XML Fragments extended with database operators

Yosi Mass, Dafna Sheinwald, Benjamin Sznajder & Sivan Yogev
IBM Research Laboratory in Haifa
Haifa 31905, ISRAEL
{yosimass, dafna, sivany, benjams}@il.ibm.com

Abstract

XML documents represent a middle range between unstructured data such as textual documents and fully structured data encoded in databases. Typically, information retrieval techniques are used to support search on the "unstructured" end of this scale, while database techniques are used for the structured part. To date, most of the works on XML query and search have stemmed from the structured side and are strongly inspired by database techniques. In a previous work we described a new query approach via pieces of XML data called "XML Fragments" which are of the same nature as the queried XML documents and are specifically targeted to support the information needs of end-users in an intuitive way. In addition to its simplicity, XML Fragments represent a natural extension to traditional free text information retrieval queries where both documents and queries are represented as vectors of words and as such it enables a natural extension of IR ranking models to rank XML documents by context and structure. In this paper, we extend XML Fragments with database operators thus allowing both IR style approach together with database "structured" query capabilities.

Introduction

XML documents became quite popular in the last few years as a way to add semantic information to data. Consequentially numerous XML collections have emerged such as Medline[1] for medical data, IEEE[2] Journals, Wikipedia[3] and more. Envisioning more and more such collections, researches started to investigate new methods for XML Retrieval. Since XML documents are regarded as semi-structured, research for XML retrieval is dominated by "data" centric query languages from one hand and "document" centric IR approaches from the other hand (Fuhr & GrossJohan, 2001).

To date, as (Broder, 2002) observed, most of the work on XML query and search has stemmed from the database communities and from the information needs of business applications, as evidenced by existing XML query languages such as W3C's XPath (XPath, Berglund *et. al.,* 2003) and XQuery (XQuery; Boag S. *et. al.* 2003), which are strongly inspired by SQL (Chamberlin & Boyce, 1974). Lately the XQuery community realized the need to add more free text features to their data centric query language and the result was XQuery-FT (XQuery-FT, 2006) which adds Full Text capabilities to XQuery.

In a previous work we presented XML Fragments (Broder *et. al.,* 2004) as a different approach for XML retrieval, motivated by document centric needs. Our motivation was to define a simple yet powerful language that can be a natural extension of queries on Full text to queries on XML. XML Fragments further followed the QBE (Query By Example) paradigm (Zloof, 1977) where pieces of XML Fragments are used for querying XML collections. This allowed a natural extension of ranking methods from classical IR to the XML domain. In (Broder *et. al.,* 2004) we

[1] http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?DB=pubmed
[2] http://www.ieee.org/portal/site
[3] http://en.wikipedia.org/

described XML Fragments as pieces of well formed XML extended with some operators such as Phrase and +/- prefix on text and structure to allow more user control for defining her needs.

This model still been a document centric approach proved to be quite powerful in many applications. For example, in the INEX Initiative for the Evaluation of XML retrieval (INEX 2002-2006) queries are expressed as NEXI which is an XPath extended with the “about” predicate (Trotman & Sigurbjörnsson, 2004). We translated all those queries to XML Fragments and achieved top results using an implementation of XML Fragments (Mass *et. al.*, 2002). Recently, another work by Chu-Carroll (Chu-Carroll *et al.*, 2006) built a full system of Question Answering using the semantic capabilities of XML Fragments.

XML Fragments were recently implemented in real Search Engines, like IBM’s Lotus Worksplace[4], or OmniFind[5]. While used in several engines we got some requirements for strengthening XML Fragments with typical database operators. Those included support for full Booleans (and/or), better control on the XML document structure and in particular on father/child relation and better discrimination between queries on attributes vs. queries on XML tags.
In this paper we take XML Fragments one step further and add some database like operators to the language. We still keep the intuitive and query by example paradigm that characterizes XML Fragments but with more data centric features. This new XML Fragments model allows database features for exact matches from one hand, while supporting all powerful IR ranking and fuzziness from the other hand. We believe that this is a unique approach since we start with an IR like query and strengthen it with database query capabilities while other approaches like XQuery-FT or XIRQL (Fuhr N. & GrossJohann, 2001) start from a full structured SQL like query and add some Full Text capabilities to it.

