Implementation for a Coherent Keyword-Based XML Query Language

Venkatakalyan Potturi
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons

Repository Citation
Potturi, Venkatakalyan, "Implementation for a Coherent Keyword-Based XML Query Language" (2007). Browse all Theses and Dissertations. 87.
https://corescholar.libraries.wright.edu/etd_all/87

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
IMPLEMENTATION FOR A COHERENT KEYWORD-BASED XML QUERY LANGUAGE

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science Computer Engineering

By

VENKATAKALYAN POTTURI
B.E., University of Madras, 2001

2007
Wright State University
March 2, 2007

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Venkatakalayan Potturi ENTITLED Implementation for a Coherent Keyword-Based XML Query Language BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science Computer Engineering.

Krishnaprasad Thirunarayan, Ph.D.
Thesis Director

Forouzan Golshani, Ph.D.
Department Chair

Committee on Final Examination

Krishnaprasad Thirunarayan, Ph.D.

Mateen M. Rizki, Ph.D.

Thomas C. Hartrum, Ph.D.

Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies
ABSTRACT


Due to the increasing pervasiveness of data sets using the XML data format, numerous query languages have been proposed that exploit the structure inherent in XML. Many such query languages, supported by specialized XML search engines, are complex and not suitable for naïve users. A simple keyword based query language is described which not only exploits the structure of XML documents to extract relevant fragments, but can also fall back on retrieval through plain text search.

This thesis focuses on developing a prototype implementation for a Coherent Keyword Based XML Query Language. It analyses the typical challenges posed by the semi-structured nature of the XML format, and then describes the design and implementation of a framework that can index and search XML datasets. The prototype, built on Apache Lucene (a Java-based Text Indexing and Search APIs), incorporates several available techniques to obtain precise and coherent results. It also provides a simple user interface to browse the vicinity of result document fragments.
# TABLE OF CONTENTS

1. INTRODUCTION AND MOTIVATION........................................................... 1

2. BACKGROUND AND RELATED RESEARCH ................................. 5

   2.1 XML Data Model .................................................................................. 5

2.2 Query languages................................................................................. 7

   2.2.1 XQuery ......................................................................................... 7

   2.2.2 XML-QL ....................................................................................... 9

2.3 Extending XML-QL with Keyword Search ...................................... 10

2.4 XSearch ............................................................................................ 11

2.5 Schema free Keyword Search in XQuery ............................................ 12

2.6 Coherent Keyword based XML Query language .............................. 13

3. SEARCH TOOL REQUIREMENTS AND DESIGN ............................ 17

   3.1 Search Tool Requirements ............................................................... 17

   3.1.1 Query Language Requirements ...................................................... 17

   3.1.2 Display Requirements ................................................................. 18

   3.2 Indexing XML Documents ............................................................. 19

   3.2.1 Text Preprocessing ...................................................................... 20

   3.2.2 Document Unit ........................................................................... 21

   3.2.3 Index Structures .......................................................................... 22

   3.2.3.1 Dewey Order Inverted Index ..................................................... 23

   3.2.3.2 XPath Inverted Index .............................................................. 24

   3.2.3.3 Comprehensive Index ............................................................ 26

   3.3 Indexing and Search API ................................................................. 29

   3.3.1 Index Construction ...................................................................... 29

   3.3.2 Index Search .............................................................................. 31

   3.4 Post Search Processing .................................................................... 34

   3.4.1 Interconnection Assumptions ....................................................... 34

   3.4.2 Grouping Related Results ......................................................... 36

   3.4.2.1 Algorithms Explained ............................................................ 37
4. SEARCH TOOL IMPLEMENTATION AND RESULTS

4.1 Structural Diagram

4.2 Specific Details of Tool Components
   4.2.1 Configurer
   4.2.2 Index Controller
   4.2.3 XML Parser
   4.2.4 Document Indexer
   4.2.5 Search Controller
   4.2.6 Query Parser
   4.2.7 Search Helper
   4.2.8 Index Searcher
   4.2.9 XPath Constructor
   4.2.10 Interconnection Checker
   4.2.11 Search User Interface
   4.2.12 Document Extractor

4.3 Search Tool Evaluation and Performance
   4.3.1 Search Tool Evaluation
   4.3.2 Search Tool Performance

4.4 Issues with the Current Implementation

5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

5.2 Future Work

6. REFERENCES
LIST OF FIGURES

Figure 1: Interconnection Algorithm ................................................................. 16
Figure 2: Dewey Order ...................................................................................... 24
Figure 3: Fragment of Sigmod Record ............................................................... 25
Figure 4: Local Order ......................................................................................... 29
Figure 5: Heterogenous XML Document .......................................................... 35
Figure 6: Handling Required Terms ................................................................. 39
Figure 7: Handling Required and Optional terms ............................................. 42
Figure 8: Prototype Structure .......................................................................... 43
Figure 9: Choosing documents to Index ......................................................... 50
Figure 10: Single Query term result ................................................................. 50
Figure 11: Multiple query term result (Both required terms) ......................... 51
Figure 12: Navigation of the XML document by selecting ancestors ............ 51
Figure 13: Both optional query terms in query ............................................... 53
Figure 14: Both required terms ....................................................................... 54
Figure 15: Both required and optional terms present ...................................... 54
Figure 16: Distinct terms returning same document fragment ..................... 55
Figure 17: Query refinement ............................................................................ 56
Figure 18: Refined Query ................................................................................ 56
Figure 19: Articles with at least six authors .................................................... 57
LIST OF TABLES

Table 1: XML Datasets, their Indexing time and Index Size ........................................... 61
Table 2: Query processing times....................................................................................... 63
1. Introduction and Motivation

Extensible Markup Language (XML) is a simple text format which was designed to describe data using custom tags defined in a Document Type Definitions (DTD) or an XML Schema. The use of custom tags made XML extremely flexible and enables it to not only describe structured data like information from a table in a relational database but also semi-structured data like annotated plays etc. An XML document along with its DTD or Schema is also self-describing which has made it a standard means of data exchange between applications and for use in configuration files of enterprise applications. The increasing preference to store and transmit data in the XML format has led to a need for searching these data stores for information.

Web search engines for HTML documents are not suited for the task of searching XML documents. These search engines ignore the information in meta-data (i.e., XML tags), are not aware of any structure in an XML document and therefore consider a whole document as one single fragment. As a result, web search engines cannot derive any relationships between different fragments of the same XML document and a query will return as a result, a whole XML document instead of relevant fragments of XML documents. These drawbacks defeat the purpose of searching XML documents with web search engines as a user would have to peruse a whole document to find the relevant information.

Query languages for XML have been proposed, which try to intelligently make
use of the structure inherent in XML documents. The essential features of an XML query language are listed by comparing four query languages, XML-QL, Lorel, YATL and XQL by way of examples [1]. XQuery, the most recent query language built on the experience with other query languages contributed by database community (XML-QL, SQL) and the document community (XQL, XPath) is now a W3C candidate recommendation [2]. Query languages like XQuery provide functionality similar to what SQL provides for relational databases. They have a well defined syntax and rules for formulating queries and also provide built in functions for comparing, aggregating and sorting results. They also support regular expressions which can be useful when the underlying structure is not known to the user. The user also has total control with regards to constructing the result set from the extracted XML fragments. These query languages can be most effectively used by users who have good knowledge of the data set they need to query and also are competent enough to construct efficient queries.

The above mentioned query languages do not cater to the needs of users who either have no inclination to learn a query language or are not competent enough to use one. Such users can also have very limited or no knowledge about the data set they need to search. Further, queries written using query languages are specific to a document they are written for. To query a different document, changes have to be made to an existing query or a new one written from scratch. Additionally, tweaking a query to get the desired results when the underlying structure of the XML document is not known is a very tedious process. These drawbacks of query languages give rise to the requirement for keyword search of XML documents. An extension of XML-QL [3] which incorporates keyword search was attempted in [4]. This enables users to search multiple
documents at once, with the knowledge about the structure of an XML document not being mandatory. This still does not solve the problem of the user having to know a complex query language.

XSearch, a search engine for XML documents which attempts to address the drawbacks of both text search engines and query languages is presented in [5]. The authors define a very simple keyword based query language where the user is given a choice to specify keywords alone, labels alone or keyword-label combinations. The user can also specify as to whether a search term is required or optional. The authors define a relationship called the interconnection relationship between individual query term’s results, which exploits the structure in XML documents to find meaningfully related nodes. For multiple query terms, the search engine returns, XML document fragments which are related by the interconnection relationship.

The query language used by XSearch however, considers only elements and keywords while ignoring attributes in XML documents completely. The widespread use of attributes in data centric XML documents and the interchangeability between the use of attributes and elements for representing data make it necessary for a search engine query language to support attributes. The work by Thirunarayan and Immaneni: A Coherent Keyword-Based XML Query Language [6] builds on [5] by extending the query language to support attributes. Further, the concept of Meaningful Lowest Common Ancestor Structure (MLCAS) introduced by Li et al [7] for improving the precision of keyword searches in XQuery has been incorporated.

This thesis provides a search tool based on the query language proposed by Thirunarayan and Immaneni [6]. The interconnection relationship suggested in [5] and
the concept of \textit{MLCAS} [7] has also been incorporated. The tool also provides a simple user interface which promotes the “human-in-the-loop” [6] approach for query refinement and allows a naïve user to quickly search XML documents.

