NTNU
Norwegian University of
Science and Technology

Full-Text Search in XML Databases

Robin Skoglund

Master of Science in Informatics
Submission date: June 2009
Supervisor: Trond Aalberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Background The Extensible Markup Language (XML) has become an increasingly popular format for representing and exchanging data. Its flexible and extensible syntax makes it suitable for representing both structured data and textual information, or a mixture of both.

The popularization of XML has lead to the development of a new database type. XML databases serve as repositories of large collections of XML documents, and seek to provide the same benefits for XML data as relational databases for relational data; indexing, transactional processing, fail-safe physical storage, querying collections etc.

There are two standardized query languages for XML, XQuery and XPath, which are both powerful for querying and navigating the structure XML. However, they offer limited support for full-text search, and cannot be used alone for typical Information Retrieval (IR) applications. To address IR-related issues in XML, a new standard is emerging as an extension to XPath and XQuery: XQuery and XPath Full Text 1.0 (XQFT).

Results XQFT is carefully investigated to determine how well-known IR techniques apply to XML, and the characteristics of full-text search and indexing in existing XML databases are described in a state-of-the-art study. Based on findings from literature and source code review, the design and implementation of XQFT is discussed; first in general terms, then in the context of Oracle Berkeley DB XML (BDB XML).

Experimental support for XQFT is enabled in BDB XML, and a few experiments are conducted in order to evaluate functionality aspects of the XQFT implementation.

A scheme for full-text indexing in BDB XML is proposed. The full-text index acts as an augmented version of an inverted list, and is implemented on top of an Oracle Berkeley DB database. Tokens are used as keys, with data tuples for each distinct (document, path) combination the token occurs in. Lookups in the index are based on keywords, and should allow answering various queries without materializing data.

Conclusions Investigation shows that XML-based IR with XQFT is not fundamentally different from traditional text-based IR. Full-text queries rely on linguistic tokens, which — in XQFT — are derived from nodes without considering the XML structure. Further, it is discovered that full-text indexing is crucial for query efficiency in large document collections. In summary, common issues with full-text search are present in XML-based IR, and are addressed in the same manner as text-based IR.

Keywords: XML, XML databases, full-text search, Information Retrieval, XQuery, XPath, inverted list, Oracle Berkeley DB XML

Preface

This master's thesis was carried out within the Information Management (IF) group under the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). I would like to thank my supervisor Trond Aalberg for his guidance, helpful suggestions and valuable feedback during the thesis's work.

Trondheim, June 1, 2009

Robin Skoglund

Contents

I Thesis Context **1**

1 Introduction **3**
1.1 Background and motivation . . . 3
1.2 Problem definition . . . 4
1.3 Method and approach . . . 4
1.4 Outline of thesis . . . 5

2 XML and Information Retrieval **7**
2.1 XML . . . 7
2.2 XML databases . . . 8
2.3 XQuery 1.0 and XPath 2.0 Data Model (XDM) . . . 8
2.4 XML Path Language (XPath) 2.0 . . . 10
2.5 XQuery 1.0: An XML Query Language . . . 11
2.5.1 Expressions . . . 11
2.5.2 Context . . . 11
2.5.3 Query processing . . . 12
2.5.4 Error handling . . . 13
2.6 Limitations . . . 14

3 XQuery and XPath Full Text 1.0 **17**
3.1 Tokens and phrases . . . 17
3.2 Full-text extensions to XQuery and XPath . . . 17
3.2.1 Processing model . . . 18
3.2.2 Full-text contains expression . . . 18
3.2.3 Score variables and weight . . . 18
3.2.4 Extensions to the static context . . . 20
3.3 Full-Text Selections . . . 20
3.3.1 Specifying search tokens and phrases . . . 20
3.3.2 Cardinality constraint . . . 21
3.3.3 Match options . . . 21
3.3.3.1 Language . . . 21
3.3.3.2 Wildcards . . . 21
3.3.3.3 Thesaurus . . . 21
3.3.3.4 Stemming . . . 22
3.3.3.5 Case sensitivity . . . 22
3.3.3.6 Diacritics . . . 23
3.3.3.7 Stop words . . . 23
3.3.3.8 Extension option . . . 23

3.3.4 Logical full-text operators . . . 23
3.3.5 Positional filters . . . 24
3.3.5.1 Ordered selection . . . 24
3.3.5.2 Window selection . . . 24
3.3.5.3 Distance selection . . . 24
3.3.5.4 Scope selection . . . 25
3.3.5.5 Anchoring selection . . . 25
3.3.6 Ignore option . . . 25
3.3.7 Extension selections . . . 25
3.4 Summary . . . 26

4 State of the art **27**
4.1 The Quark Project . . . 27
4.1.1 TeXQuery . . . 27
4.1.2 Quark . . . 28
4.1.2.1 Storage and indexing . . . 28
4.2 Sedna . . . 30
4.2.1 Indexing . . . 30
4.2.2 Full-text search . . . 31
4.3 BaseX . . . 31
4.3.1 Storage and indexing . . . 31
4.4 Qizx . . . 32
4.4.1 Storage and indexing . . . 32
4.5 Summary . . . 33

II Thesis Contribution **35**

5 Implementing full-text search in an XML database **37**
5.1 Introduction . . . 37
5.2 General implementation of full-text concepts . . . 37
5.2.1 Tokenization . . . 37
5.2.2 Thesaurus . . . 39
5.2.3 Stop words . . . 41
5.2.4 Stemming . . . 42
5.2.5 Positional filters . . . 42
5.2.6 Relevance ranking . . . 43
5.3 Oracle Berkeley DB XML . . . 44
5.3.1 Architecture . . . 44
5.3.1.1 XQuery Engine . . . 45
5.3.1.2 Storage . . . 45
5.3.1.3 Indexing . . . 46
5.3.2 Implementing XQFT in BDB XML . . . 48
5.3.2.1 Query lexer-parser . . . 48
5.3.2.2 Tokenization . . . 49
5.3.2.3 Evaluation of full-text selections . . . 49
5.3.2.4 Enabling experimental XQFT support in BDB XML . . . 50
5.3.3 Full-text indexing in BDB XML . . . 50
5.3.3.1 Inverted list structure . . . 51

5.3.3.2 Extending the index specification with full-text options . . . 52
5.3.3.3 Extending the XML Indexer . . . 52
5.4 Summary . . . 52

6 Results **55**
6.1 Overview of experiments . . . 55
6.2 Searching for a single token . . . 55
6.2.1 Query script . . . 55
6.2.2 Result . . . 55
6.3 Searching for a phrase and a token . . . 56
6.3.1 Query script . . . 56
6.3.2 Result . . . 56
6.4 Searching with a cardinality selection . . . 57
6.4.1 Query script . . . 57
6.4.2 Result . . . 57
6.5 Searching with the case sensitivity match option . . . 57
6.5.1 Query script . . . 57
6.5.2 Result . . . 58
6.6 Searching two tokens with `ftand` . . . 58
6.6.1 Query script . . . 58
6.6.2 Result . . . 58
6.7 Searching with the `not in` operator . . . 58
6.7.1 Query script . . . 59
6.7.2 Result . . . 59
6.8 Searching with the `window` positional filter . . . 59
6.8.1 Query script . . . 59
6.8.2 Result . . . 59
6.9 Searching with `order` and `scope` positional filters . . . 60
6.9.1 Query script . . . 60
6.9.2 Result . . . 60
6.10 Searching with the `distance` positional filter . . . 60
6.10.1 Query script . . . 60
6.10.2 Result . . . 61
6.11 Summary . . . 61

III Thesis Conclusion **63**

7 Evaluation and discussion **65**
7.1 Summary of thesis . . . 65
7.2 Discussion of contributions . . . 65
7.2.1 XQFT implementation in BDB XML/XQilla . . . 65
7.2.2 Proposed full-text index for BDB XML . . . 66
7.3 Evaluation . . . 66

8 Conclusions and further work **67**
8.1 Concluding remarks 67
8.1.1 XQuery is too limited for full-text search 67
8.1.2 XQFT adds IR concepts to XQuery 67
8.1.3 IR in XML/XQFT is not fundamentally different from text-based IR 67
8.1.4 Full-text indexing is critical for query efficiency 68
8.1.5 XQFT can and will be implemented in BDB XML 68
8.1.6 Full-text indexing in BDB XML may be implemented using Berkeley DB 68
8.2 Future work 68
8.2.1 Improving the XQFT implementation in BDB XML . . . 68
8.2.2 Dealing with frequent updates 69

A XQuery Full-Text Semantics **71**
A.1 Semantics for fts:lookupThesaurus 71
A.2 Semantics for fts:resolveStopWordsUri 76
A.3 Simple SKOS thesaurus 77
A.4 Stop words represented as XML 79

B XQFT in BDB XML **81**
B.1 Patch: Fix for full-text logical operators 81
B.1.1 Lexer patch 81
B.1.2 Parser patch 82
B.2 Patch: Enable XQFT in BDB XML 83
B.3 Default tokenizer implementation 83
B.4 Example data 85
B.5 Loading example data into a container 89

Bibliography **91**

List of Figures

2.1 XQuery Processing Model 13

3.1 XQuery Full-Text Processing Model 19

4.1 A path/value index in Quark, implemented as a relational table . 29
4.2 An inverted list index in Quark 29

5.1 Query expansion with a thesaurus 40
5.2 Overview of BDB XML architecture 45
5.3 Berkeley DB XML Query Processing 46
5.4 Classes used in the evaluation of full-text selections 50
5.5 Inverted list structure in a Berkeley DB database 51

6.1 AllMatches model for “information retrieval” and “web” 56

List of Tables

2.1 XML Schema primitive datatypes 10

4.1 Summary of XML databases 33

5.1 XQFT grammar productions not parsed in XQilla 49
5.2 Incorrect XQFT grammar productions in XQilla 49

Part I

Thesis Context

Chapter 1

Introduction

This chapter presents the motivation behind the thesis. A short introduction to Information Retrieval (IR) in the context of XML is given, and the rest of the thesis is outlined.

1.1 Background and motivation

The Extensible Markup Language (XML) has become a prominent format for exchanging data over the Internet. The features of XML have made it an increasingly popular format for a vast number of applications, ranging from configuration files to vector images via message interchange formats and document markup languages. Its non-limiting and flexible syntax make it possible to express different kinds of information – highly structured data as well as freely written text – from diverse sources, and it lends well to mixing structured and non-structured information in the same document.

To cope with the increasing popularity of XML, a new database type has been developed during the last decade. XML databases serve as repositories of large collections of XML documents, and seek to provide the same benefits for XML data as relational databases (RDBMS) for relational data; indexing, transactional processing, fail-safe physical storage, querying collections etc. Several implementations exist, and there is no industry standard that defines what an XML database is, and how it should operate.

Storing data in XML requires new retrieval mechanisms, which take advantage of the inherent structure XML provides. After the standardization of XML, work began on standardizing the process of querying XML documents. The XML Path Language (XPath) has been standardized in two versions[13, 8], and is a powerful language for navigating the tree structure of XML documents. As an extension to XPath 2.0, XQuery 1.0[30] was developed as a standard to allow more complex queries than what is possible with the path language of XPath.

While XPath and XQuery are both powerful languages for structural queries, they offer very limited support for querying textual information in XML. There are several XML repositories that contain mix of semi-structured data and unstructured text data; the IEEE Initiative for the Evaluation of XML (INEX), the Library of Congress documents in XML, the Digital Bibliography & Library

Project (DBLP) in XML, SIGMOD Record in XML, Shakespear's plays in XML etc. Furthermore, application domains, such as the field of library science, have a growing need to seamlessly query over both the structured and text parts of XML documents. To address this issue, the World Wide Web Consortium (W3C) started working on a standard called XQuery and XPath Full Text 1.0 (XQFT)[18]. It is an extension to XPath and XQuery that adds full-text search capabilities. XQFT is currently a Candidate Recommendation, meaning it will soon be ready for wide deployment. The standardization of XQFT marks an important milestone for XML to be accepted and used in mainstream search applications. After the popularization of web search engines, end users expect the system to take care of IR (stemming, thesauri, stop words, case sensitivity, diacritics) and serve a list of ranked results based some algorithm. XQFT should in theory solve all of this.

1.2 Problem definition

Based on the background and motivation in the previous section, the goal of this thesis can be summarized as the following problem definition:

> Investigate how full-text search and indexing apply to XML databases. Discussions should be based around the XQuery and XPath Full Text 1.0 Candidate Recommendation and related standards. Information Retrieval in XML should be explained, and common issues with indexing should be addressed.

The problem definition above can be broken down to several sub problems. Each of these sub problems should be considered guidelines for discussion during the investigation:

1. Various index types exist for various kinds of data. Which types of indexes are required for supporting full-text search in XML, and how are they implemented?

2. Text-based IR has been a subject of study for several decades, while XML has grown popular during the last decade. XML introduces a more structured approach for representing textual documents. Are there fundamental differences between text-based IR and XML IR? Does the structure of XML call for a new approach to IR?

3. The XQFT standard defines the language and formal semantics for full-text queries, but does not give specific implementation details. Are there open issues in the standard that implementers need to keep in mind? If so; how do we address these issues to best promote cross-implementation compatibility?

1.3 Method and approach

The characteristics of XML and IR in XML will first be studied closely. In light of findings from this study, existing XML database implementations will be analyzed through literature and source code review. Then, we will take a

closer look on the implementation of various IR concepts in XML databases in general, and specifically in Oracle Berkeley DB XML. Hopefully, we will be able to implement XQFT in Oracle Berkeley DB XML. After the implementation reaches a usable state, a small number of experiments will be carried out to verify correctness of various query results. In addition, a scheme for full-text indexing will be proposed for Oracle Berkeley DB XML.

1.4 Outline of thesis

The rest of the thesis is organized as follows:

Chapter 2 Describes XML standards related to the thesis, introduces XML databases, and states why the current query languages are too limited for full-text search.

Chapter 3 Studies XQFT — the full-text extension to XQuery and XPath which is currently being developed by the World Wide Web Consortium.

Chapter 4 Gives a state-of-the-art review of some existing implementations of XML databases with support for full-text search and indexing.

Chapter 5 Provides a description of the design and implementation of various IR concepts from XQFT, and proposes modifications and additions to Oracle Berkeley DB XML.

Chapter 6 Presents some experiments which were conducted to evaluate functionality aspects of concepts described in chapter 5.

Chapter 7 Sums up the thesis, and evaluates how well the thesis achieves the goals of the problem definition.

Chapter 8 States the findings of the thesis, draws some concluding remarks, and outlines future work.

Appendix A Shows formal semantics and examples related to XQFT implementation discussions in chapter 5.

Appendix B Contains patches and example data related to the XQFT implementation in Oracle Berkeley DB XML (chapters 5 and 6).

Chapter 2

XML and Information Retrieval

This chapter outlines the features of XML that are relevant to this thesis, along with current Information Retrieval methods and their limitations. XML databases are also introduced and discussed.

2.1 XML

Since its emergence, the Extensible Markup Language (XML) standard has gradually found its way as one of the preferred formats for data exchange. It is non-binary and non-proprietary, making it platform and protocol independent, and easy to implement in programs or libraries. Raw XML may be transferred as-is between programs or users, in contrast to relational databases or other database types, where raw data must be transformed before returning it to users. In addition to data exchange, the flexible and extensible design of XML has proven useful for a wide number of diverse applications, ranging from configuration settings to document markup languages and domain-specific programming languages, via metadata formats and image formats.

XML can be highly structured, semi-structured, or unstructured. In the latter case, a document contains few structural entities such as elements and attributes, and can be considered as merely a container for storing freely written text. Highly structured XML documents, on the other hand, can be said to mimic the function of relational databases or a tables in relational databases. Highly structured documents often follow a strict set of rules specified by an XML schema[1], with constraints on structure and data. The purpose of schema specification is typically to achieve higher normalization of data, but it can also be used for expressing semantic relationships and semantic properties of documents.

Semi-structured XML is a mixture of structured data and freely written text. An example of semi-structured XML could be a patient journal. It contains some structured data — the patient's name, date of birth, person identification

[1] XML schema, "schema" in lower case, refers to any general schema language. It is not limited to the W3C XSD Schema.

number, etc — and some free text, such as diagnostical considerations written by the doctor. The free text may be structured according to a scheme, e.g., one "record" per visit, but it is mainly a collection of sentences written by a human being, and would thus require another retrieval mechanism than the highly structured data found elsewhere in the journal.

Throughout this thesis, the focus will be on semi-structured documents, because these bear the closest resemblance to real-world applications of XML databases and full-text search. If your data is highly structured, it would probably fit a relational model better, and if your data has no structure at all, you are likely to have more success using a traditional text-based IR engine.

2.2 XML databases

The features mentioned in the previous section have made XML a prominent format for transporting data over the web. More and more web applications and content providers choose XML for exposing their data and services to the world or their business partners. XML is suitable for transport over the wire, and if a system also deals with XML data internally, one may argue that it makes perfect sense to use XML throughout the system. Relational databases can indeed be automatically "shredded" to similarly disposed XML documents as shown in [16], but with this usage pattern it might prove more efficient to store data as XML in the first place. The system avoids spending computational time converting data, it is less prone to errors occurring when converting, and time spent developing the system is cut down.

Another reason for using XML databases is this crucial point; not all data is "relational". A notable amount of an organization's information simply does not fit to the relational model, because it is less structured (semi-structured), it is hierarchical and maybe sequential, and made of documents.

When data is pulled from relational databases — or other database types — and served as XML, it is called an XML-enabled database, which implies that the database software has an additional layer of query processing which maps XML operations to ordinary database operations. This layer often accepts XML as input, e.g., XPath or XQuery queries, and outputs raw XML accordingly. The internals of the database system itself perform the conversions (as opposed to using middleware), so the user only relates to XML when querying.

When the fundamental storage unit in a database is an XML document, the database is said to be a native XML database. Since the primary storage unit is XML, there is no conversion or mapping of input and output when running queries.

2.3 XQuery 1.0 and XPath 2.0 Data Model (XDM)

XQuery 1.0 and XPath 2.0 share the same underlying data model, called XQuery 1.0 and XPath 2.0 Data Model (XDM) [31], which purpose is to provide an abstract representation of one or more XML documents or document fragments. XDM is also the data model for XSLT 2.0[20], and can be summarized in two sentences;

1. it defines the information contained in the input to an XSLT or XQuery processor, and

2. it defines all permissible values of expressions in the XSLT, XQuery, and XPath languages.

XDM is based on another W3C recommendation, XML Information Set (Second Edition)[12] from 2004, but differs on some points to meet the requirements[28, 15] of XPath 2.0 and XQuery 1.0:

- Support for XML Schema types[7]
- Representation of collections of documents and complex values
- Support for atomic values
- Support for (ordered, heterogeneous) sequences

To meet those requirements, XDM defines 4 basic components:

Node — There are 7 node types: Element nodes, attribute nodes, document nodes, text nodes, processing instruction nodes, comment nodes, and namespace nodes. An important feature is that every node has a unique identity. Even though there are two equal elements in a document, they will have different identities. The identity of a node is assigned by the query processor, and is not visible to the user.

Nodes may be typed or untyped. Type - simple or complex - is acquired by validating against an XML Schema, and will be annotated to the node upon successful validation.

Atomic value — A simple data value with no markup associated with it. Atomic values may be any of the 19 predefined primitives types in the XML Schema specification, or a type that is derived from any of the primitive types. Examples of atomic values include the number 1337 and the string "Hello, World". A full list of primitive types is given in Table 2.1 on the following page.

Item — A generic term that refers to either a node or an atomic value.

Sequence — An ordered list of zero, one, or more items. A sequence containing exactly one item is called a singleton.

Understanding XDM is analogous to understanding tables, columns and rows when learning SQL. With XDM, there are other primaries that make up the model than in the relational world. A sequence could be considered XDM's answer to a relation, as it is the foundation for almost all operations — much like a relation in the RDBMS world — but XDM is able to express more complicated structures than tables, columns, and rows.