The rest of the paper is organized as follows: in the next section we describe the syntax and give some examples of the new added operators and in the followed section we formally define the semantics of the extended language. We then describe an implementation of the full language on top of the Lucene search library. We conclude with summary and future directions.

XML Fragment – a powerful IR language

The formal syntax and semantics of XML Fragments is fully defined by (Broder *et al.*, 2004) while, here, we give only a short summary. Our main motivation in defining XML Fragments is to extend classical IR system in which the query and the document collection both consists of free text. We claim that the same can hold for XML collections and we suggest querying XML documents via pieces of XML documents or “XML Fragments” of the same nature as the documents that are queried. Returned results should be not only perfect matches but also “close enough” ones ranked according to some measure of relevance. XML Fragments are thus portions of valid XML, possibly combined with free text. For example, the following are valid XML Fragments:

```
1.     <element1 id=”123”> text1 text2 <element2> … </element2></element1>
2.     <element>…</element> <element>…</element>
3.     <element>…</element> text1 <element>…</element>
4.     text1 text2 text3
```

[4] http://www-142.ibm.com/software/workplace/products/product5.nsf/wdocs/homepage
[5] http://www.ibm.com/software/data/enterprise-search/omnifind-enterprise/

Since XML Fragments can have more than one root element, we add a dummy `<root>` element that wraps the whole query to get a well formed XML data. We can look at the XML query as a tree where each node is either an XML tag or a word. Intuitively the semantics of XML Fragments query Q is that a document D is a valid result for Q (or that Q is satisfied by D) if we can find a path from Q's root to one of its leaves that fully appears in D.

In order to allow more user control on XML Fragments and at the same time still keep their simple intuitive syntax, we augmented in (Broder *et al*, 2004) the XML Fragments with the following symbols:

- "+/-" prefix that can be added to elements, attributes or content. Prefixing an element with a "+" operator means that the Query subtree below that element should be fully contained in any retrieved document. Prefixing an element with "–" means that the Query sub tree below that element, should not exist in any retrieved document.
- "…" (Phrase) to enclose any free text part of the Query between quotes ("") so as to support phrase match.
- Relation terms – for parametric search. For example, the query:

  <book><year><.gt> 2000 </.gt></year></book>

  will return all books that were published after year 2000.
- Empty tag – serves as a kind of parenthesis `<> … </>` to group some query nodes together.

Extending XML Fragments with database operators

As described above, while deploying XML Fragments in some domains we got requirements for more database oriented features. We describe below the syntax we used for adding those requirements to XML Fragments and the full semantics of the extended language. Our design principle was to keep the nature of XML Fragments as an intuitive Query By Example paradigm so we added all the new extensions following this paradigm.

Boolean operators

In (Broder *et. al.*, 2004) we showed how the association of the + operator and the parenthesizes <>…</> gave the user the ability to express any Boolean constraints:

- The OR constraint between two contiguous fragment was obtained by default.
- AND constraint was achieved by the '+' operator.
- Complex full Boolean constraints were expressible by the introduction of the parenthesizes <>…</>.

While a default OR semantics between query terms is appropriate for IR usage as evident by (Holscher *et. al.*, 2000; Jansen *et. al.*, 2000), it turned out that in most search applications a default AND is assumed between fragments (Notess, 2003). To satisfy both worlds we added explicit AND/OR Booleans to XML Fragments while the semantics for an implicit set of terms/fragments is left to be implementation dependant.

Definition 1 (booleans): an **AndTerm** is an XML node of the form `<.and>Q</.and>` and an **OrTerm** is an XML node of the form `<.or>Q</.or>` where Q is an XML Fragments expression.

The semantics of an AndTerm is that all its children should exist in a returned Document while the semantics of an OrTerm is that it is enough that a single child will appear in a returned Document. The formal semantic is given in the sequel. We give now some examples.

