10 *XML retrieval*

Information retrieval (IR) systems are often contrasted with relational databases. Traditionally, IR systems have retrieved information from *unstructured text* – by which we mean "raw" text without markup. Databases are designed for querying *relational data*, sets of records that have values for predefined attributes such as employee number, title, and salary. There are fundamental differences between IR and database systems in terms of retrieval model, data structures, and query language as shown in Table 10.1.[1]

Some highly structured text search problems are most efficiently handled by a relational database; for example, if the employee table contains an attribute for short textual job descriptions and you want to find all employees who are involved with invoicing. In this case, the SQL query:

```
select lastname from employees where job_desc like 'invoic%';
```

may be sufficient to satisfy your information need with high precision and recall.

However, many structured data sources containing text are best modeled as structured documents rather than relational data. We call the search over such structured documents *structured retrieval*. Queries in structured retrieval can be either structured or unstructured, but we assume in this chapter that the collection consists only of structured documents. Applications of structured retrieval include digital libraries, patent databases, blogs, text in which entities like persons and locations have been tagged (in a process called *named entity tagging*), and output from office suites like OpenOffice that save documents as marked up text. In all of these applications, we want to be able to run queries that combine textual criteria with structural criteria. Examples of such queries are give me a full-length article on fast fourier transforms (digital libraries), give me patents whose claims mention RSA public key encryption and

STRUCTURED RETRIEVAL

[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 IR technologies.

Table 10.1 Relational database (RDB) search, unstructured IR, and structured IR. There is no consensus yet as to which methods work best for structured retrieval, although many researchers believe that XQuery (page 197) will become the standard for structured queries.

	RDB search	unstructured retrieval	structured retrieval
objects	records	unstructured documents	trees with text at leaves
model	relational model	vector space & others	?
main data structure	table	inverted index	?
queries	SQL	free text queries	?

that cite US patent 4,405,829 (patents), or give me articles about sightseeing tours of the Vatican and the Coliseum (entity-tagged text). These three queries are structured queries that cannot be answered well by an unranked retrieval system. As we argued in Example 1.1 (page 14), unranked retrieval models like the Boolean model suffer from low recall. For instance, an unranked system would return a potentially large number of articles that mention the Vatican, the Coliseum, and sightseeing tours without ranking the ones that are most relevant for the query first. Most users are also notoriously bad at precisely stating structural constraints. For instance, users may not know for which structured elements the search system supports search. In our example, the user may be unsure whether to issue the query as sightseeing AND COUNTRY:Vatican AND LANDMARK:Coliseum, as sightseeing AND STATE:Vatican AND BUILDING:Coliseum or in some other form. Users may also be completely unfamiliar with structured search and advanced search interfaces or unwilling to use them. In this chapter, we look at how ranked retrieval methods can be adapted to structured documents to address these problems.

XML

We will only look at one standard for encoding structured documents: *Extensible markup language* or *XML*, which is currently the most widely used such standard. We will not cover the specifics that distinguish XML from other types of markup such as HTML and SGML. But most of what we say in this chapter is applicable to markup languages in general.

DATA-CENTRIC XML

In the context of IR, we are only interested in XML as a language for encoding text and documents. A perhaps more widespread use of XML is to encode nontext data. For example, we may want to export data in XML format from an enterprise resource planning system and then read them into an analytics program to produce graphs for a presentation. This type of application of XML is called *data-centric* because numerical and nontext attribute-value data dominate and text is usually a small fraction of the overall data. Most data-centric XML is stored in databases – in contrast to the inverted index-based methods for text-centric XML that we present in this chapter.

