ETH zürich

ETH Library

Evaluation strategies for Xquery full-text

Master Thesis

Author(s):
Imhof, Julia

Publication date:
2008

Permanent link:
https://doi.org/10.3929/ethz-a-005676938

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Evaluation Strategies for XQuery Full-Text

Master Thesis

Systems Group

March 10, 2008 - September 10, 2008

Julia Imhof
ETH Zurich
jimhof@student.ethz.ch

Supervised by:
Dr. Peter M. Fischer
Kyumars S. Esmaili

Abstract

More and more repositories like the IEEE INEX collection [1], LexisNexis [2] or the Library of Congress collection [3] store their documents in XML format. To be able to search over these XML documents, there is a need for efficient and accurate full-text search features. There are two different approaches that could be taken into consideration: On one hand, one could use traditional full-text approaches, but these are not suitable for XML documents as they do not take the structure of the document into account. On the other hand, one could use XQuery and XPath. These languages can express a variety of queries, but queries over XML text content are limited. Full-text search in XQuery is mainly handled by the function *fn:contains($e,keyword)* that checks whether the keyword *keyword* is contained in the element denoted by $e. This function is too limited for complex queries. It cannot express queries e.g. like the following query that includes phrase matching, distance predicates and stemming:

Example The sample document contains books (see Appendix A). Find the books that contain the phrases "Beating the Dealer" with stemming and "WIN STRATEGY" with stemming and case insensitive in distance at most 10 words from each other.

Before this Master thesis, *MXQuery* Full-Text only supported keyword and phrase queries [4]. In this Master thesis, we extended *MXQuery* Full-Text with the features of the XPath 2.0 and XQuery 1.0 Full-Text W3C specification [5]. This extension includes an improved store and indexes, i.e. a stem index, an n-gram index, an nextword index and a B+ tree index. Additionally, we implemented the *MatchOptions* that support queries including stemming, the use of thesauri, diacritics sensitivity, case options and wildcards. To support logical operators, we extended *MXQuery* Full-Text with an *FTAndIterator*, an *FTOrIterator* and an *FTUnaryNotIterator*. For queries including positional filters, we added an *FTSelectionIterator* that tests the positional predicates. In addition, we had a look at several scoring models to design and implement our own scoring method. To test our implementation, we run the XPath 2.0 and XQuery 1.0 Use Cases [6] and did some benchmarking to evaluate the performance and memory usage.

Contents

1 Introduction **7**

2 XQuery 1.0 and XPath 2.0 Full-Text 1.0 **9**
2.1 Full-text Contains Expression: FTContainsExpr . . . 9
2.2 Score Variables: FTScoreVar . . . 10
2.3 Full Text Selections: FTSelection . . . 10
2.4 Logical Full-Text Operators: FTOr, FTAnd, FTMildNot, FTUnaryNot . . . 12
2.5 Positional Filters: FTOrder, FTWindow, FTDistance, FTScope, FTContent . . . 13
2.6 Cardinality Selection: FTTimes . . . 14
2.7 Ignore Option: FTIgnoreOption . . . 14
2.8 Extension Selection: FTExtensionSelection . . . 14
2.9 Minimal Conformance . . . 14

3 Initial State **17**
3.1 Parser and Runtime . . . 17
3.2 Preprocessing of the Documents . . . 17
3.3 Index . . . 17
3.4 Dewey Numbering . . . 19
3.5 Retrieval Model . . . 19
3.6 Query evaluation . . . 19

4 *MXQuery* Full-Text Design and Implementation **23**
4.1 Design and Requirements . . . 23
4.1.1 Store Requirements . . . 23
4.1.2 MatchOptions . . . 24
4.1.3 FTMildNot, FTUnaryNot, FTOr and FTAnd . . . 29
4.1.4 FTOrder, FTWindow, FTDistance, FTScope, FTContent, FTRange . . . 34
4.1.5 FTTimes . . . 39
4.1.6 FTIgnoreOption . . . 40
4.1.7 FTAnyAllOption . . . 41
4.2 Implementation . . . 42
4.2.1 Processing of Documents . . . 42
4.2.2 Indexes . . . 44
4.2.3 Store . . . 48

4.2.4 Iterators . . . 52
4.2.5 Minimal Conformance . . . 57

5 Ranking **59**
5.1 Content-Based Ranking . . . 59
5.1.1 Ranked Boolean Retrieval Model . . . 59
5.2 Content and Structure-Based Ranking . . . 60
5.2.1 XRank . . . 61
5.2.2 Vector Space Model . . . 65
5.2.3 Structure and Content Scoring for XML . . . 67
5.3 *MXQuery* Scoring Model . . . 68
5.3.1 Document Collection . . . 68
5.3.2 Single Document . . . 68

6 Benchmarking **71**
6.1 Time Measurements . . . 71
6.1.1 Loading the Data . . . 73
6.1.2 Creation of Linguistic Tokens . . . 73
6.1.3 Building the Indexes . . . 73
6.1.4 Queries . . . 73
6.1.5 Profiling Results . . . 76
6.2 Recall and Precision . . . 76

7 Summary and Future Work **77**
7.1 Summary . . . 77
7.2 Conclusion . . . 77
7.3 Future Work . . . 78
7.3.1 Dewey Identifiers and Updates . . . 78
7.3.2 Indexing . . . 79
7.3.3 Optimization Ideas for Operator Trees . . . 80
7.3.4 Ranking . . . 81
7.3.5 Ranked Boolean Retrieval Model for XML . . . 82

A Sample data **85**

B A EBNF for XQuery 1.0 Grammar with Full-Text Extensions **89**

Chapter 1

Introduction

Many different approaches have been proposed to search over XML data: One approach is JuruXML [7] that queries XML documents via pieces of XML documents or XML fragments. These fragments are of the same nature as the documents that are queried. A document is considered as a valid result if it contains the query or part of it as a subtree. An extension of the vector space model ranks the XML results by relevance.

Another approach uses XSEarch [8]. This search engine uses its own query language: the user enters search terms, an index is used to find nodes that satisfy the search terms and find out whether pairs of nodes interconnect. The results are then ranked using extended traditional information retrieval techniques and returned.

A third approach is TeXQuery [9] which is a full-text search extension to XQuery. It provides full-text primitives like boolean operators, phrase matching, proximity distance, stemming and thesauri. These primitives can be seamlessly embedded into XQuery. Additionally, it contains a scoring construct that can be used to score query results. TeXQuery is the precursor of the full-text language extensions to XPath 2.0 and XQuery 1.0 [5]. One implementation of TeXQuery is the Quark Project [10] by the Cornell University and AT&T Research Lab. A first implementation of the XPath 2.0 and XQuery 1.0 Full-Text W3C specification is GalaTex [11].

This Master thesis aims to integrate all the full-text features of the XPath 2.0 and XQuery 1.0 Full-Text W3C specification to make the implementation minimal conform [5]. To achieve this, we need to integrate an improved store and appropriate indexes. In addition, we need to design and implement a scoring method to rank the results according their relevancy.

The thesis is organized as follows: Chapter 2 gives an introduction to XQuery 1.0 and XPath 2.0 Full-Text 1.0. Chapter 3 describes the initial state of the *MXQuery Full-Text* facility. Chapter 4 is about the design and implementation of parts of the *Minimal Conformance* of XQuery Full-Text (see Section 2.9). Chapter 5 investigates different ranking strategies for XML documents and explain our ranking model and Chapter 6 measures performance and memory usage of our full-text implementation. Chapter 7 concludes the thesis with a short summary, conclusion and a discussion about future work.

Chapter 2

XQuery 1.0 and XPath 2.0 Full-Text 1.0

This chapter introduces the XQuery and XPath Full-Text specification. It is organized as follows: Section 2.1 describes the syntax and semantics of the most fundamental additional XQuery function, the *FTContainsExpr*. Section 2.2 describes the semantics and restrictions of the *Score Variable*. Section 2.3 is about the semantics of the full-text search conditions which are explained more concisely in Sections 2.4 to 2.8. The last Section, 2.9, is about the Minimal Conformance of XQuery Full-Text. The XQuery and XPath Full-Text EBNF can be found in Appendix B.

2.1 Full-text Contains Expression: FTContainsExpr

A full-text contains expression evaluates a sequence of nodes against a full-text selection and behaves like a comparison expression. Hence, it can be used anywhere a comparison expression may be used [5].

Syntax: expr "ftcontains" FTSelection

The FTContainsExpr returns a boolean value: It returns true if there is some node in the search context *expr* that matches the full-text selection *FTSelection*, otherwise it returns false. *FTSelection* can be a single search term (content only query) or a more complex full-text search expression containing boolean connectors (and, or, not) or a scope of a search token, e.g. whether they occur in the same sentence or paragraph. Other full-text search selections can include window expressions or number of occurrences of a search term. Additionally, FTMatchOptions that use stemming[1], stop-words, upper and lower case, special characters or synonyms identified by a thesaurus[2] can be applied to FTSelec-

[1] Words that have the same stem are also included as search terms. These terms can be nouns, verbs, adjectives, and adverb forms of the search term in singular and plural.

[2] Thesauri include synonyms or related terms of a search term. These can be used to expand a query.

tions.

Example

```
//book/title[ .ftcontains "BEATING THE DEALER" with stemming
lowercase]
```

2.2 Score Variables: FTScoreVar

In addition to returning a boolean value as result of a match of a node and a full-text selection, full-text search features can also associate scores with the result nodes. A score expresses the importance of a node to the query and must be in the range of [0,1]. A higher score must imply a higher degree of relevance and at the end the results are usually ordered by their score. Scoring may be influenced by the user adding weight declarations to search tokens, phrases or expressions.

Example

```
for $book score $s in
   doc("sample_data.xml")
   /books/book[. ftcontains "Party"]
where $s > 0.1
return $book/metadata/title
```

2.3 Full Text Selections: FTSelection

Full-text selections define the full-text search conditions that may include logical operators, followed by positional filter(s) (e.g. window or distance predicate) and optionally a weight value. The *Primary Full-Text Selection* is the basic form of a full-text selection and specifies words and phrases as search conditions, i.e. the *FTWords* operator. There are five different *FTAnyallOption* options on how the tokens and phrases can be matched.

- any word: At least one of the query terms needs to match
- all words: All query terms need to match
- phrase: all query terms need to match as a phrase
- any: At least one of the phrases in the sequence of strings needs to match
- all: All of the phrases in the sequence of strings need to match

The *Primary Full-Text Selection* is optionally followed by a cardinality constraint, i.e. whether the term should appear more than one time.

Example

```
//book/title[ .ftcontains {Six Millions, Vegas} all words]
```

MatchOptions modify the matching behavior of the primary full-text selection: They modify the set of keywords in the query or how they are matched against tokens in the text. There are seven different MatchOptions:

- *FTLanguageOption*: Specification of the language of search keyword(s). This option influences tokenization, stemming and stop words and may influence other *MatchOptions*.

- *FTWildcardOption*: Specification whether wildcards are used or not. The option "with wildcards" means that if a word contains a ., .?, .*, .+ or .$\{n, m\}$ at the beginning, inserted into or at the end of a query keyword, a matching keyword in the text may have one character, an optional character, zero or more characters, one or more characters or a range of characters between n and m as prefix, infix or suffix, respectively.

- *FTThesaurusOption*: Specification whether a thesaurus is used or not. If a thesaurus is used, the user must specify the location of this thesaurus. The terms determined by the thesaurus are processed as if the user had specified the terms in a disjunction. In addition, one can specify how many levels within the thesaurus have to be traversed and what kind of relationship the search term and the terms identified by the thesaurus have, e.g. in "synonyms" or "sound like" relationship.

- *FTStemOption*: Specification whether stemming is applied or not. There are the options "with stemming", which means that matches can also contain words with the same stem as the query term, or "without stemming", which means that the query term is not stemmed.

- *FTCaseOption*: Specification of how uppercase and lowercase characters are considered. There are four case options:

  - Option "case insensitive": Keywords and words in the text are matched, without considering case of characters of the keywords and words in the text.
  - Option "case sensitive": Keywords and words in the text are matched, if and only if the case of the characters in the text is the same as written in the query.
  - Option "lowercase": Keywords and words in the text are matched, if and only if they match the query terms in lower case characters.
  - Option "uppercase": Keywords and words in the text are matched, if and only if they match the query terms in upper case characters.

- *FTDiacriticsOption*: Specification of how diacritics[3] are considered. There are two possible options: either diacritics are considered ("diacritics sensitive"), i.e. there is only a match if the diacritics are contained as they are written in the query or in the text, or the diacritics are not considered at all ("diacritics insensitive").

[3]A diacritic is a small sign added to a letter to alter pronunciation or to distinguish between similar words.

- *FTStopwordOption*: Specification of whether stop words are used or not. The "with stop words option" means that if a query token is specified in the collection of stop words, it is removed and replaced by any word. In range queries, i.e. using distance or window, they are still considered and counted as words. If the stop word option is applied during indexing, i.e. the stop words are removed when building the indexes, the stop word option has no effect in the query.

- *FTExtensionOption*: MatchOption that acts in an implementation defined way.

Example

```
//book/summary[ .ftcontains "mit" case insensitive]
```

MatchOptions of the same group can not be combined. The following query is not allowed:

Example

```
//book/summary[ .ftcontains "mit" uppercase lowercase]
```

It is not allowed to combine options of the same MatchOption. However, it is allowed to combine different MatchOptions:

Example

```
//book/summary[ .ftcontains "win.+ strategy" uppercase
with wildcards with stemming with default thesaurus]
```

If different MatchOptions are combined, the application order has to satisfy the following constraints: The *FTLanguageOption* must be applied first as it influences tokenization, stemming and the use of stop words, and the *FTStemOption* must be applied before the *FTCaseOption* and the *FTDiacriticsOption*(see Section 4.1.2 for our application order).

2.4 Logical Full-Text Operators: FTOr, FTAnd, FTMildNot, FTUnaryNot

The full-text selections that are defined above can be combined using the following logical operators.

- *FTOr*: combines two full-text selections using the *ftor* operand and returns all matches that satisfy at least one of these selections.

- *FTAnd*: combines two full-text selections using the *ftand* operand and returns all matches that satisfy both of these selections.

- *MildNot*: combines two full-text selections using the *not in* operand. A *not in* B means that a search term A is matched if it is not contained in the search phrase B.

- *UnaryNot*: if a full-text selection starts with the prefix operator *ftnot*, it returns all the matches that do not satisfy this selection.

Example

```
//book/title[ .ftcontains "Vegas" ftor "Zurich"]
```

2.5 Positional Filters: FTOrder, FTWindow, FTDistance, FTScope, FTContent

The postfix operators are used to filter matches based on constraints on their positional information. There can be multiple adjacent positional filters that are applied from left to right.

- *FTOrder*: The ordered selection consists of a full-text selection followed by the postfix operator *ordered* and controls the order of the search terms to be the same as the order in which they appear in the query.
- *FTWindow*: The window selection consists of a full-text selection followed by one of the following window operators:
  - window i words
  - window i sentences
  - window i paragraphs

  where i is a user-defined integer number. A match occurs if the search terms are within the defined number of words, sentences or paragraphs, i.e. *FTUnit* and *FTBigUnit*. Element boundaries are not considered, i.e. the size of the window is not affected by the element boundaries.
- *FTDistance*: The distance selection consists of a full-text selection followed by one of the following *FTRange* operators:
  - distance [exactly i | at least i | at most i | from i to j] words
  - distance [exactly i | at least i | at most i | from i to j] sentences
  - distance [exactly i | at least i | at most i | from i to j] paragraphs

  where i and j are user-defined integer number. If zero is used, the terms need to be adjacent to each other. Again, element boundaries are not considered when determining the distance.
- *FTScope*: The scope selection is a full-text selection followed by one of the following scope operators:
  - [same | different] sentence
  - [same | different] paragraph

  The matched search terms need to be in the same or different sentence or paragraph, respectively.
- *FTContent*: The anchoring selection consists of a full-text selection followed by one of the content operators:
  - at start

- at end
- entire content

The matched search terms need to be contained at the start, at the end or in the entire content of the item being searched.

Example

```
//book/title[ .ftcontains "the wave" case insensitive ftand
"classroom" window 20 words]
```

2.6 Cardinality Selection: FTTimes

This selection consists of a search term followed by the FTTimes operator:

occurs [exactly | at least | at most i | from i to j] times.

where i and j are user-defined integer number. A match occurs if the search term occurs as many times as defined by the number i and j, respectively.

Example

```
//book/summary[ .ftcontains "MIT" occurs at least 2 times]
```

2.7 Ignore Option: FTIgnoreOption

With this option one can define a set of nodes, i.e. by giving an XQuery expression, whose content are ignored, e.g. *without content ./path* the content of element ./path is ignored in the search. **Example**

```
//book[ .ftcontains "Professor of Mathematics" without content ./p]
```

2.8 Extension Selection: FTExtensionSelection

The Extension selection is a selection whose semantics is implementation-defined, e.g. if several indexes exist, the user could define, which one should be used to execute the query.

2.9 Minimal Conformance

The first goal of this Master thesis is to fulfill the Minimal Conformance of the XQuery Full-Text specification [5]. Apart from fulfilling the minimal support for XQuery 1.0 or XPath 2.0, XQuery Full-Text needs to support the following operators:

- MatchOptions:
  - *FTCaseOption*: The *case sensitive* and *case insensitive* choices need to be supported.

- *FTDiacriticsOption*: needs to be supported.
- *FTStemOption*: needs to be supported.
- *FTThesaurusOption*: needs to be supported.
- *FTWildCardOption*: needs to be supported.
- *FTExtensionOption*: needs to be supported.

- Logical Full-Text operators
  - *FTAndOperator*: needs to be supported.
  - *FTOrOperator*: needs to be supported.
  - *FTUnaryNotOperator*: The form that can negate every kind of FT-Selection does not need to be supported. One can choose to support the negation operation in a restricted form. One of the two restrictions needs to be supported:
    * Negation Restriction 1. An FTUnaryNot expression may only appear as a direct right operand of an "ftand" (FTAnd) operation.
    * Negation Restriction 2. An FTUnaryNot expression may not appear as a descendant of an FTOr that is modified by an FT-PosFilter.

- Positional filters
  - *FTWindow*: The form of the FTWindow postfix operator that can be applied to any kind of FTSelection, is optional. However, the restricted use of FTWindow needs to be supported:
    * Window Operator Restriction: FTWindow can only be applied to an FTOr that is either a single FTWords or a combination of FTWords involving only the operators "ftand" and "ftor".
  - *FTDistance*: The form of the FTDistance postfix operator that can be applied to any kind of FTSelection, is optional. However, the restricted use of FTDistance needs to be supported:
    * Distance Operator Restriction: FTDistance can only be applied to an FTOr that is either a single FTWords or a combination of FTWords involving only the operators "ftand" and "ftor".
  - *FTOrder*: The form of the FTOrder postfix operator that can be applied to any kind of FTSelection is optional. However, the restricted use of FTOrder needs to be supported:
    * Order Operator Restriction: FTOrder may only appear directly succeeding an FTWindow or an FTDistance operator.

- Others
  - *FTAnyallOption*: needs to be supported.
  - *FTUnit* and *FTBigUnit*: Not all the choices of *FTUnit* and *FTBigUnit* need to be supported.
  - *FTIgnoreOption*: needs to be supported.
  - *Scoring*: needs to be supported.

Chapter 3

Initial State

This chapter describes the state of the *MXQuery* Full-Text facility before the work of this Master thesis (see Figure 3.1 for an overview). Section 3.1 describes how the parser was extended to parse full-text XQueries. Section 3.2, 3.3 and 3.4 are about how the document's text is preprocessed into Linguistic tokens, how the inverted list index is built and how the Linguistic tokens can be uniquely identified. Section 3.5 shortly describes what kind of retrieval model is used and the last Section 3.6 is about how full-text queries are evaluated.

3.1 Parser and Runtime

The *MXQuery* parser is extended to parse the full EBNF of XQuery 1.0 and XPath 2.0 Full-Text 1.0 (see Appendix B for the EBNF). *MXQuery* is based on the iterator model[1] [12]. Although the parser was extended to cover the whole XQuery Full-Text, the work of the research project only included keyword and phrase search. To support these kind of queries, *MXQuery* was extended by two additional iterators: the MatchIterator and the FTContainsIterator. These iterators are instantiated while parsing the full-text query.

3.2 Preprocessing of the Documents

The preprocessor contains all the methods to convert the text tokens of *MXQuery* into Linguistic tokens (see Figure 3.2) and to generate the unique identifiers. We used Dewey Numbering (see Section 3.4) as unique identifiers as it simplifies the search for matching words. Each word found during the tokenization is inserted into an inverted list.