Examples

All examples refer to the two XML documents Doc1 and Doc2 in Figures 1 and 2 respectively.

```
<Library>
  <Book isbn="1234">
       <Title>Art of computer</Title>
       <publication year="1968"/>
       <fm>
              <Author>
                 <first>Donald</first>
                 <last>Knuth</last>
              </Author>
              <Publisher>
                  Addison-Wesley
                  <State>Massachusetts</State>
              </Publisher>
       </fm>
  </Book>
</Library>
```

Figure 1 - Doc1

```
<Library>
  <Book>
       <isbn>1234</isbn>
       <Abstract>Setting of attributes for introducing databases </Abstract>
       <Title>Graph theory</Title>
       <publication year="1985"/>
       <Author>
              <first>Donald</first>
              <last>Knuth</last>
       </Author>
       <Author>
              <first>Roland</first>
              <last>Graham</last>
       </Author>
  </Book>
</Library>
```

Figure 2 - Doc2

The XML Fragment in Figure 3 below will retrieve Doc2 and not Doc1. The reason is that the `<.and>` operator requires that the two `<Author>` children should both exist. This is true only in the second Doc. and not in the first one.

```
<Book>
       <.and>
             <Author> Donald Knuth </Author>
             <Author> Roland Graham </Author>
      </.and>
</Book>
```

Figure 3 - AND operator

Note that the two terms under each Query `<Author>` tags are interpreted using the default semantics of the implementing Search Engine. So an implementation that assumes default AND semantics will return only authors that have both Donald and Knuth for the first author and Ronald and Graham for the second author. Another implementation that assumes default OR semantics will return papers that have two authors (because of the `<.and>` tag) but it can also return a paper in which the first author is "Gerald Knuth" since its enough that the `<Author>` tag will be Donald or Knuth.

The XML Fragment in Figure 4 below will retrieve both Doc1 and Doc2. The reason is that the `<.or>` operator requires only a single `<Author>` child to appear which is true for both Doc1 and Doc2 both having the `<Author>` Donald Knuth.

```
<paper>
       <.or>
             <Author> Donald Knuth </Author>
             <Author> Roland Graham </Author>
      </.or>
</paper>
```

Figure 4 - OR query

The example in Figure 5 below shows a combination of '+' operator and Booleans. It will retrieve only Doc2 since the '+' prefix on the first `<Author>` mandates that this Author in a matched document must exist.

```
<paper>
       <.or>
             +<Author> Roland Graham </Author>
             <Author> Donald Knuth </Author>
      </.or>
</paper>
```

Figure 5 – Combination of Booleans and '+'

Queries on attributes

In the base XML Fragments (Broder, 2004; Mass *et al.*, 2002), we supported queries on attributes using the XML syntax for attributes e.g. <book isbn="1234"/>. However as stated in (Broder 2004) –

> "*To simplify our notation we represent attributes as child elements, and thus the query "<book isbn="1234"/> " is equivalent to "<book><isbn>1234</isbn></book>"*

The query <book isbn="1234"/> would return so both Doc1 and Doc2 as they both have either a book attribute `isbn="1234"` (Doc1) or a child tag `<isbn>` with value 1234 (Doc2). The motivation was that from an IR perspective there is no much difference between an attribute child and a tag child. However from a Database perspective mainly in business-to-business applications, it is crucial to express explicit constraints precisely on attributes and not on tag children. XPath (XPath; Berglund *et. al.*, 2003), for example, support queries on attributes using the @ prefix. For example, the XPath query //book[@isbn='1234'] will select Doc1 which has an attribute named isbn with a value '1234' but will not select Doc2 even though it has a child 'isbn' tag containing the value '1234'.

In this paper we sharpen the definition of attribute queries to match only attributes and not children with same name.

Examples

The query

```
        <book isbn="1234"/>
```

will return only Doc1 and not Doc2 since in Doc2 "`isbn`" appears as a child tag of `<book>` and not as an attribute.

To support parametric constraints over numeric attributes we enable the following relation operators (<, >, <= and >=) on attributes. For example, the query:

```
        <book>
               <publication >= "1985"/>
        </book>
```

will return only Doc2 and not Doc1 since this query retrieves only books whose publication date was after 1985.

The depth operator

One key design element in our base XML Fragments was to allow fuzziness in the query. This was motivated by IR needs of users that don't know the exact structure of the queried collection. One such assumption was that we do not distinguish between a direct child and a descendant child. For example, the query:

```
<book>
       <Author> Donald Knuth </Author>
</book>
```

would return books authored by "Donald Knuth" regardless to the depth of the `<Author>` tag under the `<book>` element in the original document.