SEMISTRUCTURED RETRIEVAL

We call XML retrieval *structured retrieval* in this chapter. Some researchers prefer the term *semistructured retrieval* to distinguish XML retrieval from database querying. We have adopted the terminology that is widespread in

```
<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.

the XML retrieval community. For instance, the standard way of referring to XML queries is *structured queries*, not *semistructured queries*. The term *structured retrieval* is rarely used for database querying and it always refers to XML retrieval in this book.

There is a second type of IR problem that is intermediate between unstructured retrieval and querying a relational database: parametric and zone search, which we discussed in Section 6.1 (page 101). In the data model of parametric and zone search, there are parametric fields (relational attributes like *date* or *file-size*) and zones – text attributes that each take a chunk of unstructured text as value, for example, *author* and *title* in Figure 6.1 (page 101). The data model is flat; that is, there is no nesting of attributes. The number of attributes is small. In contrast, XML documents have the more complex tree structure that we see in Figure 10.2, in which attributes are nested. The number of attributes and nodes is greater than in parametric and zone search.

After presenting the basic concepts of XML in Section 10.1, this chapter first discusses the challenges we face in XML retrieval (Section 10.2). Next we describe a vector space model for XML retrieval (Section 10.3). 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 XML retrieval research. We discuss the differences between data-centric and text-centric approaches to XML in Section 10.5.

10.1 Basic XML concepts

XML ELEMENT
XML ATTRIBUTE

An XML document is an ordered, labeled tree. Each node of the tree is an *XML element* and is written with an opening and closing *tag*. An element can have one or more *XML attributes*. In the XML document in Figure 10.1, the *scene* element is enclosed by the two tags `<scene ...>` and `</scene>`. It has an attribute *number* with value *vii* and two child elements, *title* and *verse*.

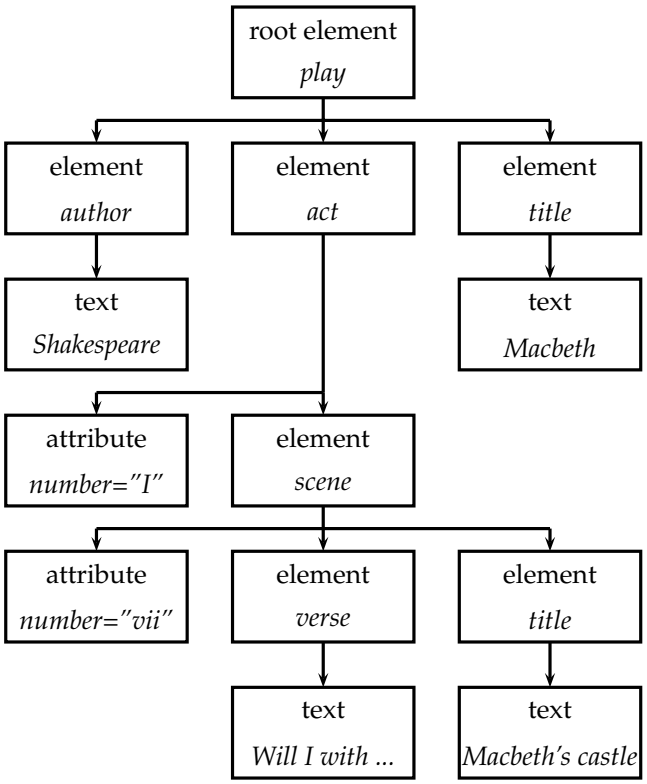

Figure 10.2 The XML document in Figure 10.1 as a simplified DOM object.

Figure 10.2 shows Figure 10.1 as a tree. The *leaf nodes* of the tree consist of text, for example, Shakespeare, Macbeth, and Macbeth's castle. The tree's *internal nodes* encode either the structure of the document (*title*, *act*, and *scene*) or metadata functions (*author*).

XML DOM

The standard for accessing and processing XML documents is the XML document object model or *DOM*. The DOM represents elements, attributes, and text within elements as nodes in a tree. Figure 10.2 is a simplified DOM representation of the XML document in Figure 10.1.[2] With a DOM API, we can process an XML document by starting at the root element and then descending down the tree from parents to children.

XPath

XML context

XPath is a standard for enumerating paths in an XML document collection. We will also refer to paths as *XML contexts* or simply *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 a single *scene* element, which

[2] The representation is simplified in a number of respects. For example, we do not show the *root* node and text is not embedded in *text* nodes. See www.w3.org/DOM/.