3.3 Index

The initial state of the index is an inverted list implemented by a hashtable. A key of the inverted list is a word that is contained in the documents. The value

[1]An iterator is an object that has two primary operations: referencing one particular element in the object collection, i.e. element access, and modifying itself so it points to the next element, i.e. element traversal

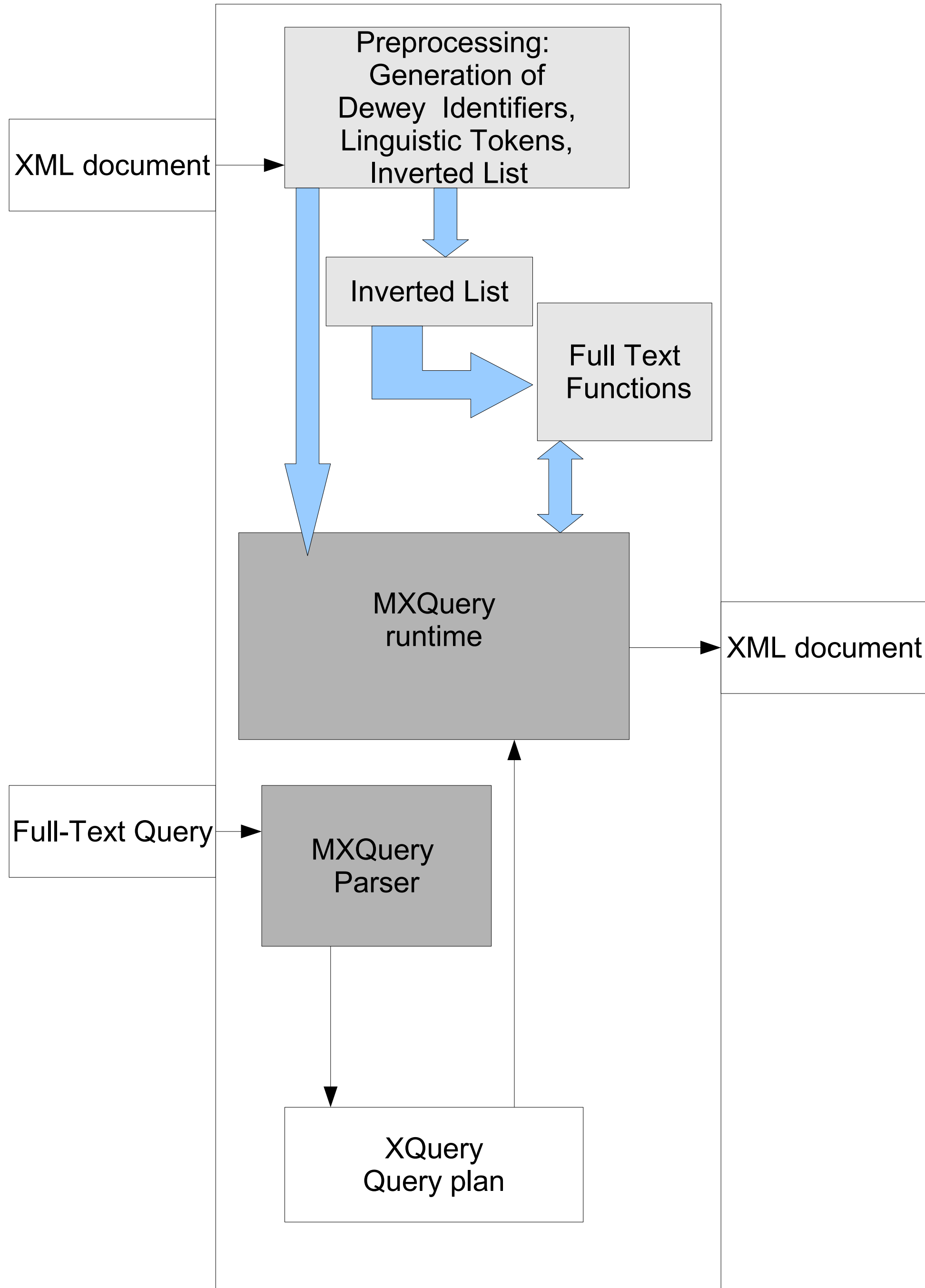


Figure 3.1: Initial State of MXQuery Full-Text Implementation

of this key are all the Linguistic tokens that contain the word. These Linguistic tokens have the following information: the linguistic term, i.e. the word, its relative position in the document, its Dewey identifier and the corresponding XDM token, i.e. the text token containing the word.

3.4 Dewey Numbering

Dewey numbers for XML nodes are generated as follows: Each node is assigned a vector that represents the depth-first node path from the document's root to that node. Text is treated as if it was a child, i.e. a Linguistic token is a child of the element that contains the Linguistic token's text [13]. In addition, the absolute position of the word in the document is appended at the end of the Dewey identifier. This is used for predicates using window or distance that operate over tag boundaries. We only use odd numbers for Dewey identifiers as even numbers are used for updates (see Section 7.3.1 about updates and see Figure 3.2 for an example of Dewey numbering).

3.5 Retrieval Model

The retrieval model of the initial state is the *Boolean Retrieval Model*. Given the nodes that fulfill the structure part of the query, it is checked whether the predicate is fulfilled, i.e. whether the keyword(s) are contained in the nodes. If the keyword(s) are contained in a node, the node is considered relevant, if they are not contained, the node is not relevant, i.e. the node is an *exact match* for the query or not. The problem with this model is its lack of ranking. A node is either relevant or not relevant and the relevant nodes are not ranked according the relevancy to the query, i.e. we do not determine the *best match* result for the query.

From the information retrieval perspective it also suffers from minimal recall: There might be nodes that are close to the relevant result, but are missed as they do not fulfill the query predicate exactly.

3.6 Query evaluation

The Query Plan consists of two levels: On the first level, the Linguistic tokens of the words that match the search term are retrieved. A *Match* is built, if the Dewey identifier of the search context's node is an ancestor of the word's Dewey identifier. This *Match* is then inserted into an *AllMatch*. MatchOptions, e.g. case sensitivity, stemming or use of a thesaurus are handled on this level. These options modify the search words and have the effect that the search term is changed or the search term is expanded to a set of search terms.

On the second level, the *AllMatches* built on the first level are manipulated and filtered, e.g. by proximity selections. If there is no *AllMatch* left after the second level an empty sequence is passed to the FTContains and a boolean token with the value false is generated. If the query has a result, the sequence of the corresponding *AllMatches* is passed to the FTContains and a boolean token with the value true is generated.

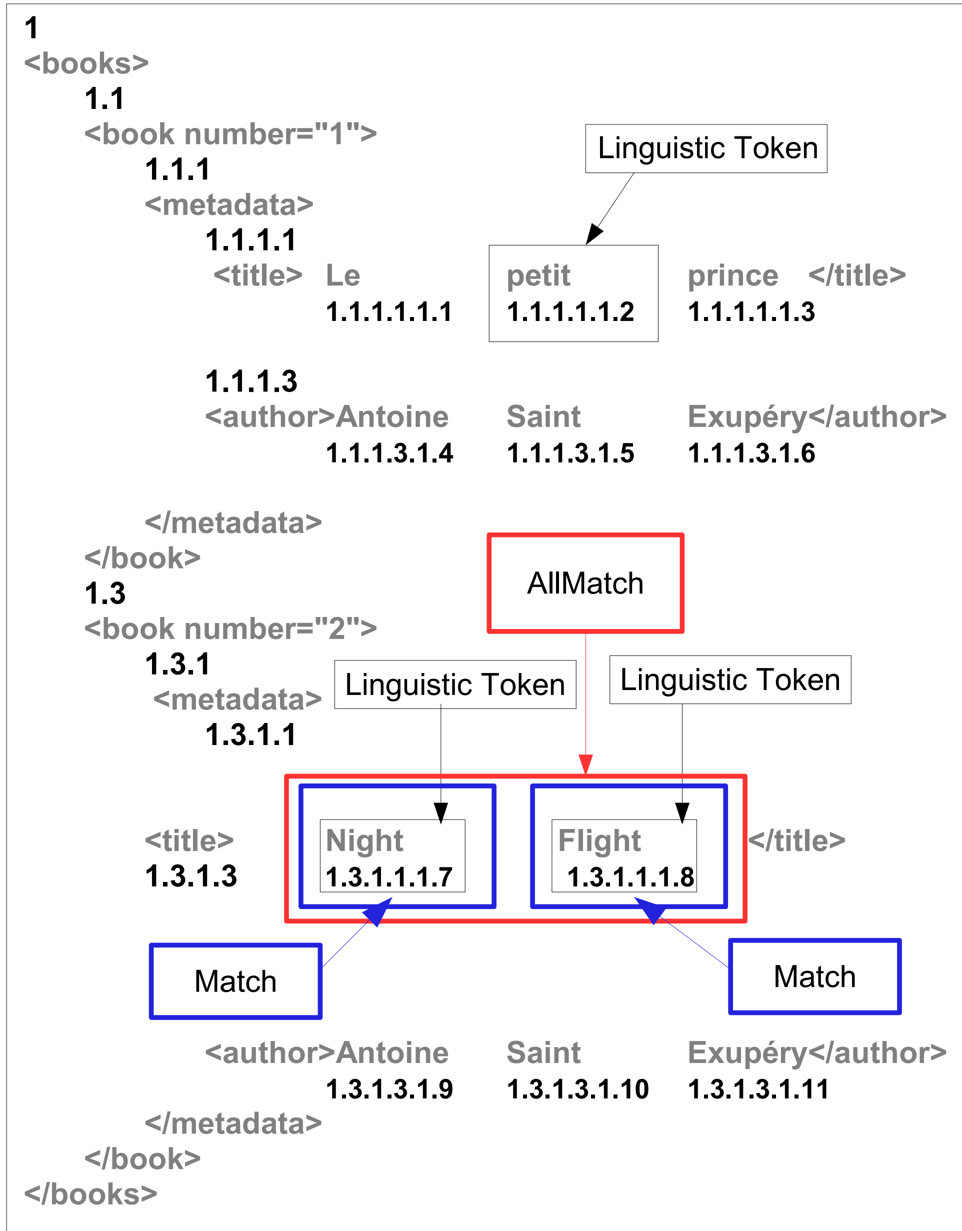


Figure 3.2: XML Document with Dewey Numbering

Example

```
//book/title[ .ftcontains "Night Flight"]
```

returns all the books whose title contain "Night Flight". The queried document, the Matches and AllMatch can be found in Figure 3.2.

Chapter 4

MXQuery Full-Text Design and Implementation

This chapter is about the design and implementation of parts of the Minimal Conformance of the XQuery Full-Text specification [5]. The initial state of *MXQuery Full-Text* has an ad-hoc store, i.e. the XDM token stream is materialized in a vector. The only index on that store is an inverted list. The MatchIterator is the only iterator which has access to the store.

In Section 4.1, we describe the design and requirements of a store that is integrated into *MXQuery*, the MatchOptions, the logical and positional operators and other full-text functions. In Section 4.2, we describe our implementation of the functionalities described in the design section.

4.1 Design and Requirements

This section describes the design approaches and requirements of the store and its access methods, the indexes on the store and operators for *ftor*, *ftand* and *ftnot* to execute the XQuery Full-Text queries. This section is organized as follows: Section 4.1.1 shortly investigates the requirements that the *MXQuery* Full-Text store should fulfill. The other sections describe the semantics and the design of each MatchOption, the logical and positional operators and additional full-text functionalities like "times" and "ignore" predicates. The examples are queries over the document in Appendix A.

4.1.1 Store Requirements

The XQuery Full-Text operators only need read access to the store. They read data from a store in two ways: random and sequential.

Random Id-Based Access and Sequential Scan

Sequential reading is necessary for queries including positional filters and/or the *FTStopword* MatchOption. For these queries we need a method that, given a starting position, returns an iterator that iterates sequentially over the next Linguistic tokens in the store.

Random Value-Based Access

In value-based store accesses, given a set of values for a target item, the matching item id is returned. In our case, the value is usually a string, i.e. a word, and not the matching Dewey identifier is returned, but the Linguistic tokens of the matching words.

4.1.2 MatchOptions

FTLanguageOption

The *FTLanguageOption* specifies the language of search keyword(s). It influences the tokenization, stemming and the stop words and must, therefore, be applied first. With regard to white spaces, tokenization should be more or less the same for Latin languages. For more sophisticated approaches, e.g. using recognition of end of sentences, there is a need for a tokenization approach for each language. For stemming and stop words, we need to provide a stemming algorithm and stop word lists for each language that should be supported.

FTWildcardOption

The *FTWildcardOption* specifies whether the search term contains a wildcard.

It performs a query expansion, i.e. all words fulfilling the wildcards are determined and included in the query. To implement the wildcard option, one can use an n-gram index that contains all the n-grams that occur in the words of a document and its corresponding list of words that contain that n-gram (see 4.2.2 for a detailed description of an n-gram index).

The evaluation of a wildcard query works as follows: The keyword of the query that contains a wildcard is fragmented into its n-grams. For each of the n-grams, we retrieve the list of corresponding words from the n-gram index and intersect them. The words that are left after intersection are possible candidates fulfilling the wildcard query. As the intersection produces false positives, the candidates are checked against the original query and the words fulfilling the original query are returned.

Example 1

```
//book[@number="1"]/title[.ftcontains "Student." with wildcards]
```

returns the title element of the first book because the title element contains "Students".

```
<title>
    Bringing Down the House: How Six Students
    Took Vegas for Millions.
</title>
```

Example 2

```
//book[@number="1"]/title[.ftcontains "Stu.ents" without wildcards]
```

returns no result because the title element of the first book does not contain the word "Stu.ents".

FTThesaurusOption

This MatchOption specifies whether a thesaurus is used or not. The default option is "without thesaurus".

If the option is "with thesaurus", it performs a query expansion, i.e. the words identified by a thesaurus are integrated into the query. A thesaurus can be stored locally or can be implemented by a web service.

The found synonyms are included as new keywords. If one of the words given by the thesaurus is contained, it is a result.

Example

```
//book/summary[.ftcontains "story" with thesaurus at
'http://localhost:8080/axis/WordNetService.jws?wsdl']
```

returns all the books that contain "story" or its synonyms, e.g. "history", in their summary.

```
<summary>
    "Shy, geeky, amiable" MIT grad Kevin Lewis, was, Mezrich learns
    at a party, living a double life winning huge sums of cash in
    Las Vegas casinos. In 1993 when Lewis was 20 years old and
    feeling aimless, he was invited to join the MIT Blackjack Team,
    organized by a former math instructor, who said, "Blackjack is
    beatable." Expanding on the "hi-lo" card-counting techniques
    popularized by Edward Thorp in his 1962 book, Beat the Dealer,
    the MIT group's more advanced team strategies were legal, yet
    frowned upon by casinos. Backed by anonymous investors, team
    members checked into Vegas hotels under assumed names and,
    pretending not to know each other, communicated in the casinos
    with gestures and card-count code words. Taking advantage of
    the statistical nature of blackjack, the team raked in millions
    before casinos caught on and pursued them.
</summary>
```

FTStemOption

The *FTStemOption* specifies whether stemming is applied or not. The default option is "without stemming".

If the option is "with stemming", it performs a query expansion, i.e. all words that have the same stem like the original query term are integrated into the query. By definition of the specification [5], it must be applied before the *FTCaseOption* and before the *FTDiacriticsOption*, because stemming needs the original version of the word to return meaningful results.

The *FTStemOption* can be implemented in the following way: to get the stem for a keyword, we apply the Porter Stemmer [14] and look up the stem in the stem index to retrieve all the words that contain the stem. These words are then looked up in the inverted list to retrieve the corresponding Linguistic tokens.

Example

```
/books/book/title[.ftcontains "Win strategy" with stemming]
```

returns the second book because its title element contains "Winning strategy".

```
<title>
    Beat the Dealer: A Winning Strategy for the Game of Twenty-One
</title>
```

FTCaseOption

The *FTCaseOption* specifies how upper and lower case characters are considered. The default option is "case insensitive". One way to evaluate the case predicate is to index all the words of the text in lower case characters. If the case predicate is "case insensitive", the query term is converted into lower case characters and all the results can be retrieved from the index. In case the predicate is "case sensitive", "lowercase" or "uppercase", all the results are retrieved but need to be post-processed:

- If the case predicate is "case sensitive", the original query term is compared to the original text term. If they are equal, the result is returned.
- If the case predicate is "lowercase", the original query term is converted into lowercase letters and compared to the original text term. If they are equal, the result is returned.
- If the case predicate is "uppercase", the original query term is converted into uppercase letters and compared to the original text term. If they are equal, the result is returned.

Example 1

```
/books/book/summary[. ftcontains 'mit' uppercase]
```

returns the first book's summary because the summary element contains "MIT" in upper case characters (see Figure 4.1 for the operator tree).

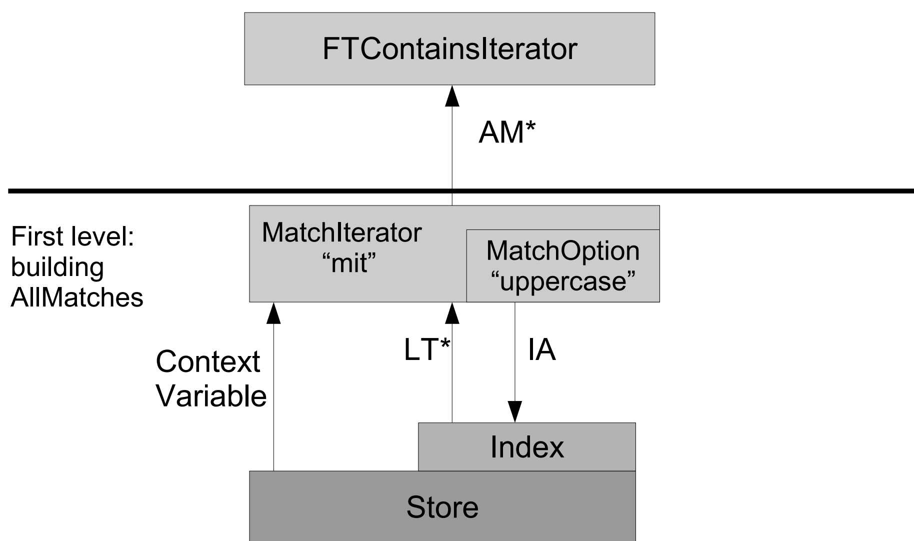
```
<summary>
    "Shy, geeky, amiable" MIT grad Kevin Lewis, was, Mezrich learns
    at a party, living a double life winning huge sums of cash in
    Las Vegas casinos...
</summary>
```

Example 2

```
/books/book/title[. ftcontains 'the wave' case insensitive]
```

returns the title of the third book because it contains "THE WAVE" and case of characters is not considered.

```
<title>THE WAVE. The Classroom is out of Control.</title>
```



LT: Linguistic Tokens
IA : Index Acess
AM: AllMatch

Figure 4.1: Operator Tree of Query with MatchOption

FTDiacriticsOption

FTDiacriticsOption specifies how diacritics are considered: "diacritics sensitive" only matches words in the text if they contain the diacritics as in the query search terms. If the option is "diacritics sensitive", it is a normal keyword or phrase match as the query term(s) and the text term(s) need to be exactly the same.

In case of "diacritics insensitive", e.g. an "é" could also be an "e", we found two possibilities to support this option:

- For every word in the text that contains a diacritic, we could store the version without diacritic, e.g. for the word "résumé" in the text, we also store "resume". The problems of this approach are the following: Especially in languages with a lot of diacritic information, we get a lot of additional tokens in the index. In addition, this approach stores both versions anyway, even if the diacritics option is not used in the query.
- It could be solved by query expansion: We generate each possible version of search terms, e.g. for "resume" we generate "résume", "resumé" and "résumé" and search for all these terms. The problem with this approach is the amount of the newly generated query terms which may make query processing very slow.

Example 1

```
//book/author[. ftcontains "Exupéry" diacritics insensitive]
```

returns the authors of the fourth and fifth books as diacritics are not considered.

```
<author>Antoine de Saint Exupéry</author>
<author>Antoine de Saint Exupery</author>
```

Example 2

```
//book/author[. ftcontains "Exupéry" diacritics sensitive]
```

returns only the author of the forth book as diacritics are considered.

```
<author>Antoine de Saint Exupéry</author>
```

FTStopWordOption

The *FTStopWordOption* specifies whether stop words are used or not. Being a stop word is only relevant to query terms, not to document terms. The default is "without stop words".

The following two approaches are possible ways to handle stop words:

1. Filter out the stop words while creating the inverted list index. Defining a list of stop words that should not be used has no influence.
2. If the query phrase contains stop words and the option is "with stop words", the stop word are excluded from the set of search terms (any term can be substituted for it) and their position is stored. For the rest of the search terms we need to check whether they fulfill:

- The positional information of the rest of the terms is correct:

  Example

  ```
  /book[ftcontains "Professor of Mathematics"
   with stop words {of}]
  ```

  "Professor" and "Mathematics" are in distance 2 of each other.

- At the positions of the stop words are any other terms than an excluded stop word.

  Example

  ```
  /book[ftcontains "Professor of Mathematics"
   with stop words {of}]
  ```

  there is any other word than "of" between "Professor" and "Mathematics".

For the second approach we need an index that returns for a given Dewey identifier the corresponding Linguistic token. This can be implemented by a B+ tree on the Dewey identifiers with the Linguistic tokens as leaves. We can search in the B+ tree for the Linguistic token at the given position and check whether its word is none of the excluded stop words.

Example

```
//book/title[. ftcontains "Game of Twenty-one"
with stop words {of}]
```

returns no result as there is only a title containing "Game of Twenty-one" including the stop word "of".

FTExtensionOption

The *FTExtensionOption* specifies a MatchOption that acts in an implementation-defined way. Depending on what the option does, it needs to be applied at a particular point in the query.

4.1.3 FTMildNot, FTUnaryNot, FTOr and FTAnd

This section describes the input and output and the semantics of the Logical Full-Text operators. The examples are queries on the document that can be found in Appendix A.

FTMildNot

The *FTMildNotOperator* takes two sequences of *AllMatch* as input. We refer to them as *leftInput* and *rightInput*. Let's say we have the following query:

Example

```
/book/summary[. ftcontains "Blackjack" not in "Blackjack Team"]
```

All books that contain "Blackjack" without neighboring "Team" in their summary are returned. There are the following possibilities:

- If the *leftInput* is empty, the output is the empty sequence.
- If only the *rightInput* is empty, the output is the *AllMatch* of the *leftInput*.
- In case both sequences are non-empty, we need to check for each *AllMatch* of the *leftInput* whether its match(es) are contained in the *rightInput AllMatch* by comparing their Dewey identifiers. If they are the same, the *AllMatch* is not a result, otherwise it is part of the output.

The result of the query above is:

```
<summary>
    "Shy, geeky, amiable" MIT grad Kevin Lewis, was, Mezrich learns
    at a party, living a double life winning huge sums of cash in
    Las Vegas casinos...
</summary>
```

FTUnaryNot

The *FTUnaryNotOperator* is used in queries that contain an *ftnot* directly followed by a *FTWords*.