This relaxation was found insufficient for database oriented applications. It should be noted that in a database language like XPath, hierarchy constraints can be expressed by either a single slash (/) or a double slash (//). The XPath query `C1/C2` matches documents with node `C2` a direct child of node `C1` while the query `C1//C2` matches documents with node `C2` a descendant of node `C1`. More complex constraints may be defined like `C1/./C2` that matches a node `C2` two levels under `C1` etc.

To add such functionality to XML Fragments we add a new depth operator:

Definition 2 (depth): a **DepthTerm** is an XML node of the form `<.depth value="$n">Q</.depth>`, where $n is a positive value and Q is an XML Fragments expression.

A query with two nested tags separated by a `<.depth value = "$n">` tag will retrieve only documents in which the two requested tags are separated by `$n` levels while a query with two nested tags without any depth tags do not distinct between direct child or a descendant. This new syntax emphases our general guideline to have XML like syntax of simple IR oriented language with advanced operators that fit application or database oriented needs.

Examples

The query in Figure 6 will retrieve Doc2 and not Doc1 since only in Doc1 the tags `<book>` and `<Author>` are separated by 2 levels while in Doc2 `<Author>` is a direct child of `<book>`.

```
<book>
      <.depth value = "2">
            <Author> Donald Knuth </Author>
      </.depth>
</book>
```

Figure 6 – Depth query

A more sophisticated example is given in Figure 7 involving nested depth tags combined with Booleans. The query below will retrieve Doc1 since it has `<book>` and `<Author>` tags separated by 2 levels and the `<State>` tag appears 3 levels below the `<book>` tag which is the sum of the 2 `<depth>` expression in the query.

```
<book>
      <.depth value = "2">
          <.and>
            <Author> Donald Knuth </Author>
            <.depth value = "1">
                  <State> Massachusetts </State>
            </.depth>
          </.and>
      </.depth>
</book>
```

Figure 7 – Depth query with Boolean tags

Target Element

An advantage of queries on XML collection over traditional IR is that the granularity of returned results can be elements inside a document instead of full documents. This may be crucial in very large documents where retrieval of the whole document is useless. We denote those returned elements as *target elements*. In our previous paper (Broder *et al.*, 2004) we left the definition of target elements for future work; we complete the definition here. To let the user define the target element, we introduce a new operator #:

Definition 3 (target element): a **TargetElement** is an XMLTerm (An XML node with a real tag name e.g. <book>) prefixed by #.

Preceding a query tag name by the # symbol marks it as a candidate to be returned by the query.

Examples

The query in Figure 8 below will return the first `<Author>` tag from both Doc1 and Doc2 from Figures 1 and 2 respectively.

```
<book>
       <#Author> Donald Knuth </#Author>
</book>
```

Figure 8 – Target element

A more sophisticated and “database oriented” query is given in Figure 9: the query will retrieve the titles of the books authored by Donald Knuth.

```
<book>
       <Author> Donald Knuth </Author>
      <#title/>
</book>
```

Figure 9 – Target element

The mechanism to return the matched target elements is implementation dependant. In INEX, for example, the target element is returned by the XPath expression that leads to that element. Other implementation may choose to return for example an offset of the target elements in the XML document (Mass & Mandelbrod, 2004).

The above target element definition is much powerful than its equivalent in XPath where the target element is implicitly the last node of the query path. Our new definition allows to define any query tag and a multiplicity of them as target elements.

Semantic of XML Fragment Query

In this section we formulate the conditions by which a given document is said to satisfy a given XML Fragment query. We use the XML Document Object Model (DOM, 2004) that represents any XML document as a tree and same for the Query. For the documents to search we use the notation <s> to represent an arbitrary element name e.g., `<Author>`, and xs to denote a particular instance of <s> in a document which is represented by the unique XPath that leads to that instance of <s>, e.g. /Book[1]/Author[1]. We use xs in order to define the notion of components as stated in Definition 4.

> **Definition 4 (component):** Let D be an XML Document and <s> an element name. Let S = { *xs* | <s> exist in D } the set of all instances of <s> in D. For each $xs \in$ S, we define the component D_{xs} as the subtree of D whose root is the element <s> and is uniquely identified by *xs*. The entire document D is identified by its topmost element and for sake of simplicity we refer to it as D_x .

For example in Doc1 from Figure 1 above, we have `<Author>` as an element name and S={/library[1]/book[1]/fm[1]/author[1],/library[1]/book[1]/fm[1]/author[2]} its corresponding component instances.

