

Algorithms and data structures to support the efficient implementation of a subset of the XQuery language

Subcontract within the Project QUESTION-HOW signed with ISTI-CNR Pisa, Italy.
(Project/Contract No. IST-2000-28767)

Subcontractor: Prof. Paolo Ferragina, Dipartimento di Informatica, University of Pisa, Italy.

1. The goal

The aim of this study was to improve and extend the functionalities offered by the *XCDE Library*, developed at the “Dipartimento di Informatica” by the algorithmic group of Prof. Paolo Ferragina, toward the support of a sophisticated query language which mixes some properties of XQuery (<http://www.w3.org/TR/xquery/>) together with some features proper of the IR-languages. This research direction has been recently suggested by the W3C with the document <http://www.w3.org/TR/xquery-full-text-requirements/>. This is, as far as we know, one of the first attempts to implement that proposal via a compressed index for XML documents. The resulting query language allows the user to perform sophisticated string queries over XML documents: proximity among multiple words, search by regular expressions or error-based matches, *snippet* extraction to select the context of a query occurrence, substring/prefix/suffix word searches, and many other word-based queries. In the following sections we recall briefly the features of the XCDE Library and then detail the query language that we have implemented on the top of it. We will also provide some running examples to better explain those functionalities.

The source code and documentation of the library can be downloaded from the site: <http://butirro.di.unipi.it/~ferrax/xcde/xcdelib.html>. The library is written in C and its use is free for non-commercial purposes.

2. The XCDE Library

The library provides a set of algorithms and data structures for indexing and searching a collection of XML documents. The documents must be *well-formed* and may be *heterogeneous* in that they may reflect different DTDs or XML Schema. The library supports the storage and management of these XML files in native form by operating directly at the File System level. The main features are: state-of-the-art algorithms and data structures for text indexing, compressed space occupancy, and novel succinct data structures for the management of the hierarchical structure of an XML document.

The main indexing data structure is the classical Inverted List where proper compression techniques are used to keep the posting lists in succinct space. Words, tags, attributes and their values are indexed using independent indices of this type. As a result prefix, suffix, substring, regular expression, approximate and proximity searches on the textual

content of the XML document as well on the attribute values can be executed in an efficient way. Resolving structural queries on tag paths can be also done efficiently by using a novel indexing of the hierarchical structure of the XML document based on geometric data structures.

The original text is compressed in such a way that random accesses to some of its parts are supported without decompressing the whole file. This is helpful for fast displaying the context of a query occurrence when visualizing the query results (usually called the *snippet* of the occurrence). Overall both the compressed XML document plus all of its indices occupy about the original file size. This is a significant space reduction with respect to other XML native indexers, like NATIX, for which the factor is at least 3, or is much less than the indexers based on Relational or OO Databases which abundantly surpass the factor 10. Clearly space reduction has an impact on the final performance because it allows to increase the disk bandwidth and to better exploit the buffering/caching strategies of the underlying OS and the hierarchy of memory levels. Furthermore, space reduction plays a crucial role in applications which run on small-memory devices like, for example, PDA and e-book.

One of the basic choices of the XCDE library is to operate at the granularity of the *single document*. This choice has one significant disadvantage and various advantages. It is clear that this fine granularity makes the searches on collections formed by many small documents much time consuming. However, on the other hand, single file indexing allows to easily implement distributed indices, to trivially update the index as a result of insertion/deletion of a document, to easily take care of the heterogeneity of XML documents, and also, to customize the indices according to the individual document features. It goes without saying that we might still index numerous XML documents without incurring in a significant query slowdown, in two different ways: either by "merging" small documents into bigger ones in order to reduce their overall number, or by adding a "light" global index which filters out some parts of the collection before the search on single documents is executed. The former approach might deploy a properly devised *xml tagging* which preserves the individuality of each document within a unique physical file containing the whole collection.

The library provides an API with a rich set of C functions to operate on its whole collection of data structures and algorithms. It may implement most of the basic functionalities of XQuery, and support more complex IR-like searches. The aim of its design is to manage efficiently and effectively XML documents that contain significant textual parts and/or a deeply nested hierarchical structure, and whose attribute values are complicated strings of numbers and letters. This is the typical scenario encountered in literary texts tagged via XML-TEI, one of our motivating application. The single-document *granularity* offered by our indices perfectly fits in this framework because it easily allows the humanists to restrict the searches on portions of the collection according to some criteria specified at query time: writing period, writing type, author, collections,...