Example

```
//book/author[. ftcontains ftnot "Edward Thorp"]
```

returns all author elements that do not contain Edward Thorp.

```
<author>Ben Mezrich</author>
<author>Morton Rhue</author>
<author>Georg Orwell</author>
<author>Antoine de Saint Exupéry</author>
<author>Antoine de Saint Exupery</author>
```

The evaluation works as follows: If the *FTUnaryNotOperator* receives an *AllMatch*, it returns the empty sequence such that the *FTContainsOperator* creates a boolean token with value false. If the *FTUnaryNotOperator* receives the empty sequence, it creates a dummy *AllMatch* such that the *FTContainsOperator* creates a boolean token with value true. In case the *ftnot* is part of a more complex *FTSelection*, we need more sophisticated ways to evaluate the query.

FTOr

The *FTOrOperator* takes two or more sequences of *AllMatch* as input. The semantics of *FTOr* states that one of the FTSelections need to be fulfilled.

Example

```
//book/author[. ftcontains "Edward Thorp" ftor "Ben Mezrich"]
```

returns all author elements that either contain "Edward Thorp" or "Ben Mezrich" or both.

```
<author>Ben Mezrich</author>
<author>Edward Thorp</author>
```

The evaluation works as follows: If all sequences are empty, the empty sequence is output. If one or more sequences contain *AllMatches*, all *AllMatches* are unified into one *SuperAllMatch*. In case this *SuperAllMatch* is further processed, every operator needs to unnest the *AllMatches* to get access to the matches and their Linguistic tokens (Figure 4.2 depicts the operator tree). The *FTOr* is commutative and associative in all cases, i.e. it makes no difference in which order the sequences are evaluated. As either one of the *FTSelection* needs to be satisfied, it makes no sense to constrain the order.

FTAnd

The *FTAndOperator* takes two or more sequences of *AllMatches* as input. The semantics of the *FTAnd* states that all the FTSelections need to be fulfilled

Example

```
//book/author[. ftcontains "Edward" ftand "Thorp"]
```

returns all author elements that contain "Edward" and "Thorp".

```
<author>Edward Thorp</author>
```

The evaluation works as follows: If one of the sequences is empty, i.e. one of the input FTSelection is not fulfilled, the empty sequence is output. If all sequences contain *AllMatch*, we apply the *Cartesian product*: We combine each *AllMatch* of an input with each *AllMatch* of the other inputs and generate new *AllMatch* containing the matches of these inputs.

The *FTAnd* is commutative in most cases as the default order constraint is "unordered", i.e. the input order can be exchanged. In case an *FTOrder* filter is defined in the query:

Example

```
//book/author[. ftcontains "Edward" ftand "Thorp" ordered]
```

the search term of the left input "Edward" and the search term of the right input "Thorp" need to be matched in the text in the same order as defined in the query.

FTAnd is associative in most cases:

Example

```
//book/summary[ .ftcontains "Edward" ftand "Thorp" ftand
"Beat the dealer" window 10 words]
```

it makes no difference if we first compute an *ftand* of "Edward" and "Thorp" and an *ftand* of its result with "Beat the dealer" or if we compute an *ftand* of "Thorp" and "Beat the dealer" and then with its result an *ftand* with "Edward". But in case of negation like in the query:

Example

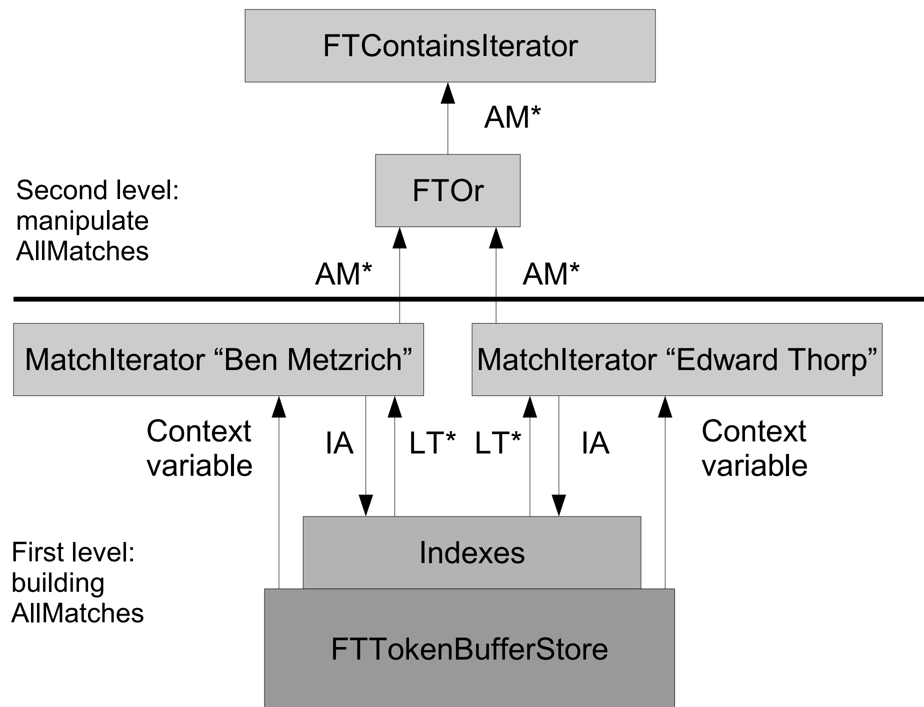


Figure 4.2: Operator Tree of FTOr Query

```
//book[ .ftcontains "blackjack" ftand "team" ftand
ftnot "blackjack team"]
```

the *FTAnd* is not associative. The semantics of the query is to return only books that contain "blackjack" and "team", but not if these terms occur next to each other. If we first perform the *FTAnd* of "blackjack" and "team", we get books that contain both terms. Then we perform an *ftand* with "blackjack team" and get all the books that contain "blackjack" and "team" but not next to each other. If we group in the following way:

```
//book[ .ftcontains ("blackjack" ftand ("team" ftand
ftnot "blackjack team")]
```

we first get all the books that contain "team", but not if they have a preceding "blackjack". If we then perform an *ftand* with "blackjack", we get all the books that contain "blackjack" and "team" and also the ones that contain "blackjack team". The information that we do not want books containing "blackjack team" is lost.

In case the right input contains a preceding *ftnot*, the *FTAndOperator* processes the *AllMatch* differently:

Example

```
//book[ .ftcontains "blackjack" ftand ftnot "roulette"]
```

If we evaluate this query in the normal *FTAnd* way, the empty sequence is output: If a book only contains "blackjack" and no "roulette" is found, the left input of the *FTAnd* is an *AllMatch* for the found "blackjack" and the right input is the empty sequence, so the empty sequence is output, although the book containing "blackjack" is a result. The semantics of the *FTAndNot* is the following: An *AllMatch* is only returned if the *ftnot-FTSelection* is not satisfied. We have the following possibilities:

- If both input sequence are empty, the empty sequence is output.
- If the left input is an empty sequence, the empty sequence is returned.
- If the right input is an empty sequence, the *FTAndOperator* outputs the *AllMatch* of the left input.
- In case both input sequences contain *AllMatch*
  - In the simple cases, i.e. the left input and the right input are *FT-Words*: The empty sequence is output as the right input is satisfied.
  - In the more complex cases, i.e. left input and the right input contain e.g. positional filters:

    Example

    ```
    /book[. ftcontains "blackjack" ftand ftnot "team"
    window 3 words]
    ```

> One way is to evaluate the *ftand* as if there is no *ftnot* and evaluate the *ftnot* with the positional filter: First we find all "blackjack-team" combinations. If a result is in window 3 words, the empty sequence is output. If a result is more than 3 words apart, the *AllMatch* are output. The *ftnot* is not evaluated with the *ftand*, but further up in the operator tree.

The specification does not define a binding order for the logical operators. If there are no parentheses that clarify the evaluation, we apply the logical operators from left to right.

4.1.4 FTOrder, FTWindow, FTDistance, FTScope, FTContent, FTRange

This section describes the in- and output and the semantics of the positional operators. The example queries are all on the sample document that can be found in Appendix A.

FTOrder

The *FTOrder* operator takes one sequence of *AllMatch* as input. The resulting *AllMatches* contain matches for which the word positions of the Linguistic tokens are in the same order as the search terms in the query. The *FTOrder* checks whether the positional information of the Linguistic tokens in the matches of the *AllMatch* is in ascending order. If they are in ascending order, the *AllMatch* is output. In case of an empty input sequence or an *AllMatch* with matches in wrong order, the empty sequence is output.

Example

```
//book/summary[ .ftcontains "blackjack" ftand "team" ordered]
```

returns all the books whose summary contain "blackjack" and "team" in this order.

```
<summary>
    "Shy, geeky, amiable" MIT grad Kevin Lewis, was, Mezrich learns
    at a party, living a double life winning huge sums of cash in
    Las Vegas casinos...
</summary>
```

FTWindow

The *FTWindowOperator* takes one sequence of *AllMatch*, one or two *AdditiveExpr*, i.e. one or two XDM Long token, and a *FTUnit* as input. The *AdditiveExpr* determines how many units (defined by *FTUnit*) the matches need to be within. The *FTUnit* is either "words", "sentences" or "paragraphs". The *AllMatch* that is returned contains matches that satisfy the window constraint,i.e. there exists a sequence of the specified number of subsequent ("words", "sentences" or "paragraphs") positions, such that all the Linguistic tokens are within that window. If the *FTUnit* is "words", the *FTWindowOperator* needs to check for the matches with the highest and lowest position whether they fulfill the

window constraint. In case of "sentences" or "paragraph", the *FTWindowOperator* needs to check the two matches with the highest and lowest "sentence" or "paragraph" information. If this constraint is fulfilled, the operator outputs the *AllMatch*, otherwise it outputs the empty sequence. In case of an empty input sequence, the empty sequence is output.

Example

```
//book/summary[ .ftcontains "Professor of Mathematics" ftand
"Professor Thorp" window 3 paragraphs]
```

returns all the books that contain a paragraph with "Professor of Mathematics" and a paragraph with "Professor Thorp" and these two paragraphs are in a window of three paragraphs.

```
<summary>
    <p>
        Ever since the time of Cardano, mathematicians have been
        delving into the theory of games of chance, but rarely
        with the stunning success achieved by Edward Thorp,
        Professor of Mathematics at the University of
        California at Irvine.
    </p>
    ...
    ...
    <p>
        Now the new revised point count system shows
        how the player can win in spite of
        present or future rule changes in Las Vegas,
        Atlantic City and Puerto Rico; how to win
        in spite of cheating by casinos. The cars in the
        book can be used in actual casino play.
  </p>
</summary>
```

FTDistance

The *FTDistanceOperator* takes a sequence of *AllMatch*, an *FTRange* (see Section 4.1.4) and a *FTUnit* as input. The *FTRange* determines whether the distance needs to be "exactly", "at least", "at most" or "from i to j" as long as the *AdditiveExpr* defines. The *FTUnit* determines whether the distance is measured in "words", "sentences" or "paragraphs". The *AllMatch* that is returned contains matches that satisfy the distance constraint, i.e. the distance for every pair of matches is within the specified interval of "words", "sentences" or "paragraphs" from the end of the preceding Linguistic token to the start of the next. The *FTDistanceOperator* checks for each pair of matches whether the distance constraint is fulfilled. If it is, the *AllMatch* is a result and is output. In case the constraint is not fulfilled or the input sequence is empty, the empty sequence is output.

Example

```
//book/summary[ .ftcontains "Professor of Mathematics" ftand
"Professor Thorp" distance at least 4 paragraphs]
```

returns all the books that contain a paragraph with "Professor of Mathematics" and a paragraph with "Professor Thorp" and these two paragraphs are in distance four or more paragraphs. In our example, the empty result is returned as phrase "Professor of Mathematics" and phrase "Professor Thorp" occur in distance one and two paragraphs from each other (see Figure 4.3 for the operator tree).

FTScope

The *FTScopeOperator* takes one sequence of *AllMatch* and a *FTBigUnit* as input which is either the string "sentence" or the string "paragraph". An *AllMatch* returned by the scope "same sentence" contains those matches whose Linguistic tokens span only a single sentence and all span the same sentence, hence an *AllMatch* returned by the scope "different sentence" contains those matches whose Linguistic tokens do not have the same sentence information. The semantics of "same paragraph" is similar to "same sentence", the only difference is that the paragraph information needs to be checked. The semantics of "different paragraph" is similar to "different sentence". The *FTScopeOperator* checks for each *AllMatch* whether the matches fulfill the given constraint. If yes, the *AllMatch* is output, otherwise the empty sequence is output. In case the input sequence is empty, the empty sequence is output.

Example

```
//book/summary[ .ftcontains "Professor of Mathematics" ftand
"Edward Thorp" different sentence]
```

returns all books that contain "Professor of Mathematics" and "Edward Thorp" in different sentences. In this case, the empty result is returned as "Professor of Mathematics" and "Edward Thorp" only occur in the same sentence.

FTContent

The *FTContentOperator* takes one sequence of *AllMatch* and one of the postfix operators "at start", "at end" or "entire content" as input. An *AllMatch* returned by the content "at start" is an *AllMatch* whose matches contain Linguistic tokens that match the first tokens of the current search context node. An *AllMatch* returned by the content "at end" is an *AllMatch* whose matches contains Linguistic tokens that match the last tokens of the current search context node. If the postfix operator is "entire content", the query is evaluated as if it was *distance exactly 0 words at start at end*. This means that the Linguistic tokens in the match need to match every token in the content of the current search context node.

To include the *FTContent* functionality into *MXQuery* we need to expand the Linguistic tokens. If a Linguistic token is the first term of an element, we need to store this information, e.g. by having an *isStartingToken* boolean flag. If a Linguistic token is the last term of an element, we also need to store this information e.g. again by having an *isEndingToken* boolean flag. When we

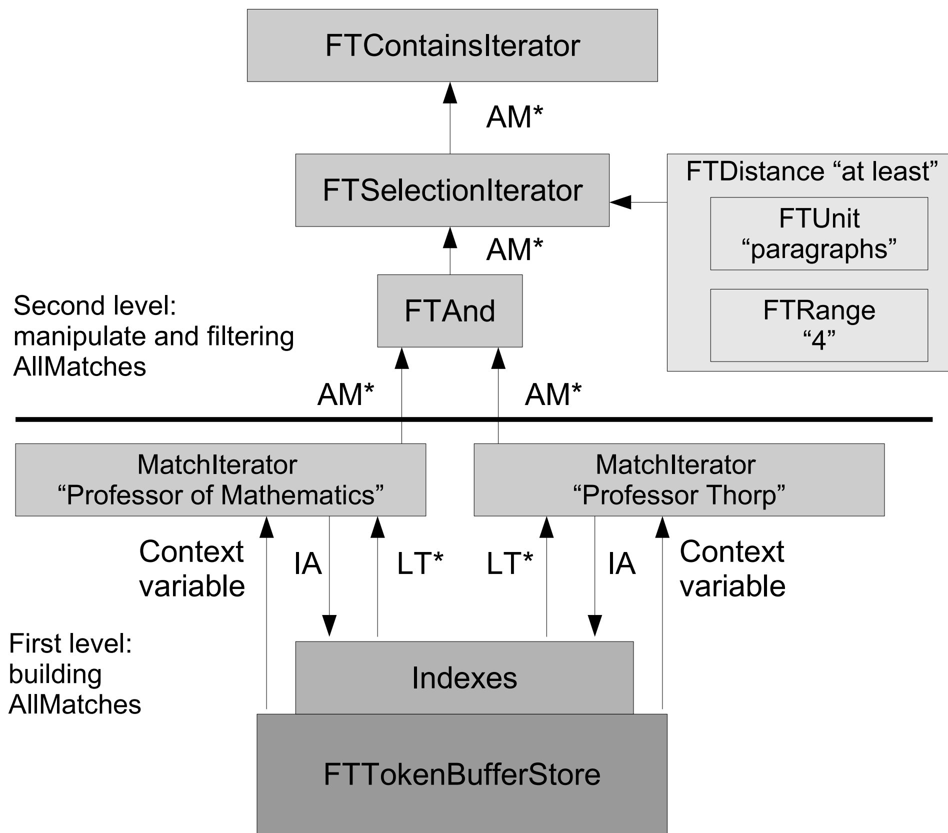


Figure 4.3: Operator Tree of FTDistance Query

have the postfix operator "at start" in the query, the *FTContentOperator* needs to check for an *AllMatch* whether the first match's Linguistic token is a starting token and whether the following matches' Linguistic tokens have positional information in ascending and subsequent order. If the postfix operator is "at end", the *FTContentOperator* needs to check whether the last match's Linguistic token is an ending token and whether the other matches are in descending, subsequent order with regard to the ending token. In case the postfix operator is "entire content", the *FTContentOperator* needs to check whether the first token in the *AllMatch* is a starting token and the last token is an ending token. In case the input sequence is an empty sequence, the empty sequence is output.

Example

```
//book/title[ .ftcontains "Beat the Dealer" at start]
```

returns all titles that contain "Beat the Dealer" at the beginning of their title element.

```
<title>
    Beat the Dealer: A Winning Strategy for the Game of Twenty-One
</title>
```

FTRange

The *FTRangeOperator* takes one or two *AdditiveExpr*, i.e. one or two XDM token and a string having one of the following values: "at least", "at most", "exactly" or "from i to j" as input and passes them as output. The *FTRange* is an input for the *FTDistanceOperator* and the *FTTimesOperator*.

Multiple Nested Positional Filters

To process queries that contain nested positional filters, e.g. a distance selection inside another distance selection, it makes sense to create an *SuperAllMatch* before evaluating the second distance selection. The semantics of the second distance selection enforces a constraint on the number of words between the last Linguistic token of the first *FTDistance* selection and the first Linguistic token of the second *FTDistance* selection.

Example

```
/books/book[ .ftcontains ((("Ben" ftand "Metzrich") distance
at most 2 words) ftand (("Edward" ftand "Thorp") distance
at most 2 words) distance at least 20 words)]/metadata
```

returns all books' metadata that contain "Ben Metzrich" and "Edward Thorp" in distance at least twenty words of each other.

```
<metadata>
    <title>
        Bringing Down the House:
        How Six Students Took Vegas for Millions.
    </title>
    <author>Ben Mezrich</author>
</metadata>
```

The inner distance constraints are evaluated as described above. The *FTDistanceOperator* sets a flag in the resulting *AllMatch* that there was a distance selection already. The following *FTAnd* does not create an *AllMatch* containing all the matches of both input *AllMatches*, but creates *SuperAllMatch* by applying the *Cartesian Product* on the *AllMatch*. When evaluating the outer *FTDistance*, the *FTDistanceOperator* checks whether the constraint is fulfilled for the last Linguistic token of the *AllMatch* with the smaller Dewey identifiers and for the first Linguistic token of the *AllMatch* with the bigger Dewey identifiers. This approach can be used for any positional filters besides *FT-Distance*. In case of deeper nesting of positional filters, the approach can be used in the same way, i.e. by creating a *SuperSuperAllMatch*;an *AllMatch* of *SuperAllMatch*. To check the positional constraint on this *SuperSuperAllMatch* we need to have a *unnest* method that works for any level of nesting.

4.1.5 FTTimes

This section describes the input and output and the semantics of the Cardinality operator. The *FTTimesOperator* takes a sequence of *AllMatches* and a *FTRange* expression as input. An *AllMatch* that is returned by the times constraint

- "occurs at least N times", is an *AllMatch* that contains at least N different matches of an FTSelection
- "occurs at most N times", is an *AllMatch* that contains at most N different matches of an FTSelection
- "occurs exactly N times", is an *AllMatch* that contains exactly N different matches of an FTSelection
- "occurs from i to j times", is an *AllMatch* that contains between i and j different matches of an FTSelection

The evaluation works as follows: The *AllMatches* in the sequence are counted and it is checked whether the times constraint is fulfilled. If it is, all the matches of the *AllMatches* are unified and the newly created *AllMatch* is returned, otherwise the output is the empty sequence. In case of an empty input sequence, the empty sequence is output.

Example

```
/books/book[ .ftcontains 'MIT' at least 2 times]/metadata
```

returns all books' metadata that contain "MIT" at least three times.

```
<metadata>
    <title>
        Bringing Down the House:
        How Six Students Took Vegas for Millions.
    </title>
    <author>Ben Mezrich</author>
</metadata>
```

4.1.6 FTIgnoreOption

Using the *FTIgnoreOption*, it is possible to modify which parts of the XML structure are available for a single match of FTSelection [15]. The *FTIgnoreOption* can only be specified on the top-level FTSelection and is an optional extension to the *FTContainsExpr*: *EvaluationContext[.ftcontains FTSelection without content IgnoreExpr]*

The *IgnoreExpr* following *without content* specifies a sequence of nodes, whose text should be ignored when searching the search context nodes. For example the following query searches for books that contain "Professor of Mathematics" but not in their p element.

Example

```
//book[. ftcontains "Professor of Mathematics"
without content .//p]
```

returns the empty result as "Professor of Mathematics" only occurs in a paragraph element.

There are three aspects to this exclusion of nodes from the search context:

- When the phrase "Professor of Mathematics" appears in a p element (or descendant element of the p element), it should not be found
- When eliminating paragraphs, the distances of terms in the remaining text are affected, i.e. when ignoring a paragraph that stands between "Professor of" and "Mathematics":

  Example

  ```
  <profession>Professor of<p>Discrete</p>
  Mathematics</profession>
  ```

  the terms become adjacent and can be matched as a single phrase [15].
- When eliminating paragraphs, the sentence information of terms in the remaining text is also affected, i.e. when ignoring a paragraph that stands between "Professor of" and "Mathematics":

  Example

  ```
  <profession>Professor of<p>Statistics.
  He is also chair of</p>Mathematics</profession>
  ```

  the terms which fall into the same sentence now, were in different sentences before.

The *FTIgnore* can be handled by the MatchIterator. If an *IgnoreExpr* is defined, the MatchIterator receives an additional XDM token that contains the ignore context. The MatchIterator compares the Dewey identifier of the search context XDM token with the Dewey identifier of the ignore context XDM token. If the search context XDM token's Dewey identifier is equal or a descendant of the

ignore context's Dewey identifier, no *AllMatch* is generated. If it is not equal or descendant, an *AllMatch* is output.