The thesis has been organized as follows: Chapter 2 provides a brief overview of the work and findings in the area of XML search. The query language proposed by Thirunarayan and Immaneni [6] is explained with examples. Chapter 3 gives a detailed description of the factors considered for implementing the query language. It also outlines the high level design, differences with the implementation provided by [6] and related issues. Chapter 4 provides the detailed implementation details, results and performance of the search tool. Finally Chapter 5 discusses the conclusions and future work.
2. Background and Related Research

This Chapter explains the data model (the nature and structure) of the data sets we are attempting to search and some of the current solutions available in this problem space. The solutions currently proposed for searching XML datasets (Query Languages) are mainly classified into languages already standardized or in the process of standardization by W3C, which are mainly structure based, and languages focused on information retrieval which are keyword based. The most recent query language which is on the way to standardization by the W3C, XQuery is introduced. XML-QL, a rule-based query language for XML developed specifically to address the W3C’s call for an XML query language (that resulted in the development of XQuery) is described, and an extension of XML-QL, which deals with adding keyword search capability [4] is introduced. The search engine XSearch which formed the basis for the Coherent Keyword-Based XML query language [6], the work by Li et.al [7] for providing a schema free search capability to XQuery and whose precision improvements are incorporated in [6] are described. Finally the work by Thirunarayan and Immaneni [6] for which this thesis provides an implementation is described.

2.1 XML Data Model

Extensible Markup Language, abbreviated XML was originally designed to replace SGML as a format for representing structured text documents. An XML
document adheres to the general syntax specified in the XML Recommendation [12] and is also conceptually modeled as an ordered tree [13]. XML documents consist of an optional document prologue where a Document Type Declaration can be specified. A DTD is used to define constraints on the logical structure and to support the use of predefined storage units. A document tree contains a root node, element nodes, attributes and character data. For our purposes, the following nodes in an XML tree were considered.

- **Root node**: There is one root node for every XML document. A root node does not occur except as the root of the tree. The element node for the document element is a child of the root node. The root node also has as children processing instruction and comment nodes for processing instructions.

- **Element nodes**: Element nodes are the mark-up in XML documents. They are defined by a character sequence (the element name) enclosed within opening and closing tags (<tagName..>) and can optionally have a set of attributes. The closing tag of elements has only the element name (<tagName>). Elements can also contain a character data between the start and end tags. In the tree view of the document, there is an element node for every element in the document. Element nodes can have other element nodes and text nodes as children.

- **Attributes**: The opening tags of elements may contain optional set of key/value pairs called attributes. Each attribute name is unique within the element node and the value is a character sequence which is enclosed in quotes.

- **Text nodes**: Character data occurring between the open and closing element tags is grouped into text modes. Each text mode is considered to be a descendant of the
element node.

In addition to the above, the data model retained for XML and most query languages [13] defines namespace nodes, processing instruction nodes, and comment nodes, and also link nodes, which make it possible to refer to other elements without explicitly including them. These nodes are ignored for our purposes.

2.2 Query languages

Query languages which have been standardized by the W3C can be classified as being structure based query languages which intelligently use the structure inherent in XML documents to retrieve information. These are heavily influenced by the database community and data-centric applications. These languages use either: (a) Navigational approach where Path-based navigation through the XML data queried is used and (b) Positional approach where query patterns are used as “examples” of the XML data queried. An example of the former is XQuery which has become the de-facto standard for structure-based query language because it is flexible, expressive, reliable, and full, with wide support from both industry and academia. An example of the latter is XML-QL which is a pattern and rule based language. A brief introduction of both these languages is given here. Bailey et al [14] can be consulted for an up-to-date and detailed survey of structure-based (non-IR) XML query languages.

2.2.1 XQuery

XQuery is derived from an XML query language called Quilt, which in turn borrowed features from several other languages, including XPath 1.0, XQL, XML-QL,
SQL, and OQL and represents the “state-of-the-art” of navigational XML query languages. At its core XQuery has the following functionality:

- **Sequences**: The results of the path expressions used to evaluate queries are sequences. Sequences can not only be composed of nodes but also atomic values like 1,2,3 etc.
- **Strong typing**: XQuery is a strongly typed language having support for the simple and complex data types of XML Schema.
- **Construction, Grouping and Ordering**: Constructors for node types and the data types in XML Schema are provided.
- **Variables**: XPath allows users to define variables in FLWOR expressions. FLWOR is an acronym for its five major component clauses: `for-let-while-order by-return`. Variables can be declared in the `let` expression inside the `for`. FLWOR expressions are terminated by the `return` clause which returns the whole result of the expression. FLWOR expressions can be followed by other expressions ad-infinitum.
- **User defined functions**: XQuery allows new functions to be defined by the user, and these functions can be recursive.
- **Un-ordered sequences**: As a means for assisting query optimization, XQuery provides the `unordered` keyword, indicating that the order of elements in sequences that are constructed or returned as result of XQuery expressions is not relevant.
- **Universal and existential quantification**: XQuery provides `some` and `all` for expressing existentially or universally quantified conditions.
• Schema validation: XQuery implementations provide support for schema validation, both of input and of constructed data, using the `validate` expression.

• Full host language. XQuery completes XPath with capabilities to set up the context of path expressions.

Note the comprehensive functionality available to a user querying XML datasets using XQuery. When the structure or schema of an XML document is not known beforehand, regular expressions can be used in a query. With this expressiveness there is also the drawback that, only users who are competent enough to learn XQuery can use it. There is also the need to have good knowledge about the schema of each XML document and a query written for one document cannot be used effectively for another document with a different schema. Further, XQuery does not provide means to query documents which are document centric where keyword search would be more effective.

### 2.2.2 XML-QL

XML-QL is a pattern-based and rule-based query language which uses element patterns in a `WHERE` clause to select elements. Such patterns can be augmented by variables for selecting data. Results of a query are produced using a construction pattern in a `CONSTRUCT` clause. This basically specifies as to how the result should be produced. A characteristic of XML-QL is that it uses query patterns containing several variables that may select several data items at a time instead of path selections that may only select one data item at a time. Note the similarity in the basic structure of a query to the structure typically used in SQL to query a database. XML-QL does not allow the use of
more than one rule (unlike SQL), and one has to use sub queries which are nested `WHERE-`.

`CONSTRUCT` rules to express complex queries. XML-QL does not support aggregation like finding the number of children of an element.

XML-QL also suffers from the same limitations of XQuery in that it is difficult to use for novice users and there is no support for keyword search.

### 2.3 Extending XML-QL with Keyword Search

Integrating Keyword Search into XML Query Processing [4] attempts to add keyword search capability to XML-QL by introducing a new `contains` predicate. The main motivation behind this being, XML documents which are not data centric and have large amount of text content can be queried. Documents whose structure is not known beforehand can also be queried without resorting to the use of regular expressions. Another aim stated, is to enable novice users to query XML documents. The `contains` predicate tests the existence of a given word within an XML element. The predicate has four arguments: an `XML element variable`, a `word`, an integer expression limiting the `depth` at which the word is found within the element and a Boolean expression over the set of constants `{tag_name, attribute_name, content, attribute_value}` imposing a constraint on the `location` of the word within the given element [4]. This work used an RDBMS to store the inverted index of the XML data. The queries are converted to SQL behind the scenes and executed over a database to gather the required result.

While this keyword extension does add keyword search capability and does simplify the process for novice users, there is a learning curve involved. This is due to the fact that the user still has to learn the basic structure of XML-QL, learn how to declare
and use variables. Further, there is no mechanism to retrieve related answers in a multiple keyword search. The user has to refine the query progressively to get the desired results.

### 2.4 XSearch

This work by Cohen et. al. [5] is the seminal work in keyword search over XML documents. The authors propose a simple query language which accomplishes the following.

- Utilize the meta data information in XML document, that is element names.
- Use plain keyword search when there is no knowledge of the structure.
- For multiple query terms, retrieve result fragments related to each other.

This simple query language addresses the drawbacks of the keyword search extension introduced above. The structure of the XML document can be used with the option to fall back on pure keyword search. The search term is of the form $l:k$, $l: $, $k: $ where $l$ is the label and $k$ is the keyword. An interior node $n$ in an ordered document tree satisfies the query $l:k$ if $n$ is labeled with $l$ and has a descendant text node containing $k$. A node $n$ satisfies the query $:k$ if $n$ has an descendant node that contains the keyword $k$.

The authors define an interconnection relationship which groups related nodes together. Each element node in an XML document represents an entity, and each entity has descendent nodes which can be considered as belonging to this entity. For example, in the Sigmod Record dataset, multiple article nodes are grouped together as children of the articles nodes. Intuitively we can see that the author and title of one article are related to each other, while an author belonging to one article node is not related to a
Two subtrees $T_a$ and $T_b$ are said to be interconnected, if the path from their roots to the lowest common ancestor in the tree does not contain two distinct nodes with the same element label, or the only distinct nodes with the same element label are these roots.

Applying this interconnection relationship to a query `article::title::` over the Sigmod Record we can group the related `author-title` pairs.

But the limitation of this interconnection relationship arises when there exists an sibling of an `article` node which is different, for example, a `journal` node which also has `title` and `author` as descendants. In this case the lowest common ancestor is the `articles` node, and an `author` belonging to the `journal` will be considered as being interconnected with the `title` of an `article` node, that is, unrelated `author-title` pairs are returned as being interconnected. Further, the query language does not deal with attributes, and is therefore not suitable for searching data-centric XML datasets where most of the information is captured in attribute bindings.