Definition 5 (nesting): Let D_{xs} be a component uniquely identified by *xs*. We define nest(D_{xs}), the nesting of the component D_{xs}, as the length of the XPath expression *xs* identifying it. We define by length of an XPath expression, the number of the nodes involved in it.

We define now the concept of satisfiability for a Query as follows. Let D_{xs} be a document component and Q = (t_1, …, t_n) be a query. To simplify our notation, we refer to a single text term by WordTerm and to a phrase by a PhraseTerm. We then distinguish between XMLTerm (a term with a real tag name e.g. `<book>`), AndTerm, OrTerm, DepthTerm and RelationTerm. We verify whether a given query Q is satisfied by a given document component D_{xs} recursively by using the function Check_satisfiability, which is abstracted in pseudo-code in Figure 10 below.

```
Check_satisfiability(Q, $D_{xs}$) :

  1. if (Q is empty) return true;
  2. if (Q=($t_1$) and $t_1$ is a WordTerm with no +/- prefix): if $t_1$ appears in the content of some node
     in $D_{xs}$ then return true; otherwise return false;
  3. if (Q=($t_1$) and $t_1$ is a PhraseTerm with no +/- prefix): if all the phrase's words appear
     consecutively in the content of some node in $D_{xs}$ then return true; otherwise return false;
  4. if (Q=($t_1$) and $t_1$ is an AndTerm with no +/- prefix) : let $t_1$= <.and>$u_1$,…,$u_k$</.and>; if for each
     $u_i$, Check_Satisfiability($u_i$, $D_{xs}$) = true, then return true; otherwise return false;
  5. if (Q=($t_1$) and $t_1$ is an OrTerm with no +/- prefix) : let $t_1$= <.or>$u_1$,…,$u_k$</.or>; if ther exist $u_i$
     such that Check_Satisfiability($u_i$, $D_{xs}$) = true then return true; otherwise return false;
  6. if (Q=($t_1$) and $t_1$ is a DepthTerm with no +/- prefix (*)) : let $t_1$= <.depth val="n">Q'</.depth>;
     if there exist some element <s'> in $D_{xs}$ such that nest($D_{xs}$) + n = nest($D_{xs'}$) and
     Check_Satisfiability(Q', $D_{xs'}$) = true then return true; otherwise return false;
  7. if (Q=($t_1$) and $t_1$ is an XMLTerm with no +/- prefix) let $t_1$= <s'>Q'</s'>; if there exists some
     element <s'> in $D_{xs}$ such that Check_satisfiability(Q', $D_{xs'}$)=true then return true ; otherwise
     return false;
  8. if (Q=($t_1$) and $t_1$ is an XMLTerm with an attribute with no +/- prefix) : let $t_1$= <s'
     att="val">Q'</s'>; if there exists some element <s' att="val"> in $D_{xs}$ such that
     Check_satisfiability(Q', $D_{xs'}$)=true then return true ; otherwise return false;
  9. if (Q=($t_1$) and $t_1$ is a RelationTerm with no +/- prefix) : if $D_{xs}$ is of the form <s>val1<s> and
     val1 is a number then
     If $t_1$ = <.gt.>Val</.gt.> and $Val1 > Val$ then return true
     Else If $t_1$ = <.ge.>Val</.ge.> and $Val1 \geq Val$ then return true;
     Else If $t_1$ = <.lt.>Val</.lt.> and $Val1 < Val$ then return true
     Else If $t_1$ = <.le.>Val</.le.>  and $Val1 \leq Val$ then return true;
     Otherwise return false;
  10. if (Q=($t_1$,…,$t_n$), n>1) :  let $Q^+$ be all the "+" terms (with the "+" removed), $Q^-$ all the "-" terms
     (with the "-" removed) and $Q^o$ the rest of the terms.
         If (for each $t_i \in Q^-$ we get Check_satisfiability(Q=($t_i$), $D_{xs}$) = false and
            for each $t_i \in Q^+$ we get Check_satisfiability(Q=($t_i$), $D_{xs}$) = true ) and
            for $Q^0$ we either apply AND semantics or we apply OR semantics and we get true)
          return true;
     Otherwise return false;
```

Figure 10 – Check_satisfiability method

(*) For ease of reading we assume that a DepthTerm can have only a single XMLTerm node. This assumption can be relaxed and then we need to modify the pseudo code above to recurse over a set of potential $\{D_{xs}\}$ elements in the desired depth.