```
//article
[.//yr = 2001 or .//yr = 2002]
//section
[about(.,summer holidays)]
```

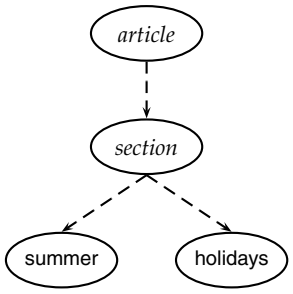

Figure 10.3 An XML query in NEXI format and its partial representation as a tree.

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 a set with two members (the play's 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 vocabulary term and separate it from the element path by the symbol #, even though this does not conform to the XPath standard. For example, `title#"Macbeth"` selects all titles containing the term Macbeth.

SCHEMA 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 can only write structured queries for an XML retrieval system if they have some minimal knowledge about the schema of the collection.

XML DTD

XML SCHEMA

NEXI A common format for XML queries is *NEXI* (Narrowed Extended XPath I). We give an example in Figure 10.3. We display the query on four lines for typographical convenience, but it is intended to be read as one unit without line breaks. In particular, `//section` is embedded under `//article`.

The query in Figure 10.3 specifies a search for sections about the summer holidays that are part of articles from 2001 or 2002. As in XPath, double slashes indicate that an arbitrary number of elements can intervene on a path. The dot in a clause in square brackets refers to the element the clause modifies. The clause `[.//yr = 2001 or .//yr = 2002]` modifies `//article`. Thus, the dot refers to `//article` in this case. Similarly, the dot in `[about(., summer holidays)]` refers to the section that the clause modifies.

The two `yr` conditions are relational attribute constraints. Only articles whose `yr` attribute is 2001 or 2002 (or that contain an element whose `yr` attribute is 2001 or 2002) are to be considered. The `about` clause is a ranking

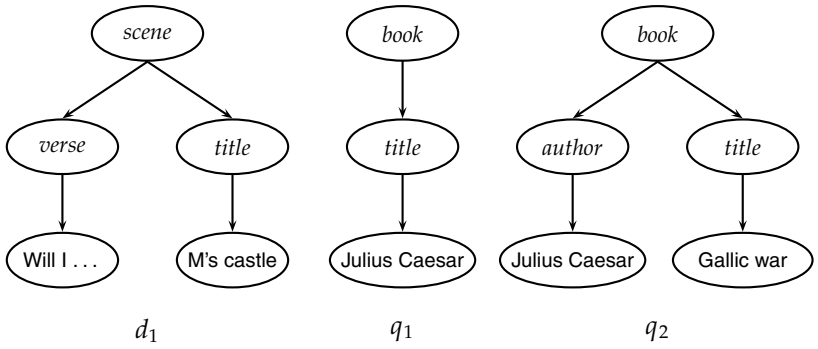

Figure 10.4 Tree representation of XML documents and queries.

constraint: Sections that occur in the right type of article are to be ranked according to how relevant they are to the topic summer holidays.

We usually handle relational attribute constraints by prefiltering or postfiltering: We simply exclude all elements from the result set that do not meet the relational attribute constraints. In this chapter, we will not address how to do this efficiently and instead focus on the core information retrieval problem in XML retrieval, namely, how to rank documents according to the relevance criteria expressed in the about conditions of the NEXI query.

If we discard relational attributes, we can represent documents as trees with only one type of node: element nodes. In other words, we remove all attribute nodes from the XML document, such as, the *number* attribute in Figure 10.1. Figure 10.4 shows a subtree of the document in Figure 10.1 as an element–node tree (labeled d_1).

We can represent queries as trees in the same way. This is a query-by-example approach to query language design because users pose queries by creating objects that satisfy the same formal description as documents. In Figure 10.4, q_1 is a search for books whose titles score highly for the keywords Julius Caesar. q_2 is a search for books whose author elements score highly for Julius Caesar and whose title elements score highly for Gallic war.[3]

10.2 Challenges in XML retrieval

In this section, we discuss a number of challenges that make structured retrieval more difficult than unstructured retrieval. Recall from page 178 the basic setting we assume in structured retrieval: The collection consists of structured documents and queries are either structured (as in Figure 10.3) or unstructured (e.g., summer holidays).

[3] To represent the semantics of NEXI queries fully, we would also need to designate one node in the tree as a "target node," for example, the *section* in the tree in Figure 10.3. Without the designation of a target node, the tree in Figure 10.3 is not a search for sections embedded in articles (as specified by NEXI), but a search for articles that contain sections.

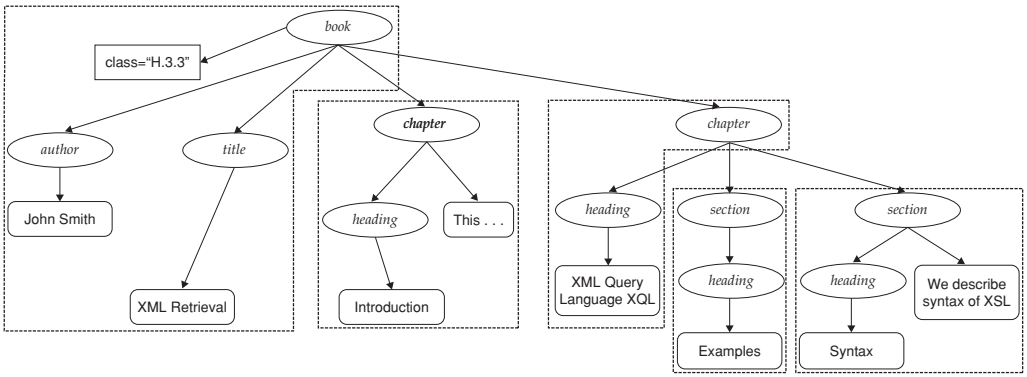

Figure 10.5 Partitioning an XML document into nonoverlapping indexing units.