### 2.5 Schema free Keyword Search in XQuery

This work by Li et. al. [7] attempts to add the capability of schema free keyword search capability to XQuery. Multiple documents can be searched irrespective of the schema if keyword search capability is introduced. Further, the authors propose the concept of Meaningful Lowest Common Ancestor (MLCAS) which addresses the limitation of the interconnection relationship introduced by Cohen et. al [5]. This notion
of interconnectedness can be stated as follows

[6]:

Two subtrees $T_a$ and $T_b$ are said to be interconnected, if the path from $T_a$’s root to their lowest common ancestor in the tree does not contain another node that is the lowest common ancestor of $T_a$ and a distinct subtree $T^1_b$, where $T^1_b$ has the same root element label as $T_b$.

This notion of interconnectedness has a concept where there is a node overriding another node when the interconnection is determined and the common ancestor of the overriding node and the node in question is the Meaningful Least Common Ancestor. When we apply Li et. al.’s notion to the case where Cohen et. al.’s notion of interconnected nodes failed above, that is, when an author in a journal node and a title in a article node are considered, the title present in the journal node will override the title present in the article node and the journal is the MLCA of author and title. In the opposite case, the article node is the MLCA of the title and author it contains. This notion prevents the grouping of unrelated author-title combinations.

The authors propose adding the mlcas keyword to XQuery to enable the schema free search capability. Although this combines the expressive ability of XQuery with Keyword search capability, the user still has to learn XQuery.

2.6 Coherent Keyword based XML Query language

Although the simple keyword based query language used by XSearch, uses the structure implicit in XML documents to deliver precise and semantically related pieces of information, it does not support the use of attributes. Thirunarayan and Immaneni [6]
smoothly extend the query language used by XSEarch to use attributes. Further, the precision improvements suggested by Li et. al.[7] have been incorporated. The additional benefits derived by the above changes include:

- Data centric XML documents where the information bearing nodes are the values of attributes can now be queried.
- Authoring variations where \(<T \ A="s"/>\) and \(<T> \ <A> \ s \ </A> \ </T>\) are used interchangeably can be handled.
- Semantic Web formalizations like RDF and Owl make extensive use of attributes, and the two forms \(<T \ A="s"/>\) and \(<T> \ <A> \ s \ </A> \ </T>\) are equivalent in RDF. Supporting attributes would make it possible to query RDF and OWL.
- Coherent or semantically related results can be generated not only for homogenous XML documents (XSearch), but also for heterogeneous XML documents (Inclusion of Li et. al’s precision improvements).
- XML documents can get refined by annotators, that is, for example, the entire name in a text may be recognized and enclosed within \(<\text{Name}> \ldots \</\text{Name}>\) tags, while in a subsequent step, it may be refined to the first name and the last name being used separately.

In the query language proposed by Thirunarayan and Immaneni, for a single search term a satisfying XML subtree can be defined as follows [6].

1. The search term \(e:a:k\) is satisfied by a tree containing a subtree with the top element \(e\) that is associated with the attribute \(a\) with value containing \(k\), or a sub element \(a\) with descendant text node containing \(k\).
2. The search term \textit{e:a:} is satisfied by a tree containing a subtree with the top element \textit{e} that is associated with the attribute \textit{a}, or sub element \textit{a}.

3. The search term \textit{:a:k} is satisfied by a tree containing a subtree with a top element that is associated with the attribute \textit{a} with value containing \textit{k}, or a sub element \textit{a} with descendant text node containing \textit{k}.

4. The search term \textit{e::k} is satisfied by a tree containing a subtree with the top element \textit{e} and that has
   
   \begin{itemize}
   \item an attribute associated with the value containing \textit{k}, or
   \item a descendant element with an associated attribute value containing \textit{k}, or
   \item a descendant text node containing \textit{k}.
   \end{itemize}

5. The search term \textit{e::} is satisfied by a tree containing a subtree with the top element name \textit{e}.

6. The search term \textit{:a:} is satisfied by a tree containing a subtree with a top element that is associated with the attribute \textit{a} or a sub element \textit{a}.

7. The search term \textit{::k} is satisfied by a tree containing
   
   \begin{itemize}
   \item a subtree with a top element that is associated with an attribute value containing \textit{k}, or
   \item the descendant text node containing \textit{k}.
   \end{itemize}

We can see that both duality between the usage of elements and attributes and the progressive refinement of XML documents has been accounted for.

Further Thirunarayan and Immaneni describe with the help of pseudo code how both the notions of interconnectedness are implemented (Figure 1). In what follows, a
“path” refers to the access path of an element node from the root of the document. The predicate \( \text{common-prefix}(p, q) \) yields the prefix string common to paths \( p \) and \( q \) (from \( \text{concatenation}(q, r) = p. \)). \( \text{is-label-disjoint}(\text{aps}, \text{apt}) \) takes two access paths, and verifies that the labels on the nodes (with the exception of the last nodes) of the two paths are disjoint [6].

This algorithm modularized as the predicate \( \text{is_interconnected} \) is used in this implementation of the query language.

```plaintext
foreach (sp,tp) in result(s) x result(t)
begin
  apsp := access-path(sp);
  aptp := access-path(tp);
  if (filename(sp) = filename(tp)) and
  is-label-disjoint(remainder-suffix(apsp, common-prefix(apsp, aptp)),
  remainder-suffix(aptp, common-prefix(apsp, aptp)))
  // Cohen et al’s constraint above
  then
  // Li et al’s constraint below
  overridden := false;
  foreach sp0 in result(s)
  begin
    apsp0 := access-path(sp0);
    if (is-prefix(common-prefix(apsp, aptp), common-prefix(apsp0, aptp))
    and (apsp0 ! = aptp))
    then begin overridden := true;
    goto FINAL;
    end;
  end;
  endforeach tp0 in result(t)
  begin
    aptp0 := access-path(tp0);
    if (is-prefix(common-prefix(apsp, aptp), common-prefix(apsp, aptp0))
    and (apsp ! = aptp0))
    then begin overridden := true;
    goto FINAL;
    end;
  end;
FINAL:
  /* pairing-found(sp) := true;
  */ pairing-found(tp) := true;
query-answer += if overridden then {}
  else { (filename(sp), { apsp, aptp }) };
end; // if
end; // foreach
```

Figure 1: Interconnection Algorithm
3. Search Tool Requirements and Design

The realization of a search tool for a particular data format requires providing a solution for the three steps involved, namely, definition of a query language, indexing the data set and searching the index. The query language defined by [6] has been described in detail in the previous Chapter. This Chapter explains how the task of indexing and searching XML documents has been accomplished. The requirements of the search tool, factors which influenced the design of the index, the indexing and search library used, the post search refinement of the results which involves applying the interconnection relationship introduced in [5] and the concept of Meaningful Lowest Common Ancestor Structure (MLCAS) [7] are explained.

3.1 Search Tool Requirements

The search tool should provide a simple user interface through which one can create and maintain indexes for a given document collection. The prototype should search the indexes created while satisfying the functional requirements of the XML query language. The requirements imposed by the query language and the user interface are discussed below.

3.1.1 Query Language Requirements

The query language proposed in [6] requires the tool prototype to support
searches on element names, attributes and keywords. It should allow the user to use attribute and element names interchangeably and return document fragments rooted at element nodes which satisfy the query. The language also requires the search process to have knowledge about descendants or conversely, ancestors, of all elements to be recorded as nodes can be returned which have descendant nodes containing the keywords or descendant elements attribute values satisfying the search keyword. Lastly, the position of each element in a document needs to be stored to determine if document fragments are related by the Interconnection relationships described before. The prototype is not required to be aware of the schema or DTD of the XML document and the user is not given the choice to index or search XML documents at a specific granularity.

To summarize, the query language requires the search tool to be aware of and distinguish between the following data in each document:

- Meta-data information like element and attribute names.
- The structure of the XML document, which is the exact location of each element and attribute in the document and also the textual content in text nodes or attribute values.

### 3.1.2 Display Requirements

The search tool should provide a simple interface where a user can choose documents to index and enter query terms. The results returned by a search query are at the granularity of element nodes in XML documents and the presence of multiple search terms in a query means that each result in the result set has multiple XML document fragments. The authors of [6] propose a “human-in-the-loop” approach where the user
can navigate through the XML tree to better refine his query. This means that the user should be shown the list of ancestors for the XML document fragments returned as a result and he should be able to navigate the document starting from a result element node.

### 3.2 Indexing XML Documents

Search operations attempt to map user requests to pertinent fragments in a document or a document collection. A sequential scan of the document set for the query pattern may be suitable for small collections, but is inefficient for large collections. Alternate representations of the data in the document set which enables effective retrieval of pertinent information by search operations needs to be created, and the process which does the same is called indexing. An index for a document serves to describe the information contained in the document which can be used by search operations to find the relevant results. The data-structures used by indexes should be efficient with respect to search operations and their design depends on the data which is being indexed, the model used in information retrieval and the query language used by the search tools. For example, fully structured data or databases use tables as the data structure, relational calculus is the model used to retrieve the information and the query language is SQL. Information retrieval on unstructured data on the other hand commonly uses inverted indexes as the data structure, some of the models used are vector space analysis, Latent Semantic Analysis, Singular Value Decomposition and others, and the query language is text queries [8] [9]. Data represented in XML on the other hand is semi-structured as explained in the previous Chapter and index creation for such data has its own particular challenges for defining the data structure and the model to use. The steps involved in
indexing XML documents, the issues considered and the solutions adopted are explained in the following sections.