The aspects that affect the positional and sentence information can be solved in different ways:

- The ignoring of nodes can be handled like an update such as a deletion. Depending on how the deletion is handled (see 7.3.1), we can handle the ignore update in the following ways:
  - Positional information is available in the Linguistic tokens and we use a delta index approach: deletion of the nodes triggers a deletion and re-insertion of the document into a delta index.
  - No positional information is available in the Linguistic tokens: The deletion has no influence. If we find a "Professor of" match, we scan the next token whether it is a "Mathematics" match.

  For the sentence information we could have a similar approach: If it is contained in the Linguistic tokens, we could again use a delta index approach, otherwise we have to scan the tokens from "Professor of" to "Mathematics" and check whether there is a full stop in between.

- Another approach could be a hybrid of the approaches described above: We use positional and sentence information, but ignore it when an ignore option is contained in the query. We then use the scanning approach as we cannot rely on the positional and sentence information anymore.

4.1.7 FTAnyAllOption

FTWords defines the search terms or phrases that need to be matched in the text. It consists of two parts: a mandatory *FTWordsValue* part and an optional *FTAnyallOption* part. If the query contains an *FTAnyallOption*, the query can be evaluated in the following way:

- {*word1, word2, ...*} *any word*: We check for every word whether it is contained. If one or more is contained, an *AllMatch* is built containing matches for the word(s). If none is contained, the empty sequence is returned.

- {*word1, word2, ...*} *all word*: We check whether all the words are contained. If they are, an *AllMatch* containing all the matches is returned (as if it was a phrase). If not all words are contained, the empty sequence is returned.

- {*word1, word2, ...*} *phrase*: We check whether all the words, together as a phrase, are contained. If they are, an *AllMatch* containing the phrase is returned. If the phrase is not contained, the empty sequence is returned.

- {*phrase1, phrase2, ...*} *any*: We check whether any of the phrases is contained. If one or more are contained, an *AllMatch* is built containing matches for the phrase(s). If none is contained, the empty sequence is returned.

AnyallOption	*Rewriting*
{*word1, word2, ...*} *any word*	*word1 ftor word2 ftor ...*
{*word1, word2, ...*} *all word*	*word1 ftand word2 ftand ...*
{*word1, word2, ...*} *phrase*	*(word1 word2 ...) ftor (word2 word1 ...) ftor ...*
{*phrase1, phrase2, ...*} *any*	*phrase1 ftor phrase2 ftor ...*
{*phrase1, phrase2, ...*} *all*	*phrase1 ftand phrase2 ftand ...*

Table 4.1: Rewriting of AnyallOptions

- {*phrase1, phrase2, ...*} *all*: We check whether all of the phrases are contained. If they are, an *AllMatch* containing all the matches for the phrases is returned. If not all phrases are contained, the empty sequence is returned.

Another way to evaluate the *FTAnyallOption*, is to perform a rewriting and to evaluate the rewritten query (see Table 4.1).

4.2 Implementation

This section describes the *MXQuery* implementation of the indexes, the store and operators to execute the XQuery Full-Text queries (see Figure 4.4 for an overview). Section 4.2.1 describes the improved tokenization algorithm, Section 4.2.2 is about the indexes on the store and Section 4.2.3 describes the implementation of the store and its access methods. Section 4.2.4 is about the implementation of the iterators for *ftor*, *ftand* and *ftnot* and the extension of the *MatchIterator* to process the MatchOptions.

4.2.1 Processing of Documents

The *LinguisticTokenGenerator* is used by the full-text store to generate Linguistic tokens out of XDM text tokens. The text token's text is tokenized and for each of the words, a new Linguistic token is generated. To process queries containing scope predicates, i.e. same or different sentence or paragraph, respectively, we added a very simple tokenization algorithm inspired by [16]. Whenever there is the pattern:

"Word beginning with a lower case letter followed by a full stop, question or exclamation mark, followed by a word beginning with a capital letter"

a new sentence is found. This algorithm identifies: "This is a simple tokenization algorithm. It identifies sentences." as two sentences, but "This is a simple tokenization algorithm by G. Grefenstette." as one sentence, although there are two "." in the sentence. This pattern identifies most of the "." correctly either as full stop or as abbreviation point.

One problem are abbreviations that are written in lower case, e.g. "He is a techn. Software Engineer.". The point after "techn" is identified as a full stop although it is an abbreviation point. To identify this case, we implemented a trie on all abbreviations of the English language. In case we identify the pattern above, we check whether the last word could be an abbreviation by looking it

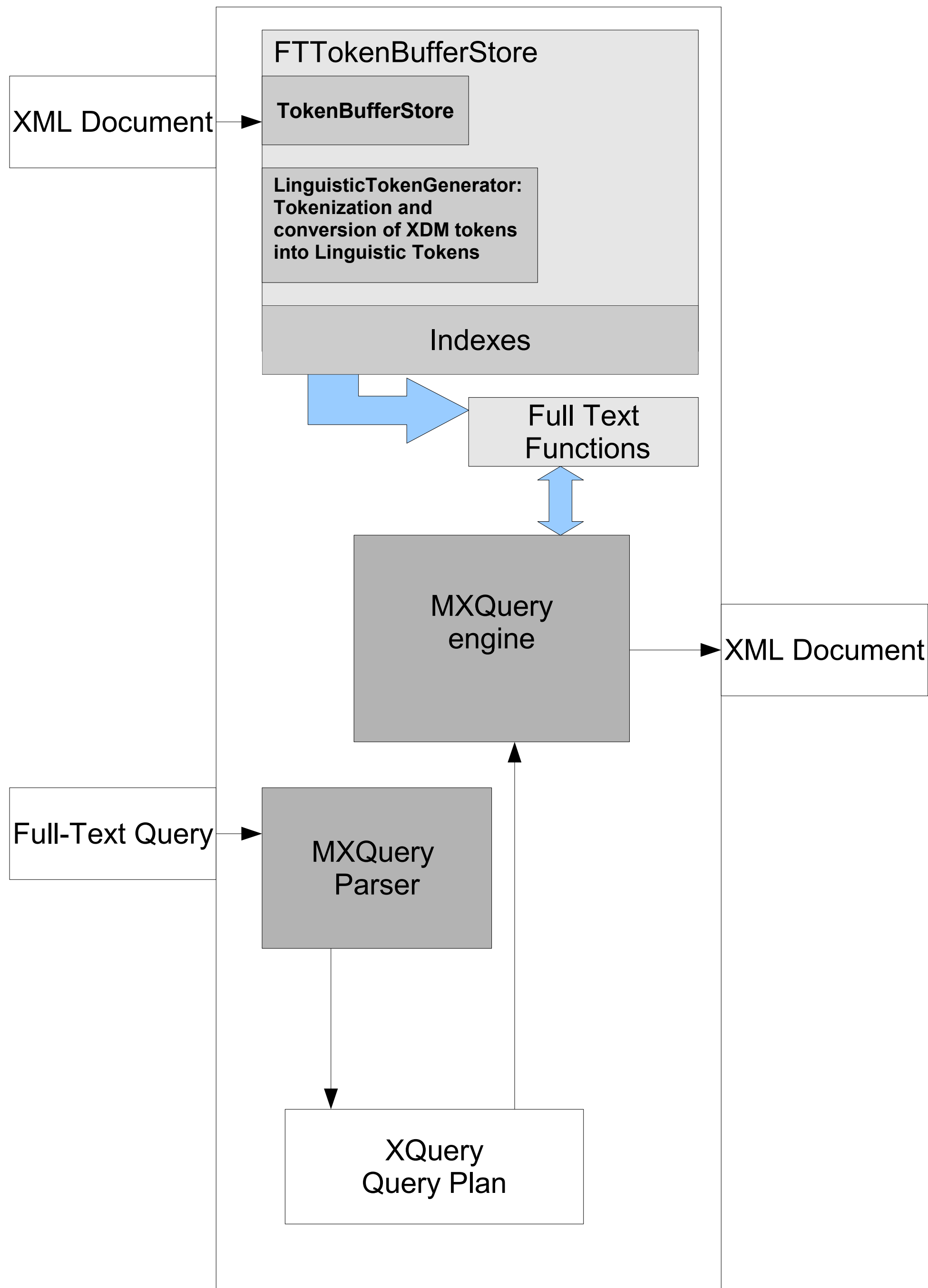


Figure 4.4: Overview of MXQuery Full-Text Implementation

up in the trie. If it is found, we identify it as an abbreviation. The sentence information, i.e. the sentence number the word is belonging to, is stored in the corresponding Linguistic token.

Paragraph information is only considered in elements that have element name *p*, i.e.

```
<p>This is a paragraph</p>
```

The Linguistic tokens of this paragraph contain paragraph information, i.e. the paragraph number the corresponding word is contained in.

In addition, the Linguistic token of the initial implementation is extended by two fields that indicate whether the token is a starting token, i.e. the Linguistic token of the first word in a text token, or an ending token, i.e. the Linguistic token of the last word in a text token. These fields can be used to evaluate the *FTContent* predicate. The word itself is not stored in the Linguistic token as a string, but two offsets indicating the start position and the end position of the word in the text token's text.

4.2.2 Indexes

This section describes the indexes that are supported by the implementation:

Inverted List

The inverted list is implemented by a hashtable. The key of an entry is a word, converted to lower case characters, contained in the document. Its value is the list of the corresponding Linguistic tokens. With the inverted list, it can be checked whether a word is contained in the document. If it is not contained in the hashtable, the word is not contained in the document. If the word is contained in the hashtable, it can be checked at which positions the word occurs by retrieving the list of its corresponding Linguistic tokens. To query the positions, i.e. the Linguistic tokens of a word, we implemented a function *getLinguisticTokensExact* that takes the word as input and returns a LinguisticTokenIterator over all the corresponding Linguistic tokens.

N-Gram Index

An n-gram is a sequence of n characters of a word, e.g. the 3-grams for castle are $ca, cas, ast, stl, tle, le$, the special character $ is used to denote the beginning and end of a word [17]. In an n-gram index the dictionary contains all grams of size 2 to n, that occur in any term in the vocabulary. Each n-gram points to the list of vocabulary terms that contain that n-gram. The n-gram index is implemented by a hashtable. The key of a hashtable's entry is an n-gram of a word that is contained in the document. Its value is the list of the words that contain that n-gram. With the n-gram index, we can evaluate queries containing wildcards (see Section 4.1.2).

Nextword Index

The nextword index is used to speed up phrase search [18]. It is implemented by a hashtable. The key of an entry is one of the n most frequent words (firstwords)

contained in the document. The value of the entry is a list of pairs of Linguistic tokens. These pairs contain one of the corresponding Linguistic token of one of the n most frequent words and the Linguistic token next to it, i.e. the Linguistic token of the word next to the common word (see Figure 4.5). With the nextword index, we can evaluate phrase queries faster than with the conventional inverted list.

If a phrase query contains a common word, we do not have to check all the common word's occurrences, but can check whether one of its nextwords is equal to the next word in the phrase. If one of the nextwords is equal to the next word in the phrase, we already have its Linguistic token and do not have to retrieve it in a second step. If every word in the text documents is used as *firstword*, the size of the *nextword index* is of approximately twice that of an inverted list. As much of the speed improvement is for phrase queries including non-rare words, it is proposed to only use common words as *firstwords*, i.e. we use the three most common words as *firstwords*. Hence, we use a combination of an inverted list on rare words and a nextword index on common words.

B+ tree

The B+ tree implementation is taken from *iMeMex* [19]. To make it compatible with the CLDC[1] API [20] we needed to make some adaptations[2]. The B+ tree is used internally for the nextword index to find a word's neighboring Linguistic token. It is also used for the FTStopWordOption (see 4.1.2).

The main classes of the B+ tree implementation are the following:

BTree This class is the BTree implementation. The most important method is *bulkload* which bulk loads an empty tree. Bulkloading is a simple algorithm to create an index based on a B+ tree very quickly. The algorithm works as follows: As input we have a sequence of Linguistic tokens that is sorted according to the DeweyIdentifiers. While the sequence is not empty, the Linguistic tokens are inserted into new leaves according to the B+ tree invariant (see survey [21]) and in the parent node we put a pointer to the leaf. In case the parent node does not exist, it is created recursively. The advantage of this algorithm is the absence of node and leaf splits as it creates a B+ tree from left to right and from bottom to top (see Figure 4.6).

BTreeNode The BTreeNode is the interface of a node in the tree. Its implementations, leaf and internal node, implement a method *bulkAdd* that adds a data pair, Dewey identifier/Linguistic token for leaf and Dewey identifier/pointer to the child BTreeNode for internal nodes respectively.

Stem Index

The stem index is implemented by a hashtable. Keys of the entries are the stems of the words that are contained in the document. The values of the entries are

[1] A specification of a framework for Java ME applications targeted at devices with very limited resources such as pagers and mobile phones.

[2] MXQuery is CLDC conform so that it can be run on a portable device

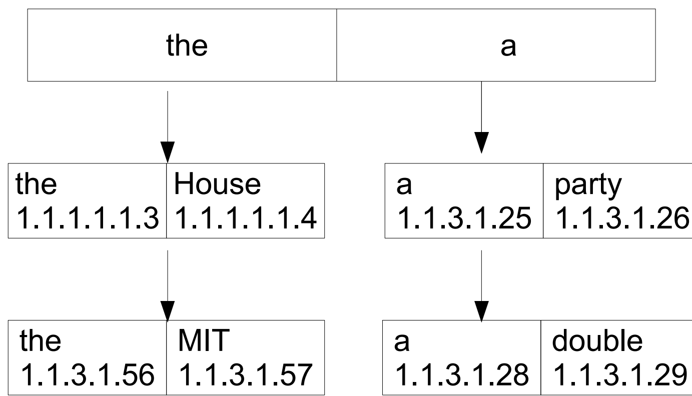


Figure 4.5: Nextword Index for Two Most Common Words

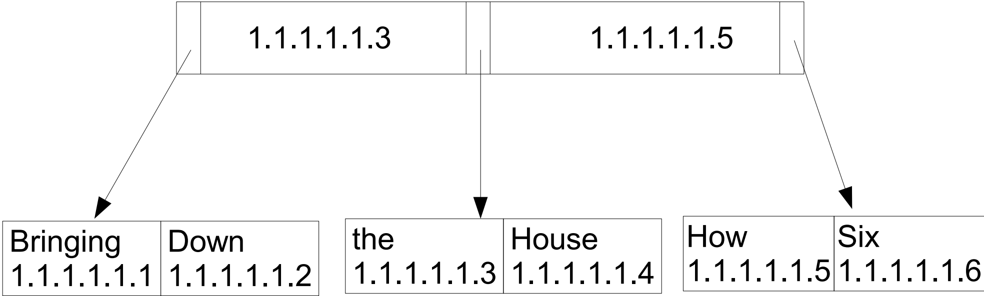


Figure 4.6: B+ tree with Linguistic Tokens in Leafs

lists of words which have that stem. With the stem index, we can evaluate queries containing the *FTStemOption* (see 4.1.2).

4.2.3 Store

The implementation of the full-text store is an extension of the *TokenBuffer* of the SMS storage [22]. This store materializes the token stream into a buffer that is implemented by an array. The text tokens are processed further: They are fragmented into Linguistic tokens by the *LinguisticTokenGenerator*. The *FTTokenBufferStore* also initiates and creates the indexes and the POS[3] trie. The access methods of the store are the following:

Random Id-Based Access and Sequential Scan

LinguisticToken getLinguisticTokens(DeweyIdentifier did) takes a Dewey identifier of a Linguistic token as input (this is the starting position) and returns an iterator over the Linguistic tokens in the store. If the input Dewey identifier is of an element that does not directly contain Linguistic tokens, null is returned. This access is implemented by a *B+ tree* on the Dewey identifiers. When searching for the Linguistic token with the Dewey identifier *did*, the *B+ tree* is traversed for *did* and the Linguistic token that has this Dewey identifier is returned, if it exists.

Random Value-Based Access

LinguisticTokenIterator getLinguisticTokensExact(String word) is used to return an iterator over the corresponding Linguistic tokens for the search term *word*. It is implemented by accessing the inverted index: When searching the term *word*, a look up in the inverted index is performed, returning a list of Linguistic tokens. An iterator over this list is then returned. If the word is not contained in the text and hence is not found in the index, an empty sequence iterator is returned.

Wildcard Queries When having queries containing wildcards, the word is, if possible, provided with $ at the beginning and/or at the end. The word is fragmented into n-grams. Those fragments are then looked up in the n-gram index and the lists of words are retrieved. To find the words that are contained in all the lists, a boolean ”and” is formed over them and the intersection is computed. Using this approach, we also get matches that contain the conjunctions of the n-grams, but do not match the original wildcard query. To cope with this, we have a post-filtering step in which terms are checked individually against the original query by a simple string matching operation [17].

Vector getWordsWithSuffix(String prefix) is used to return the set of all words that have the string ”prefix” as prefix. First the ”prefix” is prepended a $ and it is fragmented into n-grams. Those are then looked up in the n-gram index. After retrieving the lists, intersecting them and checking the words against the original query, the remaining terms are returned. If ”prefix” is not

[3]Linguistic category of words

contained in the document, i.e. it is not in the n-gram index, an empty set is returned.

Example

```
//book[ .ftcontains "Student." with wildcards]
```

First of all, "Student" is fragmented into its n-grams. In our implementation n is three, so the n-grams are:

- 3-grams: $st, stu, tud, ude, den, ent
- 2-grams: $s, st, tu, ud, de, en, nt

For each of these n-grams, we retrieve the corresponding list of words, intersect them and check the word against the original query "Student.".

Vector getWordsWithPrefix(String suffix) is used to return all words that have the string "suffix" as suffix. First the "suffix" is appended a $ and it is fragmented into n-grams. The fragments are looked up in the n-gram index to retrieve the lists of words. After intersecting them and checking the words against the original query, the remaining terms are returned. If "suffix" is not contained in the document, an empty set is returned.

Example

```
//book[ .ftcontains ".*dents" with wildcards]
```

The first step is to fragment "dents" into its n-grams:

- 3-grams: den, ent, nts, ts$
- 2-grams: de, en, nt, ts, s$

For each of these n-grams, we retrieve their list of words, intersect them and check the word against the original query ".*dents".

Vector getWordsWithInfix(String prefix, String suffix) is used to return all the words that have "prefix" as prefix and "suffix" as suffix and in between any number of characters (defined by the wildcard qualifier). First the "prefix" is prepended a $ and the "suffix" is appended a $. Prefix and suffix are fragmented into n-grams and the lists that are returned for the n-grams are intersected. The terms that survive are then checked against the original query. The remaining terms are returned. If either there are no words having "prefix" as prefix or "suffix" as "suffix" or both, an empty set is returned.

Example

```
//book[ .ftcontains "Stu.{1,2}ents" with wildcards]
```

At first, "stu" and "ents" are fragmented into their n-grams:

- 3-grams: st, stu, ent, nts, ts
- 2-grams: s, st, tu, en, nt, ts, s

For each of these n-grams, we retrieve the corresponding list of words, intersect them and check the word against the original query "Stu.{1,2}ents".

Vector getWordsForMultipleWildCard(String word) is used to return the set of words that fulfill the word "word" with its wildcards. If possible (depending on where the wildcards are), $ are prepended and/or appended and the parts of "word" that are not wildcards are fragmented into n-grams. These n-grams are looked up in the n-gram index, the lists are intersected and the terms that remain after intersection are checked against the original query. The terms that survive are returned.

Example

If the query is

- `//book[ .ftcontains ".*prov.*" with wildcards]`

  we look up:

  - 3-grams: pro, rov
  - 2-grams: pr, ro and ov

- `//book[ .ftcontains ".*pr.*vement" with wildcards]`

  we look up:

  - 3-grams: vem, eme, men, ent, nt$
  - 2-grams: pr, ve, em, me, en, nt, t$

- `//book[ .ftcontains "pr.*ve.*" with wildcards]`

  we look up:

  - 2-grams: pr, ve

in the index, intersect the returned lists of words and check whether the remaining words fulfill the original query.

Stemming

Vector getWordsForStem(String stem) is used to return all the words that have the stem "stem" as prefix. It is implemented by accessing the stem index. The original query term is stemmed and looked up in the stem index to retrieve all the words that have the same stem. The problem with this approach is its overhead: Before it is known whether stemming is used in the query, we build a special index for it.

Phrase queries Phrase search is implemented by using the *nextword index* [18].

PhraseIterator getPhraseExact(Vector v, DeweyId ignoreId) Processing a phrase query works in the following way: The phrase is tokenized into single words and for each of the words, it is checked whether

- It is a *firstword*: Its list of *firstword-nextword* pairs is traversed and it is checked whether one of its *nextwords* is the next word in the phrase. If it is, the *nextword* Linguistic tokens are retrieved and we continue with the word after the next word. If it is not contained, the whole phrase is not contained and an empty sequence is returned.

- It is not a *firstword*: Its list of Linguistic tokens is retrieved and we continue with the next word in the phrase. If the list is empty, the whole phrase is not contained and an empty sequence is returned.

If all words in the phrase have either found its *nextword* Linguistic token or its list of corresponding Linguistic tokens, it is tested whether there is a Linguistic token for each of the words in the phrase which are next to each other with regards to the position. If one or more phrases are found, an iterator over these phrases is returned. If no phrase is found, an empty sequence iterator is returned.

An additional speed-up strategy is to have a (possibly time-limited) index on phrases or word pairs from phrases that are commonly posed as queries (see Section 7).