Of course, the XCDE library is intended just as a *kernel* of a more complex *XML document engine*. It has been developed with the aim of providing researchers with a basic library for designing more complex applications. In the next section we discuss how its functionalities have been deployed to implement a sort of a mixed query language: XQuery + IR. The driving document is <http://www.w3.org/TR/xquery-full-text-requirements/>.

3. The Query Engine

In order to issue a query onto the XCDE indexes, we need to build them by means of the command:

```
xcde_build [-v][-c CONFIGURATION_FILE_NAME] [-f] -t -d filename
```

The result is a set of files containing the compressed document and the indexing data structures. One novelty with respect to the original XCDE Library is the design of another compressed data structure (option **-d**) that stores the *depth* of every element in the XML tree. This way, queries involving the *distance* between descending elements can be resolved efficiently.

After the indexes have been built, users can issue queries over the original XML document by using the *XCDE query engine*. XQuery is a powerful query language but it seems to be designed having in mind XML documents derived from *structured* data. Conversely XML is *text based* and thus its query language should be oriented mainly to the processing and management of textual data. Our goal has been therefore to design a query language which is *simple* to be used and *IR* oriented. Its specialties are the following.

- The query syntax is similar to SQL: **SELECT-FROM-RETURN**, but here the SELECT clause is specified by means of an XML piece of *well-formed text*. As a result, every user which knows a little bit of XML can formulate the query without being forced to learn the XPath syntax, as XQuery requires. Most of the IR functionalities detailed in <http://www.w3.org/TR/xquery-full-text-requirements/> have been implemented as well other powerful string-based queries are supported, like regular expressions and error matches.
- The output of the query (the *snippet*) can be formatted within the RETURN clause which, again, includes an XML piece of *well-formed text*. The key tool to build that piece of XML text is a *special attribute* (called hereafter *pivot*) whose name is *xml_var*. This attribute is added to elements within the SELECT clause in order to identify some “interesting points” in each document subtree that matches the query. The pivots are then used in the RETURN clause to indicate the way in which these points in the matching subtree must be visualized. More than one pivot can be specified within the SELECT clause, without therefore using complicated combinations of XPath expressions. Another specialty of the snippet extraction process, which we detail below, is the possibility for the user to define the size of

the snippet to be extracted, the presence or not of the tags, the well-formedness of the snippet, the retrieval of all elements including a given document part, etc..

The query may be issued by means of the following command:

```
xcde_search2 [-p] [-f FILENAME] query_expression
```

By using the options **-p** and **-f** is possible to change the behaviour of the query engine:

-f FILENAME: this option allows the user to load the query from a file.

-p: this option indicates to the query engine that the position of the document subtrees satisfying the query must be returned.

The **query_expression** is written as a well-formed XML piece of text drawn on the same set of elements and attributes of the queried document plus some special elements that allow to specify the IR functionalities and the snippet extraction process, as detailed below. The query expression may further use a RANGE clause which allows to specify the range of results to be returned to the user.

Query	::=	SELECT (Element SpecialElement) FROM NameFile+ RETURN ExprReturn+ RANGE Range;
Expression	::=	(Element SpecialElement)*;
Element	::=	StartTag Expression EndTag;
SpecialElement	::=	(SpecialElement)+ Literal WordSearch <xml_or> (Element SpecialElement)(Element SpecialElement)+</xml_or>; <xml_proximity xml_maxprox = '(1 9)(0 9)*'> (WordSearch) (WordSearch)+ </xml_proximity> <xml_anyvalue (Literal = 'TypeSearch: Literal ')+> Expression </xml_anyvalue> <xml_not [Pivot]?> (Element SpecialElement) </xml_not>
StartTag	::=	< NameTag Attribute* >;