### 3.2.1 Text Preprocessing

The first step in any indexing task is to lexically process the terms in the data set which are going to be indexed. Lexical processing includes extracting tokens or index terms by splitting words at punctuation characters while carefully considering digits, acronyms and special characters. For example if there are numbers in words having hyphens, they should not be split and should be considered as one token, internet domain names and email addresses should be identified and treated as one token. The case of words is also not important and for efficiency, both the index terms and the search terms can be converted to lower case. Another consideration is the removal of commonly occurring terms which are irrelevant for searching as they are not information or data to be searched for. Examples of such words are "a", "and", "the", "that", "for" etc. Lexical processing makes sure that only important or relevant terms are stored in the index. This decreases the index size thereby improving the efficiency of the overall search process.

Both lexical processing and syntactic processing like stop word removal were applied to the input XML document set. Care should also be taken that terms in the query also undergo the same text processing techniques so as to gather correct results from the index.

One of the syntactic processing techniques not applied was stemming. Stemming is the process of reducing words to their root form. For example, a stemming algorithm will reduce "lazy" to "lazi". The word "laziness" would also be reduced to "lazi", allowing searches for either word to find documents containing the other. For XML
documents, the META data like element names describe what the data in the text nodes represent and should not undergo this processing. For example in Sigmod Record, there is a distinction between “authors” and “author”. Applying stemming to the tag names will remove the difference between the two and as a result incorrect results will be returned. Applying stemming algorithms selectively, that is, applying stemming only to the content of text nodes and attribute values while possible, adds increased complexity to the search tool. This would mean that separate indexes would have to be created for the content in text nodes and a search operation would in effect be two sub-searches.

3.2.2 Document Unit

To create an index for a document set, a document unit or indexing unit has to be defined. A document unit is the lowest granularity at which a search tool returns a result for a query. For unstructured data for example, a document unit can be one document in a collection of documents, paragraphs in a text document, web pages on the web etc. The choice of the document unit at index time governs how efficient the search operation over the index can be. The post index search processing or support required to satisfy a query is also governed by the choice of the document unit. For example, indexing content in emails requires decisions on how to index the attachments with the emails. Text attachments, either single documents or a set of compressed documents can be indexed separately from the content and have a reference to the email body. A search in this case can return either the document in the attachment independent of the email, or both the email and the attachment together, and moreover if the attachment appears as a stand alone document elsewhere, there is no need to make the effort of indexing it twice. Another solution would be to index the content and the attachments as one document
unit. This would mean that the attachments are always bound to the email, and might require more space if the attachment is indexed twice. Both solutions while realizable have, space, efficiency and scalability trade offs.

XML documents being semi-structured have document units at the granularity of XML elements. This gives rise to the challenge of identifying to which level of granularity one should go. That is, at what depth of the tree one should consider an element a document unit. As there is no one fixed schema or DTD which authors of XML documents follow, finding a generalized document unit which would satisfy all the cases is a challenge. Another challenge is to identify each document unit uniquely in a single XML document. The choice of the document unit is also influenced by the query language which is going to be used and the expected results of the query language. Chapter 2 explains how the query language handles the concept of context of a query over a XML document and how semantically related answers at the granularity of element nodes are returned. Considering the fact that the query language is schema and DTD agnostic and that there is no requirement for the user to specify the granularity at which answers are desired, each element in a XML document is considered as a document unit. The post search processing that will be required in this case will be determination of the semantically related terms in the result set. This choice of the document unit gives us the option of indexing the attributes, attribute values and the data in the text nodes associated with the element together. The position of each element can also be stored so as to identify it uniquely in the XML document.

3.2.3 Index Structures

Once the document unit for a data set has been decided, the task now is to design
an index structure which can represent all the information present in the document unit. The index structure should also facilitate efficient searches and yield result sets which satisfy the query language requirements. This Section describes the index data structures considered and the final solution adopted.

### 3.2.3.1 Dewey Order Inverted Index

The requirement to search by element name, attribute or keyword while returning fragments rooted at element nodes containing the search terms automatically implies that the exact position of each document unit in the XML tree should be known. The Dewey order [10] provides an encoding for an XML tree where each element is numbered in relation to both its parent and its siblings. Consider the Dewey encoded XML tree in Figure 2. We can see that the Dewey ID of each element has as prefix the ID of its parent node. An inverted index which is an ordered list of all the index terms with each term associated with a list of Dewey ID and document ID pair can now be constructed. A search operation over this inverted index with the keywords specified will return the Dewey ID’s of the elements which match the query terms. Post index access processing or merging of the postings which adds an additional step to retrieval is required to find the correct element nodes containing the keywords. Further processing is required to obtain the final result, which is, for the nodes with the same document ID, making sure that the Dewey ID of the element node is a prefix of the Dewey ID of the attribute node which is in turn the prefix of the Dewey ID of the keyword.

For multiple query terms, the result set obtained, after the index search and merging operation described above for each word in the query is an intermediate result set. The interconnection relationships defined by both Cohen et al’s and Li et al’s have to
be applied to obtain the related terms which will constitute the final result. Both notions of interconnectness require the knowledge of the element labels of the ancestors.

![Diagram of Dewey Order](image)

**Figure 2: Dewey Order**

For the index structure described which uses the Dewey IDs, it is a non trivial task to find the element labels of the ancestors of a node. It would require additional searches on the index, with the Dewey ID as the key to get the ancestor node labels. It is for this reason that this approach was not implemented.

### 3.2.3.2 XPath Inverted Index

XPath is a W3C standard for addressing parts of an XML document. It is based on the tree representation of an XML document and uses path expressions for navigating through the elements and attributes in an XML document. A path expression can be built from the root to any node in the XML tree.

For example, consider the XML document fragment in Figure 3, the second child of the “authors” node will have the XPath `/SigmodRecord[1]/issue[1]/articles[1]/authors[1]/author[2]` and will access the author “Karen Botnich”, while the first child will have the path `SigmodRecord[1]/issue[1]/articles[1]/authors[1]/author[1]`, accessing the
Figure 3: Fragment of Sigmod Record

author “Anthony I. Wasserman”. We can see that each document unit can be uniquely identified in the XML tree and also its position with respect to its siblings is known. This form of XML document representation has the advantage that the label names of the ancestor nodes of each element node are known. This solves one of the issues with Dewey order tree encoding, which is that now there is no need to repeatedly query the index of the element label. An inverted index with the index terms as the key and a list of <Document ID, XPath from root> has been used as the indexing strategy by Thirunarayan and Immaneni. This indexing structure requires the search process to access the index once to get all the required information for building the intermediated result set which will be processed further by applying the interconnection relationship algorithms. As explained by Thirunarayan and Immaneni, this indexing strategy requires a post index access processing step similar to what one had to do with the Dewey order encoding.
scheme to build the intermediate result set.

The post index access processing is required due to the fact that element nodes, the attributes associated with them, the attribute values and the text nodes “contained” within element nodes have to be treated separately as the query language makes a distinction between the three. Therefore, separate XPath expressions have to be constructed to represent the attributes and text nodes even if they are associated with one document unit. Further, the prefix in the path expressions of attributes and terms in text nodes associated with an element is common with that of the element node and therefore duplicated. These additional XPaths which have to be indexed have the effect of increasing the size of the index. An index structure which attempts to remove these disadvantages is discussed in the following Section.

3.2.3.3 Comprehensive Index

The query language expects as result, fragments rooted at element nodes, even if for a query term an index search is conducted for attribute names, attribute values or information in text nodes and not by element name. Instead of an inverted index structure, the strategy of treating each document unit in the XML tree as a separate document was considered. Such an index structure will allow us to store all the information relating to a document unit (an element node), which is, attribute names, values, the terms in text nodes. Storing the above mentioned information and making it searchable by an index search process will provide us with two advantages. Firstly, the need for post search processing to build the intermediate result set is removed as, during index search itself one finds the element nodes which “contain” the attribute names, attribute values or text nodes. Secondly, storing the XPaths for terms in text nodes,
attribute names and attribute values becomes redundant information. This is because attributes, attribute values and text nodes are represented directly within element nodes with which they are associated and results in a decrease in the size of the index. The information to be stored for each document unit or element node and why it needs to be stored is explained below.

- **Document ID**: Identifies the document to which the document unit belongs to. It is required during post search support so that only document fragments from the same document are considered for calculating the interconnection relationships and also for displaying the result to the user.

- **Element Name**: The label of the document unit which is being indexed.

- **Ancestor List**: The list of ancestors for the element node starting from the root of the XML tree. This is required as the query language abstracts from the differences between attributes and elements as explained previously. Concretely, for the search term “e: a: k”, since “a” can also be a descendant element node in a tree rooted at “e”, if “a” is the name of the current element node, we check the ancestors of “a” to see if “e” is present. Secondly, the ancestor label names are required for determining the interconnection relationship. The ancestor list should be searchable by the index search process.

- **Tree Location**: Each element node or document unit needs to be identified uniquely in the XML tree. This is required for both calculating the interconnection relationship, and for retrieving the element fragment from the XML document. This need not be searchable as it is required only for post search support. The numbering scheme followed to number each node, called Local
Order Encoding [10] is shown in Figure 4. Each node is given a number relative to its sibling and the tree location of a node now consists of the node IDs of its ancestors from the root of the tree. For example, the tree location of the scene node is “1 2 2”.

- **Attribute Names:** All the attribute names associated with the element are stored and are searchable.
- **Attribute Values:** The attribute values are stored and searchable. They will be needed when the index is searched for keywords entered in the query.
- **Text Node content:** The content of the text node is stored and searchable. Like attribute values, index search for keywords will access the terms stored. The index terms or the significant words are obtained as explained before in Text processing.
- **Attribute Name and Value pair:** In addition to the attribute names and values separately, we need to store the name value-pair for all the attributes associated with the element as this is the easiest way to associate each other and search specifically for an attribute having a particular value.