Vector getPhraseWithWildcard(Vector phrase) Phrase search containing wildcard(s) is evaluated in the following way: The phrase is again tokenized into the single words. If a word contains a wildcard, all its possible words and their corresponding Linguistic tokens are retrieved. If a word contains no wildcard, it is processed as explained above. If all words in the phrase have their list of corresponding Linguistic tokens or *nextword* Linguistic tokens, it is tested whether there is a Linguistic token for each of the words in the phrase which are next to each other. If one or more phrases are found, a vector of phrases is returned. If no phrase is found, an empty vector is returned. To speed up wildcard phrase querying, we could have a n-gram index over the *nextwords* (see Section 7.3.2).

Vector getPhraseWithStemming(Vector phrase) Phrase search with stemming is evaluated in the following way: The phrase is tokenized, each of the words is stemmed and their stems are looked up in the stem index. We then compute each combination of phrases with the retrieved words, if the query is the following:

Example

```
//book[ .ftcontains "Win Strategy"]
```

we retrieve all the words that have the same stem as "Win" and "Strategy" and compute every possible combinations of these words. These newly generated phrases are then processed as described above (see Section 4.2.3).

Phrase Search with FTIgnoreOption If phrases go over ignored elements, they are processed differently.

Example

We have the following document

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book>
        <content>
            <p>
                Users can be tested at any computer workstation
                <footnote>They may be more comfortable
                at their own workstation than in a lab.
                </footnote>or in a lab.
           </p>
       </content>
   </book>
</books>
```

and the following query:

```
//book[. ftcontains 'Users can be tested at any computer
workstation or in a lab' without content .//footnote]
```

The phrase can not be found in the conventional way as the position constraint (all words need to be next to each other) is not fulfilled. If an *FTIgnoreOption* is defined and no matching phrase can be found, all words from the last matched word to the last word in the parent element are scanned (ignoring the element with the Dewey identifier of the ignore option) to check whether the rest of the phrase can found.

In our example: Having found "Users can be tested at any computer workstation", we ignore the *footnote* element and scan the rest of the text in the *p* element for "or in a lab". The scanning is implemented by accessing the B+ tree index: The Dewey identifier of the last found matching word is looked up in the B+ tree. Starting from the leaf that contains the Linguistic token of this word, we scan all leaves and look for the rest of the phrase ignoring the Linguistic tokens whose Dewey identifier is a descendant of the ignored id. We stop the scanning as soon as we reach a Linguistic token that is not a child of the current element.

4.2.4 Iterators

In this section, we describe the most important iterators that are used by the MXQuery full-text facility.

FTBaseIterator

The *FTBaseIterator* contains the basic functionalities of the full-text iterators *FTAndIterator*, *FTOrIterator*, *FTMildNotIterator*, *FTUnaryNotIterator*, *FTSelectionIterator*, and *MatchIterator*. It includes the context and subiterators of the iterator that is extending it and two methods *setContext* that sets the context of the iterator's subiterators and *reset* that resets the iterator's subiterators.

FTUnaryNotIterator

The *FTUnaryNotIterator* is the implementation of the *FTUnaryNot* operation. We only support the restricted version: The *FTUnaryNot* expression may only appear as a direct right operand of an *FTAnd* operation. The *FTUnaryNotIterator* only returns the *AllMatch* of its underlying subiterators. The actual *ftnot* is evaluated in the *FTAndIterator*.

FTAndIterator

The *FTAndIterator* is the implementation of the *FTAnd* operation. In case one or all the input iterators are an empty sequence, the *FTAndIterator* returns the empty sequence. In case all the input iterators contain *AllMatches*, we compute every possible combination of *AllMatches* that have the same parent and return it as *SuperAllMatch*, i.e. an *AllMatch* that does not contain matches, but *AllMatches*. If the right operand of the *FTAnd* is an *FTUnaryNotIterator*, the functionality of the *FTAndIterator* is inverted: If the *FTUnaryNotIterator* returns the empty sequence, a *SuperAllMatch* is created and returned. If the *FTUnaryNotIterator* returns an *AllMatch*, the empty sequence is returned.

FTOrIterator

The *FTOrIterator* is the implementation of the *FTOr* operation. If the underlying subiterators contain *SuperAllMatches* (this means that the left and/or right input of the *ftor* contains an *ftand*), the *AllMatch* is unnested to get access to the matches.

FTMildNotIterator

This iterator is not implemented yet, as it is not part of the Minimal Conformance. However, it is needed to build the query plan. It returns the *AllMatch* of its subiterator.

FTSelectionIterator

This iterator is the implementation of the *FTSelection*. It takes the sequence of *AllMatch* that fulfill the underlying *Primary Full-Text Selection* and an *FTPositional* (*FTDistance*, *FTWindow* or *FTOrder*) as input and tests the positional constraint. If is is fulfilled, a *SuperAllMatch* is created and returned.

MatchIterator

The *MatchIterator* is one of the most important iterators in the full-text model: it is the only iterator that has access to the store and the indexes. Its input are the MatchOptions that are defined in the query and two subiterators that provide the query context and keyword(s) in the form of XDM tokens. To check whether a keyword is contained in an element described by the query context, the *MatchIterator* applies the MatchOptions on the keyword and accesses the store to get all the Linguistic tokens of the result words. This list of Linguistic tokens is then traversed and it is checked whether the Linguistic token is a descendant of the query context element by comparing the Dewey identifiers. If the query context element's Dewey identifier is an ancestor of the Dewey

identifier of the Linguistic token, a match is created. In case there are several Linguistic tokens that match, an *AllMatch* for each of them is created. In case of a phrase search, each Linguistic token that is a part of the phrase and fulfills the ancestor-descendant constraint makes up one match. These matches are then all part of one *AllMatch*. In case there are several phrases that match the phrase search, an *AllMatch* for each result is created. In addition to the simple keyword/phrase search with or without MatchOptions, the *MatchIterator* also evaluates the *FTAnyallOption* and the *FTIgnoreOption*.

MatchOptions The *MatchOptions* are implemented by the superclass *MatchOption* that contains the option written in the query. Each of the *MatchOptions* has its own class storing any other information that is needed to evaluate the query. The MatchOptions are applied in the following order (if the option is part of the query):

1. FTLanguageOption: The implementation of the *FTLanguageOption* is defined by the Minimal Conformance of the specification: it compares whether the language option defined in the prolog and the language option defined in the query body is the same.

2. FTWildcardOption: Before the other options can be applied, we need whole words, so the wildcard option is evaluated as first option. In the *FTWildcardOption* implementation, it is first checked whether the option is "with wildcards" or "without wildcards". In case of "with wildcards", the words that fulfill the wildcards are retrieved from the store: If the *FTWildcardOption* is the only MatchOption in the query, the Linguistic tokens that contain words fulfilling the wildcards are retrieved. Otherwise the found words are processed further.

3. FTStemOption: As defined in the specification, stemming needs to be done before the diacritics and the case option. In the *FTStemOption* implementation, it is checked whether the option is "with stemming" or "without stemming". In the case of "with stemming", the words or the phrases that have the same stem are retrieved from the store. If the *FTStemOption* is the only MatchOption in the query, the corresponding Linguistic tokens are returned. Otherwise the found words or phrases are processed further.

4. FTDiacriticsOption: In the *FTDiacriticsOption* implementation, it is checked whether the option is "diacritics sensitive" or "diacritics insensitive". The words are stored without any diacritics in the indexes, so if the option is "diacritics insensitive", the query is processed as if it was a normal keyword or phrase query. If the *FTDiacriticsOption* is the only MatchOption in the query, the corresponding Linguistic tokens are retrieved. Otherwise the found words or phrases are processed further. If the option is "diacritics sensitive", the retrieved Linguistic tokens are checked against the original keyword or phrase with the diacritics information.

5. FTThesaurusOption: We use a local web service to implement the *FTThesaurusOption* as it is lighter in terms of memory usage and does not need to be adapted to be CLDC conform [20]. This web service calls the

getSynsets(String word) method provided by the *WordNet* Simple API. It is using the *WordNet* database in turn to return the synonyms of word "word" [23]. As *WordNet* returns several levels of synonyms, we implemented a trie over a dictionary file that contains part of speech (POS) information (see Figure 4.7). Whenever there is a call to the web service, we first look up the word in the dictionary file and return the word's POS. This information is then included in the web service call and constrains the output to the most matching synonyms. Although proposed by the specification, we did not integrate more functionality, e.g. relationship or level. We think that users are usually interested in synonyms and the level is implemented by not returning all the synonyms, but only the synonyms that have the same POS like the word for which we look for synonyms.

6. FTCaseOption: The case option is the last option to be applied. We first apply all the MatchOptions to the keyword or phrase, retrieve the Linguistic tokens from the store and then check the words of the Linguistic tokens:
   - If the option is "lowercase", the query term(s) are converted into lowercase characters and are checked against the text of the Linguistic token(s). If they are the same, the Linguistic token(s) are returned.
   - If the option is "uppercase", the query term(s) are converted into uppercase characters and are checked against the text of the Linguistic token(s). If they are the same, the Linguistic token(s) are returned.
   - If the option is "case sensitive", the query term(s) are checked against the text of the Linguistic tokens. If they are the same, the Linguistic token(s) are returned.

The *FTStopwordOption* is optional according the Minimal Conformance of the specification, so it is not implemented. If there is an extension defined by the *FTExtensionOption*, the user needs to add the corresponding implementation.

Any queries that combine wildcard, stemming and diacritics may not always return the correct result. The problem is the application order of the *MatchOptions*. When building the stem index, words are first stemmed and then the diacritics are removed. But in the query evaluation, we first look for words fulfilling the wildcard (this is already done without diacritics information) and apply the stemming algorithm on the found words. But as described in the specification, stemming needs to be applied before the diacritics option as stem(diacritics(word)) is not always the same as diacritics(stem(word)), e.g. the stem of "éducation" is "éducat", but the stem of "education" is "educ".

FTAnyallOption In the design chapter we described the rewriting of *FTAnyallOption* into queries using *ftand* and *ftor*. Our implementation of the *FTAnyallOption* evaluation is without rewriting the query:

- phrase: This option is implemented by bringing all the words in the *Expr*[4] into a phrase and using the method *getPhraseExact(Vector phrase)* of the *FTTokenBufferStore* to obtain all the phrases in the text that match phrase *phrase*.

[4]Expr: a general expression of XQuery

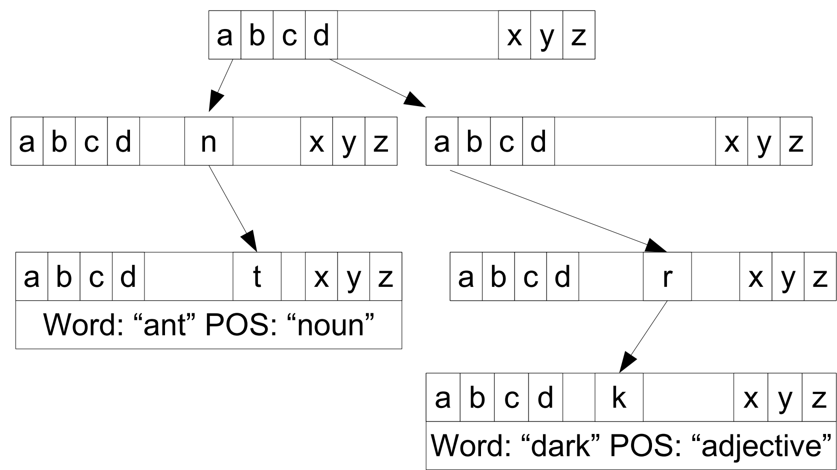


Figure 4.7: Trie with POS information

- any words: This option is implemented by using the method *getLinguisticTokenExact(String word)* of the *FTTokenBufferStore* to obtain all the words in the text that match word *word*.
- all words: This option is implemented by using the method *getLinguisticTokenExact(String word)* of the *FTTokenBufferStore* to obtain all the words in the text that match word *word*. We then find all the "phrases" that include all the words contained in the *Expr*. The "phrases" are not really phrases as the words do not have to be next to each other.
- any: This option is implemented by using the method *getLinguisticTokenExact(String word)* of the *FTTokenBufferStore* if a word is contained in the *Expr* or by using the method *getPhraseExact(Vector phrase)* if a phrase is contained in the *Expr* to obtain all the words or phrases in the text that match word *word* or match phrase *phrase*.
- all: This option is implemented by using the method *getLinguisticTokenExact(String word)* of the *FTTokenBufferStore* if a word is contained in the *Expr* or by using the method *getPhraseExact(Vector phrase)* of the *FTTokenBufferStore* if a phrase is contained in the *Expr* to obtain all the words or phrases in the text that match word *word* or match phrase *phrase*. We then find all the "phrases" that include all the words or phrases contained in the *Expr*. Again, the "phrases" are not really phrases as the words and phrases do not have to be next to each other.

FTIgnoreOption The *FTIgnoreOption* is also checked in the *MatchIterator*. If an *FTIgnoreOption* is defined and the Dewey identifier of the ignore context is an ancestor of the Linguistic token's Dewey identifier, the result is discarded and no *AllMatch* is built.

4.2.5 Minimal Conformance

The following items of the Minimal Conformance are supported in the *MXQuery* Full-Text implementation:

- MatchOptions:
  - *FTCaseOption*: all choices are supported.
  - *FTDiacriticsOption*: supported.
  - *FTStemOption*: supported.
  - *FTThesaurusOption*: supported with exception of relationships defined in ISO 2788.
  - *FTWildCardOption*: supported.
  - *FTExtensionOption*: is parsed.
- Logical Full-Text operators
  - *FTAndOperator*: supported.
  - *FTOrOperator*: supported.
  - *FTUnaryNotOperator*: Negation restriction 1 is supported.

- Positional filters
  - *FTWindow*: Window operator restriction is supported.
  - *FTDistance*: Distance operator restriction is supported.
  - *FTOrder*: Order operator restriction is supported.
- Others
  - *FTAnyallOption*: supported.
  - *FTUnit* and *FTBigUnit*: all the choices of *FTUnit* and *FTBigUnit* are supported.
  - *FTIgnoreOption*: supported.
  - *Scoring*: not yet supported.

Chapter 5

Ranking

There are several approaches to XML ranking. To compute the rank of a result, we investigated four different approaches: One classical approach that only takes the text content into account (see Section 5.1) and three approaches that consider structure and content to compute the score (see Section 5.2).

5.1 Content-Based Ranking

Content-based ranking approaches exploit the text data of a document to score and rank the results of a query. The Ranked Boolean Retrieval Model is a traditional information retrieval model to obtain score for results. Section 5.1.1 gives a short introduction to the classical model and a proposal on how to adapt it for XML retrieval and XQuery Full-Text search.

5.1.1 Ranked Boolean Retrieval Model

The main idea about the *Ranked Boolean Retrieval Model* is the observation that not every word of a text is equally indicative of the text's meaning [24]. Therefore different weights are given to different terms, i.e. a term's weight should reflect the (estimated) importance of the term. There are several ways to compute these weights, a common one are *tf.idf weights*:

Definitions

$tf_{i,j}$ = number of times term w_j occurs in document d_i
df_j = number of documents in collection that contain term w_j
n = total number of document in collection
$idf_j = \log(n/df_j)$
tf.idf weights: $tf.idf_{i,j} = tf_{i,j} * idf_j$

With these weights multiple term occurrences increase document relevance (tf) and less frequently occurring (more discriminative) terms get a higher weight (idf). These weights are then used in the following way: First the result set is

computed like in the Boolean retrieval model and then the result set is ranked based on the term weights:

Score of document i for term j: $score_i = tf_{i,j} * idf_j$.

In case of boolean operators like conjunctions or disjunctions, respectively:

$score_i(w_1 \text{ AND } \ldots \text{ AND } w_m) = min_j \; score_i(w_j)$
$score_i(w_1 \text{ OR } \ldots \text{ OR } w_m) = max_j \; score_i(w_j)$

MXQuery and Ranked Boolean Retrieval Model: The *Ranked Boolean Retrieval Model* is an easy, but powerful way to calculate the rank of a document. To use this method for XQuery Full-Text, in particular for MXQuery, we could adapt the *Ranked Boolean Model* to XML documents in the following way: First we need to materialize the documents and calculate the inverse document frequency weights idf_j for every term that is contained in the documents. The $tf_{i,j}$ weight can be calculated during query processing: $tf_{i,j}$ = number of times term k_j occurs in an XML node n_i where k_j is the keyword and n_i is a node of the result set. We then calculate all the $tf.idf_{i,j}$ for every term/node pair values and rank them based on these values [25].

To have a more sophisticated method we could also take proximity of keywords into account. If the keywords are closer, the node containing the keywords gets a higher score as a node that contains the keywords but they are all far apart [26]. This can be added easily to the existing model, as we store the absolute position in the corresponding Linguistic token of a term.

In addition, we could also take the proximity to the result node into account: If a keyword is contained in a subelement of the result node, the result is ranked lower than if it is directly contained in the result node. If there are several keywords in the query, a result node is ranked higher if it contains all the keywords directly than if the keywords are distributed over the subelements of the result node [26].

The problem of this approach is its lack of considering the structure. The model is too simple for XML retrieval as it does not take the structure of an XML document into account which in some cases might be as important as the XML document's content.

Even though the *Ranked Boolean Retrieval Model* does not take structural information into account, we consider it an appropriate ranking Model for *MX-Query* Full-Text: It can be applied for any kind of documents (containing much to few structural information) and *idf* scores can be precomputed offline, i.e. before query processing.

5.2 Content and Structure-Based Ranking

In Web search, all of the retrieving and ranking approaches, like *PageRank* or *HITS*, exploit the hyperlink structure of HTML documents.

In Section 5.2.1 we describe *XRank* which only supports ranked keyword search. When queries are only based on search terms, retrieval of structured data is more difficult than retrieval of unstructured data. One of the difficulties that we encounter is that we want to return parts of the document that contain the search term, but not the entire document. But which part should be returned? One important criterion is given by [25]: *"Structured document retrieval principle: a system should always retrieve the most specific part of a document answering the query"*. *XRank* exploits link and element-subelement relationships of the XML document to retrieve and rank the results to a query.

In Section 5.2.2 we investigate an approach that adapts the classical *Vector Space Model* to a *Vector Space Model for XML documents*.

Section 5.2.3 then describes an approach that takes content and structure into account when computing the score.

5.2.1 XRank

XRank (see [26]) is an approach for ranked keyword search over hyperlinked XML documents and collections that contain XML and HTML documents. *XRank* exploits the structure of the XML documents during query processing. XRank faces three main challenges:

- Ranking: is done at the granularity of XML elements and is not only based on hyperlinks as in HTML, but takes both hyperlinks and containment edges into account.
- Results: The results are returned at the granularity of XML elements as the deepest node containing a keyword gives more context information then returning the whole document.
- Proximity: In HTML documents the measure of proximity is the distance between the keywords, in XML documents on the other hand there are two measures: width (distance between keywords) and height (distance between keywords and result XML element) in the XML tree.