The first challenge in structured retrieval is that users want us to return parts of documents (i.e., XML elements), not entire documents as IR systems usually do in unstructured retrieval. If we query Shakespeare's plays for Macbeth's castle, should we return the scene, the act, or the entire 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 criterion for selecting the most appropriate part of a document is the *structured document retrieval principle*:

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 I, Scene vii, *Macbeth's castle*, are both good hits because they contain the matching term Macbeth. 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 difficult.

Parallel to the issue of which parts of a document to return to the user is the issue of which parts of a document to index. In Section 2.1.2 (page 20), we

INDEXING UNIT

discussed the need for a document unit or *indexing 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, and so on. In structured retrieval, there are a number of different approaches to defining the indexing unit.

One approach is to group nodes into nonoverlapping pseudodocuments as shown in Figure 10.5. In the example, books, chapters, and sections have been designated to be indexing units, but without overlap. For example, the leftmost dashed indexing unit contains only those parts of the tree dominated

by *book* that are not already part of other indexing units. The disadvantage of this approach is that pseudodocuments may not make sense to the user because they are not coherent units. For instance, the left most indexing unit in Figure 10.5 merges three disparate elements, the *class*, *author*, and *title* elements.

We can also use one of the largest elements as the indexing unit, for example, the *book* element in a collection of books or the *play* element for Shakespeare's works. We can then postprocess search results to find for each book or play the subelement that is the best hit. For example, the query Macbeth's castle may return the play *Macbeth*, which we can then postprocess to identify act I, scene vii as the best matching subelement. Unfortunately, this two-stage retrieval process fails to return the best subelement for many queries because the relevance of a whole book is often not a good predictor of the relevance of small subelements within it.

Instead of retrieving large units and identifying subelements (top down), we can also search all leaves, select the most relevant ones, and extend them to larger units in postprocessing (bottom up). For the query Macbeth's castle in Figure 10.1, we would retrieve the title *Macbeth's castle* in the first pass and then decide in a postprocessing step whether to return the title, the scene, the act, or the play. This approach has a similar problem as the last one: The relevance of a leaf element is often not a good predictor of the relevance of elements in which it is contained.

The least restrictive approach is to index all elements. This is also problematic. Many XML elements are not meaningful search results, for example, typographical elements like `<b>definitely</b>` or an ISBN number, which cannot be interpreted without context. Also, indexing all elements means that search results will be highly redundant. For the query Macbeth's castle and the document in Figure 10.1, we would return all of the *play*, *act*, *scene*, and *title* elements on the path between the root node and Macbeth's castle. The leaf node would then occur four times in the result set, once directly and three times as part of other elements. We call elements that are contained within each other *nested*. Returning redundant nested elements in a list of returned hits is not very user friendly.

NESTED ELEMENTS

Because of the redundancy caused by nested elements, it is common to restrict the set of elements that are eligible to be returned. Restriction strategies include:

- Discard all small elements
- Discard all element types that users do not look at (this requires a working XML retrieval system that logs this information)
- Discard all element types that assessors generally do not judge to be relevant (if relevance assessments are available)
- Only keep element types that a system designer or librarian has deemed to be useful search results