We can see that the list of ancestors and tree location of the element is common to all the attributes, attribute values and text node content and all the information needed for satisfying the query language is stored. This indexing structure while storing the required information without any redundancy raises a different kind of challenge. It is apparent that there is no longer a straight forward mapping of the query terms to the index terms. This is because there is no “postings list” for a straight forward look up of an index for the information about occurrences of query terms in XML documents. The task of
implementing the indexing strategy and also searching the index in a reasonably efficient way is a non trivial task. The next Section introduces a mature library, Lucene, with an easy-to-use API which was used to accomplish the tasks of indexing and searching. The open source tool kit also provides a simple query language which was used to extract information from the index.

3.3 Indexing and Search API

Lucene [12], provided by Apache, is a high performance, full-featured text search library written in Java. It provides an easy-to-use API for efficiently constructing indexes and searching them. This Section explains how the tasks of indexing and searching the index structure introduced above are accomplished with the API provided by Lucene.

3.3.1 Index Construction

Lucene indexes information as a sequence of documents which are the indexing
units returned for a search operation. Each document is a collection of field-value pairs. The value of a field is nothing but a sequence of index terms or tokens, with each term being a string which represents the smallest unit of information. The value terms of some fields can be made to undergo the textual processing described before. Lucene provides what are called Analyzers which extract indexable tokens and process the text before it is added to the index. An analyzer called the Standard Analyzer which eliminates stop words, converts terms to lower case and performs other input modifications was used. Fields provide means to organize and describe the information containing value strings in each document. Each field has the following attributes:

- **Store** – Specifies whether a field is stored in the index or not.
- **Index** – Specifies whether a field is indexed or not. A field, if indexed, means its value terms will be searchable. Further two attributes TOKENIZED and UN_TOKENIZED provide the option of passing the value terms through an Analyzer. Tokenized text will undergo textual processing by the analyzer chosen.
- **TermVector** – Specifies if a field should have a term vector, and if yes, how.

Lucene provides an API through which one can create documents and add them to the index. Lucene creates inverted indexes for the documents added to the index and stores statistics about terms contained in the text.

We can see now that the index structure designed for the search tool can be directly mapped to a Lucene document. The document ID, element name, ancestor list etc. are considered as field names and the information they denote is stored as values. All
the fields are stored in the index and, all fields except document ID and tree location are indexed. The element name, ancestor names, and attribute names are not tokenized while the other fields are. The field names used are documentID, elementName, ancestorList, treeLoc, attributeNames, attributeValues, attr#name, textValue. Here, the field attr#name has the attribute “name” appended to attr#, and has the attribute value as the field value.

### 3.3.2 Index Search

Lucene provides IndexSearcher, MultiSearcher and QueryParser classes which can be used to search the index. The MutiSearcher is used to search multiple indexes and the QueryParser translates the user query term, which can be Google-like search expressions into Lucene’s API representation of a query. The following types of search operations are supported:

- Searching by Field name and the text sought in the field.
- Wildcard searches by using ‘*’ and ‘?’ for multiple and single characters.
- Fuzzy searches based on Edit Distance algorithm to find words similar to a query term. Query terms followed by ‘~’ and an additional optional value between 0 and 1 to specify the required similarity. Proximity searches use ‘~’ at the end of a phrase with a number to find words which are a specific distance away,
- Boolean searches which use operators like AND, OR, NOT and +/- operators. Multiple clauses can be grouped using parenthesis to form sub-queries for controlling the Boolean logic.
- Escape characters are supported to enable search for the special characters.
Lucene provides means to efficiently search the index and also provides the user with the flexibility required by supporting the above mentioned features in its search API.

To search the index structure we need to primarily search by field name and use Boolean operators. Also, each query term is passed through the same Analyzer that was used during index time. Each user query should return a result satisfying the query language’s expected result set for a single term query as explained in Chapter 2. Mapping the user query to the Lucene search term to get the results satisfying the query language is done as shown by the following example queries which can be applied to the Sigmod Record listing shown before:

1. Query term: article:initPage:45 (e:a:k)
   - (elementName:article AND attr#:initPage:45)
   - (elementName:initPage AND ancestorList:article AND textValue:45)

2. Query term: article:title: (e:a)
   - (elementName:article AND attributeName:title)
   - (elementName:title AND ancestorList:article)

   - (attr# volume:11)
   - (elementName: volume AND textValue:11)

4. Query term: author::01 (e::k)
   - (elementName:author AND textValue:01)
   - (ancestorList:author AND textValue:01)
   - (elementName:author AND attributeValue:01)
5. Query term: authors:: (e::)
   - elementName: authors

6. Query term: :position: (:a:)
   - attributeName: position OR elementName: position

7. Query term: ::data design (::k)
   - attributeValue: data design OR textValue: data design

The document objects returned for Lucene query term constitute the result set for the single user query term. Each document can be considered as an XML document fragment rooted at the element at the tree location in the document. We can see that some user query types need to be mapped to two Lucene queries to get the required information from the index. This is due to the fact that the query language considers element names and attributes names to be interchangeable and secondly, keywords can appear in text nodes or in attribute values. The element node we are seeking will not be returned for some sub queries, the node returned will be in fact a descendent of the element node which we require. This is true for all the query types where we have to search by the “ancestorList”. For this case, we have to make sure that we consider the element node in the ancestor list as the answer. We can also see that no post search processing similar to what we might have to do for a Dewey order inverted index or an XPath inverted index is necessary. The only post search processing required in this case is for determining the tree location of an ancestor node. The combination of the indexing strategy and Lucene’s query language returns the intermediate result set to which post search processing can be done with minimal modifications to the intermediate results.
3.4 Post Search Processing

Multiple query terms in a query would result in multiple intermediate result sets after index access. For displaying the results to the user, the interconnected document fragments among the result sets have to be grouped together by applying the interconnection algorithms. The assumptions made while determining if two XML document fragments are interconnected and the factors which need to be taken into consideration while grouping document fragments together is explained in this Section.

3.4.1 Interconnection Assumptions

The algorithm by Thirunarayan and Immaneni for determining if two nodes are interconnected has been described in the previous Chapter. The implementation considered here for post search processing makes a subtle distinction to what was described in [6].

The primary aim of post search processing is to group together semantically related document fragments. There are instances where two result document fragments for separate query terms have the same tree location in an XML document, that is, they are the same element node. Thirunarayan and Immaneni consider two document fragments rooted at the same element node to be interconnected to each other. This implementation makes the opposite assumption, that is, two document fragments rooted at the same element node are not interconnected to each other. The distinction is explained with the following example.

Consider the XML fragment in Figure 5. The query author::,author::, elicits 4 query answers.[6]
The answers do not change for the query +author::, +author:: or +author::,
+author::, +author:: [6].

<publications>
  <book>
    <title>Modern Information Retrieval</title>
    <author>Ricardo Baeza-Yates</author>
    <author>Berthier Ribeiro-Neto</author>
    <chapter>
      <title>Digital Libraries</title>
      <author>Edward A. Fox</author>
      <author>Ohm Sornil</author>
    </chapter>
  </book>
  <article><title>The Anatomy of a Large-Scale Hypertextual Web Search Engine</title>
  <author>Sergey Brin</author>
  <author>Lawrence Page</author>
  </article>
  <article><title>An Algorithm for Suffix Stripping</title>
  <author>M.F.Porter</author>
  </article>
  <article><title>Indexing by Latent Semantic Analysis</title>
  </article>
</publications>

Figure 5: Heterogenous XML Document

For the above queries, the language does not seem to be capturing the intent of the user, which is, searching for document fragments which must have two or in the last case,
three authors. Making the distinction that two document fragments rooted at the same
element node are not interconnected can make the query language express the intent of
the user. The query +author::, +author:: over the same XML document will yield only the
two author pair nodes and remove the single author (M. F. Potter) result. The query
+author::, author:: on the other hand will include the single author result too. This is
because, only one author is required and the presence of the second author node is
optional. The query +author::,+author::,+author will not return any result as there are no
document fragments with three authors.

Making the above assumption alone is not enough as we can see for the XML
tree:

```xml
<A B="pWord qWord" />
```

The query, ::pWord, ::qWord, will not return any result as the result for both the queries
is the same element node. To solve this, the condition that the terms in the query should
be the same is added to the above assumption.

To summarize, for two document fragments returned for two query terms, if the
query has the same search terms(consider the publications example) and if the fragments
are rooted at the same element node, then they are assumed to be not interconnected.

### 3.4.2 Grouping Related Results

A user query can have multiple required and optional terms, in which case, in a result
displayed to the user, the following must be true for query satisfaction:

- For the required terms, for each result displayed to the user, a result document
  fragment corresponding to each query term must be present and, all the document
fragments in the result corresponding to the query terms must be interconnected to each other.

- For the optional terms in a query, for each result displayed to the user, it is not required for the optional terms to have corresponding result document fragments. If there is a result document fragment for an optional query term, it must be interconnected to the required query term results if they are present. If a result document fragment is not interconnected to another and both fragments are results for optional query terms, they are displayed as different results along with the required terms if any.

The grouping algorithms which use the `isInterconnected()` predicate introduced in [6] to group related nodes together are explained

### 3.4.2.1 Algorithms Explained

There are three cases to be considered in a user query. The cases being that, there are only required terms, there are required and optional terms and the last one being, there are only optional terms. The intermediate results for the query terms are differentiated into results for the required terms and results for the optional terms. There is no processing required for doing this as during the query parse time itself the difference in the query terms is noted and the results are stored in different sets. Further this implementation of the predicate `isInterconnected()` does not distinguish between optional and required terms as it does there.