XRank considers two reference types as hyperlinks: intra-document references (IDREF) and inter-document references (XLink). The collection of hyperlinked documents is modeled as a directed graph $G = (N, CE, HE)$. N is the set of nodes $N = NE \cup NV$ where NE is the set of elements and NV is the set of values. CE is the set of containment edges: $(u,v) \in CE$ iff v is a value/nested sub-element of u; u is the parent of v or the ancestor if there is a sequence of containment edges. HE is the set of hyperlinked edges and $(u,v) \in HE$ iff u contains a hyperlink reference to v. The predicate contains*(v,k) is true if node v directly or indirectly contains keyword k.

Keyword Search with n Keywords

Query $Q = k_1,\ldots, k_n$

The set of elements that directly or indirectly contain all the query keywords is modeled by $R_0 = \{v \mid v \in NE \wedge \forall\ k \in Q\ (contains^*(v,k))\}$, but the set of all independent occurrences of the keyword query is modeled by $Result(Q) = \{v \mid \forall\ k \in Q\ \exists\ c \in N\ ((v,c) \in CE \wedge c \notin R_0 \wedge (contains^*(v,k)))\}$. The idea of

Result(Q) is that if a subelement already contains all the query keywords, it (or one of its descendants) will be a more specific result for the query, so it should be returned instead of the parent element. The problem with this approach is that the returned result may be to specific. As a solution they propose to allow user to navigate in the document or to predefine a set of answer nodes. As XQuery defines the elements that need to be returned, we do not need to take this problem into consideration.

Ranking: Properties and Function

The desired properties for the ranking function are the following:

- Result specifity: More specific results should be ranked higher than less specific results, e.g. keywords are in the same element: this is a more specific result than if query keywords occur in different elements.
- Keyword proximity: The ranking function should take the proximity of query words into account
- Hyperlink Awareness: The ranking function should use the hyperlink structure of XML documents, i.e. widely referenced XML documents should be ranked higher.

The ranking function is a Google PageRank-like function that takes the nested structure of XML into account. *ElemRank(v)* is the importance of an XML element v and defined as follows: Given query $Q = k_1,..., k_n$, $R = Result(Q)$ and $v_1 \in R$. The ranking with respect to the query keyword k_i is defined by $r'(v_1,k_i) = ElemRank(v_t) \times decay^{t-1}$ where there is a sequence of containment edges (v_1,v_2), ..., (v_t,v_{t+1}) s.t. v_{t+1} directly contains k_i. The rank is scaled down by the factor of $decay \in [0,1]$. In this way less specific results get lower ranks. If there are more than one occurrence of a keyword, say m, first the rank for each occurrence is computed and the ranks are then combined: $r'(v_1,k_i) = f(r_1, ..., r_m)$ where f can be the maximization or the sum function. The overall ranking of a result element v_1 for query $Q = k_1,..., k_n$ is: $R(v_1,Q) = (\sum_{i=1}^{n} r'(v_1,k_i)) \times p(v_1,k_1, ..., k_n)$. If a user prefers a keyword over another keyword, a weight to each keyword can be given and the rank for the keyword can be weighted accordingly: $R(v_1,Q) = (\sum_{i=1}^{n} (r'(v_1,k_i)) \times weight_i) \times p(v_1,k_1, ..., k_n)$.

ElemRanks are computed for each XML element by adapting the *Page Rank* algorithm:

$$p(v) = (1-d)/N_d + d \times \sum_{(u,v) \in HE} p(u)/N_h(u)$$

where N_d is the total number of documents and $N_h(u)$ is the number of outgoing hyperlinks from document u. The first term of the equation designates the random surfer choosing the document v, the second term designates that page v is reached through references. This scheme is mapped to XML documents by mapping each element to a document and all edges to hyperlinked edges. Several other adaptations need to be done: One problem that needs to be

addressed is the difference between one-directional hyperlinks and bi-directional containment edges. If an element has a high *ElemRank*, subelements also have a high *ElemRank*. This is called *Forward ElemRank*. On the other hand if an element has many subelements that have a high *ElemRank*, the parent element should also have a high *ElemRank*. This is called *reverse ElemRank*. This is achieved by the following first change to the *PageRank* algorithm:

$$e(v) = (1\text{-}d)/N_e + d \times \sum_{(u,v)\in E} e(u)/(N_h(u) + N_e(u) + 1)$$

where N_e is the total number of XML elements and $N_e(u)$ the number of subelements of u. $E = HE \cup CE \cup CE^{-1}$ is the set of hyperlink, containment and reverse containment edges. This adaptation ensures that *ElemRanks* are transfered bi-directional and not only one directional as in the original *PageRank* algorithm. The next adaption addresses the fact that hyperlinks and containment edges are treated similarly although they are usually independent. To discriminate between containment and hyperlinked edges, the algorithm is adapted the following:

$$e(v) = (1 - d_1 - d_2)/N_e + d_1 \times \sum_{(u,v)\in HE} e(u)/N_h(u) + d_2 \times \sum_{(u,v)\in CE\cup CE^{-1}} e(u)/(N_c(u)+1)$$

where d_1 and d_2 are used to weigh the probability of navigating through hyperlinks and containment links, respectively. This method discriminated containment and hyperlinked edges, but not forward and reverse containment edges: If an element has a lot of subelements, the *ElemRank* of each section should be a fraction of the *ElemRank* of the whole element and the *ElemRank* of a parent should be proportional to the aggregate of the *ElemRanks* of its subelements. In the final state of the algorithm, the *ElemRanks* for the reverse containment relationships are aggregated:

$$e(v) = (1 - d_1 - d_2 - d_3)/(N_e \times N_{de}(v)) + d_1 \times \sum_{(u,v)\in HE} e(u)/N_h(u) + d_2 \times \sum_{(u,v)\in CE} e(u)/N_c(u) + d_3 \times \sum_{(u,v)\in CE^{-1}} e(u)$$

where d_1, d_2 and d_3 are used to weigh the probability of navigating through hyperlinked, forward or reverse containment links, respectively. N_{de}(v) is the number of elements in the XML document containing the element *v*. If not all the features are available, e.g. hyperlinks are missing, split among the other alternatives proportionally.

The goal is to evaluate the query efficiently, ranking and returning the top m results. There are three query evaluation strategies described:

Dewey inverted list (DIL): There exists an inverted list for every keyword with the keyword as key and as value the list of Dewey ids of elements that directly contain the keywords, their corresponding *ElemRank* and the position(s)

of the word in that element. The entries are sorted according the Dewey id. Key idea of the query processing is merging the lists and simultaneously computing the largest common prefix of the Dewey ids in the different lists. Computing the longest common prefix identifies the deepest ancestor that contains the query word(s). The algorithm works as follows. There are two data structures used: A Dewey stack that stores id, rank, position list of the current dewey id and keeps track of the longest common prefix and a result heap that keeps track of the top m results seen so far. The first step is to merge the inverted lists and to compute the longest common prefix of the current entry and the previous entry stored on the Dewey stack. Dewey stack components that are not part in the common prefix are popped from the stack: If any of the popped components contain all query words, it is added to the result heap, if not, the position list and the scaled down rank (as the result gets less specific) are added to the parent.

Ranked Dewey inverted list (RDIL): The disadvantage of the above approach is its long execution time especially if users like to have only few top results. In this case, all the entries in the inverted lists need to be scanned. To address this problem, the following data structure is proposed: The inverted list entries are not sorted by Dewey id, but by the *ElemRank* as higher ranked results are likely to appear first and the scanning of the inverted list can be stopped before reaching the end of the list. This approach is convenient if there is a single search term. In case of multiple search terms, the approach faces the problem of having a keyword with a high *ElemRank* which is at the beginning of the list, and a keyword with a low *ElemRank* which is at the end of the list. The *RDIL* data structure is similar to the *DIL* data structure, but the entries are sorted according the *ElemRank*. In addition, every inverted list has a B+ tree on the Dewey ids. Query processing works as follows: If we have a keyword k_i with Dewey id d we need to find the longest prefix of d that also contains the other query words. To find this longest prefix, we use the B+ tree: We probe the B+ tree of a keyword k_j and look for the smallest Dewey id d_2 that is larger than d - either this or the predecessor share the longest common prefix. The problem with this approach is that although the single keywords can have a high *ElemRank*, the result element can have a low *ElemRank*. To guarantee that the top m results are output without having a look at all the results, a Threshold Algorithm is used that computes a threshold every point in time. If there are m results that have all a rank greater or equal to the current threshold, the algorithm stops. The threshold in this approach is the sum of *ElemRanks* of the last processed elements.

Hybrid Dewey inverted list (HDIL): In cases where keywords are not correlated in the document, there are few results, but the *RDIL* structure has to scan most of the inverted lists, i.e. probing all the entries. In this case the *DIL* structure would perform better. The *HDIL* data structure contains for every keyword the full inverted list sorted by Dewey id, a fraction of the inverted list sorted by *ElemRank* and a B+ tree on the full inverted list. The query processing is an adaptive strategy: Start with the *RDIL* strategy and periodically monitor its performance by measuring a) the time spent so far: -*t*, b) the number of results above the threshold -*r* and compute the estimated remaining time: $(m\text{-}r)\times\ t/r$. If the remaining time is smaller of equal the expected time for the *DIL* approach, switch to *DIL* (its expected time is dependent on the

number of query words and the size of each query keyword inverted list).

MXQuery and XRank

As mentioned at the beginning of this Section, *XRank* is an approach for ranked keyword search over hyperlinked XML documents. But, as stated above, it can also be used if features like hyperlinks are not contained in a document. The computation of *ElemRanks* is meant to be done offline as it takes a while to converge. This works well if texts in the collections do not change often as the *ElemRanks* do not have to be re-computed often. To use this approach, we need to materialize the XML documents to compute the *ElemRanks* of their elements.

XRank is suited for XQuery queries like *//*[ftcontains "keyword(s)"]*, i.e. keyword search. If XQuery queries include a PathExpr that already define the result nodes, we do not need to look for the most specific node. But the *XRank* approach could be adapted for queries with PathExpr the following way: If a node in the result set is a direct parent, the score is higher as if the node is an ancestor of the node containing the keyword. This could be easily integrated into the existing full-text model of *MXQuery.* There already exist functions for checking whether a Dewey identifier is a ancestor or even a parent of another Dewey identifier.

XRank would be an adequate for MXQuery Full-Text, but is very time-consuming to implement and to integrate. For time reasons we prefer a simpler model. Future work may include the implementation of a more sophisticated model, e.g. the *XRank* implementation or a ranking model similar to *XRank.*

5.2.2 Vector Space Model

In the *Vector Space Model* documents are represented as vectors in high-dimensional vector space [24]. The entry j of the vector d_i contains the number of times term t_j is contained in document i. The query itself is treated as a (short) document. The similarity between a document d and a query q is calculated using cosine: $\cos(d,q)$. The smaller the angle between query and document, the more similar the document is to the query. At the end, the documents are ranked based on their similarity with the query.

Another method that could be used is proposed by [27]. This approach queries XML documents using pieces or fragments of XML. It presents an extension of the classical *Vector Space Model* that integrates a measure of similarity between XML paths: Instead of using term weights, the weight is computed regarding the term and its corresponding context $w_D(t_i,c_i)$. The context is expressed by the XPath from the root to the node containing the term. In addition, it is proposed to relax the cosine of the document and query by determining a value for context resemblance. In this way the score is not only increased when a same (t_i,c_i) is found, but when the t_i appears in a context c_j similar to a context c_i. The classical Vector Space measure

$$\rho(\text{Q,D}) = \sum_{t_i \in Q \cap D} w_Q(t_i) * w_D(t_i) / \|Q\| * \|D\|$$

is transformed into the following equation:

$$\rho(\text{Q,D}) = (\sum_{(t_i,c_i)\in Q} \sum_{(t_i,c_k)\in D} w_Q(t_i,c_k) * w_D(t_i,c_k) * cr(c_i,c_k))/(\|Q\| * \|D\|)$$

where Q is the query, D a document in the collection, t_i term i and $cr \in [0,1]$ is the measure of resemblance between contexts.

When indexing, vector of (t, c) pairs are extracted and posting lists for each pair (t, c) are created. Each (t, c) pair gets the unique key $t\#c$. The system can then identify the occurrences of a term t under context c by accessing the posting list of key $t\#c$. Index terms are stored in a trie structure and the contexts under which the term is stored, can be retrieved using the key $t\#$. Ranking is performed in the following way: First for each similar context c'[1], the algorithm retrieves the posting list (t_i,c') and scans it to accumulate document scores, i.e. weight of the term and resemblance between c and c'. The documents are then ranked in descending order according these scores. The context resemblance is determined by representing contexts as strings and using techniques from pattern and string matching. A context c' is similar to context c if these criteria are fulfilled:

- The context c' and c have as many axis steps as possible in common and in the same order.

- The occurrences of axis steps of c' similar to the axis steps of c are closer to the beginning then to the tail of the path.

- The occurrences of axis steps of c' similar to the axis steps of c are close to each other

- Of the contexts c' and c'' that match on the same axis steps, the shorter of the two paths gets a higher score

Proposals how to determine these scores are described in [27].

MXQuery and Vector Space Model: The *Vector Space Model* is also simple model, but the approach has the problem of sparseness and huge size of the vectors in large XML collections: We need to have an entry for every term contained in the documents and a vector for every possible node in a document. To use this method for XQuery Full-Text, we could adapt the model in the following ways: The most intuitive way is to consider XML nodes as documents and to build vectors $n_{i,k}$. The entry j of $n_{i,k}$ contains the number of times term w_j is contained in node k of document i.

As we need to compute the tf information and the cosine during query evaluation which makes the processing very slow, we do not consider the *Vector Space Model* as an appropriate Ranking Model for *MXQuery* Full-Text.

[1] all contexts are assigned a cr score that measure their similarity to the query term context c_i

5.2.3 Structure and Content Scoring for XML

The approach described in [28] proposes a scoring method that is based on the traditional tf∗idf measure and takes structure and content into account. Queries are *twig queries*: a rooted tree with string-labeled nodes and child and/or descendant edges. For example a query for an RSS feed document could be the following: channel/item//title and the title needs to have the value "ReuterNews" and channel/item//link and the link needs to have the value "reuters.com".

To obtain more results then just the exact matches, query relaxations are applied. There are three possible query relaxations that transform the original query in an approximate query: *edge generalization*, *leaf deletion* and *subtree promotion*. The relaxation returns additional results, but guarantee that exact matches to the original query are also matches to the approximate query. *Edge generalization* replaces a child axis with a descendant axis, *leaf deletion* makes a leaf node optional and *subtree promotion* moves a subtree from its parent node to its grandparent node. The relaxation of a query are organized in a DAG. If a query *Q'* is obtained from a query *Q* by applying a relaxation, there is a directed edge from *Q* to *Q'*.

Three scoring methods are proposed: *twig scoring*, *path scoring* and *binary scoring*. *Twig scoring* computes the score of an answer taking into account occurrences of all structural and content predicates in the query, i.e. a match to a query would be assigned an *idf* score based on the fraction of the number of results that have the very same path and content match. The *tf* score would be based on the number of query matches for the specific answer. As this approach exploits all correlations between nodes in the query, it is very time and memory consuming. *Path scoring* loosens the correlations between root-to-leaf paths in the query, i.e. for queries with different paths, a score is calculated for each path and combined for the answer score. *Binary scoring* scores binary predicates with respect to the query root and assumes independence between these predicates, i.e. scores are computed between root and its child/children, between the root and all its descendants (until leaf level) and combine them to the answer score.

To achieve a efficient top-k query processing, the scores need to be stored and organized so that one can easily determine the highest score of a match and prune the irrelevant matches. As *idf* scores are shared across all partial results of the same query, they can be precomputed and stored for all possible relaxations of the user query.

MXQuery and Structure and Content Scoring for XML

In XQuery the PathExpr, i.e. the context, is strictly defined and need to be fulfilled to be a valid result to the query. The goal of the approach described applies query relaxation to obtain fuzzy matches. We could apply the approach for XQuery as well, but need to rank the fuzzy matches lower than the exact matches.

Primarily, we like to implement XQuery Full-Text which only allows exact matches: If the PathExpr, i.e. the context, is not fulfilled in the first place, an element is not considered as a possible result. Hence, Structure and Content Scoring for XML is not an appropriate ranking Model for XQuery Full-Text as it allows fuzzy matches, i.e. that path requirements are relaxed. Another

disadvantage of this method is its lack of precomputation: *idf* and *tf* scores are computed for queries, i.e. for each possible relaxed query of the initial query. This does not allow for precomputation of values before the query processing, i.e. the query processing might be very slow.

5.3 *MXQuery* Scoring Model

We decided to use a slightly changed *Ranked Boolean Retrieval Model*:

5.3.1 Document Collection

If the queries are evaluated over a collection of documents, the *tf* and *idf* weights are defined as follows:

Tf.Idf The *term frequency* is the number of times the word(s) occur in the result element of a query. To determine this number, we do not need the store: it can be determined during query processing. The *document frequency* and *inverse document frequency* depend on the number of documents. In a collection of XML documents, each XML document is treated as a flat document, i.e. the number of documents is equal to the number of documents in the collection and the *document frequency* of a term is in how many different documents the term occurs. The *document frequency* can be precomputed and stored for each term, e.g. in the inverted list.

5.3.2 Single Document

If the data is contained in one single document, the *document frequency* cannot be determined. Therefore, w use another measure to compute scores:

dewey-idf $= did_{parent}/did_{descendant}$

where did_{parent} is the length of the Dewey identifier of the context element fulfilling the full-text predicate and $did_{descendant}$ is the length of the keywords' Dewey identifier. This measure is then combined with the term frequency of the keywords. Weight information in the query is combined with the term frequency of the keywords. The final score looks for keyword search looks as follows:

$score_{resultnode} = did_{parent}/did_{descendant} * (tf_{descendant}\ /\ \#tf_{descendant})$
$* \ weight_{descendant})$

The final score looks for phrase search looks as follows:

$$score_{resultnode} = did_{parent}/\text{avg}(did_{descendant}) * ((tf_{descendant}/\#tf_{descendant}) * weight_{descendant})$$

where $avg(did_{descendant})$ is the average length of the Dewey identifiers of all keywords in the phrase and $\#descendants$ is the number of words in the phrase.

In case of queries containing and *ftand* or *ftor* (see 2.4), we combine the scores as described in Section 5.1.1:

$$score_i\ (w_1 \text{ AND } \ldots \text{ AND } w_m) = min_j\ score_i(w_j)$$
$$score_i\ (w_1 \text{ OR } \ldots \text{ OR } w_m) = max_j\ score_i(w_j)$$

where i is a result element and w_j, $j = 1 \ldots$ m are keywords. Optimization ideas for scoring and ranking are described in Section 7.3.4.

Scoring Example

The query

