10 XML retrieval

Information retrieval systems are often contrasted with relational databases. IR systems retrieve information from unstructured text – by which we mostly mean “raw” text without markup. Databases are designed for searching structured data: sets of records that have values for predefined attributes such as employee number, title and salary. Some highly structured search problems are best solved with a relational database, for example, if the employee table contains an attribute for short textual job descriptions. There are fundamental differences between information retrieval and database systems in terms of retrieval model, data structures and query language as shown in Table 10.1.1

There are two types of information retrieval problem that are intermediate between text retrieval and search over relational data. We discussed the first type, parametric search, in Section 6.1 (page 85). The second type, XML retrieval, is the subject of this chapter. We will view XML documents as trees that have leaf nodes containing text and labeled internal nodes that define the roles of the leaf nodes in the document. We call this type of text semistructured and retrieval over it semistructured retrieval. In the example in Figure 10.2, some of the leaves shown are Shakespeare, Macbeth, and Macbeth’s castle, and the labeled internal nodes encode either the structure of the document (title, act, and scene) or metadata functions (author).

Semistructured retrieval has become increasingly important in recent years because of the growing use of Extended Markup Language or XML. XML is used for web content, for documents produced by office productivity suites, for the import and export of text content in general, and many other applications. These days, most semistructured data are encoded in XML. Here, we neglect the specifics that distinguish XML from other standards for semistructured data such as HTML and SGML.

1. In most modern database systems, one can enable full-text search for text columns. This usually means that an inverted index is created and Boolean or vector space search enabled, effectively combining core database with information retrieval technologies.
152 10 XML retrieval

<table>
<thead>
<tr>
<th>objects</th>
<th>databases</th>
<th>information retrieval</th>
<th>semistructured retrieval</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>record</td>
<td>unstructured document</td>
<td>tree with text at leaves</td>
</tr>
<tr>
<td>main data structure</td>
<td>relational calculus</td>
<td>vector space &amp; others</td>
<td>?</td>
</tr>
<tr>
<td>queries</td>
<td>table</td>
<td>inverted index</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>SQL</td>
<td>text queries</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 10.1 Databases, information retrieval and semistructured retrieval. There is no consensus yet as to what formal models, query languages and data structures are consistently successful for semistructured retrieval.

<play>
  <author>Shakespeare</author>
  <title>Macbeth</title>
  <act number="I">...
    <scene number="VII">
      <title>Macbeth’s castle</title>
      <verse>Will I with wine and wassail ...</verse>
    </scene>
  </act>
</play>

Figure 10.1 An XML document.

After presenting the basic concepts of XML, this chapter first discusses the challenges we face in semistructured retrieval. Then we describe JuruXML, a modification of the vector space model for XML retrieval. Section 10.4 presents INEX, a shared task evaluation that has been held for a number of years and currently is the most important venue for presenting XML retrieval research. Section 10.5 discusses some of the more “relational” aspects of XML.

10.1 Basic XML concepts

An XML document is an ordered, labeled tree. The nodes of the tree are XML elements and are written with an opening and closing tag. An element can have one or more XML attributes. One of the elements in the example XML document in Figure 10.1 is scene, which is enclosed by the two tags <scene ...> and </scene>. The element has an attribute number with value VII and two child elements, title and verse.

There is a standard way of accessing and processing XML documents, the XML Document Object Model or DOM. DOM represents elements, attributes
10.1 Basic XML concepts

and text within elements as nodes in a tree. Figure 10.2 shows the DOM representation of the XML document in Figure 10.1. With a DOM API, it is easy to process an XML document by starting at the root element and then descending down the tree from parents to children.

**XPath** is the standard for paths in XML. We will also refer to paths as contexts in this chapter. Only a small subset of XPath is needed for our purposes. The XPath expression `node` selects all nodes of that name. Successive elements of a path are separated by slashes, so `act/scene` selects all `scene` elements whose parent is an `act` element. Double slashes indicate that an arbitrary number of elements can intervene on a path: `play///scene` selects all `scene` elements occurring in a `play` element. In Figure 10.2 this set consists of the single `scene` element that is accessible via the path `play, act, scene` from the top. An initial slash starts the path at the root element. `/play/title` selects the play’s title in Figure 10.1, `/play///title` selects the play’s ti-
title and the scene’s title, and /scene/title selects no elements. For notational convenience, we allow the final element of a path to be a string, e.g. title/"Macbeth" for all titles containing the word Macbeth, even though this does not conform to the XPath standard.