NameTag	::=	Literal;
Attribute	::=	Literal = 'Literal' Pivot xml_dist = '(1 9)(0 9)*';
Pivot	::=	xml_var = '\$Literal';
WordSearch	::=	< xml_exact [Pivot]? [xml_case = 'CaseValue']? > Literal </ xml_exact > < xml_prefix [Pivot]? [xml_case = 'CaseValue']? > Literal </ xml_prefix > < xml_suffix [Pivot]? [xml_case = 'CaseValue']? > Literal </ xml_suffix > < xml_contained [Pivot]? [xml_case = 'CaseValue']? > Literal </ xml_contained > < xml_regexp [Pivot]?> RegExpr </ xml_regexp > < xml_error xml_maxerr = '(1 8)*'> Literal </ xml_error >;
CaseValue	::=	sensitive insensitive;
TypeSearch	::=	prefi prefs sufi sufs conti conts err :(1 9)* exi exs ;
EndTag	::=	</ NameTag >;
Literal	::=	String;
NameFile	::=	Literal;
ExprReturn	::=	\$Literal Literal StartReturnTag ExprReturn EndTag < xml_context AttrReturn> \$Literal </ xml_context >;
StartReturnTag	::=	< NameTag [Literal = 'Literal']* >;
AttrReturn	::=	xml_size = '[1-9][0-9]*' [xml_format = '(tag notag)']? [xml_view = 'backward forward bidir full stop_parent']?;
Range	::=	(1 9)(0 9)* (1 9)(0 9)*;

Let we detail the [WordSearch](#) elements, each of which may include one string whose type of search is indicated as follows.

<code><xml_prefix></code>	prefix match for the included string
<code><xml_suffix></code>	suffix match for the included string
<code><xml_contained></code>	substring match for the included string
<code><xml_regexp></code>	the included string is a regular expression following the <i>grep</i> -like syntax. The indicated word has to be searched case sensitive
<code><xml_error></code>	The included string is searched for a match with errors whose number (at most 8) is indicated as value of the attribute <code>xml_maxerr</code> . An error is a substitution, insertion or deletion of a character into the included string. The indicated word has to be searched case sensitive
<code><xml_exact></code>	exact match for the included string

All these elements can include the attribute `xml_case` which specifies if the indicated word has to be searched case (in)sensitive. If this attribute is not present the default behaviour is to consider case insensitive searches. In the case that no element is specified to denote the type of search for a string indicated by the user, the default behaviour is exact match.

With a similar syntax is possible to specify the type of search (`TypeSearch` elements) within an attribute value.

<code>prefi prefs</code>	Prefix of the attribute value, insensitive or sensitive
<code>sufi sufs</code>	Suffix of the attribute value, insensitive or sensitive
<code>conti conts</code>	Substring of the attribute value, insensitive or sensitive
<code>err</code>	Match (sensitive) with the attribute value with errors, at most 8
<code>exi exs</code>	an exact token (insensitive or sensitive). The default behavior, if any type of search is specified, is to consider an exact match

There are others special elements (`SpecialElement`) that can be used in the query expression within the SELECT clause to define a matching subtree.

<code><xml_or></code>	Boolean OR among at least two operands. Attributes <code>xml_dist</code> in its children will be ignored
<code><xml_proximity></code>	search for a set of words within the distance indicated in its attribute <code>xml_maxprox</code>
<code><xml_anyvalue></code>	search for elements that contain the attributes specified within <code>xml_anyvalue</code> . The type of searches on attribute values have been indicated above

<xml_not>

Boolean NOT on one operand

Inside a (special) element, say E, is possible to insert the attribute `xml_var = '$literal'` which defines what has been previously called a *pivot*. Any matching subtree binds the occurrence of the (special) element E, with the variable `$literal`. In the case that E is an element of the XML document, every occurrence of `$literal` in the RETURN clause is substituted with E's content in the matching subtree. If E is instead a *special element*, every occurrence of `$literal` in the RETURN clause is substituted with the matching word identified by this special element. To allow some flexibility in formatting the output, a user can include in the RETURN elements which are different from the pivots; in this case, these elements are left unchanged. Moreover, if a `$literal` occurs in the RETURN clause but it doesn't occur in the SELECT clause, then an error message is raised. Finally, we observe that it is also possible to insert within an element E the attribute `xml_dist = 'x'` which allows to select the occurrences of E that have distance at most x from its parent in the query. For example if the distance is set to 1, this means that E must occur as a child of the parent specified in the SELECT clause. Of course, the absence of the attribute `xml_dist` indicates that E must occur as a descending element of its parent specified in the SELECT clause.