In most of these approaches, result sets still contain nested elements. Thus, we may want to remove some elements in a postprocessing step to reduce redundancy. Alternatively, we can collapse several nested elements in the results list and use *highlighting* of query terms to draw the user's attention to the relevant passages. If query terms are highlighted, then scanning a medium-sized element (e.g., a section) takes little more time than scanning a small subelement (e.g., a paragraph). Thus, if the section and the paragraph both occur in the results list, it is sufficient to show the section. An additional advantage of this approach is that the paragraph is presented together with its context (i.e., the embedding section). This context may be helpful in interpreting the paragraph (e.g., the source of the information reported) even if the paragraph on its own satisfies the query.

If the user knows the schema of the collection and is able to specify the desired type of element, then the problem of redundancy is alleviated because few nested elements have the same type. But as we discussed in the introduction, users often do not know what the name of an element in the collection is (Is the Vatican a *country* or a *city*?) or they may not know how to compose structured queries at all.

A challenge in XML retrieval related to nesting is that we may need to distinguish different contexts of a term when we compute term statistics for ranking, in particular inverse document frequency (idf) statistics as defined in Section 6.2.1 (page 108). 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. It makes little sense to compute a single document frequency for Gates in this example.

One solution is to compute idf for XML-context/term pairs, for example, to compute different idf weights for `author#"Gates"` and `section#"Gates"`. Unfortunately, this scheme runs into sparse data problems – that is, many XML–context pairs occur too rarely to reliably estimate document frequency (see Section 13.2, page 240, for a discussion of sparseness). A compromise is only to consider the parent node x of the term and not the rest of the path from the root to x to distinguish contexts. There are still conflations of contexts that are harmful in this scheme. For instance, we do not distinguish names of authors and names of corporations if both have the parent node *name*. But most important distinctions, like the example contrast `author#"Gates"` versus `section#"Gates"`, will be respected.

SCHEMA HETEROGENEITY

In many cases, several different XML schemas occur in a collection because the XML documents in an IR application often come from more than one source. This phenomenon is called *schema heterogeneity* or *schema diversity* and presents yet another challenge. As illustrated in Figure 10.6, comparable elements may have different names: *creator* in d_2 vs. *author* in d_3. In other cases, the structural organization of the schemas may be different: Author names are direct descendants of the node *author* in q_3, but there are the intervening nodes *firstname* and *lastname* in d_3. If we employ strict

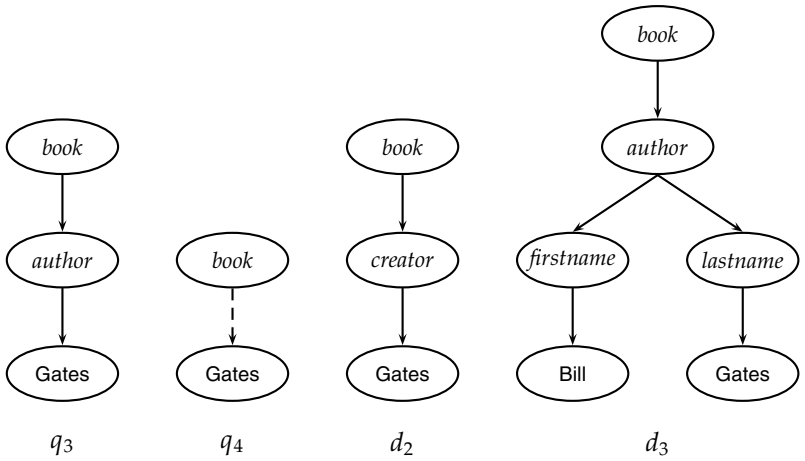

Figure 10.6 Schema heterogeneity: intervening nodes and mismatched names.

matching of trees, then q_3 will retrieve neither d_2 nor d_3, although both documents are relevant. Some form of approximate matching of element names in combination with semiautomatic matching of different document structures can help here. Human editing of correspondences of elements in different schemas will usually do better than automatic methods.

Schema heterogeneity is one reason for query–document mismatches like q_3/d_2 and q_3/d_3. Another reason is that users often are not familiar with the element names and the structure of the schemas of collections they search, as mentioned. This poses a challenge for interface design in XML retrieval. Ideally, the user interface should expose the tree structure of the collection and allow users to specify the elements they are querying. If we take this approach, then designing the query interface in structured retrieval is more complex than a search box for keyword queries in unstructured retrieval.

EXTENDED QUERY