Three algorithms for grouping the three possible cases were implemented. Although the overall approach was the similar for all three cases, there are differences in their requirements, and the separation of required and optional results before hand
enabled the use of three predicates instead of handling all the cases in one. Two of them, one for handling the required terms and one for handling the case where there are required and optional terms are explained here. The predicate for handling only optional terms is not explained, as one can simply remove a few conditions in the required and optional predicate to arrive at the optional only predicate.

The query which has only required terms is the most restrictive as a result document fragment has to be present for each term and all document fragments in a result have to be interconnected. The pseudo code modularized as the predicate, `handleRequiredTerms(resultLists)` expresses how interconnected document fragments were grouped together for the first case where there are only required query terms (Figure 6). The predicate takes as parameter, a list of intermediate result lists and returns the final result set which can be visualized as being a two dimensional array with each row being an individual result. Initially, each term in the first intermediate result list is added to a row in the final result set. The rest of the nodes in a row are added iteratively if they satisfy the interconnection relationship with the nodes already present in the row. In what follows, `isTermInterconnectedWithRow (resultNode, row, resultList, resultLists)` checks if a result node is interconnected to current nodes in the result row and returns true if yes, and false otherwise. The predicate `rowExists(newRow, finalResult)` checks if the terms in `newRow` are present in another row. It returns true if yes and `newRow` is not added to the final result set. The predicate `addCurrentNodes(row2, row1)` will add the nodes in `row1` to `row2`.

A similar approach is used when optional terms are present (Figure 7). The additional condition when compared to required only result is that we have to make sure
that there are no super sets or subsets of a new row present, before we add the row to the

```
finalResult := {}
foreach intermediateResultList in resultLists
begin
  if( intermediateResultList is first list )
  then begin
    foreach resultNode in intermediateResultList
    begin
      newRow := {};
      newRow + resultTerm;
      finalResult += newRow;
    end; //for
  end;
else begin
  newFinal := {}
  foreach resultNode in intermediateResultList
  begin
    foreach row in finalResult
    begin
      if(isTermInterconnectedWithRow(resultNode, row, resultList, resultLists))
      then begin
        newRow := {};
        addCurrentNodes(newResultRow, row);
        newRow += resultNode;
        if(!rowExists(newResultRow, finalResult) && rowExists(newResultRow, newFinal))
        then begin
          newRow += newResultRow;
          end; //if
        end; //for
      end; //else
    end; //for
  end; //else
  finalResult := newFinal;
end; //for
```

**Figure 6: Handling Required Terms**

final result set. That is, there should not be results already existing in the final result set
which have all the existing result nodes in this row. This is achieved via the predicates

`superSetExists(newResultRow, finalResult)` which returns true or false depending
on whether there exists a super set of `newResultRow` in `finalResult`. If the new row
has all the terms in some existing rows (sub-sets), then they should be removed from the
final result and this new row added. This is achieved via the predicate

`getSubSetIndexes(newResultRow, finalResult, finalListRowLocations, subSetIndexes)`. This predicate returns the indexes of the subset rows if they are present, in the
list subSetIndexes. The predicate `getNewFinal(finalResult, subSetIndexes)` removes the subsets from the final list result set and returns the new list. First the required result set is determined by invoking the predicate handling required results. The rows in this required set are iterated through and for all the terms in the optional list, the terms interconnected to all the required terms in a row are added to this row. If a optional term is related to all the required terms, but is not related to any one or more optional terms present in the row, a new row is created with the required terms and this current optional term and added to the final result set. This is done by the predicate `createNewRowReqAndOpt()`. The super set and sub set checks are done before this row is added to the final result set.

For all optional terms, there is no need to invoke the required terms predicate, and there is no need to iterate thorough a required row. The conditions specific to the required rows, mainly making sure they are always added, and making sure that the optional terms are interconnected to all the existing terms in a row are discarded. The resulting predicate is the one for handling only optional terms. All the requirements for subset and superset checks remain.
finalResult := {};  
requiredSet := handleRequired(requiredLists);  
finalListRowLocations := {};  

//Let loop counter be optListCnt
for each optList in optionalLists begin
    foreach optTerm in optList begin
        foreach reqRow in requiredSet begin
            if(isTermInterconnectedWithRow(optTerm, reqRow, optList, optionalLists)) then begin
                //For the loop below, the counter, optListCnt2 is initialized to optListCnt1+1.
                foreach optListTwo in optionalLists begin
                    foreach optionalTerm in optListTwo begin
                        if(isTermInterconnectedWithRow
                            (optTerm, reqRow, optList, optionalLists) &&
                            isInterConnected(optTerm, optionalTerm, optList, optionalList)) then begin
                            if(finalListRowLocations.size == 0) then begin
                                subsetIndexes := {};  
                                newResultRow := {};  
                                addCurrentNodes(newResultRow, reqRow);  
                                //substitute from below;
                                *checkForSuperSetsGetSubSetIndexesAndAdd
                                end;  
                            else begin
                                subsetIndexes := {};  
                                newLocList := {};  
                                foreach index in finalListRowLocations begin
                                    row = finalResult[index];  
                                    if (subsetIndexes.contains(index)) then begin
                                        //The predicate below, adds the required terms
                                        //to the newResultRow. It then adds only the
                                        //optional results which are interconnected to
                                        //the optionalTerm to the new row.
                                        //Finally the optionalTerm is added.
                                        newResultRow := createNewRowReqAndOpt();  
                                        //substitute from below
                                        *checkForSuperSetsGetSubSetIndexesAndAdd
                                        end;  
                                    end;  
                                end;  
                                //remove subset indexes from finalListRowLocations.
                                finalListRowLocations := getNewLocs(finalListRowLocations, subsetIndexes);  
                                //remove subset from final. Also return new indexes in newLocList
                                finalResult = getNewFinal(finalResult, subsetIndexes, newLocList);  
                                foreach newIndex in newLocList begin
                                    finalListRowLocations += newIndex;  
                                end;  
                            end;  
                        end;  
                    end;  
                end;  
            end;  
        end;  
    end;  
end;  
else begin
    //This means that optTerm is not interconnected
    //with any term in this optListTwo.
Figure 7: Handling Required and Optional terms
4. Search Tool Implementation and Results

4.1 Structural Diagram

The modules which make up the search tool are depicted below.

The search tool allows the user to perform the following functions:
• Uses the parameters specified in the configuration files to create and maintain indexes for XML document sets.

• Allows the user to specify queries conforming to the query language described, performs an index search and applies the interconnection algorithms to group together semantically related nodes.

• Provides a user interface for displaying the results with the interconnected nodes grouped together. It also allows the user to navigate through the XML document starting from the result document fragment.

4.2 Specific Details of Tool Components

4.2.1 Configurer

The configurer uses properties files for providing the parameters used by the modules of the search tool. The default document directory where the source documents are stored is specified. It stores information about where the indexes of the document set are located and this information is used by both the index and searcher modules. Further index configuration information is also stored in configurations files. This information can be changed depending on the system, allows the indexer to determine how much main memory access is permissible.

4.2.2 Index Controller

The main need for the index controller is to make the user interface agnostic to the parsing and indexing tool being used. The controller is also responsible for configuring the Lucene indexer with the help of the parameters in the configuration files.
It checks to see if the main index directory is present, and if not creates one. It maintains a reference to the XML document parser, and passes to it the XML file which needs to be parsed and indexed. The index-able document units are passed to the indexer module for persisting in an index.

4.2.3 XML Parser

To gather the document units for indexing, the input document set has to be parsed to extract the desired information. Further, the query language discussed in this work is DTD/Schema free. The two choices available for parsing XML data are the SAX parser and the Document Object Model (DOM).

SAX is event driven and raises an event when a node in an XML document is encountered. The advantage of this is that an XML document is handled linearly and there are no high memory requirements. Event handlers for handling the events thrown when element nodes, attributes and text nodes are encountered, have to be coded. Gathering the relationships between the elements and also extracting all the information pertaining to an element node, which is a requirement for the document unit chosen, becomes a tough task in this scenario.

It is for this reason that the DOM API was preferred. A tree representation of an XML document is loaded into memory and it can be traversed iteratively to extract the required information. The API also provides methods which make it extremely easy to gather all the required information concerning an element node. During the tree traversal, both the list of ancestors and the tree location for each element can be built. The information for each node is organized in a way similar to the indexing unit described previously. These index-able units are passed on to the index for persisting to an index.
4.2.4 Document Indexer

The Document Indexer takes as input the index-able units passed to it and creates indexes in the directory location which was set by the Index Controller. Each XML document has its index created and placed in a directory which has the same name as the source document. The indexer uses a Standard Analyzer for pre-processing the field values which are marked as TOKENIZED. Pre-processing includes case conversion, stop word elimination, removal of “s” from the ends of words, dots from acronyms etc. The indexer uses Lucene’s IndexWriter object to write to the index. The IndexWriter instance can be configured to use the system resources effectively. To speed up the indexing process, the hard disk access time should be minimized and the main memory should be used as much as possible. This can be done by setting the IndexWriter parameter MaxBufferedDoc to as high a value as possible. This parameter determines how many Lucene documents are held in memory before they are written to the hard disk. Another IndexWriter parameter which should be set is MergeFactor. The Lucene default is 10 which means, a new segment will be created for every 10 documents added. When the tenth segment of size ten is added, all ten are merged into a single segment of size 100. When ten such segments of size 100 have been added, these are merged into a single segment containing 1000 documents, and so on. Keeping the MergeFactor high will therefore prevent excessive disk access and speed up the indexing process.