We also need the concept of schema in this chapter. A schema puts constraints on the structure of allowable XML documents for a particular application. A schema for Shakespeare’s plays may stipulate that scenes can only occur as children of acts and that only acts and scenes have the number attribute. Two standards for schemas for XML documents are XML DTD (document type definition) and XML Schema. Users only can write structural queries for an XML retrieval system if they have some minimal knowledge about the schema of the underlying collection.

10.2 Challenges in semistructured retrieval

In Chapter 2 (page 18) we briefly discussed the need of a document unit in indexing and retrieval. In unstructured retrieval, it is usually clear what the right document unit is: files on your desktop, email messages, web pages on the web etc. The first challenge in semistructured retrieval is that we don’t have such a natural document unit or indexing unit. If we query Shakespeare’s plays for Macbeth’s castle, should we return the scene, the act or the whole play in Figure 10.2? In this case, the user is probably looking for the scene. On the other hand, an otherwise unspecified search for Macbeth should return the play of this name, not a subunit. One decision criterion that has been proposed for selecting the most appropriate part of a document is the structured document retrieval principle:

Structured document retrieval principle. A system should always retrieve the most specific part of a document answering the query.

This principle motivates a retrieval strategy that returns the smallest unit that contains the information sought, but does not go below this level. However, it can be hard to implement this principle algorithmically. Consider the query title/"Macbeth" applied to Figure 10.2. The title of the tragedy, Macbeth, and the title of Act 1, Scene 7, Macbeth’s castle, are both good hits in terms of term matching. But in this case, the title of the tragedy, the higher node, is preferred. Deciding which level of the tree is right for answering a query is tricky.

There are at least three different approaches to defining the indexing unit in XML retrieval. One is to index all components that are eligible to be returned in a search result. All subtrees in Figure 10.1 meet this criterion, but typographical XML elements as in <b>definitely</b> or an ISBN number without context may not. This scheme has the disadvantage that search
results will contain overlapping units that have to be filtered out in a post-
processing step to reduce redundancy.

Another approach is to group nodes into non-overlapping pseudodocu-
ments as shown in Figure 10.3. This avoids the overlap problem, but pseu-
dodocuments may not make intuitive sense to the user. And they have to 
be fixed at indexing time, leaving no flexibility to answer queries at a more 
specific or more general level.

The third approach is to designate one XML element as the substitute for 
the document unit. This is the approach taken by the system in the next 
section where the document collection is a collection of articles from IEEE 
journals and each component dominated by an article node is treated as a 
document. As with the node groups in Figure 10.3, we have the problem 
that indexing units are fixed. However, we can attempt to extract the most 
relevant subcomponent from each hit in a postprocessing step.

The lack of a clear indexing unit is related to another challenge in XML 
retrieval: We need to distinguish different contexts of a term when we com-
pute term statistics for ranking, in particular inverse document frequency 
(idf) statistics (Section 6.2.1, page 88). For example, the term Gates under the 
node Author is unrelated to an occurrence under a content node like Section if 
used to refer to the plural of gate. Computing a single document frequency 
for all occurrences of Gates is problematic unless the distribution of terms 
is similar in all content nodes. The simplest solution is to compute idf 
weights for term-context pairs (or structural terms). So we would compute different idf 
weights for author/"Gates" and section/"Gates". Unfortunately, this
A schema mismatch. A difficulty in XML retrieval is that a query may not conform to the schema of the collection. Here, no exact match is possible between query (left tree) and document (right tree) because the Name node in the query has no corresponding node in the document.

The schemas of XML documents in a collection can vary since semi-structured documents often come from different sources. This presents yet another challenge. Comparable nodes may have different names and be represented differently structurally. In Figure 10.4, the node Name in the query (the left tree) corresponds to the two nodes FirstName and LastName in the document (the right tree). Some form of approximate matching of node names in combination with semi-automatic matching of different document structures can help here. Human editing of correspondences of nodes in different schemas will usually do better than completely automatic methods.