Table 2.1: XML Schema primitive datatypes

XSD Name	Represents
string	Character strings, e.g., "Hello World"
boolean	Binary-value logic, e.g. ,true or false.
decimal	A subset of real numbers that can be represented with decimal numbers.
float	IEEE single-precision 32-bit floating point type.
double	IEEE double-prevision 64-bit floating point type.
duration	A duration of time, six dimensions (year, month, day, hour, minute, and second).
dateTime	Objects with integer-valued year, month, day, hour and minute properties, a decimal-valued second property, and a boolean timezone property
time	An instant of time that recurs every day.
date	A "date object" is an object with year, month, and day properties just like those of dateTime objects, plus an optional timezone-valued timezone property.
gYearMonth	A specific Gregorian month in a specific Gregorian year.
gYear	A Gregorian calendar year.
gMonthDay	A Gregorian date that recurs, specifically a day of the year such as the third of May.
gDay	A Gregorian day that recurs, specifically a day of the month such as the 5th of the month.
gMonth	A Gregorian month that recurs every year.
hexBinary	Arbitrary hex-encoded binary data.
base64binary	Base64-encoded arbitrary binary data.
anyURI	A Uniform Resource Identifier Reference (URI).
QName	XML qualified names, such as an element name. Contains an optional namespace prefix and a local name.
NOTATION	The NOTATION attribute type from XML.

2.4 XML Path Language (XPath) 2.0

XPath 2.0 is an *expression language* that allows the processing of values conforming to XDM. In simpler words, an XPath expression returns a sequence of items from a given XML document that match the given expression. Since XPath builds on XDM, it inherits the type system from XML Schema along with the concept of the 4 basic components (node, atomic value, item, sequence). XPath adds a number of operators, and a multitude of functions in a function library. A full list of functions and operators is available in [21].

The most interesting operator in XPath is the "/" (slash) operator. It is a binary operator that applies the expression on its right-hand side to each item on the left-hand side. An expression using the slash operator is called a *path expression*, and is useful for filtering out parts of a node, most commonly a node's descendant nodes. A path expression can navigate the tree representation of an

XML document by using *axis specifiers*. The default axis is the child axis, but this and all other axises can be specified. Example: The abbreviated expression `//ul/li` is equal to `/descendant-or-self::node()/ul/child::li`, and will select all `li` child elements of all `ul` elements, which in turn are children of the current context node.

2.5 XQuery 1.0: An XML Query Language

XQuery 1.0 is an extension of XPath 2.0, and was designed by the W3C XML Query Working Group to meet the requirements[15] for an XML query language. In 1998, just after the final recommendation of XML 1.0, interest grew in having an intelligent way of querying XML. A small number of independent implementations existed within a year or two, and XQuery ended up being derived from the Quilt[11] XML query language, which in turn was based on features from other languages; XPath 1.0[13], XQL[27], XML-QL[14], SQL[2], and OQL[3]. It is fair to say that XQuery is to XML what SQL is to relations, albeit the syntax is different.

2.5.1 Expressions

Expressions are the basic units of evaluation in a query. An expression is a string of Unicode characters, and may be as simple as `2 + 3`, or as complex as almost any other programming language, by declaring and using functions, FLWOR expressions, comparisons, and path expressions from XPath. FLWOR is pronounced "flower", and is an acronym for `For, Let, Where, Order by, Return`, which are expressions for iterating sequences and and binding variables to intermediate results. E.g., `for` expressions may be used for joining two XML documents and returning combined information or aggregated results.

2.5.2 Context

Expressions are always evaluated in a certain *context*. The context of an expression contains all information that can affect the result of the expression. The XQuery standard defines two context types; static context and dynamic context. The static context contains information that is available before evaluating the expression, during the static analysis of the query. Examples of information in the static context include base URI, statically known documents and collections, ordering mode, default namespace etc. The dynamic context, on the other hand, contains information that is not available until evaluating the query. Examples of information in the dynamic context include context item (the *item* currently being processed), context position, context size (*sequence* size), variable values, current *dateTime* etc. Most importantly, the dynamic context contains the available documents and available collections.

[2] Structured Query Language, for relational databases
[3] Object Query Language, for object-oriented databases

2.5.3 Query processing

While the XQuery standard does not specify in detail how a query should be carried out in implementations, it does outline a basis for the processing model of a query. There are 5 basic elements in the XQuery processing model:

1. Data model generation
2. Schema import processing
3. Expression processing
4. Serialization
5. Consistency constraints

Data model generation takes place before query processing. Its purpose is to parse input data and provide an XDM instance, which is the basis of further processing. Schema import processing is also outside the bounds of actual query processing, and provides in-scope schema definitions for the static context. Definitions may be extracted from existing XML schemas, or provided by other mechanisms, e.g., a component in an XML-enabled database.

The expression processing is split in two phases; static analysis and dynamic evaluation. During static analysis, the query is parsed into an operation tree, the static context is initialized and augmented with information from the query prolog[4], and implementation-specific features are applied. The operation tree is normalized by making implicit operations, and each expression is assigned a static type if the static typing feature is supported in the implementation. The purpose of the static analysis is to give developers more control of the query execution, and make sure that queries are evaluated in a consistent and orderly fashion. It also has performance implications, since a query can be compiled statically and evaluated several times dynamically. The outcome of static analysis is either success or one or more type errors, static errors, or statically-detected dynamic errors.

The dynamic evaluation phase can only be started if no errors were raised during the static analysis, and depends on the operation tree, the input data, and the dynamic context (which may use external data and the static context). A dynamic type is associated with each value as it is computed. The dynamic type of a value allows for more specific constrains, e.g., limiting the number of allowed atomic values in a sequence. The result of the dynamic evaluation phase is either a result value, a type error, or a dynamic error.

Serialization is the process of converting an XDM instance to a sequence of octets, e.g., XML output, XHTML output, , HTML output, or text output. Serialization is optional, i.e., an implementation may choose to only provide a DOM interface or an interface based on an event stream.

The system of consistency constraints is a means to ensure that the XDM instance, the static context, and the dynamic context are mutually consistent. The XQuery standard lists a number of consistency constraints that are prerequisites for a correctly functioning XQuery implementation, e.g., “For every node that has a type annotation, if that type annotation is found in the in-scope

[4] A series of declarations and imports that define the environment for a query.

schema definitions (ISSD), then its definition in the ISSD must be equivalent to its definition in the data model schema.".

Figure 2.1 is copied from [30], and shows a schematic overview of what is described above. The black border denotes the boundaries of the XQuery standard.

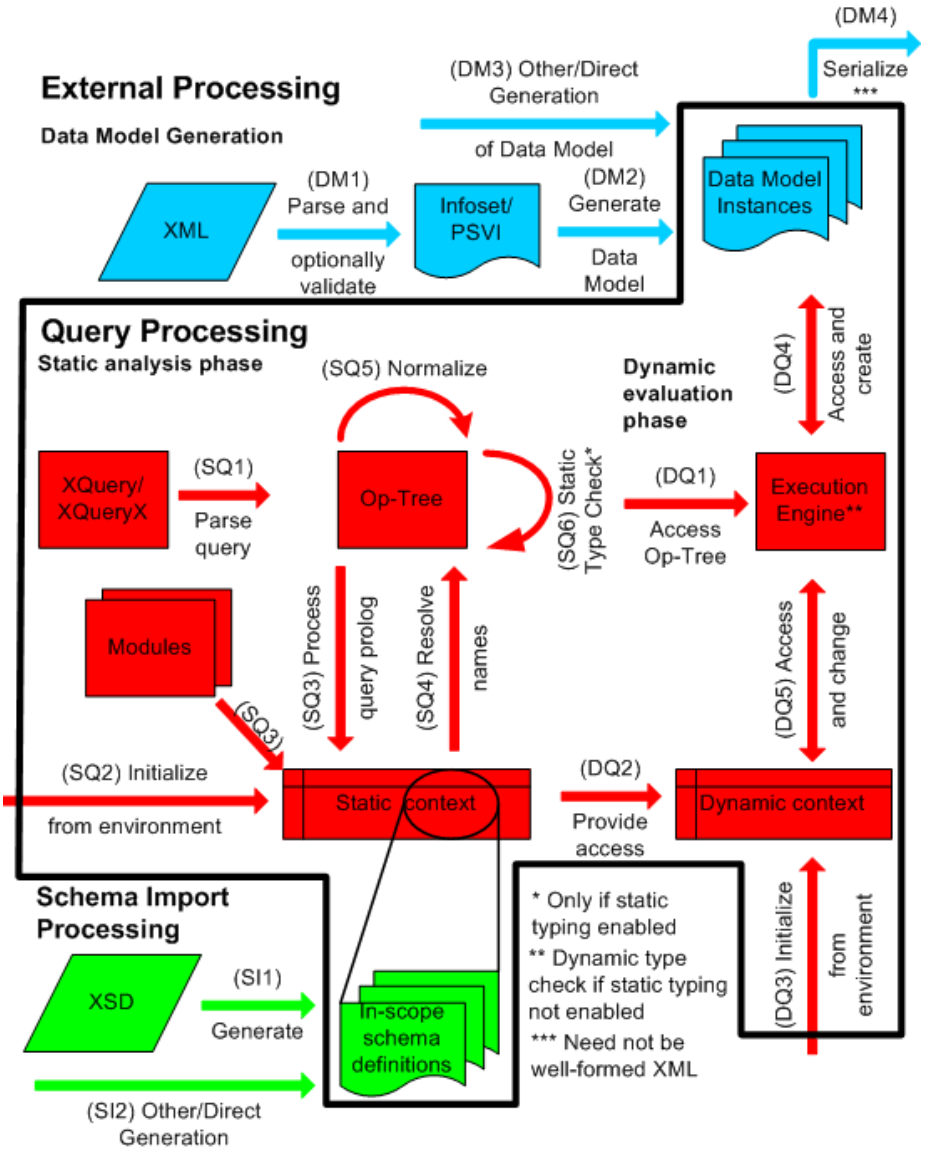

Figure 2.1: XQuery Processing Model

2.5.4 Error handling

Errors may be raised during the expression processing phases mentioned in the previous subsection. XQuery defines three error types:

A static error is an error that must be detected during the static analysis phase. A syntax error is an example of a static error.

A dynamic error is an error that must be detected during the dynamic evaluation phase and may be detected during the static analysis phase. Numeric overflow is an example of a dynamic error.

A type error may be raised during the static analysis phase or the dynamic evaluation phase. During the static analysis phase, a type error occurs when the static type of an expression does not match the expected type of the context in which the expression occurs. During the dynamic evaluation phase, a type error occurs when the dynamic type of a value does not match the expected type of the context in which the value occurs.

A vast number of conditions exist where the result of the query processing would be one or more of the aforementioned errors. Implementations may also define additional erroneous conditions outside the XQuery specification, which would result in dynamic errors.

2.6 Limitations

While XPath and XQuery are both powerful languages for structural queries, they offer very limited support for querying textual information in XML, as shown in [2, 4]. For instance, searching text in an element is in XQuery/XPath limited to boolean substring matching by use of the `fn:contains($str1, $str2)` function, which would return `true` if `$str2` is a substring of `$str1`. This may be a valid search in some situations, but the function is unable to answer queries like the following, from XML Query Use Cases[10]:

> (Use Case 2.2.6 Q6): Consider an XML document that contains books. Find all book titles which start with "improving" followed within 2 words by "usability".

Clearly, a simple substring query is unable to answer the above search, which would require the underlying implementation to support word queries, order specification, and starts-with functionality. There is also a desire for searching text across element boundaries:

> (Use Case 3.2.1 Q1): Find all book chapters containing the phrase "one of the best known lists of heuristics is Ten Usability Heuristics".

The use case above could translate into a query that looks for text spanning several elements, e.g., "... One of the best known list of heuristics is <citation url="foo">Ten Usability Heuristics by Jacob Nielson</citation> ...".

Queries with wildcards is another example of something that cannot be implemented with substring matching:

> (Use Case 5.2.5 Q5): Find all books with the word "test" with a three to four character suffix in the text.

It should match "testers" and "testing", but not "pretest", "tests" and "tested".

Wildcard queries allow any suffix. Unlike a wildcard query which could potentially match "testimony", a query using stemming would not. The following query will match "tests", "testing" and "testers", but not "testimony":

> (Use Case 6.2.1 Q1): Using stemming, find all books with the word "test" in the text.

Another IR technique that should be supported is the use of a thesaurus. An example of a search utilizing a thesaurus:

> (Use Case 7.2.1 Q1): In the same book collection, find all introductions which quote someone.

The query for the use case above should find synonyms for the word “quote”, such as “said”, “stated”, “remarks”, “replied” etc.

There are a number of other use cases involving Information Retrieval techniques that are not supported by the bare XQuery or XPath standards. This final example shows how it is desirable to give scores to results:

> (Use Case 17.2.1 Q1): Find all books which mention “usability” in the title or the text. Return book titles and scores.

Ranking results like in the query above could be done outside the boundaries of XQuery query processing, but this would prevent a query from taking advantage of scores, and is likely to be far from optimal with respect to execution time.

To summarize, XQuery has limited or no support for common IR techniques and concepts.

Chapter 3

XQuery and XPath Full Text 1.0

This chapter describes in detail the full-text extension that is currently under development for XQuery 1.0 and XPath 2.0.

3.1 Tokens and phrases

As stated in Section 2.6 on page 14, XQuery has only rudimentary support for full-text search — limited to substring matching. This is not enough to meet the expectations of full-text search from end users.

> "Where a document contains unstructured or semi-structured data, it is important to be able to apply IR techniques such as scoring and weighting." – XQFT[18]

The main difference between substring matching and full-text search is that the latter operates on *tokens* and *phrases* rather than a continuous stream of characters. Character strings are broken into linguistic tokens through the process of *tokenization.* Tokenization will typically break a string into; tokens, punctuation units, and spaces. Informally, a token is a word. Formally, it is a non-empty sequence of characters returned by a *tokenizer* as a basic unit to be searched. Further, a *phrase*, *sentence*, or *paragraph*, is an ordered sequence of any number of tokens.

Tokenization lets functions and operators operate on a part or the root of the token, which enables wildcards, stemming, and use of thesauri. Tokenization also lets functions and operators work with the relative positions of tokens, which in turn enables proximity operators.

3.2 Full-text extensions to XQuery and XPath

To add support for full-text search, XQFT extends XQuery and XPath in three ways. It 1) adds a new expression; `ftcontains`, 2) enhances the syntax of FLWOR expressions with score variables, and 3) adds full-text match options to the static context. The following subsections describe those three things in further detail.

3.2.1 Processing model

Full-text queries are defined using a full-text contains expression (`ftcontains`). To evaluate this expression, the execution engine (see Figure 2.1 on page 13) needs to tokenize both the query and the input data The input data in the case of XQFT is made from an XPath or XQuery expression, and specifies the sequence of items to be searched. This sequence is called the *search context*. With a tokenized query and search context, the execution engine will use a `Matcher` to combine tokens from each, and create instances of an `AllMatches` model, which is a means to describe all possible solutions to the query for a given search context item. Each solution is described by a `Match` instance, which contains positive (include) and negative (exclude) terms from the search context. An `AllMatches` instance is converted to a boolean value before being returned to the XQuery or XPath expression. If at least one member contains only positive terms, the result is true. If all members contain negative terms, the result is false. Figure 3.1 on the facing page shows how XQFT extends the processing model from known from XQuery. To summarize, the algorithm has the following steps:

1. Evaluate the search context expression, the ignore option (if any), and other nested expressions.
2. Tokenize the query string(s).
3. For each search context item:
   (a) Delete ignored nodes from the item.
   (b) Tokenize the result of the previous step.
   (c) Evaluate full-text selections against the search tokens.
4. Convert the topmost `AllMatches` instances into a boolean value.

3.2.2 Full-text contains expression

A full-text contains expression (`ftcontains`) is an expression that evaluates a sequence of items against a full-text selection. It returns a boolean value; true if there is some item that matches the full-text selection. The expression may be used anywhere where a comparison can be used.

This example shows an `ftcontains` expression that returns the author of each book with a title containing a token with the same root as `dog`, and the token `cat`:

```
1 for $b in /books/book
2 where $b/title ftcontains ("dog" with stemming) ftand "cat"
3 return $b/author
```

3.2.3 Score variables and weight

In addition to adding `ftcontains`, XQFT extends the `for` clause from FLWOR expressions to allow scoring of results. The score of a full-text query result expresses its relevance to the search conditions, and is the result of an implementation-dependent scoring algorithm. Score values are of type `xs:double` in the range [0, 1], and a higher score implies a higher relevance.

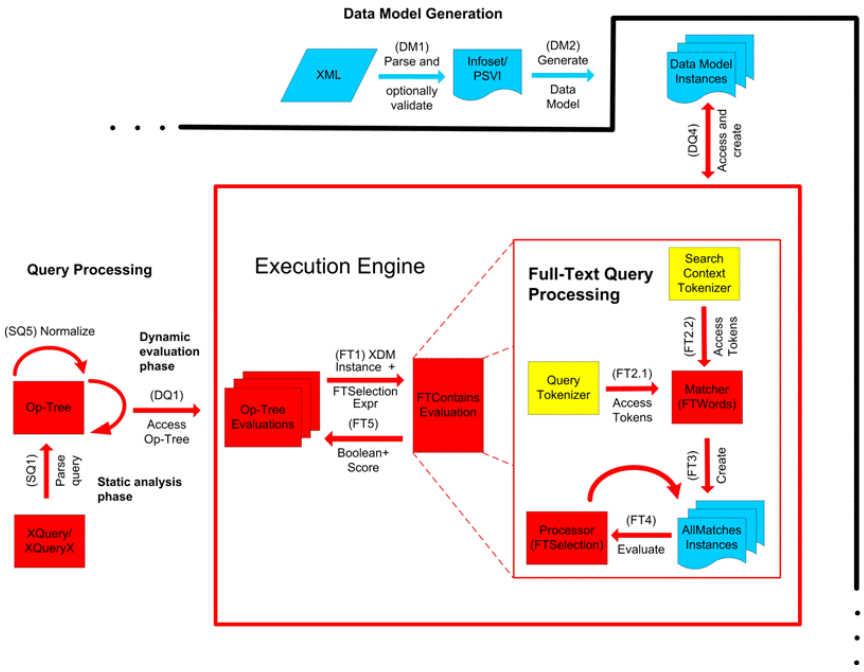

Figure 3.1: XQuery Full-Text Processing Model

This figure is taken from [18], and shows a schematic overview of how XQFT extends the processing model from Figure 2.1 on page 13. The execution engine is zoomed to show the evaluation of a `ftcontains` expression. The black border denotes the boundaries of XQuery/XQFT.

Below is an example of a query using score variables to only return a ranked list of books matching "web site" and "usability" with a relevance (score) higher than 50%:

```
1 for $b score $s
2     in /books/book[content ftcontains "web site" ftand
        "usability"]
3 where $s > 0.5
4 order by $s descending
5 return <result>
6            <title> {$b//title} </title>
7            <score> {$s} </score>
8        </result>
```

By adding weights to search tokens, a user can influence the scoring algorithm the implementation uses. Weights must have an absolute value in the range [0.0, 1000.0]. The default weight of a token is 1.0.

Example:

```
1 (: "usability" is important, "web site" not so much :)
2 for $b in /books/book
3 let score $s := $b/content ftcontains ("web site" weight 0.5)
4                                 ftand ("usability" weight 2)
```

```
5 return <result score="{$s}">{$b}</result>
```

3.2.4 Extensions to the static context

The static context (Figure 2.1 on page 13) is extended by adding support for full-text match options. Match options modify the semantics of `ftcontains` expressions, and can be declared in the query prolog using `declare ft-option <match option>`. Available match options are described in the next section.

3.3 Full-Text Selections

A full-text selection specifies the conditions for a full-text search. It contains full-text operators and match options used in an `ftcontains` expression. This subsection describes the parts involved in full-text selections.