For what concerns the FROM clause we just note that it allows to specify the XML files on which the query must be resolved.

The output format of the query engine, properly formatted with the RETURN clause, consists of the element `xcde:results` that includes all the *matching subtrees*, each visualized within an element `xcde:result`. The name of the file including a result is returned within the element `xcde:filename`, the rank of the result is indicated in the element `xcde:return` with the attribute `result_number`. The content of `xcde:return` is the snippet requested by the user according to the formatting specified in the RETURN clause. This formatting can be detailed via the following special element.

<xml_context>

In this element the user can put one or more pivots and the `xml_context`'s attributes will be applied to any pivot in an independent way. The attribute `xml_size` is required and specifies the number of tokens to be extracted in addition to the result snippet associated to a pivot (word or subtree). The attribute `xml_view` is optional and specifies if the snippet must be extracted after (`forward`) and/or before (`backward`), or both (`bidir`), the pivot. If the attribute value is `'full'`, the output will be a *well-formed* snippet with all the elements that contain that pivot. The attribute value `'stop_parent'` constraints the *well-formed* snippet to stop its extraction when the open-tag or the close-tag of the pivot's parent is found. If `xml_view` is not inserted, the default behavior is to consider the attribute value `'bidir'`. Another optional attribute is `xml_format` that specifies if the tags must be visualized or not within the snippet. The default behavior is with tags.

4. Some query examples

In this section we provide some illustrative examples to detail the use of the proposed query language. Notice the powerful functionalities to deal with *text snippets*, and to deal with *complex string queries*. Another important feature of our language is the use of the reserved attribute *xml_var* that allows to fix *pivots* within the subtrees matching the query so that the retrieval of the occurrences and their visualization becomes much easier.

Q01a. Single word query: Find all book titles containing the word "usability", case insensitive.

```
SELECT <book>
      <xml_exact xml_var = '$word'> usability </xml_exact >
    </book>
FROM exampleFile.xml
RETURN <example_search> $word </example_search>
```

The snippets are numbered and identified by means of the name of the file that contains them.

```
<xcde:results>
<xcde:result>
<xcde:filename>exampleFile.xml</xcde:filename>
<xcde:return result_number = '1'>
  <example_search> Usability </example_search>
</xcde:return>
.
.
.
<xcde:return result_number = '24'>
  <example_search> Usability </example_search>
</xcde:return>
</xcde:result>
</xcde:results >
```

Q01b. Single word query: Return snippets of 10 tokens before each occurrence of “usability”, case insensitive, including the tagging.

```
SELECT <book>
      <xml_exact xml_var = '$word'> Usability </xml_exact>
    </book>
FROM exampleFile.xml
RETURN <xml_context xml_size = '10' xml_view = 'backward'>
      $word
    </xml_context >
```

Q01c. Single word query: Return snippets of 10 tokens before and after each occurrence of “usability”, case insensitive, removing the tagging.

```

SELECT <book>
      <xml_exact xml_var = '$word'> Usability </xml_exact>
    </book>
FROM exampleFile.xml
RETURN <xml_context xml_size = '10' xml_view = 'bidir' xml_format = 'notag'>
      $word
    </xml_context >

```

Q01d. Single word query: Return well-formed snippets taking 10 tokens before and after each occurrence of “usability”, case sensitive. The occurrences are the ones from the 20th to the 30th, and they are displayed with the tag elements.

```

SELECT <book>
      <xml_exact xml_var = '$word' xml_case = 'sensitive'>
        Usability
      </xml_exact>
    </book>
FROM exampleFile.xml
RETURN <xml_context xml_size = '10' xml_view = 'full'> $word <xml_context>
RANGE 20 30