Schemas also pose a challenge for user interfaces if users are not familiar with the structure and naming conventions of the document collection. Consider the queries in Figure 10.5. For query (b) the user has made the assumption that the node referring to the creator of the document is called Author, but it could also be named Writer or Creator (the latter being the choice of the Dublin Core Metadata standard). Query (c) is a search for books that contain St. Peter anywhere. We will call them extended queries here since they will match with documents containing any number of nodes between Book and St. Peter. This corresponds to the double slash in XPath notation: book//"St. Peter". It is quite common for users to issue such extended queries without specifying the exact structural configuration in which a query term should occur.

The user interface should expose the tree structure of the collection and allow users to specify the nodes they are querying. As a consequence the query interface is more complex than a search box for keyword queries in unstruc-
10.3 A vector space model for XML retrieval

Unlike unstructured retrieval, XML retrieval requires taking into consideration the structural context of terms. A document authored by Bill Gates should match the second query in Figure 10.5, but not the first. We want to make use of the vector space model to represent this structural context. In unstructured retrieval, there would be a single axis for Gates. In XML retrieval, we must separate the title word Gates from the author name Gates. Thus, the axes must represent not only words, but also their position within the XML tree.

One of the first systems that took the vector space approach to XML retrieval was JuruXML. The basic document representation adopted in JuruXML is shown in Figure 10.6. The dimensions of the vector space are defined to be subtrees of documents that contain at least one lexicon term. We call a subtree that functions as an axis a structural term. A subset of the possible structural terms is shown in the figure, but there are others (e.g., the subtree corresponding to the whole document with the leaf node Gates removed). There is a tradeoff between index size and accuracy of query results. In general, query results will only be completely accurate if we index all subtrees...
(Exercise 10.1), but this results in a very large index. A compromise is to index all paths that end in a single lexicon term. The document in the figure would then have 9 structural terms.

We can treat structural terms just like regular terms in ordinary vector space retrieval. For instance, we can compute term frequency weights and perform stemming and case folding. To compute idf weights for structural terms, we need a document unit. JuruXML assumes that there is such a document unit, e.g., the article or scene elements.

We also want to weight contexts in the query. Users often care more about some parts of the query than others. In query (b) in Figure 10.5, the user may want to give more weight to the author because she is not sure whether she remembers the title correctly. This weighting can either be done in the user interface as part of query input; or by the system designer in cases where the importance of different parts of the query is likely to be the same across users.

In JuruXML, it is assumed that all queries are extended (“double slash”) queries. If a query can be made to match a document by inserting additional nodes as in Figure 10.7, then the document is a potential hit. We interpret every query as an extended query because requiring a decision for each part of the query tree as to whether the corresponding link is intended as exact or extended would put too much of a burden on the user.

Even if we allow extended matches, we still prefer documents that match the query structure closely. We ensure that retrieval results respect this preference by computing a weight for each match. A simple measure of the
Figure 10.7  Query-document matching for extended queries. Extended queries match a document if the query path can be transformed into the document path by insertion of additional nodes. The context resemblance function $cr$ is a measure of how similar two paths are. Here we have $cr(q,d_1) = 3/4 = 0.75$ and $cr(q,d_2) = 3/5 = 0.6$.

“goodness of fit” between two paths is the following context resemblance function $cr$:

$$cr(q,d) = \frac{1 + |q|}{1 + |d|}$$

where $q$ and $d$ are the number of nodes in the query path and document path, respectively. The context resemblance function returns 0 if the query path cannot be extended to match the document path. Its value is 1.0 if $q$ and $d$ are identical. Two additional examples are given in Figure 10.7.

The query processing component needs to identify those structural terms in the dictionary that have a non-zero match with the query terms. This step is analogous to tolerant retrieval in unstructured retrieval that also requires a mapping from query terms to index terms (see Chapter 3).

The final score for a document is computed as a variant of the cosine measure (Equation (7.1), page 98). The context resemblance similarity between query and document is defined as:

$$sim-cr(q,d) = \frac{\sum_{t \in V} \sum_{c_k \in C} \sum_{c_l \in C} \text{weight}(q,t,c_k) \text{weight}(d,t,c_l) cr(c_k,c_l)}{||\vec{q}|| ||\vec{d}||}$$

where $V$ is the vocabulary of (non-structural) terms; $C$ is the set of all contexts (paths) occurring in the collection and the queries; and $\text{weight}(d,t,c_k)$ is the weight of term $t$ in context $c_k$ in document $d$.