We can also support the user by interpreting all parent–child relationships in queries as descendant relationships with any number of intervening nodes allowed. We call such queries *extended queries*. The tree in Figure 10.3 and q_4 in Figure 10.6 are examples of extended queries. We show edges that are interpreted as descendant relationships as dashed arrows. In q_4, a dashed arrow connects *book* and Gates. As a pseudo-XPath notation for q_4, we adopt `book//#"Gates"`: a book that somewhere in its structure contains the word Gates where the path from the *book* node to Gates can be arbitrarily long. The pseudo-XPath notation for the extended query that in addition specifies that Gates occurs in a *section* of the *book* is `book//section//#"Gates"`. It is convenient for users to be able to issue such extended queries without having to specify the exact structural configuration in which a query term should occur – either because they do not care about the exact configuration or because they do not know enough about the schema of the collection to be able to specify it.

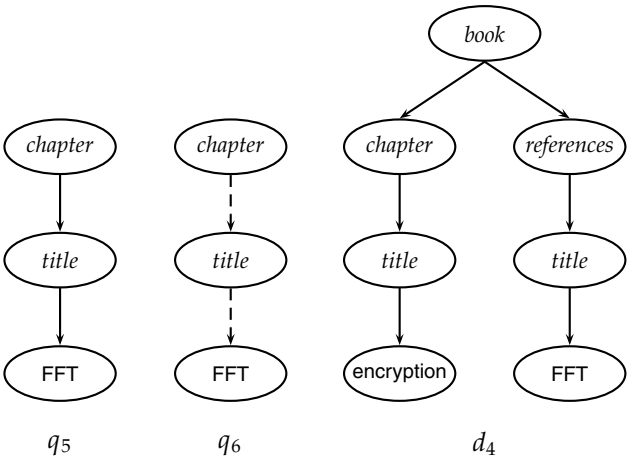

Figure 10.7 A structural mismatch between two queries and a document.

In Figure 10.7, the user is looking for a chapter entitled FFT (q_5). Suppose there is no such chapter in the collection, but that there are references to books on FFT (d_4). A reference to a book on FFT is not exactly what the user is looking for, but it is better than returning nothing. Extended queries do not help here. The extended query q_6 also returns nothing. This is a case where we may want to interpret the structural constraints specified in the query as hints as opposed to as strict conditions. As we will discuss in Section 10.4, users prefer a relaxed interpretation of structural constraints: Elements that do not meet structural constraints perfectly should be ranked lower, but they should not be omitted from search results.

10.3 A vector space model for XML retrieval

In this section, we present a simple vector space model for XML retrieval. It is not intended to be a complete description of a state-of-the-art system. Instead, we want to give the reader a flavor of how documents can be represented and retrieved in XML retrieval.

To take account of structure in retrieval in Figure 10.4, we want a book entitled *Julius Caesar* to be a match for q_1 and no match (or a lower weighted match) for q_2. In unstructured retrieval, there would be a single dimension of the vector space for Caesar. In XML retrieval, we must separate the title word Caesar from the author name Caesar. One way of doing this is to have each dimension of the vector space encode a word together with its position within the XML tree.

Figure 10.8 illustrates this representation. We first take each text node (which in our setup is always a leaf) and break it into multiple nodes, one for each word. So the leaf node Bill Gates is split into two leaves, Bill and Gates. Next we define the dimensions of the vector space to be *lexicalized subtrees*

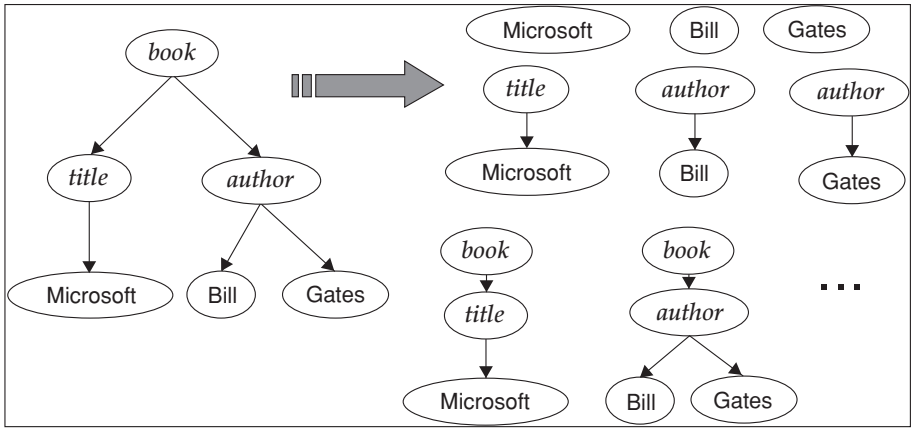

Figure 10.8 A mapping of an XML document (left) to a set of lexicalized subtrees (right).

of documents – subtrees that contain at least one vocabulary term. A subset of these possible lexicalized subtrees is shown in the figure, but there are others – for example, the subtree corresponding to the whole document with the leaf node Gates removed. We can now represent queries and documents as vectors in this space of lexicalized subtrees and compute matches between them. This means that we can use the vector space formalism from Chapter 6 for XML retrieval. The main difference is that the dimensions of vector space in unstructured retrieval are vocabulary terms, whereas they are lexicalized subtrees in XML retrieval.

There is a tradeoff between the dimensionality of the space and accuracy of query results. If we trivially restrict dimensions to vocabulary terms, then we have a standard vector space retrieval system that will retrieve many documents that do not match the structure of the query (e.g., Gates in the title as opposed to the author element). If we create a separate dimension for each lexicalized subtree occurring in the collection, the dimensionality of the space becomes too large. A compromise is to index all paths that end in a single vocabulary term, in other words, all XML-context/term pairs. We call such an XML-context/term pair a *structural term* and denote it by $\langle c, t \rangle$: a pair of XML-context c and vocabulary term t. The document in Figure 10.8 has nine structural terms. Seven are shown (e.g., `"Bill"` and `Author#"Bill"`) and two are not shown: `/Book/Author#"Bill"` and `/Book/Author#"Gates"`. The tree with the leaves Bill and Gates is a lexicalized subtree that is not a structural term. We use the previously introduced pseudo-XPath notation for structural terms.