A general full-text selection (`FTSelection`) is specified as containing logical operators (`FTOr`), optionally followed by positional filters (3.3.5 on page 24) and a weight. Semantically, the `FTOr` translates into a primary full-text selection optionally followed by match options (3.3.3 on the facing page). A primary full-text selection (`FTPrimary`) is the basic form of a full-text selection, and specifies tokens and phrases, optionally followed by a cardinality constraint (3.3.2 on the next page). A primary full-text selection may also be a general full-text selection or an extension selection (3.3.7 on page 25).

3.3.1 Specifying search tokens and phrases

Tokens and phrases that an `ftcontains` expression should search for are specified by giving a literal or an XQuery expression, which will be converted to a sequence of strings. Semantically in XQFT, this is defined as `FTWords`. The `FTWords` production also allows specifying an option for how the query tokens should be matched:

- "any" is the default if the option is omitted. This will treat the sequence of tokens for each string as a phrase. The resulting matches must contain at least one of the phrases.
- "all" will also treat the sequence of tokens for each string as a phrase. Resulting matches must contain all the phrases.
- "phrase" will concatenate the given string sequence, and a resulting match must contain the generated phrase.
- "any word" will join combine the string sequence into a single set, and a resulting match must contain at least one token from the set.
- "all words" will also combine the sequence of strings into a single set, and a resulting match must contain all the words

Examples:

```
1 (: specify a single word :)
2 //book[./title ftcontains "Expert"]
3
```

```
4 (: specify a phrase :)
5 //book[./title ftcontains "Expert Reviews"]
6
7 (: specify two words, and that all/both must match :)
8 //book[./title ftcontains {"Expert", "Reviews"} all]
```

3.3.2 Cardinality constraint

In a primary full-text selection, `FTWords` may be followed by `FTTimes`, which is the "`occurs`" operator. This is used for specifying the cardinality of a successful match of a `FTWords` operand. Valid ranges for "`occurs`" are: "`exactly <n> times`", "`at least <n> times`", "`at most <n> times`", and "`from <n> to <m>`".

Example:

```
1 (: specify cardinality using the occurs operator :)
2 //book[. ftcontains "usability" occurs at least 2 times]
```

3.3.3 Match options

3.3.3.1 Language

The language option (`FTLanguageOption`) is used for localization of full text queries. It affects tokenization, stemming, and stop words in an implementation-defined way. The default language is specified in the static context.

Example:

```
1 (: specify language option to select appropriate stop word list :)
2 //book[@number="1"]//editor ftcontains "salon de the"
3 with default stop words language "fr"
```

3.3.3.2 Wildcards

The wildcard option (`FTWildCardOption`) enables wildcard matching of tokens and phrases. Valid options are `with wildcards` and `without wildcards`, the latter being the default. The syntax for wildcards is the same as for regular expressions: a period (.) indicates a wildcard, and quantifiers (`?`, `*`, `+`, `{n,m}`) may be used for specifying the range of characters that should match the wildcard.

Example:

```
1 (: specify wildcard option :)
2 //book[@number="1"]/title ftcontains "improv.*" with wildcards
```

3.3.3.3 Thesaurus

A thesaurus option (`FTThesaurusOption`) modifies token and phrase matching by specifying whether and how thesauri should be used. An option can specify 1) the location of thesauri (either a `default` or a URI reference), and optionally 2) the relationship to be applied (see below) and 3) how many levels hierarchical relationships should be traversed. (2) and (3) are only used if a thesaurus location is given as a URI reference.

When enabled, a thesaurus will modify the words in the query, or add tokens and phrases to the query. The search will then be performed as though the used had specified additional tokens in a disjunction (`FTOr`). How the thesaurus is represented is implementation-dependent, and may be e.g., a topic map, an ontology, a taxonomy, a soundex etc.

XQFT states that relationships include, but are not limited to, the relationships and their abbreviations in ISO 2788[1], and their equivalents in other languages. Listed here in English, with abbreviations in brackets:

1. equivalence relationships (synonyms): PREFERRED TERM (USE), NON-PREFERRED USED FOR TERM (UF);
2. hierarchical relationships: BROADER TERM (BT), NARROWER TERM (NT), BROADER TERM GENERIC (BTG), NARROWER TERM GENERIC (NTG), BROADER TERM PARTITIVE (BTP), NARROWER TERM PARTITIVE (NTP), TOP Terms (TT); and
3. associative relationships: RELATED TERM (RT).

Example:

```
1 (: use thesaurus to also match synonyms of "duties", e.g., "tasks"
     :)
2 count(//book/content ftcontains "duties" with
3 thesaurus at "http://cdn.example.com/thesaurus.xml"
4 relationship "UF")
```

3.3.3.4 Stemming

The stemming option (`FTStemOption`) modifies token and phrase matching by specifying whether stemming should be used. Valid values are `with stemming` and `without stemming` (the default). How stemming should be performed is implementation-defined.

Example:

```
1 (: use stemming to also match "improving" :)
2 /books/book[@number="1"]/title ftcontains "improve" with stemming
3
4 (: the result would be different from a wildcard match, which
5    potentionally could match "improvising" and others :)
6 /books/book[@number="1"]/title ftcontains "improv.+" with wildcards
```

3.3.3.5 Case sensitivity

A case option (`FTCaseOption`) specifies how case should be considered. There are four possible options: `case insensitive` (default), `case sensitive`, `lowercase`, and `uppercase`. The second will only match tokens with the same case as specified in the query. The last two options will only match tokens of the specified case.

Example:

```
1 (: specify that only lowercase tokens/phrases should be matched :)
2 //book[@number="1"]/title ftcontains "Usability" lowercase
```

[1] ISO 2788: 1986: Documentation – Guidelines for the establishment and development of monolingual thesauri.

3.3.3.6 Diacritics

A diacritics option (`FTDiacriticsOption`) specifies whether token matching should be sensitive to diacritics. Possible options are `diacritics insensitive` (default) or `diacritics sensitive`.

```
1 (: specifying "diacritics sensitive" will not match "Verá" :)
2 //book[@number="1"]/editors ftcontains "Vera" diacritics sensitive
```

3.3.3.7 Stop words

A stop word option (`FTStopWordOption`) influences matching of `FTWords` by specifying whether stop words are used. Like with the thesaurus option, the stop word option may specify a default list that should be used (from static context or otherwise in an implementation-defined manner) or giving a URI reference to a location where a stop word list could be found. In addition, a list of stop words may be given as a comma-separated list of string literals directly in the query. Multiple lists can be combined using one of the keywords `union` or `except` between lists. The default is `without stopwords`.

If stop words are used, any token matching a stop word will be removed from the search, and may be substituted by any other token.

Example:

```
1 (: will match "propagating few errors" :)
2 /books/book[@number="1"]//p ftcontains "propagation of errors"
3 with stemming with stop words ("a", "the", "of")
```

3.3.3.8 Extension option

The extension option (`FTExtensionOption`) is defined by XQFT as an implementation-defined match option. The reason for specifying this syntax is to allow queries utilizing custom match options to be successfully parsed by any implementation. An extension option consists of the literal "`option`" followed by a `QName` and a string literal.

Example:

```
1 (: pass an extension option to the implementation's internals :)
2 //para[. ftcontains
3          ("Kinder" ftand "Platz" distance exactly 1 words)
4          with stemming
5          option exq:compounds "distance=1" ]
```

3.3.4 Logical full-text operators

Full-text selections can be combined with logical operators:

- `ftor`; finds all matches that satisfy at least one of the operands,
- `ftand`; finds matches that satisfy all of the operands,
- `ftnot`; finds matches that do not satisfy the operand, and
- `not in`; is a milder form of the operators `ftand ftnot`. The selection `A not in B` finds tokens matching A, but not if the token is part of a match in B.

Example:

```
(: or :)
//book[.//author ftcontains "Millicent" ftor "Voltaire"]

(: and :)
//book/author ftcontains "Millicent" ftand "Montana"

(: and not :)
//book ftcontains "information"
            ftand "retrieval"
      ftand ftnot "information retrieval"

(: not in :)
/books/book ftcontains "usability" not in "usability testing"
```

3.3.5 Positional filters

Positional filters are postfix operators used for filtering matches based on their positional information, such as constraints on distance and document/semantic scope. An arbitrary number of filters may be specified after a full-text selection. Multiple filters will be applied from left to right (output from first filter is input to second filter etc.).

3.3.5.1 Ordered selection

An ordered selection is a full-text selection followed by the postfix operator `ordered`. An ordered selection will only match if the tokens in the search text appears in the same order as in the query.

Example:

```
(: will not match the string "lorem ipsum dolor sit amen" :)
//book/title ftcontains ("ipsum" ftand "lorem") ordered
```

3.3.5.2 Window selection

A window selection is a full-text selection followed by one of the window operators. A window operator specifies that each matching token in the search text must occur within a given number of units from each other. Possible units are `words`, `sentences`, or `paragraphs`. A window selection may cross element boundaries, and stop words are included when computing the window size.

Example:

```
(: would not match the string
   "a web site about how to improve usability" :)
//book/content ftcontains "web site ftand "usability" window 3
    words
```

3.3.5.3 Distance selection

A distance selection is a full-text selection followed by one of the distance operators. It can be considered a more detailed version of a window selection, as it specifies the distance of units (a range) that each matching token in the search text must occur within. The range is the same as for cardinality selection

(`exactly`, `at least`, `at most`, `from <n> to <m>`), and the unit is the same as for window selection (`words`, `sentences`, `paragraphs`).

Example:

```
1 (: will not match "the usability of a web site" :)
2 /books/book ftcontains "web" ftand "site" ftand "usability"
3 distance at most 2 words
```

3.3.5.4 Scope selection

A scope selection is a full-text selection followed by one of the scope operators. A scope operator specifies that matching tokens must occur in either the `same` or `different` unit. Valid units are `sentence` and `paragraph`.

Example:

```
1 (: will not match if tokens are in different paragraphs :)
2 //book[. ftcontains "usability" ftand "testing" same paragraph]
```

3.3.5.5 Anchoring selection

An anchoring selection is a full-text selection followed by one of the operators `at start`, `at end`, or `entire content`.

Example:

```
1 (: will match a string ending in "propagating very few errors" :)
2 /books//p[. ftcontains "propagat.*" with wildcards
3                 ftand "few errors"
4               distance at most 2 words at end]
5
6 (: will return each b element whose entire content is "web site" :)
7 /books//b[. ftcontains "web site" entire content]
```

3.3.6 Ignore option

The ignore option specifies a set of nodes whose content are ignored when processing the query.

Example:

```
1 //book[./chapter ftcontains "usability"
2             without content chapter//annotation]
```

3.3.7 Extension selections

As with the extension match option, XQFT defines an implementation-defined extension selection to enable custom selections in implementations without breaking parsing in others. An extension selection consists of one or more pragmas followed by a full-text selection closed in curly braces. A pragma is an identifying QName and implementation-defined content, delimited by "`(#`" and "`#)`".

Example:

```
1 (: tell the underlying implementation to use indexing :)
2 /books/book/author[name
3   ftcontains (# exq:use-index #) {'Berners-Lee'}]
```

3.4 Summary

We have seen that XQFT on an abstract level defines a full-text extension to the existing XQuery standard. The most notable addition is the *full-text contains expression*, which performs *full-text selections* on *sequences* of *items*. Full-text selections allow specifying a multitude of options and filters as conditions for the search. All of those, however, require that both the query and the sequence of items are segmented into *tokens* and *phrases*, through the process of *tokenization*.

XQFT does not specify how parts of the standard should be implemented. To borrow a term from object-oriented programming; XQFT merely defines the interface to which an implementation must conform. Concepts such as indexing, storage mechanisms and access methods are outside the scope of the standard, and no hints are given on how these concepts should be dealt with (except that indexing may or may not influence search conditions).

Chapter 4

State of the art

This chapter provides an analysis of how full-text search and indexing are implemented in existing XQuery extensions and XML database projects.

4.1 The Quark Project

The Quark Project[1] at the Cornell Database Group was a project that aimed to provide an XML database with efficient search using a mix of structured queries and full-text predicates. They developed TeXQuery[3]; a full-text extension to XQuery that was a precursor to XQFT, and Quark[6, 29]; an efficient XQFT implementation.

4.1.1 TeXQuery

The structural part of queries in Quark are regular XQuery queries, which retrieve sequences of items. The full-text part of queries was originally based on TeXQuery — a full-text extension to XQuery that was a precursor to XQFT. TeXQuery adds full-text support by defining a full-text contains expression (`FTContainsExpr`) and a score expression (`FTScoreExpr`). Both expressions work on full-text selections, which as in XQFT specify the conditions (tokens, phrases, boolean connectivities, scoping, term weights) for a full-text search. TeXQuery also adds *context modifiers*, which are equivalent to match options in XQFT. The difference between TeXQuery and XQFT is that the latter has matured more, leading to a more generalized approach (i.e., a *standard*). For example, the full-text contains expression in TeXQuery works only on full-text selections, whereas the XQFT version works on the more general *range expression*. The score expression in TeXQuery is defined as score variables in XQFT, which are used in conjunction with FLWOR expressions to make it easier to write queries. All in all, TeXQuery was very influential on the standardization of XQFT (the authors of TeXQuery are also editors of XQFT).

[1] http://www.cs.cornell.edu/bigreddata/Quark

4.1.2 Quark

The Quark Project identifies an interesting issue that is likely to occur in large-scale XML search applications: traditional IR engines rely on the assumption that documents are *materialized*, i.e., static documents that can be parsed, tokenized, and indexed when documents are loaded into the system. Quark states that there are several search applications in XML/IR where queries do not operate on materialized base data, but rather a *virtual view* made up of fragments of several documents, or even aggregated from web services. Examples:

- *personalized views* for large-scale web portals or enterprise search; the view a user is searching is specific for the user, e.g., based on personal interests or access levels in the system, and
- *information integration*; a user is searching a view that is aggregated from web services.

Quark argues that maintaining materialized virtual views is often undesirable and impractical, because:

- a system with many users implies having many views that need to be constantly updated,
- content might be overlapping between users, leading to duplicates and wasted resources, and
- materialized views might be out-of-date with respect to the base data they aggregate.

To accommodate for keyword searches in virtual views, Quark utilizes indexes on the *base* data to efficiently determine which parts of the data that are relevant to the query. Only those that are relevant will eventually be materialized.

4.1.2.1 Storage and indexing

Documents are stored in a compressed binary format in the file system, allowing stream based XML processing. Each element is given a unique identificator, a Dewey ID, which is a way to encode the order of elements. The Dewey ID for an element is prefixed with the ID of the parent element, which allows for traversing an XML tree in both directions (upward and downward) using IDs. Using Dewey IDs in various forms of inverted lists was shown to be successful for ranked keyword searches[17]. This is reflected in Quark's indexing strategy.

Quark uses two types of indexes internally:

- Structure+Value Based Index (SVBI), a path index used in path queries, and
- Structure Based Inverted List Index (SBILI), used to evaluate keyword searches over materialized documents.

The SVBI could be implemented as a relational table, as in Figure 4.1. Each row contains a unique pair of Path and Value, and a list of IDs to elements on the Path with a matching Value. A B+-Tree index could be built on the (Path, Value) pair to allow fast lookups. The index would be used as follows:

- A path query with value predicates (`/book/author/fn[. = 'Jane']`) uses the index to find paths matching the value "Jane".
- A path query without predicates merges the IDLists corresponding to the given path.
- A path query with descendant axes (`/book//fn`) probes the index for each full data path (e.g., `/book/name/fn`)
- A *twig query*[2] is evaluated in two steps; 1) by evaluating each individual path query in the twig and 2) merging the results based on Dewey ID.

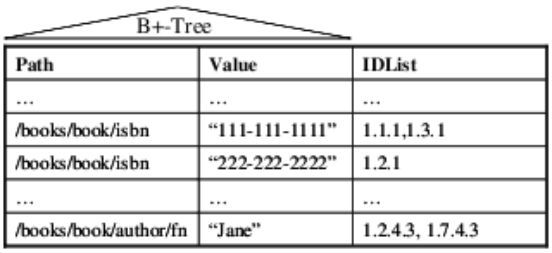

Figure 4.1: A path/value index in Quark, implemented as a relational table

This figure is copied from [29] fig. 5.

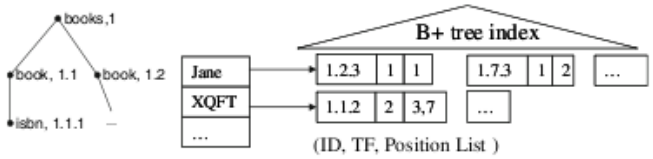

Figure 4.2: An inverted list index in Quark

The tree on the left-hand side represents an example document with Dewey IDs for each node, which are mapped to keywords in the inverted list on the right-hand side. This figure is copied from [29] fig. 4.

The SBILI, shown in Figure 4.2, is used for linking keywords with elements that directly contain them. Each keyword in the document collection has a list of element IDs, in addition to extra information, such as term frequency and positional information. As with the SVBI, a B+-tree index can be built on top of the SBILI, which allows for efficiently checking whether a specific element contains a given keyword.

By using indexes to create and materialize (pruned) views, the query processor avoids doing expensive joins and accessing base data directly. This is crucial for large-scale search applications, and Quark claims to be a very efficient solution.

[2] "A twig query is a small tree, whose nodes n represent simple predicates p_n on the content (text) or the structure (elements) of a queried XML document, whereas its edges define the desired relationship between the items to match." — Quote from [6].

4.2 Sedna

Sedna[3] is an open source native XML database with support for XQuery and full-text search. The full-text extension is based on the commercial dtSearch Text Retrieval Engine[4], and supports:

- token/phrase searches,
- boolean operators,
- proximities,
- thesauri,
- stemming,
- wildcards,
- fuzzy searching; a proprietary algorithm that finds misspelled words, and
- relevance ranking with scoring and weights.

4.2.1 Indexing

Sedna supports two index types: value indexes and full-text indexes.

A value index is specified by giving a general path, a key path, and an atomic type the content will be cast to before stored in the index. Example:

```
1 CREATE INDEX "books"
2 ON doc("books")/book BY author/lastname
3 AS xs:string
```

The index above will index `book` elements by `author/lastname`. Authors' last names will be cast to `xs:string` values and used as keys in the index.

A full-text index is a path index that stores an element's contents either as XML or as a string value, or a customized mixture of those. If an XML index is created, the XML representation of a node will be stored as-is. A string value index will use the `fn:string` function to make string versions of the node (including all descendant nodes). Example:

```
1 CREATE FULL-TEXT INDEX "abstracts"
2 ON doc("books")//abstract
3 TYPE "string-value"
```

Unfortunately, Sedna does not provide documentation on how indexes are utilized by the query processor. The dtSearch engine is a commercial third-party product which is not bundled with Sedna, and none of the companies provide sufficient information on implementation details. However, if the full-text index stores a serialized string value of the entire node, then index cannot be used for queries with positional filters, because this information is lost when the node is converted to a string. Furthermore, if the index stores XML intact, certain lookups in the index will require a scan of the entire node to determine which parts of the XML that are relevant to the query.

[3] Sedna XML Database: http://modis.ispras.ru/sedna/

[4] dtSearch Text Retrieval Engine: http://www.dtsearch.com/PLF_engine_2.html

4.2.2 Full-text search

Sedna defines two auxiliary XQuery functions for performing full-text search:

1. `ftindex-scan($title as xs:string, $query as xs:string) as node()*`
2. `ftscan($seq as node()*, $query as xs:string, $type as xs:string, $customization_rules as xs:string) as node()*`

The first function searches an index directly, e.g., using the `$query` "apple and not pear". The second function searches a sequence of nodes that are not indexed, and allows specifying the parameters known from the full-text index. Both functions return a sequence of matching nodes, whereas XQFT returns a sequence of items.

4.3 BaseX

BaseX is an open source native XML database developed by the Database and Information Systems Group at the University of Konstanz, Germany. BaseX supports full-text search by implementing XQFT. They claim to offer the most complete implementation of the standard. As an extension to XQFT, they add a match option called `fuzzy`, which uses Levenshtein distance to match misspelled words (fuzzy searching). Example of a fuzzy query:

```
1 (: will also match "usability" :)
2 //book[./title ftcontains 'usabiliti' with fuzzy]
```

4.3.1 Storage and indexing

BaseX supports 3 indexes for databases, which are enabled/disabled upon creation of a database. If enabled, the indexes will be used for all queries automatically, i.e., a user is not required to specify in a query that indexes should be used.