An example of an inverted index search in extended query matching is given in Figure 10.8. The structural term ST in the query occurs as ST1 in the
Figure 10.8 Inverted index search for extended queries.

Indexing
For each indexing unit \(i\):
- Compute structural terms for \(i\)
- Construct index

Search
Compute structural terms for query
For each structural term \(t\):
- Find matching structural terms in dictionary
  - For each matching structural term \(t'\):
    - Compute matching weight \(cr(t, t')\)
- Search inverted index with computed terms and weights
- Return ranked list

Figure 10.9 Indexing and search in JuruXML.

Index and has a match coefficient of 0.63 with a second term ST5 in the index. In this example, the highest ranking document is Doc9 with a similarity of \(1.0 \times 0.2 + 0.63 \times 0.6 = 0.578\). Query weights are assumed to be 1.0.

Figure 10.9 summarizes indexing and query processing in JuruXML.

Idf weights for the just introduced retrieval algorithm are computed separately for each structural term. We can call this strategy NoMerge since contexts of different structural terms are not merged. An alternative is a Merge strategy that computes statistics for term \((t, c)\) by collapsing all contexts \(c'\) that have a non-zero context resemblance with \(c\). So for computing the document frequency of the structural term \((atl, recognition)\), occurrences of recognition in contexts \(fm/atl, article//atl\) etc. would also be counted. This scheme addresses the sparse data problems that occur when computing idf weights for structural terms.
12,107  number of documents
494 MB  size
1995–2002  time of publication of articles
1,532  average number of XML nodes per document
6.9  average depth of a node
30  number of CAS topics
30  number of CO topics

► Table 10.2  INEX 2002 collection statistics. There are two types of topics: content only (CO) and content and structure (CAS). An example of a CAS topic is given in Figure 10.11.

A second innovation in the Merge run is the use of a modified similarity function sim-qc, which replaces sim-cr when computing cosine similarity:

\[
\text{sim-qc}(q, d) = \frac{\sum_{t \in V} \sum_{c \in C} \text{weight}(q, t) \text{weight}(d, t) \text{qc}(c)}{|q||d|} \tag{10.2}
\]

where \( \text{qc}(c) = |c| + 1 \). So weight is a linear function of the path length or specificity of the context.

An evaluation of Merge and NoMerge and a comparison to other methods are the subject of the next section.

10.4 Evaluation of XML Retrieval

A large part of academic research on XML retrieval is conducted within the INEX (INitiatíve for the Evaluation of XML retrieval) program, a collaborative effort that includes reference collections, sets of queries, relevance judgments and a yearly meeting to present and discuss research results. What TREC is to unstructured information retrieval, INEX is to XML retrieval. We will learn in this section how a reference collection for XML retrieval can be set up, what evaluation measures have been proposed and what level of performance to expect on these measures.

The INEX 2002 collection consists of about 12,000 articles from IEEE journals. We give collection statistics in Table 10.2 and show the structure of the documents in Figure 10.10.

There are two types of queries, called topics, in INEX: content-only or CO topics and content-and-structure or CAS topics. CO topics are regular keyword queries as in unstructured information retrieval. CAS topics have structural constraints in addition to keywords as shown in the example in Figure 10.11. These two different components of CAS queries make relevance assessments more complicated than in unstructured retrieval. INEX defines
Figure 10.10  Schema of the documents in the INEX collection. The element tags are front matter (fm), title (ti), article title (atl), body (bdy), section (sec), section title (st) and paragraph (p). Not shown is title group (tig).

\[
\begin{align*}
&\text{<te>article</te>} \\
&\text{<cw>non-monotonic reasoning</cw> } \text{<ce>bdy/sec</ce>} \\
&\text{<cw>1999 2000</cw> } \text{<ce>hdr//yr</ce>} \\
&\text{<cw>-calendar</cw> } \text{<ce>tig/atl</ce>}
\end{align*}
\]

Figure 10.11  An INEX CAS topic. The topic specifies a search for articles on non-monotonic reasoning from 1999 or 2000 that are not calendars of events. The first line specifies that retrieved elements should be articles (te = target element). The other three lines are pairs of content word (cw) and content element (ce) conditions, indicating the content words that should occur (non-monotonic reasoning, 1999, 2000) or should not occur (calendar) in a particular context.