```
for $book score $s in
   doc("sample_data.xml")
   /books/book[. ftcontains "party" case insensitive]
where $s => 0.1
order by $s
return $book/metadata/title
```

returns the result:

```
<title>1984</title>
<title>Bringing Down the House: How Six Students Took Vegas for Millions</title>
```

The scores are the following:

- for book with title "Bringing Down the House: How Six Students Took Vegas for Millions":

  $$score_{House} = length(1.1.3)/length(1.1.1.3.1.24) * (1/5) = 1/2 * 1/5 = 1/10 = 0.1$$

- for book with title "1984":

  $$score_{1984} = length(1.7.3)/avg_{length}(1.7.3.3.1.1.pos, 1.7.3.3.3.1.pos, 1.7.3.3.3.1.pos, 1.7.3.3.3.1.pos) * (1/5 + 1/5 + 1/5 + 1/5) = 1/3 * 4/5 = 4/15 = 0.267$$

The book with title "1984" is a slightly better match as the term occurs more times. This could be an indicator that the book is relevant. However, the score is not that high because the search term is not directly contained in the book, but in a descendant of book. The book with title "Bringing Down the House: How Six Students Took Vegas for Millions" is ranked lower, because the term occurs only once.

Chapter 6

Benchmarking

This chapter presents our benchmark results. We used the following data for the benchmarking: Auctions.xml is auction data converted to XML from web sources. Mondial.xml is data out of the World geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources. AllShakes.xml contains all Shakespeare's plays converted to XML. In the auction data, the text and structure data ration is balanced. The mondial data is heavily structured and does not contain a lot of text data. However, Shakespeare is text data with annotation markup.

Table 6.1 presents the three test documents. **Elements** is the number of elements that the document contains. **Max depth** is the maximal depth, **Avg depth** is the average depth of hierarchy. **Size** denotes the size of the document and the **Date** is the creation date of the data. All the data can be found on [29] and [30]. Section 6.1.1 presents the time measurements for loading the data into the store, Section 6.1.2 shows the time measurements for creating Linguistic tokens and Section 6.1.3 is about the time measurements of building the single indexes. Section 6.1.5 is about the profiling results: we measured the memory usage for the store, the indexes, the Dewey identifiers and the Linguistic tokens. The last Section 6.2 gives a short description of how recall and precision could be measured in XQuery Full-Text.

6.1 Time Measurements

As we do not have a reference implementation, we cannot compare our measurements with the measurements of another XQuery Full-Text engine.

Doc	Elements	Max depth	Avg depth	Size	Date
Auctions.xml	1151	5	3.76	125 KB	2001
Mondial.xml	22423	5	3.6	1MB	2002
AllShakes.xml	179 690	7	NA	7MB	2002

Table 6.1: The Three Test documents

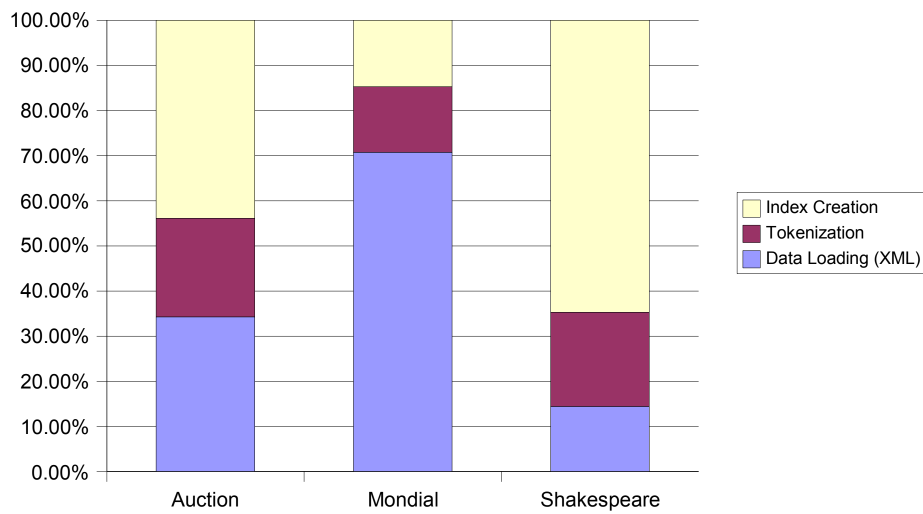


Figure 6.1: Data Loading, Tokenization and Index Creation

Doc	Time
Auctions.xml	539ms
Mondial.xml	3183ms
AllShakes.xml	8483ms

Table 6.2: Loading Times

Doc	Time
Auctions.xml	344ms
Mondial.xml	656ms
AllShakes.xml	12313ms

Table 6.3: Tokenization Times

6.1.1 Loading the Data

We measured the time to load the data into the *FTTokenBufferStore* (only XDM tokens). Table 6.2 presents the measurement results: **Time** is the processing time.

6.1.2 Creation of Linguistic Tokens

We measured the time to tokenize the text into Linguistic tokens, i.e. the work of the LinguisticTokenGenerator. Table 6.3 shows the measurement results: **Time** is the processing time.

6.1.3 Building the Indexes

We measured the time to build the indexes on the FTTokenBufferStore given that the data is already loaded into the store. The nextword index is dependent on the inverted list and the B+ tree. The inverted index, stem index, n-gram index and the B+ tree are independent from each other. While building the inverted list, a frequency table is built to determine the n most common words in the nextword index. Table 6.4 shows the measurements. **Index** is the index whose creation time is measured and **Time** is the processing time. For a comparison of loading time, tokenization time and index creation time see Figure 6.1. We observe that for the auction data and the mondial data the loading of the XML data takes the most time. For the Shakespeare data it is the tokenization and the index creation that is time consuming.

6.1.4 Queries

The following queries were run and their execution time measured. $Time_p$ is the time to evaluate the path, $Time_{ft}$ is the time to evaluate the *FTContainsExpr* and $Time_{total}$ is the total time. Table 6.5 presents the results for queries over the auction data, Table 6.6 and Table 6.7 present the results for queries over the mondial data and Shakespeare's plays, respectively. The query processing times are dominated by the path navigation because it is not indexed. Keyword or phrase queries (with or without logical operators) are fast, queries containing *MatchOptions* need more time for processing.

Doc	Index	Time
Auctions.xml		
	inverted list	122ms
	n-gram index	274ms
	B+ tree index	49ms
	nextword index	92ms
	stem index	154ms
Mondial.xml		
	inverted list	144ms
	n-gram index	267ms
	B+ tree index	65ms
	nextword index	49ms
	stem index	137ms
AllShakes.xml		
	inverted list	3409ms
	n-gram index	23527ms
	B+ tree index	995ms
	nextword index	1154ms
	stem index	9102ms

Table 6.4: Indexing Times

Query	$Time_p$	$Time_{ft}$	$Time_{total}$
fn:doc("auctions.xml")/auctions[. ftcontains "Mastercard"]	2447ms	27ms	2474ms
fn:doc("auctions.xml")/auctions[. ftcontains "Mb" case insensitive]	2395ms	58ms	2453ms
fn:doc("auctions.xml")/auctions[. ftcontains "power" with stemming]	2256ms	165ms	2421ms
fn:doc("auctions.xml")/auctions[. ftcontains "American Express"]	2439ms	33ms	2472ms
fn:doc("auctions.xml")/auctions[. ftcontains "Mastercard" ftor "American Express"]	2299ms	139ms	2438ms
fn:doc("auctions.xml")/auctions[. ftcontains "Mastercard" ftor "Postfinance"]	2344ms	117ms	2461ms
fn:doc("auctions.xml")/auctions[. ftcontains "Mastercard" ftand "American Express"]	2332ms	157ms	2489ms

Table 6.5: Query Processing Times for Auction Data

Query	$Time_p$	$Time_{ft}$	$Time_{total}$
fn:doc(”mondial.xml”)/mondial[. ftcontains ”Zurich”]	10423ms	693ms	11116ms
fn:doc(”mondial.xml”)/mondial[. ftcontains ”Africa.*” with wildcards]	10435ms	877ms	11312ms
fn:doc(”mondial.xml”)/mondial[. ftcontains ”Santa Cruz”]	10605ms	453ms	11058ms
fn:doc(”mondial.xml”)/mondial[. ftcontains ”Santa Cruz” ftor ”Santa Fe”]	10005ms	1557ms	10120ms
fn:doc(”mondial.xml”)/mondial[. ftcontains ”Santa Cruz” ftand ”Santa Fe”]	10382ms	1161ms	11543ms
fn:doc(”mondial.xml”)/mondial[. ftcontains ”San.1,3 Cruz” with wildcards]	10283ms	467ms	10750ms

Table 6.6: Query Processing Times for Mondial Data

Query	$Time_p$	$Time_{ft}$	$Time_{total}$
fn:doc(”allShakes.xml”)/plays[. ftcontains ”Julia”]	95924ms	797ms	96721ms
fn:doc(”allShakes.xml”)/plays[. ftcontains ”Julia” uppercase]	92416ms	8041ms	100457ms
fn:doc(”allShakes.xml”)/plays[. ftcontains ”Demetrius” ftor ”Hamlet”]	92763ms	1499ms	94262ms
fn:doc(”allShakes.xml”)/plays[. ftcontains ”Julia” ftand ”Romeo”]	93534ms	1032ms	94566ms
fn:doc(”allShakes.xml”)/plays[. ftcontains ”speak” with stemming]	92752ms	3497ms	96249ms
fn:doc(”allShakes.xml”)/plays[. ftcontains ”king henry VI” uppercase]	93346ms	1851ms	95197ms

Table 6.7: Query Processing Times for Shakespeare Data

Doc	Item	**Number of Instances**	**Memory usage**
Auctions.xml			
	Text token	2304	73728Bytes
	Dewey identifier	13488	316kB
	Linguistic token	10032	391kB
Mondial.xml			
	Text token	34948	1092kB
	Dewey identifier	113273	2654kB
	Linguistic token	8479	331kB
AllShakes.xml			
	Text token	357741	11179kB
	Dewey identifier	1444110	33846kB
	Linguistic token	906679	35417kB

Table 6.8: Memory Usage for Dewey Identifiers, Text Tokens and Linguistic Tokens

6.1.5 Profiling Results

We measured the memory usage of full-text items and found some bottlenecks, mainly in the Dewey identifier generation and the tokenization of the text into Linguistic tokens. Table 6.8 contains the results: **Doc** is the document, **Number of Instances** is the number of created objects and **Memory usage** is the amount of memory that the objects use.

6.2 Recall and Precision

Recall and Precision are information retrieval measures that are used to evaluate the quality of the results returned by a search engine. Recall is defined as the number of relevant documents retrieved divided by the total number of relevant documents. Precision is the number of relevant documents retrieved divided by the total number of documents retrieved. In our case, we often deal with one document, but we can define the elements bound by the path as documents. For example; If we look for all title elements that contain a certain keyword, we can define the title elements as documents. Hence, the number of relevant documents retrieved is the number of relevant title elements retrieved. In case of a document collection, the definition of a documents is a document in the document collection.

Chapter 7

Summary and Future Work

The first two sections of this chapter, Section 7.1 and Section 7.2 summarize the results of this Master thesis and review the XQuery Full-Text specification, respectively. Section 7.3 describes optimization ideas and future work: Section 7.3.1 explains how Dewey identifiers could be extended to handle updates. Section 7.3.2 describes ideas for other indexes and explains how existing indexes could be extended to speed up query processing. Section 7.3.3 is about optimizations of the operator trees and the last Section 7.3.4 describes how the ranking could be optimized.

7.1 Summary

The goal of this Master thesis was to integrate the full-text features of the Minimal Conformance of the XPath 2.0 and XQuery 1.0 Full-Text W3C specification into *MXQuery* Full-Text. This was achieved by making the following extensions: we implemented a better store and additional indexes, i.e. a stem index, an n-gram index, an nextword index and a B+ tree index. This indexes allow us to support keyword and phrase queries including stemming and wildcards. In addition, we extended *MXQuery* with a local web service to expand the query terms by synonyms. Other included options support queries with case sensitivity or queries ignoring content of user-defined elements. For queries including logical operators, we integrated an *FTAndIterator*, an *FTOrIterator* and an *FTUnaryNotIterator*. To support queries with window, distance and order predicates, we added an *FTSelectionIterator* that process and test them. Additionally, we investigated several scoring models to design and implement a scoring method to rank the results according their relevancy. We tested our implementation by running the XPath 2.0 and XQuery 1.0 Use Cases [6]. The performance of our implementation, e.g. time to load the data into the store, building the indexes and query processing and memory usage, e.g. memory used by the store and the indexes, was measured and some bottlenecks optimized.

7.2 Conclusion

The goal of XPath 2.0 and XQuery Full-Text is to provide the user with full-text search over XML documents. But on one hand, XQuery is a query language that

excepts exact results and on the other hand, full-text search returns anything that has something to do with the search terms. To unify these two worlds, the W3C working group specified a rich set of features: the *MatchOptions* do not only provide a useful advanced search, but also allow the user to find elements or documents, respectively that are not accurate matches to the query, i.e. they enable some kind of fuzzy matches. Despite of these *MatchOptions*, there is no room for fuzzy matches: If an element does not fulfill the path defined by the query, it is not match, although it might be a result the user is looking for. XQuery Full-Text assumes that the user knows the structure of the document(s). It may make sense to find a way to loosen the strict path constraint and allow for fuzzy path matches.

7.3 Future Work

7.3.1 Dewey Identifiers and Updates

There are four different types of updates in XQuery: *insert*, *delete*, *replace* and *rename*. The initial state of the Dewey number implementation is not yet able to handle updates, e.g. insertion of an XML node in the document. To avoid renumbering of all the nodes following an insertion, we implement ORDPATH (see [13]).

If we want to insert one or more nodes, we use different strategies depending on where we want to insert the nodes: The easiest case is inserting a sibling node to the right of all existing children of a node. We only need to add two to the last position of the last child. If we would like to insert the nodes to the left of a set of siblings and the left most child is number one, i.e. there are no numbers left, we use negative odd numbers. Arbitrary ORDPATH insertions that insert a new node Y between any two siblings of a parent node X where there are no in-between numbers left, are called *careting in*. In this case we create a component with an even number between the final (odd) numbers of the two siblings and a following odd number which is usually one. The even numbers are only used as caret, but do not increase the depth of the tree and, therefore, do not count for ancestry. Deletion of nodes does not require special handling. Problems occur with regard to positional information of text tokens:

- *Insert*: Inserting Dewey numbers while updating have one problem: If the inserted node(s) contain text data, we not only need to add position information to the newly inserted text, but also need to update all the position information of the text following the inserted node. The following ways can be used to address this problem:

  - One could do without the position information: In case of updates, no position information needs to be updated. The lack of position information influences the execution of window and distance queries. The distances and windows need to be evaluated while processing the query

    Example: when processing the query

    ```
    book//title[ . ftcontains "Bringing"
    ftand "House" ftand "Students"
    window at most 15]
    ```

we need to check for every possible arrangement of "Bringing", "House" and "Students" that are within 15 words. If we find one of the words, we need to check whether we find the other two in the next 15 words. If we had position information, we could retrieve the inverted list entries for the three words and calculate their distance.

  – One could have a delta index approach: As text indexing systems understand update as a delete followed by an insert, one could delete the updated document and re-insert it into a delta index. Queries are then send to both indexes and the results are merged. If a query accesses a updated document, it finds it deleted and ignores the old index.

- *Delete*: When deleting nodes, the Dewey numbering face a similar problem as when inserting nodes: If the deleted node contains text data, the position information is out of date after a deletion. Distance or window queries cannot rely on the position information anymore. The two possible solutions described above could also be used for this case. If there exists no position information, a *delete* operation does not influence the numbering of the document at all. If a delta index approach is used, a deletion of a node triggers a deletion and re-insertion of the document into a delta index.

- *Replace*: There are two different *replace* operations: value or node replacement. One can replace the value of a node with another value or one can replace a node with no node, another node or a sequence of other nodes. In case of replacing the value of a node with another value, there is a problem if the value to be replaced is text data, i.e. the positional numbering changes. In case a node is replaced by no node, it behaves like a deletion. In case the node is replaced by another or a sequence of other nodes, the position information can be out of date, if the new node or the sequence of nodes contain text data.

- *Rename*: The *rename* operation does not change the hierarchy or content of the XML document. It changes the name of an element, attribute or processing instruction. This has no influence on the positional numbering.

7.3.2 Indexing

Attribute Tokenization and Indexing

The current version does not consider attribute content as text content. It is not indexed, so full-text queries on attribute content are not supported. A future version of *MXQuery* Full-Text could include support for queries on attribute content.

Indexing for Phrases

Future work may contain other indexes to speed up phrase query processing. One approach is suggested by [31]: In addition to the inverted list with lists of *nextwords* for common words, one can have an index over frequent past phrase queries. This approach is referred to as *Partial phrase indexes*: Selected phrase

queries of lengths $l \geq 2$ are stored as terms in a conventional inverted index structure. This index could also be dynamic: If the same phrase query occurs several times, it can be integrated in the index. In order that the index is not getting too big, older phrase queries could be evicted from the index.

Another approach is described by [32]. They describe another kind of a common phrase index: The *common phrases* are sequences of two or more contiguous words that start with a common word (words with highest frequencies) and end with a terminal word[1]. These *common phrases* are then stored as keys in an inverted index. The values are the list of positions and the frequencies of this *common phrases*. This approach also speeds up phrase query processing, but the preprocessing of the documents takes more time: For each term in the document it needs to be determined of which part of speech it is (by looking it up in the POS trie).

Indexing for Wildcard Phrase Queries

The processing and evaluation of wildcard queries could also be speeded up: Depending on the position of the term with the wildcards, we have the following approaches:

- If the term containing the wildcard is at the beginning of the phrase and some of the terms fulfilling the wildcard constraint are *firstwords*, we can check their *nextword* lists for the next term.

  Example

  If the query is "Win.+ Strategy" and e.g. "Winning" is a *firstword* term, we can check its *nextword* list whether it contains "Strategy". For the other terms that are not *firstword* terms we need to evaluate in the common way: Retrieve the list of Linguistic tokens and merge to find matches that contain each of the tokens next to each other.

- If the term containing the wildcard is at any other position and the previous word in the phrase is a *firstword* term, we need a n-gram index over the *nextwords*. For example: If the query is "of Math.*" and "of" is a *firstword* term we can use the n-gram index over the *nextwords* to find all terms that fulfill the wildcard constraint "Math.*".

7.3.3 Optimization Ideas for Operator Trees

FTOrder

One possible optimization is the embedding of *FTOrder* into *FTAnd*, if we have a query like

Example

```
/book[ .ftcontains "Edward"
ftand "Thorp" ordered]}
```

[1]terminal words are words that are not a preposition, an adverb, a conjunction, a definite article, and indefinite article or a pronoun

we do not build every possible pair of "Edward" and "Thorp" and check with the *FTOrder* whether the Dewey identifier of "Edward" is smaller than the Dewey identifier of "Thorp". We can integrate the *FTOrder* into *FTAnd* by adapting the generation of the *Cartesian product*: We only output an AllMatch if the Dewey identifier of the *leftInput AllMatch*'s match is smaller than the Dewey identifier of the *rightInput AllMatch*'s match. For example if "Edward" is at position 1, 5 and 10 and "Thorp" at position 2,4 and 20, we only output the pairs (1,2), (1,5), (1,20), (5,20) and (10,20), instead of outputting all possible combinations and filtering them again by applying the *FTOrder*.

FTScope

Instead of having an *FTScope* in an *FTSelectionIterator*, we could embed the scope constraint into the *FTAnd*, if we have a query like

Example

```
/book[ .ftcontains "Edward"
ftand "Thorp" same sentence]}
```

we do not generate every pair of "Edward" and "Thorp" and check with the *FTScope* whether the occur in the same sentence. We could embed the *FTScope* into the *FTAnd*: Before outputting an *AllMatch* pair, the *FTAndOperator* also checks whether the sentence information is the same.

Access on XDM tokens

There is an index needed that returns the set of corresponding XDM tokens when giving a path expression as input, i.e. when defining a search context in the query, we need to have all the XDM tokens that fulfill this search context.

7.3.4 Ranking

Evaluation

We investigated several scoring and ranking approaches of other full-text engines for XML. Inspired by these strategies, we designed our own scoring model and integrated it into *MXQuery* Full-Text. What is missing is a survey of these scoring models that compare them with each other. Future work could include implementing additional scoring methods for XQuery Full-Text and comparing their results and precision.

Optimization for Ranking

When traversing the inverted list for search terms that are descendants of the query context, for every new query context we start at the beginning of the list. This could be optimized: The query context nodes are processed in document order and as the search terms are stored ordered according their Dewey identifier, we could remember the current position in the inverted list when changing the query context.

Another optimization that solves that problem could be the use of a B+ tree on the Dewey identifiers: We could directly check whether there are descendants

for a certain query context by looking up its Dewey identifier and getting all its descendant Linguistic tokens by doing a range scan on the leaf level (until the Dewey identifier does not have the same prefix anymore).

7.3.5 Ranked Boolean Retrieval Model for XML

To have a more sophisticated *Ranked Boolean Retrieval Model*, e.g. to integrate some structural information, we could enhance the model as described in Section 5.1.1: We could take the proximity to the result node into account. If a keyword is contained in a subelement of the result node, the result is ranked lower than if it is directly contained in the result node. We could even calculate the distance from the result node to the keyword's parent. The shorter it is, the higher the node is ranked.

If there are several keywords in the query, a result node is ranked higher if it contains all the keywords directly than if the keywords are distributed over the subelements of the result node [26]. In addition, if the keywords are closer, the node containing the keywords gets a higher score than a node that contains the keywords but they are all far apart [26]. Hence, the shorter the distance between the keywords the higher the score.

In case the query contains the use of a thesaurus, the words retrieved by the thesaurus could be weighted differently. If the original search term is contained in the context node, the node is ranked higher than if it contains a term retrieved by the thesaurus. We could even go further: If the thesaurus' terms are retrieved in a specific order, i.e. according their semantical closeness to the original term, we could give the terms a descending weight. If a context node contains a closer term than another context node, it is ranked higher.

Acknowledgments

I want to conclude with saying thank you to Prof. Donald A. Kossmann, his research group and especially Dr. Peter M. Fischer and Kyumars S. Esmaili who supported my interesting work in the Systems Group at the Department of Computer Science at the Swiss Federal Institute of Technology ETH Zurich, and gave many useful suggestions and ideas concerning the *MXQuery* Full-Text design and implementation.

Appendix A

Sample data

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
   <book number="1">
      <metadata>
         <title>
          Bringing Down the House: How Six Students Took Vegas for Millions.
         </title>
         <author>Ben Mezrich</author>
      </metadata>
      <summary>
          "Shy, geeky, amiable" MIT grad Kevin Lewis, was, Mezrich learns at a party,
          living a double life winning huge sums of cash in Las Vegas casinos.
          In 1993 when Lewis was 20 years old and feeling aimless, he was invited
          to join the MIT Blackjack Team, organized by a former math instructor,
          who said, "Blackjack is beatable." Expanding on the "hi-lo" card-counting
          techniques popularized by Edward Thorp in his 1962 book, Beat the Dealer,
          the MIT group's more advanced team strategies were legal, yet frowned upon
          by casinos. Backed by anonymous investors, team members checked into Vegas
          hotels under assumed names and, pretending not to know each other,
          communicated in the casinos with gestures and card-count code words.
          Taking advantage of the statistical nature of blackjack, the team raked
          in millions before casinos caught on and pursued them.
      </summary>
   </book>
   <book number="2">
      <metadata>
         <title>
          Beat the Dealer: A Winning Strategy for the Game of Twenty-One
         </title>
         <author>Edward Thorp</author>
      </metadata>
      <summary>
         <p>
          Ever since the time of Cardano, mathematicians have been
          delving into the theory of games of chance, but rarely
```

```
        with the stunning success achieved by Edward Thorp,
        Professor of Mathematics at the University of
        California at Irvine.
       </p>
       <p>
        Professor Thorp has devised a gambling system that
        really works, as proved by the winnings of the
        author himself, and by the thousands who have
        used this system.
       </p>
       <p>
        The essentials, consolidated in simple charts, can be
        understood and memorized by the average player:
        Professor Thorp first published this strategy in 1962.
        The system proved so successful that the Las Vegas
        casinos were forced to change their rules.
       </p>
       <p>
        Now the new revised point count system shows
        how the player can win in spite of
        present or future rule changes in Las Vegas,
        Atlantic City and Puerto Rico; how to win
        in spite of cheating by casinos. The cars in the
        book can be used in actual casino play.
       </p>
   </summary>
</book>
<book number="3">
   <metadata>
      <title>THE WAVE. The Classroom is out of Control.</title>
      <author>Morton Rhue</author>
   </metadata>
    <summary>
       Laurie isn't sure what to make of 'The Wave'.
       It had begun as a simple history experiment to
       liven up their World War II studies and had become
       a craze that was taking over their lives. Laurie's
       classmates were changing from normal teenagers into
       chanting, saluting fanatics. 'The Wave' was sweeping
       through the school - and it was out of control.
       Laurie's friends scoff at her warnings but she knows
       she must make them see what they have become before
       it's too late. This book is based on a nightmarish
       true episode in a Californian high school.
   </summary>
</book>
<book number="4">
   <metadata>
      <title>
       1984
```

```
      </title>
      <author>Georg Orwell</author>
   </metadata>
   <summary>
      <p>
       The year is 1984; the scene is London,
       largest population center of Airstrip One.
      </p>
      <p>
       Airstrip One is part of the vast political entity Oceania,
       which is eternally at war with one of two other vast entities,
       Eurasia and Eastasia. At any moment, depending upon current
       alignments, all existing records show either that Oceania
       has always been at war with Eurasia and allied with Eastasia,
       or that it has always been at war with Eastasia and allied
       with Eurasia. Winston Smith knows this, because his work at
       the Ministry of Truth involves the constant "correction"
       of such records. "'Who controls the past,' ran the Party slogan,
       'controls the future: who controls the present controls the past.'"
      </p>
      <p>
       In a grim city and a terrifying country, where Big Brother
       is always Watching You and the Thought Police can practically
       read your mind, Winston is a man in grave danger for the simple
       reason that his memory still functions. He knows the Party's
       official image of the world is a fluid fiction. He knows the
       Party controls the people by feeding them lies and narrowing
       their imaginations through a process of bewilderment and
       brutalization that alienates each individual from his
       fellows and deprives him of every liberating human pursuit
       from reasoned inquiry to sexual passion. Drawn into a forbidden
       love affair, Winston finds the courage to join a secret
       revolutionary organization called The Brotherhood, dedicated
       to the destruction of the Party. Together with his
       beloved Julia, he hazards his life in a deadly match
       against the powers that be.
      </p>
   </summary>
</book>
<book number="5">
   <metadata>
      <title>
       Le petit prince
      </title>
      <author>Antoine de Saint Exupéry</author>
   </metadata>
</book>
<book number="6">
   <metadata>
      <title>
```

```
          Night Flight
         </title>
         <author>Antoine de Saint Exupery</author>
      </metadata>
   </book>
</books>
```