4.2.5 Search Controller

The search controller is responsible for managing the search process. It configures the Lucene IndexSearcher instance with the location of the index. The query string
entered by the user is passed to the Query Parser and the individual query terms which it
gets back are now passed to the Search Helper module which takes care of the index
search. The Search Controller is aware of whether search terms and their corresponding
intermediate result sets received from the Search Helper are required or optional. The
appropriate post search grouping algorithm in the Interconnection checker is invoked
depending on the presence (absence) of required and optional terms in the query.

4.2.6 Query Parser

The query parser parses the search string entered by the user. It ensures that the
syntax of the query is right and separates the query into required and optional terms. A
Java bean is created for each query term and, the element node name, attribute name and
the keyword(s) entered in each term are stored as attributes. This information is used not
only for index search, but also for post-search processing.

4.2.7 Search Helper

This module checks the query terms in each query and determines the type of the
query, the query types being e:a:k, e::k, :a:k etc. Depending on the query type, the
appropriate method on the Index Searcher is invoked. The Index Searcher returns Hits
objects which are nothing but lists of Lucene Document objects. Another reason for the
query type to be kept track of is that there are result sets for sub queries to which some
modifications need to done as explained in Section 3.3.2. The modifications pertain to
sub queries where we search the ancestor list. The element node which is returned in this
case is the descendant of the element node which is the required answer. The Search
Helper passes a List of Hits (One Hits object per Lucene Sub query) objects to the XPath
Constructor module which constructs the XPath for the result sets taking into account the type of the Lucene sub query.

### 4.2.8 Index Searcher

The index searcher module is responsible for searching the indexes created in the root index directory. An instance of the Lucene class IndexSearcher is created for each index to be searched. Recall that each file has its own index and placed in a directory. Each IndexSearcher is initialized with the same Analyzer used during index time, which is the StandardAnalyzer. This analyzer performs the same lexical processing on the user query as was done for the index tokens. These IndexSearcher instances are placed in an Array of the Lucene type Searcher and a MultiSearcher instance is now created which can initiate searchers on all the indexes in the root index directory. Depending on the query type, methods are invoked on the Index Searcher by the Search Helper. The Index Searcher queries the index using the Lucene query language mappings described in Section 3.3.2. A list of Document objects is returned for each query on the index. These lists from the sub queries are grouped together and returned to the Search Helper. Lucene in fact ranks the results of each sub query, the order of the Document objects in the Hits list denoting the ranking. During post search processing however, this ranking is lost during the grouping of the result sets.

### 4.2.9 XPath Constructor

The XPath Constructor as the name suggests converts the tree location to an XPath which can be used by an XPath processor to extract the document fragment from an XML document. For constructing the XPath, the “position()” function is used. For
example, the XPath “/*[position() = 1]” returns the first child of the root node. This way
the tree location can be directly converted to an XPath. The more important role for this
module is to make sure that for Lucene sub queries on the ancestor list, the XPath
constructed is for the ancestor node, the reason for which has been explained before in
Section 3.3.2. It is a straightforward task of using the tree location string and the ancestor
list to construct the XPath for the ancestor in question. Instead of modifying the Lucene
Document objects, the XPath constructor creates Java Beans for each result in the Lucene
Hits list. In the case of the sub query on ancestor lists, the ancestor information which is,
its document ID, tree location, XPath, and its ancestor list, is added as attribute values to
the java beans. This result set is returned to the Search Helper which in turn returns it to
the Search Controller.

4.2.10 Interconnection Checker

The Search Controller uses the Interconnection checker for the post search
processing where related results are grouped together. This process which involves
applying the interconnection algorithms has been explained in the previous Chapter. This
module returns the final result, which can be visualized as a two dimensional array with
each row representing one result to the Search Controller.

4.2.11 Search User Interface

The user interfaces satisfies the requirements laid out in Section 3.1.2. It provides
a simple interface to enable the user to enter search queries and choose XML documents
to index. Choosing the “Index” button will open up the default document source
directory, from which files can be chosen (Figure 9). Entering the query and choosing
“Search” will display the results, either individually for single query terms or each result as a set of related documents for multiple query terms (Figure 10 and 11). The display will show hundred results at a time if the number of results is large. The “Next” and

![Figure 9: Choosing documents to Index.](image)

![Figure 10: Single Query term result](image)
Figure 11: Multiple query term result (Both required terms)

Figure 12: Navigation of the XML document by selecting ancestors
“Back” buttons at the bottom (Figure 10) allows the user to navigate among result sets. The initial result shown will display the document ID, tree location, XPath and the ancestor list. An individual result’s document fragment (more than one document fragment in the case of multiple query terms) will be displayed by choosing the “show” button. Users browse the XML document by choosing the ancestor name from a drop down. This will display the document fragment rooted at the ancestor node (figure 12). In the case of multiple query terms, ancestor nodes of any component result fragments of a single result can be chosen.

4.2.12 Document Extractor

This module is used to display the document fragments chosen by the user, either all the document fragments present in each result, or the fragment rooted at the ancestor node chosen by the user. This module uses the XPath and the document ID in the result to first retrieve the DOM node using a XPath processor. This node is recursively iterated through and displayed to the user.

4.3 Search Tool Evaluation and Performance

4.3.1 Search Tool Evaluation

This implementation of the search tool was tested on wide range of XML documents. The query answers confirm to the definition given in [6] which is, satisfying the requirement for precision, adequacy, and coherence.

The only difference between the implementation in [6] and the implementation presented here has been explained in Section 3.4.1. The results of this implementation for
examples 11, 12, 13, and 14 in Section 3.6 in [6] where the difference is apparent is shown here. Consider the same XML fragment, publications.xml:

For the queries \texttt{author::, title:: and +author::, +title}, the results

\textbf{Figure 13: Both optional query terms in query}

are the same as in [6]. This is due to the fact that, for both terms in the above queries, there will be no result fragments which have the same tree location. For the queries to determine the co-authors, we can see the distinction, as there will be result fragments which have the same tree location for the query terms. For the query, \texttt{author::, author::} where both the terms are optional, we can see that there are four results with the last result being the single author case (Figure 13). For the query \texttt{+author::, +author::}, we can see that there are only three results with the single author case eliminated and for \texttt{+author::, author::}, we again have four results with the single author case also included (Figure 14 and 15). This expresses the intent of the user and adds to the expressive power of the query language. For distinct query terms which
Figure 14: Both required terms

Figure 15: Both required and optional terms present
Figure 16: Distinct terms returning same document fragment

return document fragments rooted at the same node, the result is not suppressed as shown (Figure 16) for the query :pword,:qword over the tree <A B="pWord qWord"></A>

The following is an example of the user-in-the-loop search process. A user searching for the percentage of Catholics in the USA might use the query +country::usa,+::catholic,+::percentage: on the Mondial dataset. The from the result (Figure 17) one can see that the element “name” is used to denote the country and further there is a “religions” element with the attribute “percentage”. Since we do not need all the information displayed, we can change the query to +name::USA,+religions::catholic which returns the required nodes (Figure 18). Also the first query includes “percentage” in a query term, and note that there are more than one “religions” elements (Figure 17). Now, it is the same case as that for co-author nodes in the publications.xml example discussed before where, the same
Figure 17: Query refinement

Figure 18: Refined Query
Figure 19: Articles with at least six authors

Document fragments were returned for two query terms. The assumption that two results rooted at the same node are not interconnected prevents sub-set results from being present in this result.

For some searches, a little knowledge of the structure makes searching an easier task. For example, for the SIGMOD dataset, searches for articles which have at least six authors are shown in Figure 19. Here the knowledge that the authors have a position attribute and if there is an author at position six, there should be five more authors is used. The query used is:

+author:position:01, +author:position:06, +article: [57}
Data centric XML documents like the Mondial dataset contain world geographic information and have data in text nodes which is not natural language and which cannot be known beforehand. Further, the associations between elements are not straightforward as in a document centric data set. For example, each country, city etc in this dataset is denoted by a unique “id” attribute. This “id” is used to refer to child elements or for denoting relationships between elements. A country’s capital for instance is referred to by an attribute “capital” and has as value the “id” of the capital city. Searches on this dataset, brings out the expressive power of the query language. For example, to find the capital of the USA, the query “::USA, +:capital::”, returns the document fragments rooted at the “country: nodes among others. We can see what the “id” of the capital is, and a second query with a single term searching by the “id” value returns the required
Figure 21: Searching Mondial by attribute value

information (Figures 20 and 21). For searches by keywords alone, splitting the search term in a keyword into multiple search terms yielded coherent results. For example, the search query `+::survey of radio sources sally hales` on the NASA dataset, yielded 2302 results (Figure 22), while the query `+::survey of radio sources, +::sally hales` gave 25 results (Figure 23). One can also see that the second query due to applying the interconnection relationships was able to group related nodes.
Figure 22: Single keyword query

Figure 23: Single query split into two
### 4.3.2 Search Tool Performance

The indexing and search performance of the search tool has been tested with mainly the SIGMOD, Mondial, a truncated DBLP document and the NASA datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Index Size</th>
<th>Time for Indexing</th>
<th>MaxBufferedDocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGMOD</td>
<td>524 KB</td>
<td>2.04 MB</td>
<td>7.73 Sec</td>
<td>1000</td>
</tr>
<tr>
<td>Mondial</td>
<td>1.3 MB</td>
<td>5.21 MB</td>
<td>18.01 Sec</td>
<td>1000</td>
</tr>
<tr>
<td>DBLP small</td>
<td>1.6 MB</td>
<td>6.3MB</td>
<td>33.6 Sec</td>
<td>1000</td>
</tr>
<tr>
<td>NASA</td>
<td>23.8 MB</td>
<td>91.3 MB</td>
<td>6Min 3.6 Sec</td>
<td>1000</td>
</tr>
</tbody>
</table>