Elements, documents, text nodes, and attributes are stored in binary form in a node table (not relational). For each node type, the table stores various information, such as unique node ID, number of attributes and descendants, distance to parent node, and references to text values and attributes. The query processor communicates with a data access layer, which encapsulates the storage mechanism(s) and available indexes. The various index types are described below:

Text index speeds up text-based queries by indexing text nodes. It is based on a B-tree, and supports exact and range queries. The data access layer probes the index for text values, and get offsets in return, which may then be used in the node table to retrieve more information for the match. An example query that will utilize the text index:

```
1 //book[title = 'Usability']
```

Attribute index works exactly like the text index, but for attribute values (instead text nodes in elements). An example query that will utilize the attribute index:

```
1 //book[@id = '42']
```

Full-text index comes in two versions; fuzzy and trie (specified when creating the index). The fuzzy version is implemented as a sorted array, and is optimized for simple and fuzzy searches. It uses a binary search to find indexed keywords, and will — as with the text index — return offsets to the node table where more information is found. The trie version is implemented as a compressed trie, and allows using the index for wildcard searches, though it requires more memory and is less efficient for fuzzy searches.
For each token in the query, BaseX calculates and stores references to the current sentences and paragraph, along with positional information and other flags (match options). This allows BaseX to use the index to answer queries with scoping and proximities.

With its indexes, the BaseX system should be able to only materialize the necessary parts of a query result.

4.4 Qizx

Qizx[5] is an "XML indexing and searching engine" developed by XMLMind, a division of Pixware[6]. Qizx supports most of XQFT. *Big units* ("sentence" and "paragraph") are not supported in distance and window selections, and scope selection (which relies solely on big units) is not supported. Stop words are also not supported, as Qizx considers stop words a feature of the past (when reducing index size was essential).

Scoring in Qizx is on a document-level, i.e., the same score will be given to all matching nodes in a document. The default scoring implementation is the result of a normalized token weight multiplied by relative term frequency ($\frac{number\ of\ occurences\ in\ document}{average\ number\ of\ occurences\ in\ all\ documents}$). Scoring, stemming, thesaurus, and parsing (tokenization) are pluggable features, meaning developers can provide customized implementations. No stemming algorithm or thesaurus is provided by default.

4.4.1 Storage and indexing

Qizx stores XDM representations of documents in *libraries*, which are persisted on disk in compressed form. There are four indexes available, which — as in Quark and BaseX — are utilized automatically by the query processor (as opposed to voluntarily in queries). Indexes are also stored in compressed form.

Qizx does not disclose how storage mechanisms and indexing are implemented, although one may assume that indexes are organized as a sorted data structure which allows fast lookups, and also contains positional information and term frequencies to avoid expensive I/O and materialization of documents. There is reason to believe that records in indexes contain pointers/offsets to the actual nodes in the binary document storage. The following index types are available:

[5] Qizx: http://www.xmlmind.com/qizx/
[6] Pixware: http://www.pixware.fr/

Element index Contains element names and information about child/descendant relationships. A lookup takes an element name, and returns all elements in all documents of a library matching the given name.

Attribute index Contains attribute names and their values. A lookup takes an attribute name and a value, and returns all elements with the given attribute and value.

Simple content index Contains element values (text nodes) that are recognized as tokens, i.e., the element contains a single value without any whitespace. A lookup takes an element name and a simple value, and returns all elements with the given name and value.

Full-text index Contains all keywords/tokens with a given minimum and maximum length. A lookup takes a keyword, and returns all *text nodes* containing an occurrence of the given word.

4.5 Summary

We have investigated four state-of-the-art XML databases through literature and source code. Several other databases have been left out of the study, because they a) do not use XQuery for querying, b) do not use XQFT (or similar approaches) for full-text search, or c) are undocumented and source code not available.

Indexing has proven to be essential for retrieval performance (except for very small document collections). It is simply too expensive to query large data sets without an index, since common queries will require a scan of entire document collection (sometimes several scans), leading to massive I/O traffic.

Most implementations seem to store XML data in a compressed binary format, allowing random access (streaming capabilities). Each stored node is given a unique identifier which allows addressing nodes directly in the physical storage. Indexes will typically store the IDs as pointers/offsets combined with term frequencies and positional information. Depending on the information stored in indexes, a query may be answered entirely by index lookups, thus reducing I/O to only materializing the relevant parts of a document/element/node.

The table below summarizes relevant features of the mentioned XML databases.

	Quark	**Sedna**	**BaseX**	**Qizx**
Full-text engine	XQFT	dtSearch	XQFT	XQFT
Structure index	Yes	No	No	Yes
Value index	Yes	Yes	Yes	Yes
Full-text index	Yes	Yes	Yes	Yes
Index utilization	Automatic	Voluntary	Automatic	Automatic

Table 4.1: Summary of XML databases

Part II

Thesis Contribution

Chapter 5

Implementing full-text search in an XML database

This chapter describes and discusses the design and implementation details of various IR concepts from XQFT; first in general terms, then in the context of Oracle Berkeley DB XML.

5.1 Introduction

In this chapter, we will see how XQFT may be implemented in an XML database that already supports XQuery, but has no full-text search capabilities. When implementing XQFT as an extension to XQuery, there are a number of issues that need to be considered.

As shown in Chapter 3, the processing model is different, and requires modifications to the execution engine (query processor) beyond the syntactical additions XQFT introduces. Queries and data must be tokenized to enable full-text search at all, and several queries require more information than just tokens to be answered, e.g., a token's position and scope, or match options which should be tailored to the content of the database, such as thesauri and stemming.

In Chapter 4, we investigated a few XML databases that already implement XQFT, and learned that choice of indexing strategy is crucial for efficiently querying large data sets. A full-text index is different from a regular value index, and should leverage positional queries and scoring in addition to simple keyword lookups.

In the following sections, we take a deeper look at how XQFT and full-text indexing may be implemented an XML database. We start by discussing a few selected IR concepts in general, and proceed to see if and how those may be implemented in Oracle Berkeley DB XML.

5.2 General implementation of full-text concepts

5.2.1 Tokenization

Tokenization is at the core of every full-text search, be it in XQFT or any other text-based IR system. As we will see in this subsection, tokenization of XML

for full-text search is not fundamentally different from tokenization of regular text.

The following definition is taken from [18]:

> "Formally, tokenization is the process of converting an XDM item to a collections of tokens, taking any structural information of the item into account to identify token, sentence, and paragraph boundaries. Each token is assigned a starting and ending position."

The definition above is elaborated in a list of six constraints that implementations must conform to:

1. Each token must consist of one or more characters.
2. Tokenization of an item must include only tokens derived from the string value of that item.
3. The tokenizer should, when tokenizing two equal items, identify the same tokens in each.
4. The starting and ending position of a token must be integers, and the starting position must be less than or equal to the ending position.
5. In the tokenization of an item, consider the range of token positions from the smallest starting position to the largest ending position; every token position in that range must be covered by some token in the tokenization. That is, for every token position P, there must exist some token T such that T's starting position <= P <= T's ending position.
6. The tokenizer must preserve the containment hierarchy (paragraphs contain sentences contain tokens) by adhering to the following constraints:
   (a) Each token is contained in at most one sentence and at most one paragraph. (In particular, this means that no tokens of any sentence are contained in any other sentence, and no tokens of any paragraph are contained in any other paragraph.)
   (b) All tokens of a sentence are contained in at most one paragraph.
   (c) The range of token positions from the smallest starting position to the largest ending position in a sentence does not overlap with the token position range from any other sentence.
   (d) The range of token positions from the smallest starting position to the largest ending position in a paragraph does not overlap with the token position range from any other paragraph.

As we see, for an implementation to conform to the XQFT standard, the tokenization process must take account of a token's positional information and scope. In addition, constraint 2 states that tokens must be derived from the string value of an item, i.e., `fn:string($item)`, which implicitly leaves out attributes and comments, and more importantly; it dissolves element boundaries. An element containing child elements will after tokenization be represented as a single string (segmented in tokens) without the notion of child elements.

A typical tokenizer for western languages will consider each word a token. However, a sophisticated tokenizer might be language-dependent, yielding different results for a word depending on the language. For example, in Norwegian, a compound word like "busstopp" might result in the tokens "busstopp", "buss", "stopp", whereas a language-independent tokenizer will only see one word.

We propose a general language-independent tokenizer with the following rules:

1. Tokens are split by whitespace (space, horizontal tabulation, newline), and common punctuation marks (period, comma, quotation mark, semicolon, etc.). For each token, increase a position counter (integer) by one.

2. Sentences are split by end-of-sentence punctuation marks (period, colon, exclamation mark, question mark). For each sentence, increase a sentence counter (integer) by one.

3. Paragraphs are split by newline whitespace. For each paragraph, increase a paragraph counter (integer) by one.

4. A token stores its word, along with position, sentence, and paragraph relative to the item the token was derived from.

With those rules, an item in the search context needs only a single scan (the toknization) to answer queries with positional filters.

Besides the ability to parse XDM instances, a tokenizer in XQFT is not remarkably different from a tokenizer in text-based IR systems.

5.2.2 Thesaurus

As mentioned in 3.3.3.3 on page 21, a thesaurus will expand queries by adding related terms in a disjunction. An example of this is shown in Figure 5.1 on the following page. A query asking for "car" might be expanded with narrower terms, such as "sedan" and "roadster". In the example, the query specifies that a thesaurus can be found at a given URL. The query processor will fetch the specified document, parse it, find related terms with respect to the match option (relationship, range), and add terms to the query accordingly. This implies that the query processor has knowledge of the schema for thesauri, so it will be able to parse it and extract information from it. How a thesaurus should be represented is up to implementers.

An issue for implementers is that there is no standard or preferred way for representing thesauri. The ISO standard mentioned in XQFT is only used for suggesting the relationships an implementation *should* support. Consequently, a query using thesauri may yield different results in different implementations, or even raise errors depending on the implementation.

XQFT being an XML and web standard, it would make sense to represent thesauri as XML. In this regard, there have been made some efforts by various organizations; Medical Subject Headings[1] by the US National Library of Medicine; the Alexandria Digital Library Thesaurus Protocol[2] at the University of California, Santa Barbara; The Open University Thesaurus[3]; and others.

[1] MeSH: http://www.nlm.nih.gov/mesh/
[2] ADL Thesaurus Protocol: http://www.alexandria.ucsb.edu/~gjanee/thesaurus/
[3] Open University Thesaurus Schemas: http://guardians.open.ac.uk/schemas/thesaurus/

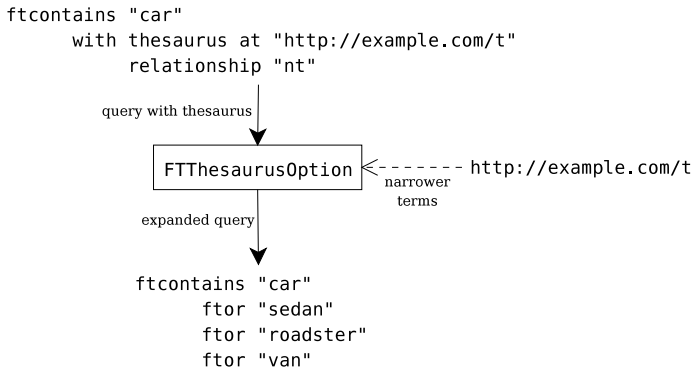

Figure 5.1: Query expansion with a thesaurus

The most standardized approach, however, is the SKOS Simple Knowledge Organization System[23] published by the Semantic Web Deployment Working Group[4] as part of the W3C Semantic Web Activity[5]. SKOS defines a common data model for knowledge organization systems, based on the OWL Web Ontology Language[25], using a Resource Description Framework (RDF)[9] syntax, such as RDF/XML[5]. The following example is based on an example from [22], and shows an entry for the word "automobile" expressed as RDF/XML:

```
1  <rdf:RDF
2    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3    xmlns:skos="http://www.w3.org/2008/05/skos#">
4
5      <skos:Concept rdf:about="http://example.com/t#1">
6          <skos:prefLabel>automobile</skos:prefLabel>
7          <skos:altLabel>car</skos:altLabel>
8          <skos:broader rdf:resource="http://example.com/t#2"/>
9          <skos:narrower rdf:resource="http://example.com/t#3"/>
10         <skos:narrower rdf:resource="http://example.com/t#4"/>
11         <skos:narrower rdf:resource="http://example.com/t#5"/>
12         <skos:related rdf:resource="http://example.com/t#6"/>
13     </skos:Concept>
14
15 </rdf:RDF>
```

As the example above shows, thesaurus terms are referenced by `rdf:resource` attributes. When parsing the thesaurus, each term must be retrieved based on the `rdf:resource` attribute. The retrieved terms will be similar to the example listing above, and the keywords to expand the query with are retrieved from either the `skos:prefLabel` element or the `skos:altLabel` element (if the specified relationship is "uf"). A complete example for the thesaurus above is given in Appendix A.3.

XQFT defines the following semantics for applying an `FTThesaurusOption`:

```
1 declare function fts:applyThesaurusOption (
2         $matchOption as element(fts:thesaurus),
3         $queryTokens as element(fts:queryToken)* )
4     as element(fts:queryItem)*
```

[4]Semantic Web Deployment Working Group: http://www.w3.org/2006/07/SWD/
[5]W3C Semantic Web Activity: http://www.w3.org/2001/sw/Activity

```
{
    if ($matchOption/@thesaurusIndicator = "with") then
        fts:lookupThesaurus( $queryTokens,
                             $matchOption/fts:thesaurusName,
                             $matchOption/@language,
                             $matchOption/fts:relationship,
                             $matchOption/fts:range )
    else if ($matchOption/@thesaurusIndicator = "without") then
        <fts:queryItem>
        {$queryTokens}
        </fts:queryItem>
    else ()
};
```

The semantics for the `fts:lookupThesaurus` function is defined as a stub in XQFT:

```
declare function fts:lookupThesaurus (
        $tokens as element(fts:queryToken)*,
        $thesaurusName as xs:string?,
        $thesaurusLanguage as xs:string?,
        $relationship as xs:string?,
        $range as element(fts:range)? )
    as element(fts:queryItem)* external;
```

An example implementation of the semantics for the `fts:lookupThesaurus` function is given in Appendix A.1.

Except for how they are represented, thesauri in XQFT are not very different from their text-based siblings in traditional IR systems. Queries are expanded by adding related keywords based on lookups in a thesaurus.

5.2.3 Stop words

Contrary to a thesaurus — which adds tokens to a query — stop words will remove tokens from a query. If a token in a query matches a stop word, that token will not be matched against tokens in the search context. A query token matching a stop word will retain its positional information, which is considered in distance or window searches (positional filters).

The `FTStopWordsOption` allows for stop words to be retrieved from a URI reference. As with thesauri, this introduces an issue for cross-implementation compatibility, because there is no defined standard way of representing stop words. We looked at the SKOS W3C standard for representation of thesauri, but no equivalent standard exists for stop words. However, a list of stop words is in its nature less complex than a thesaurus, and may thus be represented with a simpler (less verbose) syntax.

Semantically, a list of stop words resolves to a sequence of `xs:string` items. XQFT defines the semantics for applying a thesaurus option (`fts:applyThesaurusOption`), and a function stub for resolving a URI reference to a list of stop words (`fts:resolveStopWordsUri`). We propose the following syntax for representing stop words as XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<stopWords>
    <stopWord>a</stopWord>
    <stopWord>of</stopWord>
    <stopWord>or</stopWord>
</stopWords>
```

A more complete example is given in Appendix A.4. The semantics for `fts:applyThesaurusOption` and our example implementation of `fts:resolveStopWordsUri` are given in Appendix A.2.

As it turns out, stop words in XQFT are not remarkably different than stop words in text-based IR systems. The implementation of stop words does not need to consider a document's structure, because this information is discarded during tokenization.

5.2.4 Stemming

In XQFT, the stemming option is a simple on/off switch. XQFT imposes no restrictions on how stemming should be implemented. Stemming is generally independent of document structure, i.e., stemming tokens in an XML document is no different than stemming tokens in any other text document. Consequently, a stemmer in XQFT may be implemented in the same way as stemmers in text-based IR systems.

There are several approaches to implementing stemmers; brute force dictionary lookups, suffix stripping algorithms, lemmatization algorithms etc. A popular approach is to use an algorithm based on the Porter Stemming Algorithm[26], such as the open source Snowball[6] project. This is a suffix stripper that works well for most western languages, and can be customized for specific application needs.

5.2.5 Positional filters

The implementation of positional filters relies on having available the positional information for tokens, such as

1. a token's relative position in the query,
2. a token's relative position in the search context,
3. (1) versus (2), or
4. which sentence or paragraph a token resides in.

From 5.2.1, we recall that a tokenizer is required to store this information per token. With this in mind, implementing positional filters should be trivial.

An ordered selection iterates each token in the query, and verifies that matching tokens in the search context are positioned in the same ascending order.

A window selection verifies that the first and last matching token does not span wider than the given number of words, sentences, or paragraphs.

A distance selection verifies that two matching tokens adhere to the given distance, i.e., that the position of the matching tokens are are greater/-less/equal to the given number of words, sentences, or paragraphs.

A scope selection verifies that matching tokens occur in the same (or different) paragraph or sentence.

[6] Snowball: http://snowball.tartarus.org/

An anchoring selection verifies that matching tokens a) range successively from position 0 and outwards, b) range successively so that the last matching token has the maximum position possible for the search context, or c) succeed from position 0 to the maximum position possible for the search context.

Answering those selections is ultimately a matter of simple arithmetic operations (integer comparisons), performed on the positional information stored in tokens. The XML structure of documents could potentially influence the result of positional filters. However, the structure is discarded by the tokenizer, thus the implementation of positional filters is not substantially different from similar concepts in text-based IR systems.

5.2.6 Relevance ranking

The implementation of IR-style relevance ranking for XML documents has been investigated in several publications [1, 19, 17]. A recurring subject is how to apply scoring for content-and-structure (CAS) queries, and how to best utilize indexes for retrieval efficiency. There seems to be a general agreement that a matching token's importance — and hence relevance — is determined by three factors:

1. where in a document the token is located (e.g., title vs. footnote),
2. how frequently the token occurs in the matching document (tf; term frequency), and
3. how frequently the token occurs in the document corpus as a whole (idf; inverse document frequency).

The first factor is related to the XML structure of documents. If query tokens match the title of a document word by word, it is likely to be more relevant than a document matching a footnote or an annotation. However, an XML document by itself has no knowledge of what the title of a document is. A `title` element in one document may have a different meaning in another document, which effectively leaves out ranking based on element names alone. Applying XML schemas does not help, unless the query processor is tied to a *specific* schema, which again would not be ideal for documents not using that schema. XQFT has solved this issue — handling diversity of documents and schemas — by letting users specify weights (3.2.3) per query. Consider the following query:

```
1 for $b score $s
2     in /books/book[title ftcontains ("usability" weight 2.0) or
3                    content ftcontains ("usability" weight 0.8)
4     order by $s descending
5     return <result>{$b/title}{$b/abstract}</result>
```

The query above will emphasize matches in the `title` element under books, in effect giving it higher relevance than matches in the `content` element. As such, a token's relevance based on location (structure) may be calculated using weights.

Term frequency and inverse document frequency are the other factors influencing relevance. There are two issues related to calculating tf and idf in XQFT. First, there is the question of "what is a document?", i.e., which parts of an XML

tree should logically be considered a document? For the example query above; is `/books` a single document, or is each `/book/book` a separate document? A common approach in implementations is to operate with *collections* of documents, and define methods for inserting XML documents into collections. Each inserted document will logically be a distinct document, and the collection acts as the document corpus. Consequently, tf is calculated by counting occurrences in the unit (XML document) that was inserted into the collection, and idf is calculated by counting each matching document in the collection. Documents and collections potentially contain large amounts of data, which introduces the second issue.

Scanning entire documents (or indeed entire collections) to count tf and idf is time-consuming, and is not a conceivable option unless the document corpus is very small. This calls for indexes. A common indexing strategy for relevance ranking is to augment inverted lists. If an inverted list contains a token's tf for the documents it occurs in, a measure for relevance can be calculated without materializing and scanning the actual documents.

5.3 Oracle Berkeley DB XML

When choosing an XML database to implement XQFT in, the decision was made to use Oracle Berkeley DB XML (BDB XML)[7]. BDB XML was chosen for a number of reasons:

- the source code is available (open source),
- it is well-documented,
- it is and active project which is still maintained, and
- it has an active community

Most of all, BDB XML was chosen because it is used in research projects at NTNU, which could benefit from having the possibility run full-text queries on large collections of XML documents.

The subsequent subsections are based on BDB XML version 2.4.16.

5.3.1 Architecture

BDB XML is an embeddable native XML database. The term "embeddable" refers to the process architecture; rather than having a full-blown client/server stack with networking, BDB XML is a use-at-will library — with bindings for many languages — that is linked (embedded) in applications. This has implications for performance, as there is no inter-process communication or context switching.

BDB XML builds on top of Berkeley DB, which provides common database features; transactions, database replication, locking mechanisms, intermediate caching and indexing, and logging. BDB XML adds features for managing and indexing XML, and an XQuery Engine to run queries. An overview of the architecture is given in Figure 5.2 on the next page.

[7]Oracle Berkeley DB XML: http://www.oracle.com/technology/products/berkeley-db/xml/index.html

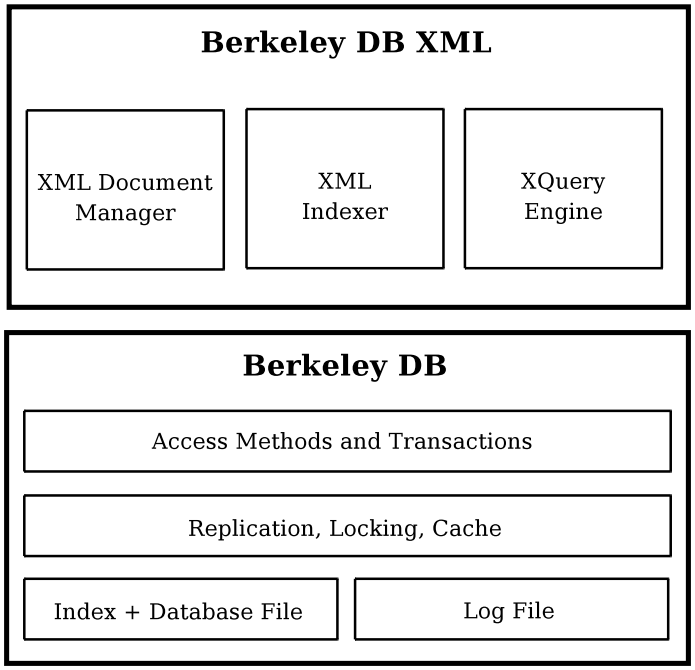

Figure 5.2: Overview of BDB XML architecture

5.3.1.1 XQuery Engine

The XQuery engine in BDB XML is, based on XQilla[8]. XQilla is an open source project, which builds on top of the Xerces-C++[9] XML parser — another open source project, developed by The Apache XML Project[10]. Xerces-C++ features numerous XML-related standards, and provides DOM and SAX programming interfaces. XQilla adds XQuery and XPath support to Xerces-C++, and BDB XML "glues" the projects together with Berkeley DB.

Figure 5.3 on the following page shows a rough overview of query processing in BDB XML, and how the various architectural components interact with each other. Users interact with the XmlManager, which acts as a façade for operations on lower levels. XQilla parses the query, and uses Xerces-C++ to represent information combined from Berkeley DB databases (documents and indexes). Query plan optimization is not shown in the figure, but is an important part of the XQuery engine in BDB XML.

5.3.1.2 Storage

BDB XML uses Berkeley DB for data storage. Documents are stored in containers, which are logical groupings of documents, indexes and statistics, data dictionaries, and other metadata. Each container comprises several Berkeley DB databases (for content, indexes, etc), and represents a collection of documents.

[8] XQilla: http://xqilla.sourceforge.net/
[9] Xerces-C++: http://xerces.apache.org/xerces-c/
[10] The Apache XML Project: http://xml.apache.org/

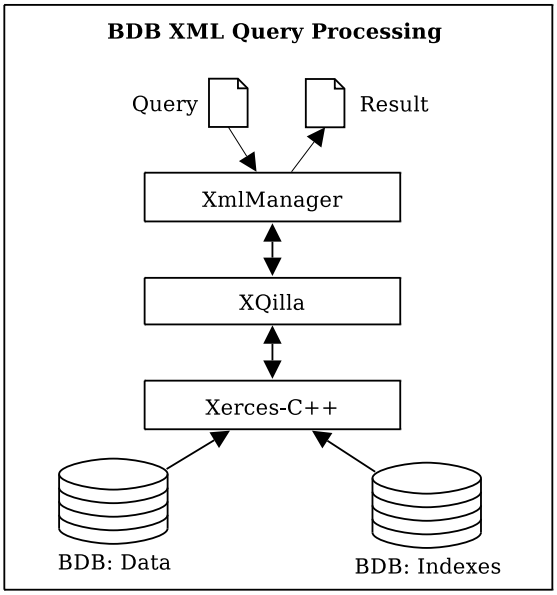

Figure 5.3: Berkeley DB XML Query Processing

A Berkeley DB database is, in essence, a high-performance key/value storage for arbitrary data, where each key may contain several data items. Data is stored in byte arrays, and Berkeley DB does not by itself force any schemas, i.e., applications — such as BDB XML — are free to define schemas for their data. Berkeley DB offers four access methods; B-tree, Hash, Queue, and Recno. BDB XML uses the B-tree access method for its databases, because a B-tree offers decent efficiency for searches, insertions, *and* deletions, and thus supports the varying usage patterns an application may have.

Containers may store XML documents intact, or break documents into nodes that are stored individually. Intact storage is required for round-tripping[24] and makes for efficient materialization and serialization of documents, at the cost of slowing down query processing; raw XML must be parsed during evaluation to answer queries. Node storage makes query processing faster, because whole documents need not be materialized to answer structure or content queries. Answers are generated from direct node lookups, and only the necessary parts of documents are materialized.

5.3.1.3 Indexing

Indexes are not created and utilized automatically, instead BDB XML allows a variety of indexes to be created by users. There are five properties that define an indexing strategy:

Uniqueness indicates whether the indexed value must be unique within the container.

Path type defines the path type to index for values, and is either `node` or `edge`. Node type stores the direct parent node of a value (e.g., `title`),

while edge type stores the "edge" of a value, i.e., the parent node and the parent's parent (e.g., `book/title`).

Node type defines the type of node being indexed; `element`, `attribute`, or `metadata`.

Key type defines the sort of test that the index supports; `presence`, `equality`, or `substring`. The substring type improves performance for queries using the fn:contains() function, while the other improve performance for queries testing for node existence or equality.

Syntax type defines the syntax to use for the indexed value, and maps to any of the XML Schema primitive types (Table 2.1 on page 10), or one of the following; `dayTimeDuration`, `yearMonthDuration`, `untypedAtomic`, or `none`.

Consider the following example document from a collection of books[11]:

```
<book id="424">
  <title>Modern Information Retrieval</title>
  <publisher>Addison Wesley; 1st edition (May 15, 1999)</publisher>
  <isbn n="10">020139829X</isbn>
  <isbn n="13">978-0201398298</isbn>
  <author>Richardo Baeza-Yates</author>
  <author>Berthier Ribeiro-Neto</author>
  <description>
    Discusses the changes in modern information retrieval and
    the provision of relevant information with minimal noise.
    Softcover. DLC: Information storage and retrieval systems.
  </description>
  <toc>
    <chapter>
      <name>Introduction</name>
      <section>
        <name>Motivation</name>
      </section>
      <section>
        <name>Basic Concepts</name>
      </section>
    </chapter>
    <chapter>
      <name>Modeling</name>
      <section>
        <name>Introduction</name>
      </section>
      <section>
        <name>A Taxonomy of Information Retrieval Models</name>
      </section>
    </chapter>
  </toc>
</book>
```

Index strategies are specified as strings of the form "[unique]-{path type}-{node type}-{key type}-{syntax type}". Index strategies are associated with a URI and a name for the attribute element to be indexed. The following index strategies are candidates for the collection of books as in the example:

[11] Example data copied from amazon.com: http://www.amazon.com/Modern-Information-Retrieval-Ricardo-Baeza-Yates/dp/020139829X/

- "unique-edge-attribute-equality-double" for the name "id", to allow fast lookups based on the `id` attribute for `book` elements,
- "edge-element-equality-string" for the names "title", "isbn", and "author", to a allow fast lookups based on string equality,
- "edge-element-substring-string" for the name "description", to allow fast substring matching book descriptions.

The query optimizer will inspect the query and detect situations where an index may apply., and use the index to retrieve a subset of possible documents.

5.3.2 Implementing XQFT in BDB XML

On a high level, enabling XQFT in BDB XML involves three tasks; 1) implementing the XQFT standard in XQilla, 2) adding support for full-text indexing, and 3) extending the query optimizer to support XQFT expressions. The query processing model (Figure 5.3 on page 46) remains the same after enabling XQFT; documents are stored in the same Berkeley DB database, XQilla handles full-text queries, and indexes are used to speed up the query process.

At the time of this writing, work has begun on implementing XQFT in XQilla and BDB XML. The implementation of XQuery and XQFT in XQilla follows the same path as other open-source implementations we have examined:

- A lexer-parser combination is generated from the EBNF[12] grammars defined in XQuery[30] and XQFT[18]. There is generally a one-to-one mapping of EBNF productions and classes, i.e., each formal rule in the grammar results in the instantiation of a class representing the rule. These classes contain the program logic required for implementing the functions/operators/expressions/productions they represent.
- In addition to the grammar classes, there are classes for representing the non-syntactical notions of XQuery and XQFT; static and dynamic context, query results, supporting classes for XDM, etc.

5.3.2.1 Query lexer-parser

The lexer-parser mentioned above parses input (queries) to programmatic units. Table 5.1 on the next page shows grammar productions of XQFT that are not yet implemented in the lexer-parser. These productions will be ignored if used, and the parser will print an error string identifying the production. This will cause invalid results.

Scoring and weights (i.e., relevance ranking) are not listed in the table. The productions for score variables and weights are parsed, but they are not yet implemented in the classes they are used.

Table 5.2 on the facing page shows productions that are incorrectly defined in XQilla. We have modified the lexer-parser to follow the grammar defined in the XQFT standard. Patches for the lex specification file and parser generator file are provided in Appendix B.1 on page 81.

[12] Extended Backus–Naur Form

Production	Description
`FTTimes`	Cardinality selection.
`FTLanguageOption`	Language match option.
`FTWildCardOption`	Wildcards match option.
`FTThesaurusOption`	Thesaurus match option.
`FTStemOption`	Stemming match option.
`FTCaseOption`	Case sensitivity match option.
`FTDiacriticsOption`	Diacritics sensitivity match option.
`FTStopwordOption`	Stop words match option.
`FTIgnoreOption`	Ignore option.

Table 5.1: XQFT grammar productions not parsed in XQilla

Production	Is	Should be
FTOr	`FTAnd ( "||" FTAnd )*`	`FTAnd ( "ftor" FTand )*`
FTAnd	`FTMildnot ( "&&" FTMildnot )*`	`FTMildnot ( "ftand" FTMildnot )*`
FTUnaryNot	`("!")? FTWordsSelection`	`("ftnot")? FTWordsSelection`

Table 5.2: Incorrect XQFT grammar productions in XQilla

5.3.2.2 Tokenization

The query and search context tokenizer (shown in Figure 3.1 on page 19) is based on the lexer-parser above, and parses XDM instances to linguistic tokens. The default implementation follows the same rules as we proposed in 5.2.1 on page 37; tokens are split by whitespace, sentences are split by punctuation, and paragraphs are split by newline characters.

The code for the default tokenizer class is shown in Appendix B.3 on page 83. On line 35, we see that the `tokenize()` method — if given a node — will generate and use a string value of the node for further tokenization. This is in accordance with the XQFT semantics, and will dissolve element boundaries and discard attribute nodes.

5.3.2.3 Evaluation of full-text selections

The evaluation of full-text selections in XQilla to a large degree follows the formal semantics defined in XQFT. Figure 5.4 on the next page shows the class structure of `AllMatches`, `Match`, `StringMatch`, and `TokenInfo`, known from Section 3.2.1 on page 18.

The classes that represent full-text selections and operators all contain an `execute(FTContext *ftcontext)` method that will return a pointer to an `AllMatches` instance. The `FTContext` class wraps the dynamic context, a tokenizer, a token store, and a variable holding the current query position. The selections and operators evaluate themselves, instantiating `Match` instances and adding `stringIncludes` or `stringExcludes` as fit.

When the selections and operators are evaluated, the `FTContains` class will atomize the top-most `AllMatches` instances to a boolean value, as described in Section 3.2.1 on page 18.

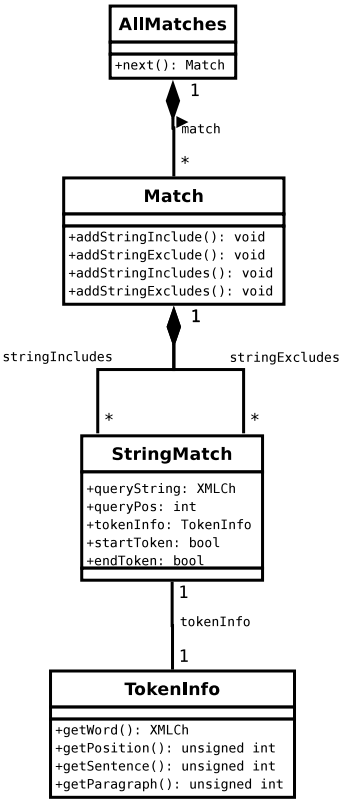

Figure 5.4: Classes used in the evaluation of full-text selections

5.3.2.4 Enabling experimental XQFT support in BDB XML

XQilla uses a set of class constants as flags to determine which language (XPath, XQuery, XQuery + XQuery Updates, XQuery + Full-Text, etc.) the lexer-parser should use. To enable experimental XQFT support in BDB XML, we need to use one of the full-text flags in the internal `QueryExpression` class of BDB XML. A patch for this is given in Appendix B.2 on page 83.

5.3.3 Full-text indexing in BDB XML

In Chapter 4, we examined full-text indexing in a few state-of-the-art XML databases. We recall that inverted lists were used to allow lookups on tokens, retrieving a subset of matching documents. Further, if tuples in the inverted list are augmented with positional information and term frequency, more queries can be answered with index lookups. We now present an inverted list index for BDB XML based on Berkeley DB.

It turns out that Berkeley DB databases are well suited for implementing inverted lists. Berkeley DB is a field-proven technology, and provides features for data access and management which would otherwise have to be implemented at some level. Its flexible no-enforced-scheme policy lends well to our purposes; storing keywords and tuples. Tokens are used as keys, and the data tuples consist

of references to documents/nodes, the token's term frequency, and position lists.

5.3.3.1 Inverted list structure

The structure of the inverted list is shown in Figure 5.5. Tokens are used as keys, and for each document the token occurs in, there is a data tuple containing;

1. the DocID[13] for the document the token occurs in,
2. the path, as specified by the indexing strategy's "name",
3. the token's term frequency, and
4. the token's positions.

(3) and (4) are relative to the stored <path> in the referenced <DocID>.

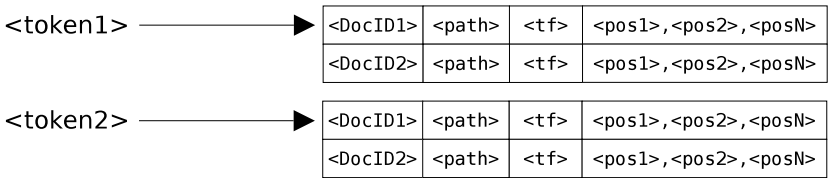

Figure 5.5: Inverted list structure in a Berkeley DB database

Multiple full-text indexes will share the same Berkeley DB database, i.e., all full-text index strategies are stored in the same database, even if they have different paths. For each element that is included by an index strategy's "name", the indexer stores a distinct tuple. Consequently, a document with many occurrences of matched elements will have many tuples in the inverted list. Depending on the document's structure, this could result in a large number of tuples per token. However, several queries may be answered with information found in the inverted list, thus avoiding materialization of documents not included in the final result. Here are some of the advantages by this approach:

- A query specifying several tokens will first consult the index to see if the inverted list contains each token. E.g., if a query uses the `ftand` logical operator and one of the tokens are not defined as keys in the index, the query is determined without even retrieving data from the index. For each of the tokens found in the index, the data tuples are retrieved and merged by DocID. This avoids materialization of documents not containing all terms specified in the query.
- A token's relevance may be calculated on document-level or element-level. If the FLWOR+scoring expression is iterating documents, it would be desirable to calculate relevance per document, i.e., sum up term frequencies grouped by DocID. On the other hand, if a FLWOR-+scoring expression iterates elements matching the indexed path, the sum of term frequencies could be grouped by the path, resulting in a finer-than-document granularity relevance.

[13] DocID is a 64-bit integer assigned for documents upon insertion to a container.

- Positional filters may be answered without any materialization, if the queried path matches indexed path.

5.3.3.2 Extending the index specification with full-text options

We recall that index strategies in BDB XML are defined by strings of the form "[unique]-{path type}-{node type}-{key type}-{syntax type}", and are identified by a URI and a name for the elements or attributes to be indexed. To add full-text indexes, we propose that "fulltext" is added as an option to {key type}. A key type of "fulltext" implies some restrictions on the other options:

- Unique must be omitted, because it does not apply to full-text indexing.
- Node type must be "element", because attributes and metadata are not considered by XQFT tokenization.
- Syntax type must be omitted (or ignored), because it does not apply to inverted lists; the indexed element's values (tokens) are used as *keys* in the Berkeley DB database.

5.3.3.3 Extending the XML Indexer

The XML Indexer in BDB XML must be extended to include support for our proposed full-text index. The full-text indexer will find elements to index in the same fashion as the existing indexers. For each found element, the steps to index are as follows:

1. Get a string value of the element.
2. Tokenize the string using the default tokenizer.
3. For each token:
   (a) If the index does not contain the token as a key; insert an initial data tuple.
   (b) If the token exists as a key in the index;
      i. retrieve or create the data tuple matching the current DocID and path
      ii. update term frequency and add the current position

5.4 Summary

We have seen that the implementation of IR concepts in XQFT is not fundamentally different from text-based IR systems. This is largely due to the fact that both XQFT and text-based IR work with *tokens*, as opposed to full document representations. The tokenizer in XQFT discards the structural information of XML (attributes, element boundaries) when parsing nodes, because tokens must be derived from the string value `(fn:string())` of items in the search context.

We have investigated XQFT in BDB XML/XQilla; work has begun, but the implementation is not yet complete. The following parts are not implemented in XQilla:

- match options,
- cardinality selection ("occurs"), and
- relevance ranking, i.e., scoring and weights.

In addition, the logical operators `"ftand"`, `"ftor"`, and `"ftnot"` were not defined correctly in the lexer-parser. This was fixed, and patches are provided in Appendix B.

The tokenizer in XQilla has been scrutinized, and we have taken a deeper look into the evaluation of full-text selections. The tokenizer functions as expected, and the evaluation of full-text selections is in accordance with the XQFT standard.

BDB XML has not yet incorporated full-text support in its query optimizer, and no plans for full-text indexing have been published. We present an inverted list index based on Berkeley DB. The index allows keyword lookups, and contains a (DocID, path, term frequency, position list) data tuple for each document and distinct path the token occurs in. This allows answering several queries without materializing data. A potential disadvantage of this approach is that keywords may contain many data tuples, resulting in retrieving more data from the index than required for answering the query. Another disadvantage is that updating the index is expensive.

Chapter 6

Results

This chapter describes the results of some experiments which have been conducted in order to evaluate the implementations discussed in Section 5.3 on page 44.

6.1 Overview of experiments

All experiments are based on the example data in Appendix B.4 on page 85. Each book element in the document is added as a document in a container. This process is shown in Appendix B.5 on page 89.

The following sections each show a query script that is executed using BDB XML's shell utility (`dbxml -s <script>`). BDB XML is compiled in Ubuntu 9.04 with the patches in Appendix B.

6.2 Searching for a single token

This query searches for a single token — "information" — in the `title` element of `book` documents. Books 1, 3, and 4 should be found.

6.2.1 Query script