Semantics of a Target Element

Once a document D is proved to satisfy a query *Q* we can find all the occurrences of its *Target Elements.* These occurrences are defined as follows: Let *t* be a target element in the query (i.e., it appears as `<#t>` in the query text), and let T = {*xt*} be the set of all instances of `<t>` in document D. An instance *xt* in T is called an *occurrence of target element t,* if there is a proof that document *D* satisfies query *Q*, a proof in which *xt* participates as an XMLTerm, i.e., by virtue of item 7 of Check_satisfiability()**.**
In a less formal way, we can say that *xt* is an *occurrence of target element t*, if *xt* participates in an occurrence, within document *D*, of the query twig associated with query *Q*.

A reference implementation of XML Fragments

We describe now an example implementation of XML Fragments in the popular open source Apache Lucene[6] search engine library which was originally written in Java, and is now available in various programming languages. In order to allow XML Fragment queries over Lucene, it was necessary to implement three components:

1. An XML parser which indexes XML documents both their textual and structural data into an inverted index.
2. An XML Fragment query parser,
3. A Runtime search algorithm to evaluate XML Fragment queries.

We have implemented these components in Java, and will hereby describe some aspects of the query parser implementation. The indexing and the runtime search algorithm are outside the scope of this paper, and will therefore not be described.

The tree structure of the XML Fragment is implemented using Lucene's `BooleanQuery`[7] class: all internal nodes of the XMLFragment query are `BooleanQuery` objects. Each `BooleanQuery` object contains an array of sub-queries (`BooleanClause`[8] objects), where each clause is either a nested `BooleanQuery` object, or another Lucene `Query`. Boolean constraints in the query are implemented using `BooleanQuery` as described below, while tags and attributes extend it by keeping an additional encapsulated `Query` object with the tag or attribute name.

An important characteristic of a clause in a `BooleanQuery` is its occurrence state, which can be MUST, MUST_NOT or SHOULD. The first two states correspond naturally to our +,- symbols. In addition, the `BooleanQuery` has a minimal number of SHOULD clauses that must be satisfied. XML Fragments Boolean nodes can be thus implemented as follows:

[6] http://lucene.apache.org/java
[7] http://lucene.apache.org/java/docs/api/org/apache/lucene/search/BooleanQuery.html
[8] http://lucene.apache.org/java/docs/api/org/apache/lucene/search/BooleanClause.html

- AND is a `BooleanQuery` with all contained clauses assigned MUST,
- OR is a `BooleanQuery` with all contained clauses assigned SHOULD, and the minimal number of SHOULD clauses that must be satisfied set to 1.
- Tag or Attribute are `BooleanQuery` with contained clauses assigned MUST or SHOULD according the modifier +/- preceding them and the default semantic of the search engine.

Our implementation of the XML Fragment parser encapsulates a Lucene query parser (given upon construction of the XML Fragment parser), which is used in two different contexts:

- As mentioned above, Lucene supports queries with either `AND` or `OR` semantics. The encapsulated Lucene query parser comes with a default operator which can be either `AND` or `OR`, and the semantic of the XML Fragment parser is set according to this operator.
- The textual parts of the XML Fragment query are parsed by the Lucene query parser, and the returned Lucene query is processed by the XML Fragment parser and added to its query tree.

Using OR semantic, the XML Fragment query parser will therefore parser the query *q* given in Figure 11 below:

```
<book year = 1968>
 +<Author> Donald Knuth </Author>
        <.depth value = "1">
             <title> "art of computer" </title>
        </.depth>
</book>
```

Figure 11 – Query q

into the following query:

```
Contains BooleanQuery(minimum=0), ElementQuery:"book"
       MUST Contains BooleanQuery(minimum=1), ElementQuery:"author"
              SHOULD TermQuery:"Donald"
              SHOULD TermQuery:"Knuth"
       SHOULD Contains(Depth=1) BooleanQuery(minimum=1), ElementQuery:"title"
              SHOULD PhraseQuery:"art of computer"
       SHOULD Contains BooleanQuery(minimum=0), AttributeQuery:"year"
              MUST TermQuery:"1968"
```

The usage of Lucene's existing query parser in free-text parsing complies with the notion of XML Fragments being an extension of free-text queries, in that the XML Fragment query parser has special treatment only to the structural parts of the query. We further elaborate this notion in cases where the query contains only free-text without structural limitations. In such cases the XML Fragments parser does not change the query, and the runtime algorithm uses the native Lucene query when scoring and ranking documents.