<table>
<thead>
<tr>
<th>COMPONENT COVERAGE</th>
<th>TOPICAL RELEVANCE</th>
</tr>
</thead>
</table>

\textit{component coverage} and \textit{topical relevance} as orthogonal dimensions of relevance. The component coverage dimension evaluates whether the element retrieved is “structurally” correct, i.e., neither too low nor too high in the tree. We distinguish four cases:

- Exact coverage (E). The information sought is the main topic of the component and the component is a meaningful unit of information.

- Too small (S). The information sought is the main topic of the component, but the component is not a meaningful (self-contained) unit of information.

Preliminary draft (c) 2007 Cambridge UP
• Too large (L). The information sought is present in the component, but is not the main topic.
• No coverage (N). The information sought is not a topic of the component.

The topical relevance dimension also has four levels: highly relevant (3), fairly relevant (2), marginally relevant (1) and irrelevant (0). Components are judged on both dimensions and the judgments are then combined into a digit-letter code. 2S is a fairly relevant component that is too small and 3E is a highly relevant component that has exact coverage. In theory, there are 16 combinations of coverage and relevance, but many cannot occur. For example, a non-relevant component cannot have exact coverage, so the combination 3N is not possible.

The relevance-coverage combinations are then “quantized” as follows:

\[
q(\text{rel}, \text{cov}) =
\begin{cases}
1.00 & \text{if } (\text{rel}, \text{cov}) = 3E \\
0.75 & \text{if } (\text{rel}, \text{cov}) \in \{2E, 3L\} \\
0.50 & \text{if } (\text{rel}, \text{cov}) \in \{1E, 2L, 2S\} \\
0.25 & \text{if } (\text{rel}, \text{cov}) \in \{1S, 1L\} \\
0.00 & \text{if } (\text{rel}, \text{cov}) = 0N
\end{cases}
\]

This evaluation scheme takes account of the fact that yes/no relevance judgments are less appropriate for XML retrieval than for unstructured retrieval. A 2S component provides incomplete information and may be difficult to interpret without more context, but it does answer the query partially. Quantization of relevance-coverage combinations avoids a binary choice and lets us grade it as “half-relevant”. The number of relevant components in a retrieved set \(C\) of components can then be computed as:

\[
\#(\text{relevant items retrieved}) = \sum_{c \in C} q(\text{rel}(c), \text{cov}(c))
\]

As an approximation, the standard definitions of precision, recall and F from Chapter 8 can be applied to this modified definition of relevant items retrieved, with some subtleties because we sum “fractional” relevance assessments. See the references given in Section 10.6 for further discussion.

One flaw of measuring relevance this way is that overlap is not accounted for. We discussed the concept of marginal relevance in the context of unstructured retrieval in Chapter 8 (page 123). This problem is worse in XML retrieval than unstructured retrieval because the same component can occur multiple times in a ranking as part of different higher-level components. The play, act, scene and title components on the path between the root node and Macbeth’s castle in Figure 10.1 can all be returned in a result set, so that the leaf node occurs four times.
Table 10.3 shows four runs from the three top-ranked groups at INEX 2002. The Merge and NoMerge runs are from the JuruXML system as described in the last section. The submission Allow-duplicate is from a database system that was extended with the capability of storing and searching for trees. UAmst02NGram uses a vector space retrieval approach with pseudo-relevance feedback and applies structural XML constraints in a postfiltering step. The best run is the Merge run, which incorporates fewer structural constraints than the other systems and mostly relies on keyword matching. The fact that it does so well demonstrates the challenge of XML retrieval: Methods for structural matching need to be designed and executed well to achieve an improvement compared to unstructured retrieval methods.

The average precision numbers in Table 10.3 are respectable, but there obviously remains a lot of work to be done. Further analysis presented in the INEX 2002 proceedings shows that the best systems get about 50\% of the first 10 hits right. However, this is only true on average. The variation across topics is large. The best run, Merge, had a median of 0.147, so performance for most topics is poor. A smaller number of topics, about a quarter, is handled well by Merge, with average precision higher than 0.5. These numbers are meant to give the reader a sense of what level of performance to expect from XML retrieval systems.