```
1  openContainer books.dbxml
2  query '
3  for $book in collection("books.dbxml")//book
4  let $title := $book/title
5  where $title ftcontains "information"
6  return
7  <book>
8      {$book/@id}
9      {$title/text()}
10 </book>
11 '
12 print
13 exit
```

6.2.2 Result

Script output:

```
1 <book id="1">Modern Information Retrieval</book>
2 <book id="3">Introduction to Information Retrieval</book>
3 <book id="4">Information Retrieval: Algorithms and Heuristics (The
      Information Retrieval Series)(2nd Edition)</book>
```

The results are as expected, and show that XQFT is enabled in BDB XML.

6.3 Searching for a phrase and a token

This query searches for a phrase ("information retrieval") and a token ("web") in the `description` element of `book` documents. Books 1 and 3 should be found.

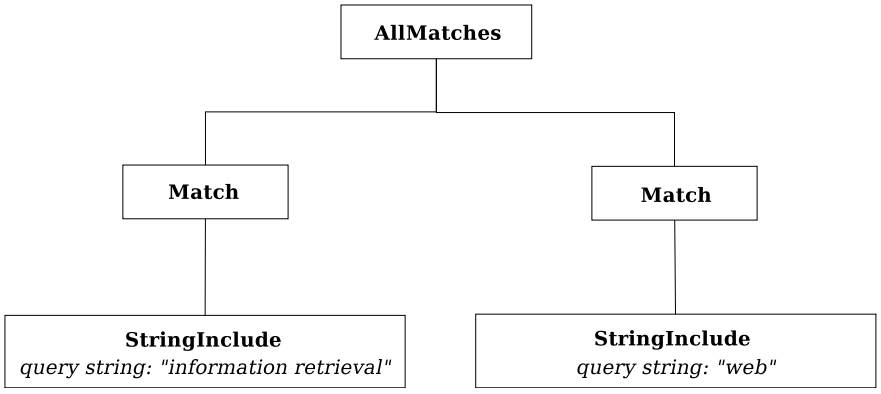

Figure 6.1: AllMatches model for "information retrieval" and "web"

6.3.1 Query script

```
1  openContainer books.dbxml
2  query '
3  for $book in collection("books.dbxml")//book
4  let $title := $book/title
5  let $desc := $book/description
6  where $desc ftcontains {"information retrieval", "web"} all
7  return
8  <book>
9      {$book/@id}
10     {$title/text()}
11 </book>
12 '
13 print
14 exit
```

6.3.2 Result

Script output:

```
1 <book id="1">Modern Information Retrieval</book>
2 <book id="3">Introduction to Information Retrieval</book>
```

The results are expected, and shows that the implementation builds an AllMatches model, as shown in Figure 6.1. The figure is based on similar models shown in XQFT[18].

6.4 Searching with a cardinality selection

This query searches for books where the phrase "information retrieval" occurs at least two times in the `title`. Book 4 should be found. However, the expected result is is to find books 1, 3, and 4, because cardinality selection is not yet implemented.

6.4.1 Query script

```
1  openContainer books.dbxml
2  query '
3  for $book in collection("books.dbxml")//book
4  let $title := $book/title
5  where $title ftcontains "information retrieval"
6                  occurs at least 2 times
7  return
8  <book>
9      {$book/@id}
10     {$title/text()}
11 </book>
12 '
13 print
14 exit
```

6.4.2 Result

Script output:

```
1 occurs
2 <book id="1">Modern Information Retrieval</book>
3 <book id="3">Introduction to Information Retrieval</book>
4 <book id="4">Information Retrieval: Algorithms and Heuristics (The
      Information Retrieval Series)(2nd Edition)</book>
```

The results are as expected; books containing the phrase "information retrieval" are found, and "occurs" is printed by the parser. This shows that cardinality selection is is not implemented.

6.5 Searching with the case sensitivity match option

This query searches for books where the phrase "information retrieval" all lowercase in the `title` element. Nothing should be found. However, as in the previous section, the expected result is is to find books 1, 3, and 4, because match options are not yet implemented.

6.5.1 Query script

```
1  openContainer books.dbxml
2  query '
3  for $book in collection("books.dbxml")//book
4  let $title := $book/title
5  where $title ftcontains "information retrieval" lowercase
6  return
7  <book>
8      {$book/@id}
```

```
9       {$title/text()}
10  </book>
11  '
12  print
13  exit
```

6.5.2 Result

Script output:

```
1  lowercase
2  <book id="1">Modern Information Retrieval</book>
3  <book id="3">Introduction to Information Retrieval</book>
4  <book id="4">Information Retrieval: Algorithms and Heuristics (The
       Information Retrieval Series)(2nd Edition)</book>
```

The results are as expected, and show that the case sensitivity match option is not implemented

6.6 Searching two tokens with `ftand`

This query searches for books where both "algorithms" and "retrieval" occur in the `description` element. Books 2 and 4 should be found.

6.6.1 Query script

```
1  openContainer books.dbxml
2  query '
3  for $book in collection("books.dbxml")//book
4  let $title := $book/title
5  let $desc := $book/description
6  where $desc ftcontains "algorithms" ftand "retrieval"
7  return
8  <book>
9       {$book/@id}
10      {$title/text()}
11  </book>
12  '
13  print
14  exit
```

6.6.2 Result

Script output:

```
1  <book id="2">Managing Gigabytes: Compressing and Indexing Documents
       and Images</book>
2  <book id="4">Information Retrieval: Algorithms and Heuristics (The
       Information Retrieval Series)(2nd Edition)</book>
```

The results are as expected, and show that our parser modifications to the `FTAnd` production work successfully.

6.7 Searching with the `not in` operator

This query searches for books where the `description` element contains the token "search", but not if the matched token is followed by "engines". Books 3 and 4 should be found. Book 1 does not contain "search" as a word, and book 2 uses should be omitted because it contains "search engines".

6.7.1 Query script

```
openContainer books.dbxml
query '
for $book in collection("books.dbxml")//book
let $title := $book/title
let $desc := $book/description
where $desc ftcontains "search" not in "search engines"
return
<book>
    {$book/@id}
    {$title/text()}
</book>
'
print
exit
```

6.7.2 Result

Script output:

```
<book id="3">Introduction to Information Retrieval</book>
<book id="4">Information Retrieval: Algorithms and Heuristics (The
    Information Retrieval Series)(2nd Edition)</book>
```

The results are as expected, and show that mild-not selections are implemented. It also shows that *tokens* are searched, as opposed to substring matching (which would also return book 1 because it contains the substring in "researchers").

6.8 Searching with the `window` positional filter

This query searches for books where the `description` element contains the tokens "retrieval" and "web" within a window of 10 words. Books 2 and 3 should be found. Book 1 matches both tokens in the same sentence, but the distance is greater than 10 words.

6.8.1 Query script

```
openContainer books.dbxml
query '
for $book in collection("books.dbxml")//book
let $title := $book/title
let $desc := $book/description
where $desc ftcontains "retrieval" ftand "web" window 10 words
return
<book>
    {$book/@id}
    {$title/text()}
</book>
'
print
exit
```

6.8.2 Result

Script output:

```
<book id="2">Managing Gigabytes: Compressing and Indexing Documents
     and Images</book>
<book id="3">Introduction to Information Retrieval</book>
```

The results are as expected, and show that the implementation stores positional information for matched tokens.

6.9 Searching with `order` and `scope` positional filters

This query searches for books where the `description` element contains the tokens "retrieval" and "information" in the same paragraph, and in the same order as in the query. Books 1 and 3 should be found. All books contain both tokens, but book 2 should be omitted because the tokens do not appear in the same paragraph. Book 4 should be omitted because the order is not the same as in the query.

6.9.1 Query script

```
1  openContainer books.dbxml
2  query '
3  for $book in collection("books.dbxml")//book
4  let $title := $book/title
5  let $desc := $book/description
6  where $desc ftcontains ("retrieval" ftand "information")
7                         ordered same paragraph
8  return
9  <book>
10     {$book/@id}
11     {$title/text()}
12 </book>
13 '
14 print
15 exit
```

6.9.2 Result

Script output:

```
1 <book id="1">Modern Information Retrieval</book>
2 <book id="3">Introduction to Information Retrieval</book>
```

The results are as expected, and show that a) the implementation stores positional information for query tokens, and b) the implementation understands the notion of *big units* (sentence, paragraph). In conclusion, the tokenizer in XQilla works expected, and matches the one we proposed in Section 5.2.1 on page 37.

6.10 Searching with the `distance` positional filter

This query searches for books where the `description` element contains the tokens "retrieval" and "information" with a distance of exactly 12 words. Only book 2 should be found.

6.10.1 Query script

```
openContainer books.dbxml
query '
for $book in collection("books.dbxml")//book
let $title := $book/title
let $desc := $book/description
where $desc ftcontains ("retrieval" ftand "information")
                       distance exactly 12 words
return
<book>
    {$book/@id}
    {$title/text()}
</book>
'
print
exit
```

6.10.2 Result

Script output:

```
<book id="2">Managing Gigabytes: Compressing and Indexing Documents
    and Images</book>
```

The results are as expected, and show that distance selection is implemented.

6.11 Summary

We have tested 10 full-text queries with experimental XQFT support in BDB XML. The results were as expected:

- The implemented parts we have described were functional.
- The non-implemented parts were ignored, and error messages were printed to `stderr`.
- Our modifications to the parser were successful.

A noteworthy result we did not initially expect, was that distance selection is functional, while cardinality selection is not. Cardinality selection was expected not to work, but we had a suspicion the reason was a lacking implementation for the `FTRange` production (“`at least`”, “`at most`”, etc.), which would spread to distance selections as well.

Part III

Thesis Conclusion

Chapter 7

Evaluation and discussion

This chapter gives a summary of the thesis. Contributions are briefly discussed, and the results of the thesis are highlighted and evaluated with respect to the problem definition.

7.1 Summary of thesis

Searching XML — let alone full-text search — is a relatively new field research, which has been of interest the last decade. Existing query language standards (XQuery and XPath) are powerful for querying XML structure, but are too limited for full-text search. Several approaches to XML full-text search have been researched by various organizations, and unified in the XQuery and XPath Full-Text 1.0 (XQFT) standard.

We have investigated the possibilities of XQFT, and discussed various design and implementation issues in further detail. Our findings indicate that a) XQFT solves many traditional IR-related problems, and b) traditional IR challenges apply to XML as well. The most pertinent challenge is how to utilize full-text indexes for improving query efficiency.

In this thesis, we have contributed the following:

- A description of the design and implementation of XQFT; first in general terms, then in the context of Oracle Berkeley DB XML.
- Patches for fixing and enabling parts of XQFT in Oracle Berkeley DB XML.
- A proposal for a full-text index in Oracle Berkeley DB XML.

7.2 Discussion of contributions

The following subsections discuss two of our contributions, comparing them with what was originally planned.

7.2.1 XQFT implementation in BDB XML/XQilla

The original plan was to implement XQFT fully in BDB XML. However, a considerable amount of time was spent learning the architecture and getting to

know the API[1]. During this phase, work began on an administration interface for BDB XML, which would be used to generate and evaluate results. This interface has been left out of the thesis, because it does not contribute to the problem definition.

Had time been better spent, more of XQFT could be implemented in XQilla.

7.2.2 Proposed full-text index for BDB XML

The proposed full-text index is not implemented, for the same reasons as the XQFT implementation. The original plan was to implement and experiment with various index configurations on a large collection of documents, and present empirical data for further discussion of our claims.

7.3 Evaluation

The paramount objective of this thesis was to investigate how full-text search and indexing apply to XML databases (specifically Oracle Berkeley DB XML). This goal has been achieved, and is described in Chapters 2, 3, 4, and 5. Unfortunately, there was not enough time to fully implement XQFT in BDB XML. The implementation has very limited functionality, which is reflected by the small number of results in Chapter 6.

Several sub problems were derived from the main problem definition. First, there was the question of which index types are suitable for supporting full-text search. This has been answered by analyzing state-of-the-art implementations (Chapter 4), and proposing a full-text index for BDB XML (Section 5.3.3).

Second, we raised concerns about text-based IR versus IR in XML, and asked the question of whether the structured nature of XML requires a new approach to IR. This question has been answered by examining the design and implementation of various IR concepts in XQFT, and is described in further detail in Section 5.2.

Third, we wondered whether there were open issues in XQFT that would impede cross-implementation compatibility. We have identified two such cases; thesauri and stop words. Those are subject to implementation-dependent differences, because XQFT does not define schemas for how they should be represented. For thesauri, we have proposed a standards-compliant representation based on SKOS (Section 5.2.2), and for stop words, we have proposed a simple schema that adheres to the predefined semantics of XQFT (Section 5.2.3).

[1] Application Programming Interface

Chapter 8

Conclusions and further work

This chapter concludes the thesis. Specific findings are summarized, and an outline for future work is given.

8.1 Concluding remarks

The following subsections summarize findings and results from the thesis.

8.1.1 XQuery is too limited for full-text search

XQuery offers only rudimentary support for full-text search, limited to substring matching with the `fn:contains()` function. This is not sufficient to answer many of the use cases defined in XML Query Use Cases[10].

8.1.2 XQFT adds IR concepts to XQuery

XQuery and XPath Full Text 1.0[18] (XQFT) is a W3C standard which adds full-text capabilities to XQuery. The standard defines new expressions for performing full-text searches on XDM instances; token or phrase matching, boolean connectivities, proximity/scope restrictions. Other IR concepts — thesauri, stop words, stemming, relevance ranking (scoring and weights) — are also defined by the standard, without imposing specific restrictions on implementation details.

8.1.3 IR in XML/XQFT is not fundamentally different from text-based IR

As with text-based Information Retrieval, operations in XQFT rely on tokenized text. Tokenization in XQFT discards structural information (attributes and element boundaries) from XML, and items in the search context are represented as collections of tokens. The same concepts from text-based IR — as mentioned in the previous subsection — apply to XQFT. The difference between text-based IR and XML IR is the underlying representation of documents and tokens.

8.1.4 Full-text indexing is critical for query efficiency

Full-text search is a heavy process with respect to computational effort and I/O traffic. Unless a database's document collection is very small, indexing is required for query efficiency. Without indexing, large amounts of data needs to be materialized, tokenized, and searched, potentially several times per query. Path indexes help for limiting the amount of documents that need to be materialized for the structural part of queries, but are not directly useful for full-text queries. Full-text indexing is a way of indexing text that better resembles the tokenized representation of text, and allows lookups based on tokens.

A common approach to full-text indexing is to use augmented forms of inverted lists. In this approach, keywords (tokens) are linked with tuples that contain references to documents in which the keyword occur, in addition to positional information and term frequency (or other relevance measures). By combining path indexes and full-text indexes, document materialization is kept to a minimum, and queries may be answered without accessing the real data.

8.1.5 XQFT can and will be implemented in BDB XML

Work has begun on implementing the XQFT standard in BDB XML. BDB XML uses Xerces-C++ and XQilla internally to parse XML and run queries. XQilla currently has experimental support for XQFT, which is disabled in BDB XML. When the XQilla implementation is complete, BDB XML will inherently support XQFT. Furthermore, BDB XML must take measures to optimize full-text queries for its internal representation of physical data.

8.1.6 Full-text indexing in BDB XML may be implemented using Berkeley DB

Oracle has not yet revealed their plans for full-text indexing in BDB XML. A promising approach is to implement augmented inverted lists using Berkeley DB databases; tokens are used as keys, with data tuples for each distinct (document, path) pair in which the token occurs.

8.2 Future work

The following subsections outline candidates for future work related to this thesis.

8.2.1 Improving the XQFT implementation in BDB XML

The current XQFT implementation in BDB XML is experimental, and not feature-complete. Future work involves:

- implementing the remaining parts of the XQFT standard in XQilla,
- improve support for XQFT expressions in BDB XML's query optimizer,
- implement the proposed full-text index for improving full-text query efficiency.

8.2.2 Dealing with frequent updates

The primary focus in this thesis has been on the design and implementation of well-known IR techniques in XML databases. As such, the discussed topics mainly apply to IR-style applications, where data is "stored once, read many times", i.e., there are few or no updates to data already existing in the database. A new set of issues emerges when an XML database is used in database-style applications, with frequent updates to existing data.

The main issue with frequent updates is to keep indexes consistent with the data they represent. Continuously updating and re-indexing requires considerable amounts of computational time and I/O, and there is a desire for partial re-indexing; only re-indexing the updated data. This needs to be investigated more with respect to full-text indexes.

Appendix A

XQuery Full-Text Semantics

A.1 Semantics for fts:lookupThesaurus

The following XQuery code (from XQFT[18]) defines the semantics for applying an `FTThesaurusOption`:

```
declare function fts:applyThesaurusOption (
        $matchOption as element(fts:thesaurus),
        $queryTokens as element(fts:queryToken)* )
    as element(fts:queryItem)*
{
    if ($matchOption/@thesaurusIndicator = "with") then
        fts:lookupThesaurus( $queryTokens,
                             $matchOption/fts:thesaurusName,
                             $matchOption/@language,
                             $matchOption/fts:relationship,
                             $matchOption/fts:range )
    else if ($matchOption/@thesaurusIndicator = "without") then
        <fts:queryItem>
        {$queryTokens}
        </fts:queryItem>
    else ()
};
```

The following XQuery code shows an example implementation of the semantics for the `fts:lookupThesaurus` function used in the `fts:applyThesaurusOption` function above. The function expects a SKOS thesaurus as in Appendix A.3.