Another important benefit gained from using Lucene's query parser is the easy addition of features to the free-text part of XML Fragment syntax definition given above. Two examples that we implemented are wildcard queries and fuzzy queries, but every existing or future Lucene query type can be added. In fact, similar to the definition of the default semantic, the free-text portion of XML Fragments can be search engine dependent, supporting all features of the search engine query syntax.

Conclusion and future works

We presented XML Fragments as a query paradigm for document centric XML retrieval and extended it with some database oriented operators. This approach matches the general attempt to narrow the gap between search and retrieval capabilities over unstructured, semi-structured and structured data. (Raghavan & Garcia-Molina, 2001). Our approach is quite unique in the sense that we start from IR oriented query and add database oriented operators to it while other known approaches such as XQuery-FT start from a database oriented query language and add Full Text capabilities to it. The resulted XML Fragments query is still intuitive and follow the Query By Example paradigm for a novice user yet it keeps the same structure for the advanced added database oriented operators.
As a next step we plan to complete the Lucene based implementation of XML Fragments and deploy it in several IR and database oriented applications. Based on user feedback we will consider adding more operators to the language.

References

Berglund A. , Boag, S. ,Chamberlin D. , Fernandez M.F. ,Kay M. , Robie J. & Simeon J. (2003). "XML Path Language (XPath) 2.0 . *W3C Working Draft*, 12 Nov 2003. See http://www.w3.org/TR/xpath20/

Boag S. , Chamberlin D. , Fernandez M.F. , Florescu D. , Robie J. & Simeon J. (2003). "XQuery 1.0: An XML Query Language", *W3C Working Draft*, 12 Nov 2003. See http://www.w3.org/TR/xquery/

Broder A. (2002). *A taxonomy of Web search*, SIGIR Forum, 36:2, Fall 2002.

Broder A., Maarek Y.S, Mandelbrod M & Mass Y. “Using XML to Query XML – From Theory to Practice”. *In Proceeding of RIAO,* 2004

Chamberlin D., Boyce R. “SEQUEL: A structured English query language”, Proceedings of the ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control, 1974.

Carmel D. , Maarek Y. , Mandelbrod M. , Mass Y. & Soffer A.(2003). "Searching XML Documents via XML Fragments". In the *Proceedings of SIGIR' 2003*, Toronto, Canada, Aug. 2003.

Chu-Carroll J, Prager J, Czuba K, Ferrucci D & Duboue P (2006). “Semantic Search via XML Fragments: A High-Precision Approach to IR”. In the *Proceedings of SIGIR’ 2006*, Seattle, Washington, Aug. 2006.

DOM (2004). Document Object Model http://www.w3.org/DOM/

Fuhr N. & GrossJohann K. (2001). “XIRQL: A Query Language for Information Retrieval in XML Documents”. In *Proceedings of SIGIR’2001*, New Orleans, LA, 2001

Fuhr N. & GrossJohann K. (2001). “XIRQL: A Query Language for Information Retrieval in XML Documents”. In *Proceedings of SIGIR’2001*, New Orleans, LA, 2001

Holscher C., Strube G. (2000) “Web search behavior of Internet experts and newbies” In *International Journal of Computer Networks*, Volume 33, 1-6, 337-346, June 2000

INEX. Initiative for the Evaluation of XML Retrieval, 2002-2206, http://inex.is.informatik.uni-duisburg.de/

Jansen B.J., Spink A. & Saracevic T. (2000) “Real life, real users and real needs: A study and analysis of user queries on the Web”. In *Information Processing & Management*. Volume 36, 207-227

Mass Y., Mandelbrod M., Amitay E., Carmel D., Maarek Y. & Soffer A. (2002). "JuruXML - an XML Retrieval System". In *Proceedings of INEX'02*, Schloss Dagstuhl, Germany, Dec. 2002