Appendix B

A EBNF for XQuery 1.0 Grammar with Full-Text Extensions

```
[1]    Module                 ::= VersionDecl? (LibraryModule | MainModule)
[2]    VersionDecl            ::= "xquery" "version" StringLiteral
                              ("encoding" StringLiteral)? Separator
[3]    MainModule             ::= Prolog QueryBody
[4]    LibraryModule          ::= ModuleDecl Prolog
[5]    ModuleDecl             ::= "module" "namespace" NCName "=" URILiteral Separator
[6]    Prolog                 ::= ((DefaultNamespaceDecl | Setter | NamespaceDecl | Import
                              | FTOptionDecl) Separator)* ((VarDecl | FunctionDecl
                              | OptionDecl) Separator)*
[7]    Setter                 ::= BoundarySpaceDecl | DefaultCollationDecl | BaseURIDecl
                              | ConstructionDecl | OrderingModeDecl | EmptyOrderDecl
                              | CopyNamespacesDecl
[8]    Import                 ::= SchemaImport | ModuleImport
[9]    Separator              ::= ";"
[10]   NamespaceDecl          ::= "declare" "namespace" NCName "=" URILiteral
[11]   BoundarySpaceDecl      ::= "declare" "boundary-space" ("preserve" | "strip")
[12]   DefaultNamespaceDecl   ::= "declare" "default" ("element" | "function")
                              "namespace" URILiteral
[13]   OptionDecl             ::= "declare" "option" QName StringLiteral
[14]   FTOptionDecl           ::= "declare" "ft-option" FTMatchOptions
[15]   OrderingModeDecl       ::= "declare" "ordering" ("ordered" | "unordered")
[16]   EmptyOrderDecl         ::= "declare" "default" "order" "empty" ("greatest" | "least")
[17]   CopyNamespacesDecl     ::= "declare" "copy-namespaces" PreserveMode "," InheritMode
[18]   PreserveMode           ::= "preserve" | "no-preserve"
[19]   InheritMode            ::= "inherit" | "no-inherit"
[20]   DefaultCollationDecl   ::= "declare" "default" "collation" URILiteral
[21]   BaseURIDecl            ::= "declare" "base-uri" URILiteral
[22]   SchemaImport           ::= "import" "schema" SchemaPrefix? URILiteral
                              ("at" URILiteral("," URILiteral)*)?
[23]   SchemaPrefix           ::= ("namespace" NCName "=") | ("default"
```

```
                                    "element" "namespace")
[24]    ModuleImport                ::= "import" "module" ("namespace" NCName "=")?
                                    URILiteral ("at" URILiteral ("," URILiteral)*)?
[25]    VarDecl                     ::= "declare" "variable" "$" QName TypeDeclaration?
                                    ((":=" ExprSingle) | "external")
[26]    ConstructionDecl            ::= "declare" "construction" ("strip" | "preserve")
[27]    FunctionDecl                ::= "declare" "function" QName "(" ParamList? ")"
                                    ("as" SequenceType)? (EnclosedExpr | "external")
[28]    ParamList                   ::= Param ("," Param)*
[29]    Param                       ::= "$" QName TypeDeclaration?
[30]    EnclosedExpr                ::= "{" Expr "}"
[31]    QueryBody                   ::= Expr
[32]    Expr                        ::= ExprSingle ("," ExprSingle)*
[33]    ExprSingle                  ::= FLWORExpr| QuantifiedExpr| TypeswitchExpr
                                    |IfExpr| OrExpr
[34]    FLWORExpr                   ::= (ForClause | LetClause)+ WhereClause?
                                    OrderByClause? "return" ExprSingle
[35]    ForClause                   ::= "for" "$" VarName TypeDeclaration?
                                    PositionalVar? FTScoreVar? "in" ExprSingle
                                    ("," "$" VarName TypeDeclaration?
                                    PositionalVar? FTScoreVar? "in" ExprSingle)*
[36]    PositionalVar               ::= "at" "$" VarName
[37]    FTScoreVar                  ::= "score" "$" VarName
[38]    LetClause                   ::= (("let" "$" VarName TypeDeclaration?) |
                                    ("let" "score" "$" VarName)) ":=" ExprSingle
                                    ("," (("$" VarName TypeDeclaration?)
                                    | FTScoreVar) ":=" ExprSingle)*
[39]    WhereClause                 ::= "where" ExprSingle
[40]    OrderByClause               ::= (("order" "by") | ("stable" "order" "by"))
                                    OrderSpecList
[41]    OrderSpecList               ::= OrderSpec ("," OrderSpec)*
[42]    OrderSpec                   ::= ExprSingle OrderModifier
[43]    OrderModifier               ::= ("ascending" | "descending")?
                                    ("empty" ("greatest" | "least"))?
                                    ("collation" URILiteral)?
[44]    QuantifiedExpr              ::= ("some" | "every") "$" VarName
                                    TypeDeclaration? "in" ExprSingle
                                    ("," "$" VarName TypeDeclaration?
                                    "in" ExprSingle)* "satisfies" ExprSingle
[45]    TypeswitchExpr              ::= "typeswitch" "(" Expr ")" CaseClause+
                                    "default" ("$" VarName)? "return"
                                    ExprSingle
[46]    CaseClause                  ::= "case" ("$" VarName "as")?
                                    SequenceType "return" ExprSingle
[47]    IfExpr                      ::= "if" "(" Expr ")" "then"
                                    ExprSingle "else" ExprSingle
[48]    OrExpr                      ::= AndExpr ( "or" AndExpr )*
[49]    AndExpr                     ::= ComparisonExpr ( "and" ComparisonExpr )*
[50]    ComparisonExpr              ::= FTContainsExpr ( (ValueComp| GeneralComp|
NodeComp)FTContainsExpr )?
```

```
[51]   FTContainsExpr         ::= RangeExpr ( "ftcontains" FTSelection
                              FTIgnoreOption? )?
[52]   RangeExpr              ::= AdditiveExpr ( "to" AdditiveExpr )?
[53]   AdditiveExpr           ::= MultiplicativeExpr
                              ( ("+" | "-") MultiplicativeExpr )*
[54]   MultiplicativeExpr     ::= UnionExpr ( ("*" | "div" | "idiv"
                              | "mod") UnionExpr )*
[55]   UnionExpr              ::= IntersectExceptExpr ( ("union" | "|")
                              IntersectExceptExpr )*
[56]   IntersectExceptExpr    ::= InstanceofExpr ( ("intersect" | "except")
                              InstanceofExpr )*
[57]   InstanceofExpr         ::= TreatExpr ( "instance" "of" SequenceType )?
[58]   TreatExpr              ::= CastableExpr ( "treat" "as" SequenceType )?
[59]   CastableExpr           ::= CastExpr ( "castable" "as" SingleType )?
[60]   CastExpr               ::= UnaryExpr ( "cast" "as" SingleType )?
[61]   UnaryExpr              ::= ("-" | "+")* ValueExpr
[62]   ValueExpr              ::= ValidateExpr | PathExpr | ExtensionExpr
[63]   GeneralComp            ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[64]   ValueComp              ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[65]   NodeComp               ::= "is" | "<<" | ">>"
[66]   ValidateExpr           ::= "validate" ValidationMode? "{" Expr "}"
[67]   ValidationMode         ::= "lax" | "strict"
[68]   ExtensionExpr          ::= Pragma+ "{" Expr? "}"
[69]   Pragma                 ::= "(#" S? QName (S PragmaContents)? "#)"
[70]   PragmaContents         ::= (Char* - (Char* '#)' Char*))
[71]   PathExpr               ::= ("/" RelativePathExpr?)| ("//" RelativePathExpr)|
                              RelativePathExpr
[72]   RelativePathExpr       ::= StepExpr (("/" | "//") StepExpr)*
[73]   StepExpr               ::= FilterExpr | AxisStep
[74]   AxisStep               ::= (ReverseStep | ForwardStep) PredicateList
[75]   ForwardStep            ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[76]   ForwardAxis            ::= ("child" "::")| ("descendant" "::")| ("attribute" "::")|
                              ("self" "::")| ("descendant-or-self" "::")|
                              ("following-sibling" "::")| ("following" "::")
[77]   AbbrevForwardStep      ::= "@"? NodeTest
[78]   ReverseStep            ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[79]   ReverseAxis            ::= ("parent" "::")
                              | ("ancestor" "::")
                              | ("preceding-sibling" "::")
                              | ("preceding" "::")
                              | ("ancestor-or-self" "::")
[80]   AbbrevReverseStep      ::= ".."
[81]   NodeTest               ::= KindTest | NameTest
[82]   NameTest               ::= QName | Wildcard
[83]   Wildcard               ::= "*"
                              | (NCName ":" "*")
                              | ("*" ":" NCName)
[84]   FilterExpr             ::= PrimaryExpr PredicateList
[85]   PredicateList          ::= Predicate*
[86]   Predicate              ::= "[" Expr "]"
```

```
[87]   PrimaryExpr              ::= Literal | VarRef | ParenthesizedExpr
                                | ContextItemExpr | FunctionCall | OrderedExpr
                                | UnorderedExpr | Constructor
[88]   Literal                  ::= NumericLiteral | StringLiteral
[89]   NumericLiteral           ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
[90]   VarRef                   ::= "$" VarName
[91]   VarName                  ::= QName
[92]   ParenthesizedExpr        ::= "(" Expr? ")"
[93]   ContextItemExpr          ::= "."
[94]   OrderedExpr              ::= "ordered" "{" Expr "}"
[95]   UnorderedExpr            ::= "unordered" "{" Expr "}"
[96]   FunctionCall             ::= QName "(" (ExprSingle ("," ExprSingle)*)? ")"
[97]   Constructor              ::= DirectConstructor
                                | ComputedConstructor
[98]   DirectConstructor        ::= DirElemConstructor
                                | DirCommentConstructor
                                | DirPIConstructor
[99]   DirElemConstructor       ::= "<" QName DirAttributeList ("/>" |
                                (">" DirElemContent* "</" QName S? ">"))
[100]  DirAttributeList         ::= (S (QName S? "=" S? DirAttributeValue)?)*
[101]  DirAttributeValue        ::= (’"’ (EscapeQuot | QuotAttrValueContent)* ’"’)
                                | ("’" (EscapeApos | AposAttrValueContent)* "’")
[102]  QuotAttrValueContent     ::= QuotAttrContentChar
                                | CommonContent
[103]  AposAttrValueContent     ::= AposAttrContentChar
                                | CommonContent
[104]  DirElemContent           ::= DirectConstructor
                                | CDataSection
                                | CommonContent
                                | ElementContentChar
[105]  CommonContent            ::= PredefinedEntityRef | CharRef | "{{" | "}}"
                                | EnclosedExpr
[106]  DirCommentConstructor    ::= "<!--" DirCommentContents "-->"
[107]  DirCommentContents       ::= ((Char - ’-’) | (’-’ (Char - ’-’)))*
[108]  DirPIConstructor         ::= "<?" PITarget (S DirPIContents)? "?>"
[109]  DirPIContents            ::= (Char* - (Char* ’?>’ Char*))
[110]  CDataSection             ::= "<![CDATA[" CDataSectionContents "]]>"
[111]  CDataSectionContents     ::= (Char* - (Char* ’]]>’ Char*))
[112]  ComputedConstructor      ::= CompDocConstructor
                                | CompElemConstructor
                                | CompAttrConstructor
                                | CompTextConstructor
                                | CompCommentConstructor
                                | CompPIConstructor
[113]  CompDocConstructor       ::= "document" "{" Expr "}"
[114]  CompElemConstructor      ::= "element" (QName | ("{" Expr "}"))
                                "{" ContentExpr? "}"
[115]  ContentExpr              ::= Expr
[116]  CompAttrConstructor      ::= "attribute" (QName | ("{" Expr "}"))
                                "{" Expr? "}"
```

```
[117]   CompTextConstructor      ::= "text" "{" Expr "}"
[118]   CompCommentConstructor   ::= "comment" "{" Expr "}"
[119]   CompPIConstructor        ::= "processing-instruction" (NCName |
                                ("{" Expr "}")) "{" Expr? "}"
[120]   SingleType               ::= AtomicType "?"?
[121]   TypeDeclaration          ::= "as" SequenceType
[122]   SequenceType             ::= ("empty-sequence" "(" ")")
                                 | (ItemType OccurrenceIndicator?)
[123]   OccurrenceIndicator     ::= "?" | "*" | "+"
[124]   ItemType                 ::= KindTest | ("item" "(" ")") | AtomicType
[125]   AtomicType               ::= QName
[126]   KindTest                 ::= DocumentTest | ElementTest |
                                 AttributeTest | SchemaElementTest
                                 | SchemaAttributeTest | PITest |
                                 CommentTest | TextTest | AnyKindTest
[127]   AnyKindTest              ::= "node" "(" ")"
[128]   DocumentTest             ::= "document-node" "(" (ElementTest |
                                 SchemaElementTest)? ")"
[129]   TextTest                 ::= "text" "(" ")"
[130]   CommentTest              ::= "comment" "(" ")"
[131]   PITest                   ::= "processing-instruction" "(" (NCName |
                                 StringLiteral)? ")"
[132]   AttributeTest            ::= "attribute" "(" (AttribNameOrWildcard
                                 ("," TypeName)?)? ")"
[133]   AttribNameOrWildcard     ::= AttributeName | "*"
[134]   SchemaAttributeTest      ::= "schema-attribute" "(" AttributeDeclaration ")"
[135]   AttributeDeclaration     ::= AttributeName
[136]   ElementTest              ::= "element" "(" (ElementNameOrWildcard
                                 ("," TypeName "?"?)?)? ")"
[137]   ElementNameOrWildcard    ::= ElementName | "*"
[138]   SchemaElementTest        ::= "schema-element" "(" ElementDeclaration ")"
[139]   ElementDeclaration       ::= ElementName
[140]   AttributeName            ::= QName
[141]   ElementName              ::= QName
[142]   TypeName                 ::= QName
[143]   URILiteral               ::= StringLiteral
[144]   FTSelection              ::= FTOr FTPosFilter* ("weight" RangeExpr)?
[145]   FTOr                     ::= FTAnd ( "ftor" FTAnd )*
[146]   FTAnd                    ::= FTMildNot ( "ftand" FTMildNot )*
[147]   FTMildNot                ::= FTUnaryNot ( "not" "in" FTUnaryNot )*
[148]   FTUnaryNot               ::= ("ftnot")? FTPrimaryWithOptions
[149]   FTPrimaryWithOptions     ::= FTPrimary FTMatchOptions?
[150]   FTPrimary                ::= (FTWords FTTimes?) | ("(" FTSelection ")") |
                                 FTExtensionSelection
[151]   FTWords                  ::= FTWordsValue FTAnyallOption?
[152]   FTWordsValue             ::= Literal | ("{" Expr "}")
[153]   FTExtensionSelection     ::= Pragma+ "{" FTSelection? "}"
[154]   FTAnyallOption           ::= ("any" "word"?) | ("all" "words"?) | "phrase"
[155]   FTTimes                  ::= "occurs" FTRange "times"
[156]   FTRange                  ::= ("exactly" AdditiveExpr)| ("at" "least" AdditiveExpr)
```

```
                                | ("at" "most" AdditiveExpr)
                                | ("from" AdditiveExpr "to" AdditiveExpr)
[157]  FTPosFilter              ::= FTOrder | FTWindow | FTDistance
                                | FTScope | FTContent
[158]  FTOrder                  ::= "ordered"
[159]  FTWindow                 ::= "window" AdditiveExpr FTUnit
[160]  FTDistance               ::= "distance" FTRange FTUnit
[161]  FTUnit                   ::= "words" | "sentences" | "paragraphs"
[162]  FTScope                  ::= ("same" | "different") FTBigUnit
[163]  FTBigUnit                ::= "sentence" | "paragraph"
[164]  FTContent                ::= ("at" "start") | ("at" "end") |
                                ("entire" "content")
[165]  FTMatchOptions           ::= FTMatchOption+
[166]  FTMatchOption            ::= FTLanguageOption | FTWildCardOption |
                                FTThesaurusOption | FTStemOption |
                                FTCaseOption | FTDiacriticsOption|
                                FTStopWordOption | FTExtensionOption
[167]  FTCaseOption             ::= ("case" "insensitive") | ("case" "sensitive")|
                                "lowercase" | "uppercase"
[168]  FTDiacriticsOption       ::= ("diacritics" "insensitive") |
                                ("diacritics" "sensitive")
[169]  FTStemOption             ::= ("with" "stemming") | ("without" "stemming")
[170]  FTThesaurusOption        ::= ("with" "thesaurus" (FTThesaurusID | "default"))
                                |("with" "thesaurus" "(" (FTThesaurusID | "default")
                                ("," FTThesaurusID)* ")" | ("without" "thesaurus")
[171]  FTThesaurusID            ::= "at" URILiteral ("relationship" StringLiteral)?
                                (FTRange "levels")?
[172]  FTStopWordOption         ::= ("with" "stop" "words" FTStopWords
                                FTStopWordsInclExcl*)
                                | ("without" "stop" "words")
                                | ("with" "default" "stop" "words"
                                FTStopWordsInclExcl*)
[173]  FTStopWords              ::= ("at" URILiteral) | ("(" StringLiteral
                                ("," StringLiteral)* ")")
[174]  FTStopWordsInclExcl      ::= ("union" | "except") FTStopWords
[175]  FTLanguageOption         ::= "language" StringLiteral
[176]  FTWildCardOption         ::= ("with" "wildcards") | ("without" "wildcards")
[177]  FTExtensionOption        ::= "option" QName StringLiteral
[178]  FTIgnoreOption           ::= "without" "content" UnionExpr
[179]  IntegerLiteral           ::= Digits
[180]  DecimalLiteral           ::= ("." Digits) | (Digits "." [0-9]*)
[181]  DoubleLiteral            ::= (("." Digits) | (Digits ("." [0-9]*)?))
                                [eE] [+-]? Digits
[182]  StringLiteral            ::= ('"' (PredefinedEntityRef | CharRef |
                                EscapeQuot | [^"&])* '"') |
                                ("'" (PredefinedEntityRef |
                                CharRef | EscapeApos | [^'&])* "'")
[183]  PredefinedEntityRef      ::= "&" ("lt" | "gt" | "amp" |
                                "quot" | "apos") ";"
[184]  EscapeQuot               ::= '""'
```

```
[185]   EscapeApos              ::= "’’"
[186]   ElementContentChar      ::= Char - [{}<&]
[187]   QuotAttrContentChar     ::= Char - ["{}<&]
[188]   AposAttrContentChar     ::= Char - [’{}<&]
[189]   Comment                 ::= "(:" (CommentContents | Comment)* ":)"
[190]   PITarget                ::= [http://www.w3.org/TR/REC-xml#NT-PITarget]XML
[191]   CharRef                 ::= [http://www.w3.org/TR/REC-xml#NT-CharRef]XML
[192]   QName                   ::= [http://www.w3.org/TR/REC-xml-names/#NT-QName]Names
[193]   NCName                  ::= [http://www.w3.org/TR/REC-xml-names/#NT-NCName]Names
[194]   S                       ::= [http://www.w3.org/TR/REC-xml#NT-S]XML
[195]   Char                    ::= [http://www.w3.org/TR/REC-xml#NT-Char]XML
```

Bibliography

[1] http://www.inex.otago.ac.nz/about.html

[2] http://www.lexisnexis.com/

[3] http://www.loc.gov/index.html

[4] Julia Imhof. XQuery Full-Text support in MXQuery, 2007.

[5] http://www.w3.org/TR/xpath-full-text-10/

[6] http://www.w3.org/TR/2008/WD-xpath-full-text-10-use-cases-20080516/

[7] Yosi Mass, Matan Mandelbrod, Einat Amitay, David Carmel, Yoelle Maarek, Aya Soffer. JuruXML-an XML retrieval system at INEX'02

[8] Sara Cohen, Jonathan Mamou, Yaron Kanza, Yehoshua Sagiv. XSearch: A Semantic Search Engine for XML, Proceedings of the 29th VLDB Conference, Berlin, Germany 2003

[9] Sihem Amer-Yahia, Chavdar Botev, Jayavel Shanmugasundaram. TeX-Query: A Full-Text Search Extension to XQuery, May 2004

[10] Chadvar Botev, Sihem Amer-Yahia, Jayavel Shanmugasundaram. A TeXQuery-Based XML Full-Text Search Engine, SIGMOD 2004

[11] Emiran Curtmola, Sihem Amer-Yahia, Philip Brown, Mary Fernández. GalaTex: A Conformant Implementation of the XQuery Full-Text Language, June 2005

[12] www.mxquery.org/

[13] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels, June 2004.

[14] http://tartarus.org/ martin/PorterStemmer/

[15] http://www.research.ibm.com/journal/sj/452/amer.html

[16] Gregory Grefenstette, Pasi Tapanainen. What is a Word, What is a Sentence? Problems of Tokenisation, Proceedings of the 3rd Conference on Computational Lexicography and Text Research, Budapest, 1994.

[17] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. Introduction to Information Retrieval, Chapter 3 Preliminary Draft, Cambridge University Press, 2008.

[18] Dirk Bahle, Hugh E. Williams, Justin Zobel. Efficient Phrase Querying with an Auxiliary Index, In Proceedings of the ACM-SIGIR Conference on Research and Development in Information Retrieval. Tampere, Finland, 2002.

[19] http://imemex.ethz.ch/

[20] http://java.sun.com/javame/reference/apis/jsr139/

[21] Douglas Comer. The Ubiquitous B-Tree, in ACM Computing Surveys Volume 11, Issue 2 (June 1979), Pages: 121 - 137

[22] Gustavo Alonso, Irina Botan, Peter M. Fischer, Donald Kossmann, Nesime Tatbul. Flexible and Scalable Storage Management for Data-intensive Stream Processing, 2008.

[23] http://wordnet.princeton.edu/

[24] Thomas Hofmann. Information Retrieval, Lecture on Retrieval Models, Autumn Term 2007.

[25] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze. Introduction to Information Retrieval, Chapter 10 Preliminary Draft, Cambridge University Press, 2008.

[26] Lin Guo, Feng Shao, Chavdar Botev, Jayavel Shanmugasundaram. XRank: Ranked Keyword Search over XML Documents, SIGMOD 2003.

[27] David Carmel, Nadav Efraty, Gad M. Landau, Yoelle S. Maarek, Yosi Mass. An Extension of the Vector Space Model for Querying XML Documents via XML Fragments, XML and Information Retrieval (Workshop),14-25, Tampere, Finland, August 2002.

[28] Sihem Amer-Yahia, Nick Koudas, Amélie Marian, Divesh Srivastava, David Toman. Structure and Content Scoring for XML, Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005.

[29] http://www.ii.uni.wroc.pl/ inikep/research/Wratislavia/

[30] http://www.cs.washington.edu/research/xmldatasets/www/repository.html

[31] Hugh E. Williams, Justin Zobel, Dirk Bahle. Fast Phrase Querying with Combined Indexes, ACM Transactions on Information Systems. 22(4):573-594, 2004.

[32] Matthew Chang, Chung Keung Poon. Efficient Phrase Querying with Common Phrase Index, ECIR 2006, LNCS 3936, pp 61-71, 2006.