STRUCTURAL TERM

As we discussed in the last section, users are bad at remembering details about the schema and at constructing queries that comply with the schema. We will therefore interpret all queries as extended queries – that is, there can be an arbitrary number of intervening nodes in the document for any parent–child node pair in the query. For example, we interpret q_5 in Figure 10.7 as q_6.

But we still prefer documents that match the query structure closely by inserting fewer additional nodes. We ensure that retrieval results respect this preference by computing a weight for each match. A simple measure of the similarity of a path c_q in a query and a path c_d in a document is the following *context resemblance* function CR:

CONTEXT RESEMBLANCE

$$\text{CR}(c_q, c_d) = \begin{cases} \frac{1+|c_q|}{1+|c_d|} & \text{if } c_q \text{ matches } c_d \\ 0 & \text{if } c_q \text{ does not match } c_d \end{cases} \tag{10.1}$$

where $|c_q|$ and $|c_d|$ are the number of nodes in the query path and document path, respectively, and c_q matches c_d iff we can transform c_q into c_d by inserting additional nodes. Two examples from Figure 10.6 are $\text{CR}(c_{q_4}, c_{d_2}) = 3/4 = 0.75$ and $\text{CR}(c_{q_4}, c_{d_3}) = 3/5 = 0.6$ where c_{q_4}, c_{d_2}, and c_{d_3} are the relevant paths from top to leaf node in q_4, d_2, and d_3, respectively. The value of $\text{CR}(c_q, c_d)$ is 1.0 if q and d are identical.

The final score for a document is computed as a variant of the cosine measure (Equation (6.10), page 111), which we call SIMNOMERGE for reasons that will become clear shortly. SIMNOMERGE is defined as follows:

$$\begin{aligned} &\text{SIMNOMERGE}(q, d) \\ &= \sum_{c_k \in B} \sum_{c_l \in B} \text{CR}(c_k, c_l) \sum_{t \in V} \text{weight}(q, t, c_k) \frac{\text{weight}(d, t, c_l)}{\sqrt{\sum_{c \in B, t \in V} \text{weight}^2(d, t, c)}} \end{aligned} \tag{10.2}$$

where V is the vocabulary of nonstructural terms; B is the set of all XML contexts; and $\text{weight}(q, t, c)$ and $\text{weight}(d, t, c)$ are the weights of term t in XML context c in query q and document d, respectively. We compute the weights using one of the weightings from Chapter 6, such as, $\text{idf}_t \cdot \text{wf}_{t,d}$. The inverse document frequency idf_t depends on which elements we use to compute df_t, as discussed in Section 10.2. The similarity measure $\text{SIMNOMERGE}(q, d)$ is not a true cosine measure because its value can be larger than 1.0 (Exercise 10.11). We divide by $\sqrt{\sum_{c \in B, t \in V} \text{weight}^2(d, t, c)}$ to normalize for document length (Section 6.3.1, page 111). We have omitted query length normalization to simplify the formula. It has no effect on ranking; for a given query, the normalizer $\sqrt{\sum_{c \in B, t \in V} \text{weight}^2(q, t, c)}$ is the same for all documents.

The algorithm for computing SIMNOMERGE for all documents in the collection is shown in Figure 10.9. The array *normalizer* in Figure 10.9 contains $\sqrt{\sum_{c \in B, t \in V} \text{weight}^2(d, t, c)}$ from Equation (10.2) for each document.

We give an example of how SIMNOMERGE computes query–document similarities in Figure 10.10. $\langle c_1, t \rangle$ is one of the structural terms in the query. We successively retrieve all postings lists for structural terms $\langle c', t \rangle$ with the same vocabulary term t. Three example postings lists are shown. For the first one, we have $\text{CR}(c_1, c_1) = 1.0$ because the two contexts are identical. The next context has no context resemblance with c_1: $\text{CR}(c_1, c_2) = 0$ and the corresponding postings list is ignored. The context match of c_1 with c_3 is $0.63 > 0$, and it will be processed. In this example, the highest ranking document is d_9 with a