Mass Y. & Mandelbrod M. , Component Ranking and Automatic Query Refinement for XML Retrieval, Advances in XML Information Retrieval, LNCS 3493, INEX 2004, Dagstuhl Germany, December 2004, pg. 73-84

Notess G.R. (2003) “Search Engine features chart”. http://searchengineshowdown.com/features/

Raghavan S. & Garcia-Molina H (2001)., “Integrating diverse information management systems: A brief survey”. In *IEEE Data Engineering Systems.* Volume 24(4): 44-52 – 2001

Trotman A. & Sigurbjörnsson B. “Narrowed Extended XPath I”. In *Proceedings of the INEX 2004 Workshop, 2004*

XPath – XML Path Language (XPath) 2.0, http://www.w3.org/TR/xpath20/

XQuery – XML Query (XQuery), http://www.w3.org/XML/Query/
XQuery-FT - XQuery 1.0 and XPath 2.0 Full-Text, W3C Working Draft 1 May 2006, http://www.w3.org/TR/xquery-full-text/
Zloof M. (1977). Query by example. IBM Systems Journal, 16(4):324-343, 1977)

Appendix A - Query Syntax for extended XMLFragment.

We assume that the reader is familiar with XML and we give now a formal definition of the Extended XML Fragments. We use *V** or *(V)** to denote a sequence of zero or more items of type *V* . *V?* or (*V)?* to denote a sequence of exactly zero or one items of type *V*, and *V+* or *(V)+* to denote a sequence of one or more items of type *V*.

```
XMLFragment          ::= (XMLTerm*  PhraseTerm*  WordTerm* BooleanTerm* )*

XMLTerm[9]           ::= Space Operator? Stag Content Etag | Space Operator?StagNoBody
BooleanTerm          ::= Space Operator? SBool Content EBool
PhraseTerm           ::= Space Operator?Phrase
WordTerm             ::= Space Operator?Word

Content              ::= RXMLFragment* | RelationTerm*
RXMLFragment         ::= SDepth? [10] XMLTerm EDepth? | PhraseTerm | WordTerm | BooleanTerm
RelationTerm[11]     ::= Space Operator?SRelation Number ERelation

Operator             ::= Plus | Minus
Plus                 ::= '+'
Minus                ::= '-'
Word[12]             ::= Sequence of zero or more characters without white spaces
Number               ::= A real number
Phrase               ::= Quote (Word Space)* Word Quote


Stag                 ::= StartTag TagContent CloseTag
TagContent           ::= TargetElement? TagName Space (Attribute Space)*

STagNoBody           ::= StartTag TagContent  CloseTagNoBody
Etag                 ::= EndTag TagName CloseTag

SBool                ::= StartTag BooleanTag CloseTag
Ebool                ::= EndTag BooleanTag CloseTag
BooleanTag           ::= AndTag | OrTag
AndTag               :;= '.and'
OrTag                ::= '.or'

SDepth               ::= StartTag DepthStr  = 'Int' CloseTag
EDepth               ::= EndTag DepthStr CloseTag
DepthStr             ::= '.depth'
Int                  ::= A positive integer value

SRelation            ::= StartTag RelationName CloseTag
ERelation            ::= EndTag RelationName Space* CloseTag
TagName[13]          ::= Word
RelationName         ::= Greater | GreaterEqual | Less | LessEqual

Attribute            ::= Operator?Word Equals Phrase  |  Operator?Word RelationOperator Number

RelationOperator     ::=  '=' | '>=' | '<=' | '>' | '<'
TargetElement        ::= '#'
```

[9] To have a well formed XML fragment the TagName in Stag and Etag must be equal
[10] To have a well formed XML fragment the SDepth must be followed by an EDepth
[11] To have a well formed XML fragment the RelationName in SRelation and ERelation must be equal
[12] For simplicity we use an intuitive definition of Word here
[13] For simplicity, we do not give here a formal definition of what must an XML tag name be.

```
StartTag            ::= ‘<’
EndTag              ::= ‘</’
CloseTag            ::= ‘>’
CloseTagNoBody::= ‘/>’
Equals              ::= ‘=’
Greater             ::= ‘.gt.’
GreaterEqual        ::= ‘.ge.’
Less                ::= ‘.lt.’
LessEqual           ::= ‘.le.’
Space               ::= ‘ ‘ (‘ ‘)*
Quote               ::= ‘ " ‘
```