```
declare namespace skos = "http://www.w3.org/2008/05/skos#"
declare namespace rdf =
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
declare namespace fts = "fts"

(: finds alternative terms for query tokens :)
declare function fts:lookupThesaurus (
    $tokens as element(fts:queryToken)*,
    $thesaurusName as xs:string?,
    $thesaurusLanguage as xs:string?,
    $relationship as xs:string?,
    $range as element(fts:range)? )
  as element(fts:queryItem)*
{
  let $thesaurus := doc($thesaurusName)//skos:Concept
  let $maxPos := max($tokens/@queryPos)
```

```
16
17    let $additionalTokens :=
18      for $token in $tokens
19      let $word := string($token/@word)
20      return
21        if ($relationship eq "use") then
22          fts:lookupThesaurusUse($word, $thesaurus,
                $thesaurusLanguage)
23        else if ($relationship eq "uf") then
24          fts:lookupThesaurusUF($word, $thesaurus,
                $thesaurusLanguage)
25        else if ($relationship eq "bt") then
26          fts:lookupThesaurusBT(0, $word, $thesaurus,
                $thesaurusLanguage, $range)
27        else if ($relationship eq "nt") then
28          fts:lookupThesaurusNT(0, $word, $thesaurus,
                $thesaurusLanguage, $range)
29        else if ($relationship eq "btg") then
30          fts:lookupThesaurusBT(0, $word, $thesaurus,
                $thesaurusLanguage, $range)
31        else if ($relationship eq "ntg") then
32          fts:lookupThesaurusNT(0, $word, $thesaurus,
                $thesaurusLanguage, $range)
33        else if ($relationship eq "btp") then
34          fts:lookupThesaurusBT(0, $word, $thesaurus,
                $thesaurusLanguage, $range)
35        else if ($relationship eq "ntp") then
36          fts:lookupThesaurusNT(0, $word, $thesaurus,
                $thesaurusLanguage, $range)
37        else if ($relationship eq "tt") then
38          fts:lookupThesaurusTT($word, $thesaurus,
                $thesaurusLanguage)
39        else if ($relationship eq "rt") then
40          fts:lookupThesaurusRT($word, $thesaurus,
                $thesaurusLanguage)
41        else
42          ()
43
44    return <fts:queryItem>
45             {$tokens}
46             {
47             for $token at $pos in distinct-values($additionalTokens)
48             return <fts:queryToken word="{$token}"
                   queryPos="{$maxPos + $pos}" />
49             }
50           </fts:queryItem>
51  };
52
53  (: finds preferred terms for a token :)
54  declare function fts:lookupThesaurusUse (
55      $token as xs:string,
56      $concepts as element(skos:Concept)*,
57      $lang as xs:string? )
58    as xs:string*
59  {
60    if ($lang) then
61      $concepts[skos:altLabel[@xml:lang = $lang] = $token]
62                /skos:prefLabel[@xml:lang = $lang]
63    else
64      $concepts[skos:altLabel = $token]
65                /skos:prefLabel
66  };
```

```
67
68  (: finds non-preferred terms for a token :)
69  declare function fts:lookupThesaurusUF (
70      $token as xs:string,
71      $concepts as element(skos:Concept)*,
72      $lang as xs:string? )
73    as xs:string*
74  {
75    if ($lang) then
76      $concepts[skos:prefLabel[@xml:lang = $lang] = $token]
77               /skos:altLabel[@xml:lang = $lang]
78    else
79      $concepts[skos:prefLabel = $token]
80               /skos:altLabel
81  };
82
83  (: finds narrower terms for a token :)
84  declare function fts:lookupThesaurusNT (
85      $level as xs:integer,
86      $token as xs:string,
87      $concepts as element(skos:Concept)*,
88      $lang as xs:string?,
89      $range as element(fts:range)? )
90  {
91    let $terms :=
92    if ($lang) then
93      for $c in $concepts[skos:prefLabel[@xml:lang = $lang] eq $token
           or
94                          skos:altLabel[@xml:lang = $lang] eq $token]
95      return $concepts[@rdf:about = $c/skos:narrower/@rdf:resource]
96                      /skos:prefLabel[@xml:lang = $lang]
97    else
98      for $c in $concepts[skos:prefLabel eq $token or
99                          skos:altLabel eq $token]
100     return $concepts[@rdf:about = $c/skos:narrower/@rdf:resource]
101                     /skos:prefLabel
102
103   let $next := $level + 1
104
105   return
106   if ($range) then
107     if ($range/@type = "exactly") then
108       if ($level = $range/@n) then
109         $terms
110       else if (count($terms) > 0) then
111         for $t in $terms
112         return fts:lookupThesaurusNT($next, $t, $concepts, $lang,
              $range)
113       else ()
114
115     else if ($range/@type = "at least") then
116       if ($level < $range/@n and count($terms) > 0) then
117         for $t in $terms return
118         fts:lookupThesaurusBT($next, $t, $concepts, $lang, $range)
119       else if ($level >= $range/@n and count($terms) > 0) then
120         (
121           $terms,
122           for $t in $terms
123           return fts:lookupThesaurusNT($next, $t, $concepts, $lang,
                $range)
124         )
125       else ()
```

```
126
127       else if ($range/@type = "at most") then
128         if ($next > $range/@n) then
129           $terms
130         else if (count($terms) > 0) then
131           (
132             $terms,
133             for $t in $terms
134             return fts:lookupThesaurusNT($next, $t, $concepts, $lang,
                      $range)
135           )
136         else ()
137
138       else if ($range/@type = "from to") then
139         if ($level < $range/@m) then
140           if (count($terms) > 0) then
141             for $t in $terms
142             return fts:lookupThesaurusNT($next, $t, $concepts, $lang,
                      $range)
143           else ()
144         else if ($next > $range/@n) then
145           $terms
146         else
147           if (count($terms) > 0) then
148             (
149               $terms,
150               for $t in $terms
151               return fts:lookupThesaurusNT($next, $t, $concepts,
                        $lang, $range)
152             )
153           else $terms
154       else $terms
155     else
156       if (count($terms) > 0) then
157         (
158           $terms,
159           for $t in $terms
160           return fts:lookupThesaurusNT($next, $t, $concepts, $lang,
                      $range)
161         )
162       else $terms
163   };
164
165   (: finds broader terms for a token :)
166   declare function fts:lookupThesaurusBT (
167       $level as xs:integer,
168       $token as xs:string,
169       $concepts as element(skos:Concept)*,
170       $lang as xs:string?,
171       $range as element(fts:range)? )
172     as xs:string*
173   {
174     let $terms :=
175     if ($lang) then
176       for $c in $concepts[skos:prefLabel[@xml:lang = $lang] eq $token
              or
177                           skos:altLabel[@xml:lang = $lang] eq $token]
178       return $concepts[@rdf:about = $c/skos:broader/@rdf:resource]
179                       /skos:prefLabel[@xml:lang = $lang]
180     else
181       for $c in $concepts[skos:prefLabel eq $token or
182                           skos:altLabel eq $token]
```

```
183       return $concepts[@rdf:about = $c/skos:broader/@rdf:resource]
184                       /skos:prefLabel
185
186     let $next := $level + 1
187
188     return
189     if ($range) then
190       if ($range/@type = "exactly") then
191         if ($level = $range/@n) then
192           $terms
193         else if (count($terms) > 0) then
194           for $t in $terms
195           return fts:lookupThesaurusBT($next, $t, $concepts, $lang,
                  $range)
196         else ()
197
198       else if ($range/@type = "at least") then
199         if ($level < $range/@n and count($terms) > 0) then
200           for $t in $terms return
201           fts:lookupThesaurusBT($next, $t, $concepts, $lang, $range)
202         else if ($level >= $range/@n and count($terms) > 0) then
203           (
204             $terms,
205             for $t in $terms
206             return fts:lookupThesaurusBT($next, $t, $concepts, $lang,
                      $range)
207           )
208         else ()
209
210       else if ($range/@type = "at most") then
211         if ($next > $range/@n) then
212           $terms
213         else if (count($terms) > 0) then
214           (
215             $terms,
216             for $t in $terms
217             return fts:lookupThesaurusBT($next, $t, $concepts, $lang,
                      $range)
218           )
219         else ()
220
221       else if ($range/@type = "from to") then
222         if ($level < $range/@m) then
223           if (count($terms) > 0) then
224             for $t in $terms
225             return fts:lookupThesaurusBT($next, $t, $concepts, $lang,
                      $range)
226           else ()
227         else if ($next > $range/@n) then
228           $terms
229         else
230           if (count($terms) > 0) then
231             (
232               $terms,
233               for $t in $terms
234               return fts:lookupThesaurusBT($next, $t, $concepts,
                        $lang, $range)
235             )
236           else $terms
237       else $terms
238     else
239       if (count($terms) > 0) then
```

```
240         (
241           $terms,
242           for $t in $terms
243           return fts:lookupThesaurusBT($next, $t, $concepts, $lang,
                  $range)
244         )
245       else $terms
246   };
247
248   (: finds top terms for a token :)
249   declare function fts:lookupThesaurusTT (
250       $token as xs:string,
251       $concepts as element(skos:Concept)*,
252       $lang as xs:string? )
253     as xs:string*
254   {
255     if ($lang) then
256       for $id in $concepts[skos:prefLabel[@xml:lang = $lang] = $token
              or
257                            skos:altLabel[@xml:lang = $lang] = $token]
258                           /skos:inScheme/@rdf:resource
259       return $concepts[skos:topConceptOf[@rdf:resource = $id]]
260                       /skos:prefLabel[@xml:lang = $lang]
261     else
262       for $id in $concepts[skos:prefLabel = $token or
263                            skos:altLabel = $token]
264                           /skos:inScheme/@rdf:resource
265       return $concepts[skos:topConceptOf[@rdf:resource = $id]]
266                       /skos:prefLabel
267   };
268
269   (: finds related terms for a token :)
270   declare function fts:lookupThesaurusRT (
271       $token as xs:string,
272       $concepts as element(skos:Concept)*,
273       $lang as xs:string? )
274     as xs:string*
275   {
276     if ($lang) then
277       for $id in $concepts[skos:prefLabel[@xml:lang = $lang] = $token
              or
278                            skos:altLabel[@xml:lang = $lang] = $token]
279                           /skos:related/@rdf:resource
280       return $concepts[@rdf:about = $id]/skos:prefLabel[@xml:lang =
              $lang]
281     else
282       for $id in $concepts[skos:prefLabel = $token or
283                            skos:altLabel = $token]
284                           /skos:related/@rdf:resource
285       return $concepts[@rdf:about = $id]/skos:prefLabel
286   };
```

A.2 Semantics for fts:resolveStopWordsUri

The following XQuery code (from XQFT[18]) defines the semantics for applying an `FTStopWordOption`:

```
1  declare function fts:applyStopWordOption (
2        $stopWordOption as element(fts:stopwords)? )
3     as xs:string*
```

```
4  {
5     if ($stopWordOption) then
6        let $swords :=
7           typeswitch ($stopWordOption/*[1])
8              case $e as element(fts:stopword)
9                 return $e/text()
10             case $e as element(fts:uri)
11                return fts:resolveStopWordsUri($e/text())
12             case element(fts:default-stopwords)
13                return fts:resolveStopWordsUri(())
14             default return ()
15       return fts:calcStopWords( $swords, $stopWordOption/fts:oper )
16    else ()
17 };
18
19 declare function fts:calcStopWords (
20       $stopWords as xs:string*,
21       $opers as element(fts:oper)* )
22    as element(fts:queryToken)*
23 {
24    if ( fn:empty($opers) ) then $stopWords
25    else
26       let $swords :=
27          typeswitch ($opers[1]/*[1])
28             case $e as element(fts:stopword)
29                return $e/text()
30             case $e as element(fts:uri)
31                return fts:resolveStopWordsUri($e/text())
32             default return ()
33       return
34          if ($opers[1]/@type eq "union") then
35             fts:calcStopWords( ($stopWords, $swords),
36                                $opers[fn:position() gt 2] )
37          else (: "except" :)
38             fts:calcStopWords( $stopWords[fn:not(.)=$swords],
39                                $opers[fn:position() gt 2] )
40 };
```

The following XQuery code shows an example implementation of the semantics for the `fts:resolveStopWordsUri` function used in the `fts:applyStopWordsOption` function above. The function expects a stop word file as in Appendix A.4

```
1 (: extracts stop words from a given $uri :)
2 declare function fts:resolveStopWordsUri( $uri as xs:string? )
3    as xs:string*
4 {
5     doc($uri)//stopWord
6 };
```

A.3 Simple SKOS thesaurus

The following example shows a simple SKOS thesaurus in RDF/XML syntax.

```
1 <rdf:RDF
2   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3   xmlns:skos="http://www.w3.org/2008/05/skos#">
4
5     <skos:Concept rdf:about="http://example.com/t#0">
6        <skos:prefLabel xml:lang="en">automotive
             vehicle</skos:prefLabel>
7        <skos:topConceptOf rdf:resource="http://example.com/t"/>
```

```
8       </skos:Concept>
9
10      <skos:Concept rdf:about="http://example.com/t#1">
11          <skos:prefLabel xml:lang="en">automobile</skos:prefLabel>
12          <skos:prefLabel xml:lang="nb">automobil</skos:prefLabel>
13          <skos:altLabel xml:lang="en">car</skos:altLabel>
14          <skos:altLabel xml:lang="nb">bil</skos:altLabel>
15          <skos:broader rdf:resource="http://example.com/t#2"/>
16          <skos:narrower rdf:resource="http://example.com/t#3"/>
17          <skos:narrower rdf:resource="http://example.com/t#4"/>
18          <skos:narrower rdf:resource="http://example.com/t#5"/>
19          <skos:related rdf:resource="http://example.com/t#6"/>
20          <skos:inScheme rdf:resource="http://example.com/t"/>
21      </skos:Concept>
22
23      <skos:Concept rdf:about="http://example.com/t#2">
24          <skos:prefLabel xml:lang="en">motor
                vehicle</skos:prefLabel>
25          <skos:prefLabel
                xml:lang="nb">motorkjoeretoey</skos:prefLabel>
26          <skos:narrower rdf:resource="http://example.com/t#1"/>
27          <skos:broader rdf:resource="http://example.com/t#7"/>
28          <skos:inScheme rdf:resource="http://example.com/t"/>
29      </skos:Concept>
30
31      <skos:Concept rdf:about="http://example.com/t#3">
32          <skos:prefLabel xml:lang="en">sedan</skos:prefLabel>
33          <skos:prefLabel xml:lang="nb">sedan</skos:prefLabel>
34          <skos:broader rdf:resource="http://example.com/t#1"/>
35          <skos:inScheme rdf:resource="http://example.com/t"/>
36      </skos:Concept>
37
38      <skos:Concept rdf:about="http://example.com/t#4">
39          <skos:prefLabel xml:lang="en">roadster</skos:prefLabel>
40          <skos:prefLabel xml:lang="nb">roadster</skos:prefLabel>
41          <skos:broader rdf:resource="http://example.com/t#1"/>
42          <skos:inScheme rdf:resource="http://example.com/t"/>
43      </skos:Concept>
44
45      <skos:Concept rdf:about="http://example.com/t#5">
46          <skos:prefLabel xml:lang="en">van</skos:prefLabel>
47          <skos:prefLabel xml:lang="nb">van</skos:prefLabel>
48          <skos:broader rdf:resource="http://example.com/t#1"/>
49          <skos:inScheme rdf:resource="http://example.com/t"/>
50      </skos:Concept>
51
52      <skos:Concept rdf:about="http://example.com/t#6">
53          <skos:prefLabel xml:lang="en">motorcar</skos:prefLabel>
54          <skos:prefLabel xml:lang="nb">motorvogn</skos:prefLabel>
55          <skos:related rdf:resource="http://example.com/t#1"/>
56          <skos:inScheme rdf:resource="http://example.com/t"/>
57      </skos:Concept>
58
59      <skos:Concept rdf:about="http://example.com/t#7">
60          <skos:prefLabel xml:lang="en">vehicle</skos:prefLabel>
61          <skos:narrower rdf:resource="http://example.com/t#2"/>
62          <skos:inScheme rdf:resource="http://example.com/t"/>
63      </skos:Concept>
64  </rdf:RDF>
```

A.4 Stop words represented as XML

The following example shows a simple XML representation of stop words.

```
<?xml version="1.0" encoding="UTF-8"?>
<stopWords>
    <stopWord>a</stopWord>
    <stopWord>also</stopWord>
    <stopWord>am</stopWord>
    <stopWord>an</stopWord>
    <stopWord>and</stopWord>
    <stopWord>as</stopWord>
    <stopWord>at</stopWord>
    <stopWord>be</stopWord>
    <stopWord>by</stopWord>
    <stopWord>can</stopWord>
    <stopWord>could</stopWord>
    <stopWord>do</stopWord>
    <stopWord>else</stopWord>
    <stopWord>for</stopWord>
    <stopWord>he</stopWord>
    <stopWord>her</stopWord>
    <stopWord>here</stopWord>
</stopWords>
```

Appendix B

XQFT in BDB XML

All patches are applied to the unpatched version of Oracle Berkeley DB XML 2.4.16, available from Oracle's web site[1]. Before compiling, the pre-generated lexical scanner (xqilla/src/lexer/XQLexer.cpp) must be deleted. Builds have been tested on 32-bit and 64-bit versions of Ubuntu Desktop 9.04. For debugging purposes, BDB XML has been compiled with the `--enable-debug` flag.

B.1 Patch: Fix for full-text logical operators

The following patches generated by Git fixes the incorrect parsing of full-text logical operators as mentioned in Table on page 49. They are applied respectively to the lex specification file (consumed by Flex) and the parser generator file (consumed by Yacc/Bison). Note that regenerating the lexical scanner requires a Unicode-patched version of Flex 2.5.4a, as per build instructions on XQilla's Wiki[2].

B.1.1 Lexer patch

```
1  diff --git a/xqilla/src/lexer/XQLexer.l
       b/xqilla/src/lexer/XQLexer.l
2  index b12b2d0..de7c259 100644
3  --- a/xqilla/src/lexer/XQLexer.l
4  +++ b/xqilla/src/lexer/XQLexer.l
5  @@ -248,9 +248,6 @@ void XQLexer::undoLessThan()
6    <INITIAL>"["                          RECOGNIZE(_LSQUARE_);
7    <INITIAL>"]"                          RECOGNIZE(_RSQUARE_);
8    <INITIAL>"?"                          RECOGNIZE(_QUESTION_MARK_);
9  -<INITIAL>"||"                          RECOGNIZE(_BAR_BAR_);
10 -<INITIAL>"&&"                          RECOGNIZE(_AMP_AMP_);
11 -<INITIAL>"!"                           RECOGNIZE(_BANG_);
12
13   <INITIAL>"external"                   RECOGNIZE_VALUE(_EXTERNAL_,
         yytext);
14   <INITIAL>"ascending"                  RECOGNIZE_VALUE(_ASCENDING_,
         yytext);
15 @@ -373,6 +370,9 @@ void XQLexer::undoLessThan()
```

[1] http://www.oracle.com/technology/software/products/berkeley-db/xml/index.html
[2] http://xqilla.sourceforge.net/FlexandBison

```
16  <INITIAL>"start"                    RECOGNIZE_VALUE(_START_,
        yytext);
17  <INITIAL>"end"                      RECOGNIZE_VALUE(_END_, yytext);
18  <INITIAL>"most"                     RECOGNIZE_VALUE(_MOST_, yytext);
19 +<INITIAL>"ftor"                     RECOGNIZE(_FTOR_);
20 +<INITIAL>"ftand"                    RECOGNIZE(_FTAND_);
21 +<INITIAL>"ftnot"                    RECOGNIZE(_FTUNARYNOT_);
22  <INITIAL>"revalidation"             RECOGNIZE_VALUE(_REVALIDATION_,
        yytext);
23  <INITIAL>"updating"                 RECOGNIZE_VALUE(_UPDATING_,
        yytext);
24  <INITIAL>"skip"                     RECOGNIZE_VALUE(_SKIP_, yytext);
```

B.1.2 Parser patch

```
1  diff --git a/xqilla/src/parser/XQParser.y
       b/xqilla/src/parser/XQParser.y
2  index 7a00c23..75ea511 100644
3  --- a/xqilla/src/parser/XQParser.y
4  +++ b/xqilla/src/parser/XQParser.y
5  @@ -273,8 +273,6 @@ namespace XQParser {
6   %token _LSQUARE_ "["
7   %token _RSQUARE_ "]"
8   %token _QUESTION_MARK_ "?"
9  -%token _BAR_BAR_ "||"
10 -%token _AMP_AMP_ "&&"
11  %token _LESS_THAN_OP_OR_TAG_ "<"
12  %token _START_TAG_CLOSE_ "> (start tag close)"
13  %token _END_TAG_CLOSE_ "> (end tag close)"
14 @@ -303,7 +301,6 @@ namespace XQParser {
15  %token _LBRACE_EXPR_ENCLOSURE_ "{ (expression enclosure)"
16  %token _RBRACE_ "}"
17  %token _SEMICOLON_ ";"
18 -%token _BANG_ "!"
19
20  %token <str> _INTEGER_LITERAL_ "<integer literal>"
21  %token <str> _DECIMAL_LITERAL_ "<decimal literal>"
22 @@ -352,6 +349,9 @@ namespace XQParser {
23  %token <str> _START_ "start"
24  %token <str> _END_ "end"
25  %token <str> _MOST_ "most"
26 +%token <str> _FTOR_ "ftor"
27 +%token <str> _FTAND_ "ftand"
28 +%token <str> _FTUNARYNOT_ "ftnot"
29  %token <str> _SKIP_ "skip"
30  %token <str> _COPY_ "copy"
31  %token <str> _VALUE_ "value"
32 @@ -2971,7 +2971,7 @@ FTSelectionOptions:
33
34  // [145]    FTOr    ::=    FTAnd ( "||" FTAnd )*
35  FTOr:
36 -  FTOr _BAR_BAR_ FTAnd
37 +  FTOr _FTOR_ FTAnd
38     {
39       if($1->getType() == FTSelection::OR) {
40         FTOr *op = (FTOr*)$1;
41 @@ -2987,7 +2987,7 @@ FTOr:
42
43  // [146]    FTAnd    ::=    FTMildnot ( "&&" FTMildnot )*
44  FTAnd:
45 -  FTAnd _AMP_AMP_ FTMildnot
46 +  FTAnd _FTAND_ FTMildnot
```

```
47      {
48        if($1->getType() == FTSelection::AND) {
49          FTAnd *op = (FTAnd*)$1;
50  @@ -3012,7 +3012,7 @@ FTMildnot:
51
52    // [148]     FTUnaryNot     ::=     ("!")? FTWordsSelection
53    FTUnaryNot:
54  -   _BANG_ FTWordsSelection
55  +   _FTUNARYNOT_ FTWordsSelection
56      {
57        $$ = WRAP(@1, new (MEMMGR) FTUnaryNot($2, MEMMGR));
58      }
```

B.2 Patch: Enable XQFT in BDB XML

The following patch generated by Git enables experimental XQFT support in BDB XML. It is applied to the internal class for representing arbitrary query expressions.