10.5 Text-centric vs. structure-centric XML retrieval

In JuruXML, XML structure serves as a framework within which conventional text matching is performed, exemplifying the text-centric approach to XML retrieval. While both structure and text are important, we give higher priority to text matching. We adopt unstructured retrieval methods to handle additional structural constraints.

In contrast, structure-centric XML retrieval (as exemplified by the University of Michigan system) puts the emphasis on the structural aspects of a user query. A clear example is: “chapters of books that have the same title as the book”. This query has no text component. It is purely structural.
Text-centric approaches are appropriate for data that are essentially text documents, marked up as XML to capture document structure. This is becoming a de facto standard for publishing text databases since most text documents have some form of interesting structure (paragraphs, sections, footnotes etc.). Examples include assembly manuals, journal issues, Shakespeare’s collected works and newswire articles.

Structure-centric approaches are commonly used for data collections with complex structures that contain text as well as non-text data. A text-centric retrieval engine will have a hard time dealing with proteomic data that may be part of a biomedical publication – or with the representation of a street map that (together with street names and other textual descriptions) forms a navigational database.

We have treated XML retrieval here as an extension of conventional text retrieval: text fields are long (e.g., sections of a document), we must support inexact matches of paths and words and users want relevance-ranked results. Relational databases do not deal well with this “use case”.

We have concentrated on applications where XML is used for little more than encoding a tree. For these applications, text-centric approaches suffice. But in reality, XML is a much richer representation formalism. In particular, attributes and their values usually have a database flavor and are best handled by a structure-centric approach or by parametric search.

Two other types of queries that cannot be handled in a vector space model are joins and ordering constraints. The following query requires a join:

Find figures that describe the Corba architecture and the paragraphs that refer to those figures.

This query imposes an ordering constraint:

Retrieve the chapter of the book Introduction to algorithms that follows the chapter Binomial heaps.

The Corba query requires a join of paragraphs and figures. The Binomial heap query relies on the ordering of nodes in XML, in this case the ordering of chapter nodes underneath the book node. There are powerful query languages for XML that can handle attributes, joins and ordering constraints. The best know of these is XQuery, a language proposed for standardization by the W3C. It is designed to be broadly applicable in all areas where XML is used. At the time of this writing, little research has been done on testing the ability of XQuery to satisfy the demands of typical information retrieval settings. Efficiency is a major concern in this regard. Due to its complexity, it is challenging to implement an XQuery-based information retrieval system with the performance characteristics that users have come to expect.

Relational databases are better equipped to handle many structural constraints, particularly joins. But ordering is also difficult in a database frame-
work – the tuples of a relation in the relational calculus are not ordered. Still, many structure-centric XML retrieval systems are extensions of relational databases. If text fields are short, exact matches for paths and text are desired and retrieval results in form of unordered sets are ok, then a relational database is sufficient.

10.6 References and further reading

There are many good introductions to XML, including (Harold and Means 2004). The structured document retrieval principle is due to Fuhr and Großjohann (2001). JuruXML is presented in (Carmel et al. 2003). Section 10.4 follows the overview of INEX 2002 by Gövert and Kazai (2003), published in the proceedings of the meeting (Fuhr et al. 2003). The proceedings also contain papers by IBM Haifa Laboratory (Mass et al. 2002), the University of Amsterdam (Kamps et al. 2002), and the University of Michigan (Yu et al. 2002) with further details on the runs submitted by these groups.

The survey of automatic schema matching given by Rahm and Bernstein (2001) for databases is also applicable to XML.

The proposed standard for XQuery can be found at http://www.w3.org/TR/xquery/.

10.7 Exercises

Exercise 10.1
We discussed the tradeoff between accuracy of results and index size in vector space XML retrieval. If we only index structural terms that are paths ending in lexicon terms, what type of query can we not answer correctly?

Exercise 10.2
If we only index structural terms that are paths ending in lexicon terms, how many structural terms does the document in Figure 10.1 yield?

Exercise 10.3
If we only index structural terms that are paths ending in lexicon terms, what is the size of the index as a function of text size?

Exercise 10.4
If we index all structural terms (i.e., all subtrees), what is the size of the index as a function of text size?