**Table 1: XML Datasets, their Indexing time and Index Size**

These three datasets represent both document and text centric XML data with the MONDIAL dataset being purely data centric and having more attribute nodes than text nodes, while NASA has a lot of textual data. DBLP and SIGMOD do not have a lot of information captured in attribute nodes and use element nodes to describe the data. All the tests were carried out with a HP Desktop with Pentium 4 CPU 3.20 GHz processor 1.49 GB RAM and 80 GB 5400 rpm hard drive running Windows XP. The indexing time required for indexing the datasets and the index size is shown in Table 1. We can see that the space requirement of this implementation is roughly one third of the requirement of the implementation in [6] for comparable datasets. This is mainly due to the indexing unit design adopted by this implementation as explained in the previous Chapter, and due to the decrease in the index size, the indexing time is also less. For the index process, the Java Virtual Machine was configured to use a maximum heap size of 900 MB using the runtime argument –Xmx900M. Increasing the MaxBufferedDoc to 10000 for indexing
the three smaller datasets actually increased the time required to index by one second. Due to memory constraints, the NASA data set could not be tested with this configuration. The MergeFactor was kept at 10000 for all the trials.

The time required to compute and display the answers depends on the type of the query and the datasets indexed in the index. Queries, in which individual query terms have huge number of hits, require more time to compute the interconnected results. For example for the query `author::, title::`, on the SIGMOD data, the terms had 3737 and 1504 hits respectively and took 78 seconds to compute the interconnected results using the machine mentioned above. For more realistic queries like the ones shown in the previous Section, the result generation time was not significant and was in the order of milliseconds or a few seconds.

An important factor is the order of the terms in the queries. This is significant due to the design of the algorithms used for post search processing as explained in the previous Chapter. For example, the query `+author::kim,+author::,+title::` and `+author::,+title::,+author::kim` return the same answer set, for a search over the index with all the above mentioned datasets indexed. But the first instance executed in 13 seconds, while for the second instance, the program was terminated after a few minutes (The number of author nodes was 21485 and title nodes 19112 and in the second case the interconnected results have to be determined for this pair first.). For the same queries applied over the SIGMOD index alone, the time taken for the first query was 60 milliseconds, and for the second case, it was 82 seconds. The use of optional terms takes significance in this regard. This is due to the fact that while required terms limit the number of possible results, optional terms do the opposite,
in that none of the individual results are discarded, and they increase the size of the final result

<table>
<thead>
<tr>
<th>Index Searched</th>
<th>Individual Query Term result set size</th>
<th>Total Time</th>
<th>Size of final result set</th>
</tr>
</thead>
<tbody>
<tr>
<td>All indexes</td>
<td>63 12 19112</td>
<td>70 ms</td>
<td>2</td>
</tr>
<tr>
<td>All indexes</td>
<td>11 21485 5464</td>
<td>1 sec</td>
<td>29</td>
</tr>
<tr>
<td>SIGMOD</td>
<td>996 139 1504</td>
<td>6 sec</td>
<td>139</td>
</tr>
<tr>
<td>Mondial</td>
<td>3 1311 101</td>
<td>350ms</td>
<td>628</td>
</tr>
<tr>
<td>Nasa</td>
<td>17 3 36</td>
<td>10ms</td>
<td>2</td>
</tr>
<tr>
<td>Nasa</td>
<td>2302 8</td>
<td>14ms</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 2: Query processing times

set. For example, the query with the second author optional, +author::kim,author::don,+title:: returns 61 results and took 1 second, while +author::kim,+author::don,+title:: returned 2 results and took 70 milliseconds to complete. The index searched included indexes for all the above mentioned datasets. Query terms which return huge number of results can be used in a search provided the terms are ordered so as to limit number of results first, and by careful placing of the optional terms.

Overall, for realistic queries which consider the above factors, the time taken to compute the interconnected nodes was acceptable. Further, the display time was always in the order of milliseconds as only 100 results are displayed at any point of time, and the user has to choose any one result and navigate through the tree. The query answer
generation time is shown in Table 2 for the realistic queries. The number of results for each term is shown and the index searched is specified. These search cases were typical of the realistic searches one can expect for the datasets in question.

4.4 Issues with the Current Implementation

After experimenting with various document collection and query types, the following concerns were noted.

One of the issues was that, some queries produce answers at a very low level of granularity. A good example being the search for :name:Vienna on the Mondial dataset returns the sub tree <name>Vienna</name>. The answer returned, forces the user to navigate around the returned document fragment to make sense of whether it is the relevant information. Allowing the user to choose the granularity of the result desired before hand will solve this problem.

When the number of results is large for each of the query terms in a multi query search, gathering the interconnected nodes is time consuming. Due to the locality of searches, indexing the interconnected nodes for certain queries can be considered. This node interconnection index approach has been taken in [5]. Indexing all the interconnected nodes in a document has very high space requirements. Therefore indexing the results for some queries like +author::, +author, while adding to the index size, can have very good time trade off and also improve the flexibility of the search tool.

In Example 14 of [6], we can see where the query language does not confirm to what can be considered as intuitive results expected. This is due to the fact that the query
language is XML schema/DTD free, and without more information about the META data itself, one cannot arrive at an “intuitively correct” answer.

Support for fuzzy searches needs to be included to improve the search experience. Fuzzy search support is needed for keyword search and can be helpful while searching highly technical documents like the NASA dataset. Fuzzy search support using ‘~’ symbol based on Edit-distance algorithm is supported by Lucene.

Allowing a user to search indexes for particular documents should be considered due to the fact that some XML element names are very common. For example, a search for the `author` nodes when the truncated DBLP, SIGMOD and NASA datasets are indexed returns 21485 results. This would make some queries like the query for co-authors prohibitively expensive. A combination of partial interconnected nodes index for each document along with the ability to choose the index to be searched makes the tool more practical.

A match highlighter is necessary for highlighting the query terms in the document fragments returned. This is a must for searching document centric XML documents like the NASA dataset. The “para” elements containing a lot of information had to be read through to determine if it was relevant.
5. Conclusion and Future work

5.1 Conclusion

The query language proposed by Thirunarayan and Immaneni has a simple syntax which allows the user to consider the meta data information in XML datasets in addition to the text data for formulating search queries. The language also takes into account the duality in the usages of elements and attributes in XML datasets. This work attempts to provide a reasonably efficient implementation satisfying the requirements of the query language. The tool was developed iteratively and its capabilities augmented to perform single query term searches, incorporating the notion of semantically related terms by implementing the interconnection algorithms, developing a GUI for further refining a query etc.

The tool was successful in achieving all the requirements laid out by the query language. Also, the indexing data structure and Lucene search strategy used by this implementation has a time and space efficiency which is acceptable for a prototype but requires improvement for a large scale system and is a contemporary research topic. The GUI allows the user to iteratively improve the search query and also browse the XML document set. The interface also cuts down on the result display time by displaying only a reasonable number of results at a time.

The search tool has been completely developed using Java and Java based
open-source libraries. It can therefore be deployed to both UNIX and Windows platforms.

5.2 Future Work

The search tool has to incorporate a few important improvements to develop into an effective component for searching XML datasets. Some of the most important enhancements needed are discussed below.

The query language makes use of the structure inherent in XML documents to retrieve related results. There are instances where the results which best capture the information needed are not displayed first and the user has to go through the result sets to find the “best fit” results. For document centric XML datasets using this query language for searching such datasets, one can retrieve the relevant information as shown before (Figure 23). But when the data in the text nodes is huge, traditional information retrieval challenges emerge. Therefore for document centric datasets there is a need for a ranking mechanism to rank the retrieved results.

For data centric applications which are highly structured and which have minimal information in text nodes, a ranking mechanism, at first glance does not seem to be add much functionality. But the tests on the search tool proved otherwise. For example, the query `+:name:India,+:percentage:,+::christian` over the Mondial database gave the “best fit” result as the twentieth one in a result set of 287. In the first few results, instead of the “country” node, one of the “organization” nodes which has an attribute “name” and the keyword “india” in its value was grouped with nodes returned for the second and third terms in the query (Figures 24 and 25). Tweaking the query to include
the following \texttt{+country:name:India,+\texttt{+}:percentage:+\texttt{+}:christian} would have eliminated the results with “organization” nodes, and in this case, the “best fit” result is the first result. A ranking mechanism would remove the necessity for such query refinements and the first query itself would order the result set such that the “best fit” result is on top with “organization” nodes occurring later in the result set.

Figure 24: "Best fit" result shown later

A composite rank of each related result in the result set needs to be computed. This composite rank should be calculated by considering the content in the text nodes (Document centric) and the distance between the related nodes in each result of the result set. The results can then be ordered by the ranking.

Incorporation of a ranking scheme would require a change to the index data structure used. Different data structures should be explored, and if possible extending the
existing data structure should be considered. Lucene’s term weighting and ranking scheme should be explored to see how it can be used.

Figure 25: Query answer occurring before "Best fit" answer

The possibility of indexing an interconnection index should be considered. To save time, a strategy of computing the interconnected terms at a given level of granularity during document index time can be developed.
6. References