```
1  diff --git a/dbxml/src/dbxml/QueryExpression.cpp
       b/dbxml/src/dbxml/QueryExpression.cpp
2  index ae4ab76..ed95a02 100644
3  --- a/dbxml/src/dbxml/QueryExpression.cpp
4  +++ b/dbxml/src/dbxml/QueryExpression.cpp
5  @@ -42,7 +42,7 @@ QueryExpression::QueryExpression(const
       std::string &query, XmlQueryContext &cont
6          context_(context),
7          qec_(context_, /*debugging*/false),
8          conf_(context, txn, &ci_),
9  -       xqContext_(XQilla::createContext(XQilla::XQUERY_UPDATE,
       &conf_, Globals::defaultMemoryManager)),
10 +
       xqContext_(XQilla::createContext(XQilla::XQUERY_FULLTEXT_UPDATE,
        &conf_, Globals::defaultMemoryManager)),
11         expr_(0)
12   {
13       ((Manager &)((QueryContext &)getContext()).getManager())
```

B.3 Default tokenizer implementation

The following code shows the default implementation for the tokenizer in XQilla.

```
1  /*
2   * Copyright (c) 2001-2008
3   *      DecisionSoft Limited. All rights reserved.
4   * Copyright (c) 2004-2008
5   *      Oracle. All rights reserved.
6   *
7   * Licensed under the Apache License, Version 2.0 (the "License");
8   * you may not use this file except in compliance with the License.
9   * You may obtain a copy of the License at
10  *
11  *      http://www.apache.org/licenses/LICENSE-2.0
12  *
13  * Unless required by applicable law or agreed to in writing,
        software
14  * distributed under the License is distributed on an "AS IS"
        BASIS,
15  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
        implied.
```

```
16   * See the License for the specific language governing permissions
       and
17   * limitations under the License.
18   *
19   * $Id: DefaultTokenizer.cpp 475 2008-01-08 18:47:44Z jpcs $
20   */
21
22 #include "../config/xqilla_config.h"
23 #include <xqilla/fulltext/DefaultTokenizer.hpp>
24 #include <xqilla/framework/XPath2MemoryManager.hpp>
25 #include <xqilla/context/DynamicContext.hpp>
26
27 #include <xercesc/util/XMLString.hpp>
28
29 #if defined(XERCES_HAS_CPP_NAMESPACE)
30 XERCES_CPP_NAMESPACE_USE
31 #endif
32
33 TokenStream::Ptr DefaultTokenizer::tokenize(const Node::Ptr &node,
       DynamicContext *context) const
34 {
35   return new DefaultTokenStream(node->dmStringValue(context),
         context->getMemoryManager());
36 }
37
38 TokenStream::Ptr DefaultTokenizer::tokenize(const XMLCh *str,
       XPath2MemoryManager *mm) const
39 {
40   return new DefaultTokenStream(str, mm);
41 }
42
43 DefaultTokenizer::DefaultTokenStream::DefaultTokenStream(const
       XMLCh *str, XPath2MemoryManager *mm)
44   : string_(XMLString::replicate(str, mm)),
45     current_(string_),
46     tokenStart_(0),
47     position_(0),
48     sentence_(0),
49     paragraph_(0),
50     seenEndOfSentence_(false),
51     mm_(mm)
52 {
53 }
54
55 DefaultTokenizer::DefaultTokenStream::~DefaultTokenStream()
56 {
57   mm_->deallocate(string_);
58 }
59
60 #define REPORT_TOKEN \
61       if(tokenStart_ != 0) { \
62         *current_ = 0; \
63         result = new
              DefaultTokenInfo(mm_->getPooledString(tokenStart_),
              position_, sentence_, paragraph_); \
64         ++position_; \
65         tokenStart_ = 0; \
66       }
67
68
69 TokenInfo::Ptr DefaultTokenizer::DefaultTokenStream::next()
70 {
```

```
71    TokenInfo::Ptr result(0);
72    while(result.isNull()) {
73      switch(*current_) {
74      case '\n': {
75        REPORT_TOKEN;
76        if(seenEndOfSentence_) {
77          ++paragraph_;
78          seenEndOfSentence_ = false;
79        }
80        break;
81      }
82      case '!':
83      case '?':
84      case ':':
85      case '.': {
86        REPORT_TOKEN;
87        if(!seenEndOfSentence_) {
88          ++sentence_;
89          seenEndOfSentence_ = true;
90        }
91        break;
92      }
93      case '\r':
94      case '\t':
95      case ' ':
96      case '"':
97      case '\'':
98      case '`':
99      case ';':
100     case ',': {
101       REPORT_TOKEN;
102       break;
103     }
104     case 0: {
105       REPORT_TOKEN;
106       return result;
107     }
108     default: {
109       if(tokenStart_ == 0) {
110         tokenStart_ = current_;
111         seenEndOfSentence_ = false;
112       }
113       break;
114     }
115     }
116
117     ++current_;
118   }
119
120   return result;
121 }
```

B.4 Example data

The following example data is loaded into a BDB XML container (shown in Appendix B.5 on page 89), and used in experiments in Chapter 6 on page 55.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <books>
3   <book id="1">
```

```
4       <title>Modern Information Retrieval</title>
5       <publisher>Addison Wesley; 1st edition (May 15,
            1999)</publisher>
6       <isbn n="10">020139829X</isbn>
7       <isbn n="13">978-0201398298</isbn>
8       <author>Richardo Baeza-Yates</author>
9       <author>Berthier Ribeiro-Neto</author>
10      <description>Information retrieval (IR) has changed
            considerably in recent years with the expansion of the
            World Wide Web and the advent of modern and inexpensive
            graphical user interfaces and mass storage devices. As a
            result., traditional IR textbooks have become quite out of
            date and this has led to the introduction of new IR books.
            Nevertheless, we believe that there is still great need for
             a book that approaches the field in a rigorous and
            complete way from a computer-science perspective (as
            opposed to a user-centered perspective). This book is an
            effort to partially fulfill this gap and should be useful
            for a first course on information retrieval as well as for
            a graduate course on the topic.
11
12  The book comprises two portions which complement and balance each
        other. The core portion includes nine chapters authored or
        co-authored by the designers of the book. The second portion,
        which is fully integrated with the first, is formed by six
        state-of-the-art chapters written by leading researchers in
        their fields. The same notation and glossary are used in all
        the chapters. Thus, despite the fact that several people have
        contributed to the text, this book is really much more a
        textbook than an edited collection of chapters written by
        separate authors. Furthermore, unlike a collection of chapters,
         we have carefully designed the contents and organization of
        the book to present a cohesive view of all the important
        aspects of modern information retrieval.
13
14  From IR models to indexing text, from IR visual tools and
        interfaces to the Web, from IR. multimedia to digital
        libraries, the book provides both breadth of coverage and
        richness of detail. It is our hope that, given the now clear
        relevance and significance of information retrieval to modern
        society. the book will contribute to further disseminate the
        study of the discipline at information science, computer
        science, and library science departments throughout the
        world.</description>
15      <toc>
16        <chapter>
17          <name>Introduction</name>
18          <section>
19            <name>Motivation</name>
20          </section>
21          <section>
22            <name>Basic Concepts</name>
23          </section>
24        </chapter>
25        <chapter>
26          <name>Modeling</name>
27          <section>
28            <name>Introduction</name>
29          </section>
30          <section>
31            <name>A Taxonomy of Information Retrieval Models</name>
32          </section>
```

```
33        </chapter>
34      </toc>
35    </book>
36
37    <book id="2">
38      <title>Managing Gigabytes: Compressing and Indexing Documents
            and Images</title>
39      <publisher>Morgan Kaufmann; 2 Sub edition (May 15,
            1999)</publisher>
40      <isbn n="10">1558605703</isbn>
41      <isbn n="13">978-1558605701</isbn>
42      <author>Alistair Moffat</author>
43      <author>Timothy C. Bell</author>
44      <description>In this fully updated second edition of the highly
             acclaimed Managing Gigabytes, authors Witten, Moffat, and
            Bell continue to provide unparalleled coverage of
            state-of-the-art techniques for compressing and indexing
            data. Whatever your field, if you work with large
            quantities of information, this book is essential
            reading---an authoritative theoretical resource and a
            practical guide to meeting the toughest storage and access
            challenges. It covers the latest developments in
            compression and indexing and their application on the Web
            and in digital libraries. It also details dozens of
            powerful techniques supported by mg, the authors' own
            system for compressing, storing, and retrieving text,
            images, and textual images. mg's source code is freely
            available on the Web.
45
46    * Up-to-date coverage of new text compression algorithms such as
        block sorting, approximate arithmetic coding, and fat Huffman
        coding
47    * New sections on content-based index compression and distributed
        querying, with 2 new data structures for fast indexing
48    * New coverage of image coding, including descriptions of de facto
        standards in use on the Web (GIF and PNG), information on
        CALIC, the new proposed JPEG Lossless standard, and JBIG2
49    * New information on the Internet and WWW, digital libraries, web
        search engines, and agent-based retrieval
50    * Accompanied by a public domain system called MG which is a fully
        worked-out operational example of the advanced techniques
        developed and explained in the book
51    * New appendix on an existing digital library system that uses the
        MG software</description>
52      <toc>
53        <chapter>
54          <name>Overview</name>
55          <section>
56            <name>Document databases</name>
57          </section>
58          <section>
59            <name>Compression</name>
60          </section>
61        </chapter>
62        <chapter>
63          <name>Text Compression</name>
64          <section>
65            <name>Models</name>
66          </section>
67          <section>
68            <name>Adaptive models</name>
69          </section>
```

```
70        </chapter>
71      </toc>
72    </book>
73
74    <book id="3">
75      <title>Introduction to Information Retrieval</title>
76      <publisher>Cambridge University Press; 1 edition (July 7,
            2008)</publisher>
77      <isbn n="10">0521865719</isbn>
78      <isbn n="13">978-0521865715</isbn>
79      <author>Christopher D. Manning</author>
80      <author>Prabhakar Raghavan</author>
81      <author>Hinrich Schutze</author>
82      <description>Class-tested and coherent, this groundbreaking new
             textbook teaches web-era information retrieval, including
            web search and the related areas of text classification and
             text clustering from basic concepts. Written from a
            computer science perspective by three leading experts in
            the field, it gives an up-to-date treatment of all aspects
            of the design and implementation of systems for gathering,
            indexing, and searching documents; methods for evaluating
            systems; and an introduction to the use of machine learning
             methods on text collections. All the important ideas are
            explained using examples and figures, making it perfect for
             introductory courses in information retrieval for advanced
             undergraduates and graduate students in computer science.
            Based on feedback from extensive classroom experience, the
            book has been carefully structured in order to make
            teaching more natural and effective. Although originally
            designed as the primary text for a graduate or advanced
            undergraduate course in information retrieval, the book
            will also create a buzz for researchers and professionals
            alike.</description>
83      <toc>
84        <chapter>
85          <name>Boolean retrieval</name>
86          <section>
87            <name>An example of information retrieval problem</name>
88          </section>
89          <section>
90            <name>A first take at building an inverted index</name>
91          </section>
92        </chapter>
93        <chapter>
94          <name>The term vocabulary and postings lists</name>
95          <section>
96            <name>Document delineation and character sequence
                  decoding</name>
97          </section>
98          <section>
99            <name>Determining the vocabulary of terms</name>
100         </section>
101       </chapter>
102     </toc>
103   </book>
104
105   <book id="4">
106     <title>Information Retrieval: Algorithms and Heuristics (The
            Information Retrieval Series)(2nd Edition)</title>
107     <publisher>Springer; 2nd edition (December 20,
            2004)</publisher>
108     <isbn n="10">1402030045</isbn>
```

```
109       <isbn n="13">978-1402030048</isbn>
110       <author>David A. Grossman</author>
111       <author>Ophir Frieder</author>
112       <description>Interested in how an efficient search engine
              works? Want to know what algorithms are used to rank
              resulting documents in response to user requests? The
              authors answer these and other key information retrieval
              design and implementation questions.
113
114   This book is not yet another high level text. Instead, algorithms
          are thoroughly described, making this book ideally suited for
          both computer science students and practitioners who work on
          search-related applications. As stated in the foreword, this
          book provides a current, broad, and detailed overview of the
          field and is the only one that does so. Examples are used
          throughout to illustrate the algorithms.
115
116   The authors explain how a query is ranked against a document
          collection using either a single or a combination of retrieval
          strategies, and how an assortment of utilities are integrated
          into the query processing scheme to improve these rankings.
          Methods for building and compressing text indexes, querying and
           retrieving documents in multiple languages, and using parallel
           or distributed processing to expedite the search are likewise
          described.
117
118   This edition is a major expansion of the one published in 1998.
          Besides updating the entire book with current techniques, it
          includes new sections on language models, cross-language
          information retrieval, peer-to-peer processing, XML search,
          mediators, and duplicate document detection.</description>
119       <toc>
120         <chapter>
121           <name>Retrieval Strategies</name>
122           <section>
123             <name>Vector Space Model</name>
124           </section>
125           <section>
126             <name>Probabilistic Retrieval Strategies</name>
127           </section>
128         </chapter>
129         <chapter>
130           <name>Retrieval Utilities</name>
131           <section>
132             <name>Relevance Feedback</name>
133           </section>
134           <section>
135             <name>Clustering</name>
136           </section>
137         </chapter>
138       </toc>
139     </book>
140   </books>
```

B.5 Loading example data into a container

The following script (`load.xquery`) is loaded and and executing the command `dbxml -s load.xquery`. Note that the `$uri` variable in the query refers to a file in the local file system. Appendix B.4 on page 85 shows the contents of the

file `books.xml`.

```
createContainer books.dbxml
putDocument "" '
let $uri := "books.xml"
for $book in doc($uri)//book
return $book
' q
sync
exit
```

Bibliography

[1] Shurug Al-Khalifa, Cong Yu, and H. V. Jagadish. Querying structured text in an xml database. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 4–15, New York, NY, USA, 2003. ACM.

[2] S. Amer-Yahia, C. Botev, J. Dörre, and J. Shanmugasundaram. Xquery full-text extensions explained. *IBM Syst. J.*, 45(2):335–351, 2006.

[3] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. Texquery: a full-text search extension to xquery. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 583–594, New York, NY, USA, 2004. ACM.

[4] Sihem Amer-Yahia and Jayavel Shanmugasundaram. Xml full-text search: challenges and opportunities. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1368–1368. VLDB Endowment, 2005.

[5] Dave Beckett. RDF/xml syntax specification (revised). W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[6] Anand Bhaskar, Chavdar Botev, Muthiah M. Muthaia Chettiar, Lin Guo, Jayavel Shanmugasundaram, Feng Shao, and Fan Yang. Quark: an efficient xquery full-text implementation. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 781–783, New York, NY, USA, 2006. ACM.

[7] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[8] Scott Boag, Anders Berglund, Don Chamberlin, Jérôme Siméon, Michael Kay, Jonathan Robie, and Mary F. Fernández. XML path language (XPath) 2.0. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/.

[9] Jeremy J. Carroll and Graham Klyne. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[10] Don Chamberlin, Jonathan Robie, Peter Fankhauser, Daniela Florescu, and Massimo Marchiori. XML query use cases. W3C note, W3C, March 2007. http://www.w3.org/TR/2007/NOTE-xquery-use-cases-20070323/.

[11] 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*, pages 1–25, London, UK, 2001. Springer-Verlag.

[12] John Cowan and Richard Tobin. XML information set (second edition). W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-xml-infoset-20040204.

[13] Steven DeRose and James Clark. XML path language (XPath) version 1.0. W3C recommendation, W3C, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[14] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for xml. In *WWW '99: Proceedings of the eighth international conference on World Wide Web*, pages 1155–1169, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[15] Daniel Engovatov and Daniel Engovatov. XML query (XQuery) 1.1 requirements. W3C working draft, W3C, March 2007. http://www.w3.org/TR/2007/WD-xquery-11-requirements-20070323.

[16] Joseph Fong, Francis Pang, and Chris Bloor. Converting relational database into xml document. In *DEXA '01: Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, pages 61–65, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. Xrank: ranked keyword search over xml documents. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 16–27, New York, NY, USA, 2003. ACM.

[18] Mary Holstege, Jochen Doerre, Jim Melton, Pat Case, Chavdar Botev, Sihem Amer-Yahia, Jayavel Shanmugasundaram, Michael Rys, and Stephen Buxton. XQuery and XPath full text 1.0. W3C candidate recommendation, W3C, May 2008. http://www.w3.org/TR/2008/CR-xpath-full-text-10-20080516/.

[19] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 779–790, New York, NY, USA, 2004. ACM.

[20] Michael Kay. XSL transformations (XSLT) version 2.0. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[21] Jim Melton, Ashok Malhotra, and Norman Walsh. XQuery 1.0 and XPath 2.0 functions and operators. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath-functions-20070123/.

[22] Alistair Miles. Quick guide to publishing a thesaurus on the semantic web. W3C working draft, W3C, May 2005. http://www.w3.org/TR/2005/WD-swbp-thesaurus-pubguide-20050517.

[23] Alistair Miles and Sean Bechhofer. SKOS simple knowledge organization system reference. W3C working draft, W3C, March 2009. http://www.w3.org/TR/2009/CR-skos-reference-20090317/.

[24] Oracle. Anatomy of an xml database: Oracle berkeley db xml. Technical report, Oracle Corporation, 2006.

[25] Peter F. Patel-Schneider, Ian Horrocks, and Patrick Hayes. OWL web ontology language semantics and abstract syntax. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.

[26] Martin F. Porter. An algorithm for suffix stripping. pages 313–316, 1997.

[27] Jonathan Robie, Joe Lapp, and David Schach. XML query language (XQL). W3C proposal, W3C, September 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[28] Mark Scardina, Mary F. Fernández, and K. Karun. XPath requirements version 2.0. W3C working draft, W3C, June 2005. http://www.w3.org/TR/2005/WD-xpath20req-20050603/.

[29] Feng Shao, Lin Guo, Chavdar Botev, Anand Bhaskar, Muthiah Chettiar, Fan Yang, and Jayavel Shanmugasundaram. Efficient keyword search over virtual xml views. *The VLDB Journal*, 18(2):543–570, 2009.

[30] Jérôme Siméon, Don Chamberlin, Daniela Florescu, Scott Boag, Mary F. Fernández, and Jonathan Robie. XQuery 1.0: An XML query language. W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[31] Norman Walsh, Mary Fernández, Ashok Malhotra, Marton Nagy, and Jonathan Marsh. XQuery 1.0 and XPath 2.0 data model (XDM). W3C recommendation, W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/.