```

Q01e. Single word query: Return snippets taking 10 tokens before and after each occurrence of “usability”, case insensitive. The extraction of the snippet stops if the parent tag of the query occurrence is met. The occurrences are the ones from the 20th to the 30th, and they are displayed with the tag elements.

```

SELECT <book>
      <xml_exact xml_var = '$word'> Usability </xml_exact>
    </book>
FROM exampleFile.xml
RETURN <xml_context xml_size = '10' xml_view = 'stop_parent'>
      $word
    <xml_context>
RANGE 20 30

```

Q02. Complex query with more than one pivot: Find all <books> containing in the <subject> element the words “usability” and “testing”, case insensitive. The constraint is also that at least one element among <title> (at distance one from <metadata>), <authors> and <dateRevised> must be present below the element <book>. The extraction of the snippet returns the content of the three elements (if any).

```

SELECT <book>
      <subject>Usability testing</subject>
      <xml_or>
        <metadata>
          <title xml_var = '$t' xml_dist = '1'></title>
        </metadata>
        <authors xml_var = '$a'></authors>
        <dateRevised xml_var = '$d'></dateRevised>
      </xml_or>
    </book>
FROM exampleFile.xml
RETURN $t $a $d

```

Q03. Complex query across element boundaries: Find all <books> containing in the <content> element the words in “usability testing once the problems”, case insensitive. The constraint is that the proximity among those words is 4, and there must exist<title> at distance at most 2 from <book>. The extraction of the snippet returns the content of <title> and <content>.

```
SELECT <book>
      <title xml_var = '$title' xml_dist = '2'></title>
      <content xml_var = '$content'>
          <xml_proximity xml_maxprox = '4'>
              usability testing once the problems
          </xml_proximity>
      </content>
  </book>
FROM exampleFile.xml
RETURN $title $content
```

Q04. Query on element and attribute: Find all <books> containing a <title> element including the phrase “MANUSCRIPT guide” as a case insensitive suffix of the attribute “shortTitle”; and a <content> element including the words “user” and “profiling”, case insensitive. The extraction of the snippet returns the entire content of <book>.

```
SELECT <book xml_var = '$book'>
      <title shortTitle = 'sufi:MANUSCRIPT Guide'></title>
      <content>User profiling</content>
  </book>
FROM exampleFile.xml
RETURN $book
```

Q05. OR Query on more than two words: Find all <books> with the words “web” or “software” or “internet” in the <content> element, and with a <title> element descending from <book>. The extraction of the snippet returns the content of <title> and <content>.

```
SELECT <book>
      <title xml_var = '$title'></title>
      <content xml_var = '$content'>
          <xml_or>web software internet</xml_or>
      </content>
  </book>
FROM exampleFile.xml
RETURN $title $content
```

Q06. NOT Query on more than two words: Find all <books> which do not contain the two words “usability” and “testing” consecutive, but have a <title> element descending from <book>. The extraction of the snippet returns the content of <title>.

```
SELECT <book>
      <title xml_var = '$title'></title>
      <xml_not>
          <xml_proximity xml_maxprox= '1'>
              usability testing
          </xml_proximity>
      </xml_not>
  </book>
FROM exampleFile.xml
```

```
RETURN $title
```

Q07. Query with errors: Find the string “ksabimity” with at most 2 errors and then print them.

```
SELECT <xml_error xml_maxerr = '2' xml_var = '$word'>
      ksabimity
    </xml_error>
FROM exampleFile.xml
RETURN $word
```

Q08. Query on attribute without any constraints on the element name: Find all elements with attribute name “type” containing the word “bax” with at most 1 error. Return the content of that element.

```
SELECT <xml_anyvalue xml_var = '$tag' type = 'err:1:bax' >
      </xml_anyvalue >
FROM exampleFile.xml
RETURN $tag
```

Q09. Query on regular expression: Find all occurrences of the words “testing” or “testong”. Return the retrieved word. The syntax for the regular expression is the one used by *grep* command under Unix.

```
SELECT <xml_regexp xml_var = '$word'>
      test[i|o]ng
    </xml_regexp>
FROM exampleFile.xml
RETURN $word
```

5. Conclusions

For the source code, the documentation and some example files please have a look at the home page of the XCDE Library: <http://butirro.di.unipi.it/~ferrax/xcde/xcdelib.html>.