```
SCOREDOCUMENTSWITHSIMNOMERGE(q, B, V, N, normalizer)
 1  for n ← 1 to N
 2  do score[n] ← 0
 3  for each ⟨c_q, t⟩ ∈ q
 4  do w_q ← WEIGHT(q, t, c_q)
 5     for each c ∈ B
 6     do if CR(c_q, c) > 0
 7           then postings ← GETPOSTINGS(⟨c, t⟩)
 8                for each posting ∈ postings
 9                do x ← CR(c_q, c) * w_q * weight(posting)
10                   score[docID(posting)] += x
11  for n ← 1 to N
12  do score[n] ← score[n]/normalizer[n]
13  return score
```

Figure 10.9 The algorithm for scoring documents with SIMNOMERGE.

similarity of $1.0 \times 0.2 + 0.63 \times 0.6 = 0.578$. To simplify the figure, the query weight of $\langle c_1, t \rangle$ is assumed to be 1.0.

The query–document similarity function in Figure 10.9 is called SIMNOMERGE because different XML contexts are kept separate for the purpose of weighting. An alternative similarity function is SIMMERGE, which relaxes the matching conditions of query and document further in the following three ways.

- We collect the statistics used for computing weight(q, t, c) and weight(d, t, c) from *all* contexts that have a nonzero resemblance to c (as opposed to just from c as in SIMNOMERGE). For instance, for computing the document frequency of the structural term `atl#"recognition"`, we also count occurrences of recognition in XML contexts `fm/atl`, `article//atl` etc.

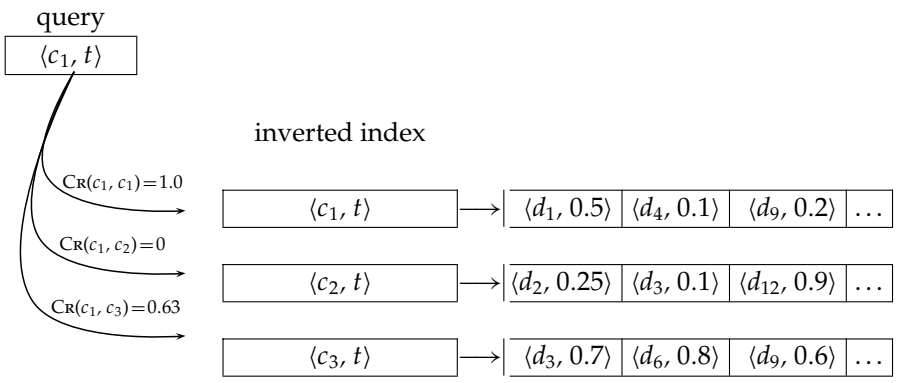

Figure 10.10 Scoring of a query with one structural term in SIMNOMERGE.

- We modify Equation (10.2) by merging all structural terms in the document that have a nonzero context resemblance to a given query structural term. For example, the contexts `/play/act/scene/title` and `/play/title` in the document will be merged when matching against the query term `/play/title#"Macbeth"`.
- The context resemblance function is further relaxed: Contexts have a nonzero resemblance in many cases where the definition of CR in Equation (10.1) returns 0.

See the references in Section 10.6 for details.

These three changes alleviate the problem of sparse term statistics discussed in Section 10.2 and increase the robustness of the matching function against poorly posed structural queries. The evaluation of SIMNOMERGE and SIMMERGE in the next section shows that the relaxed matching conditions of SIMMERGE increase the effectiveness of XML retrieval.

? **Exercise 10.1** Consider computing the document frequency for a structural term as the number of times that the structural term occurs under a particular parent node. Assume the following: The structural term $\langle c, t\rangle =$ `author#"Herbert"` occurs once as the child of the node *squib*; there are ten *squib* nodes in the collection; $\langle c, t\rangle$ occurs 1,000 times as the child of *article*; there are 1,000,000 *article* nodes in the collection. The idf weight of $\langle c, t\rangle$ then is $\log_2 10/1 \approx 3.3$ when occurring as the child of *squib* and $\log_2 1{,}000{,}000/1000 \approx 10.0$ when occurring as the child of *article*. (i) Explain why this is not an appropriate weighting for $\langle c, t\rangle$. Why should $\langle c, t\rangle$ not receive a weight that is three times higher in articles than in squibs? (ii) Suggest a better way of computing idf.

Exercise 10.2 Write down all the structural terms occurring in the XML document in Figure 10.8.

Exercise 10.3 How many structural terms does the document in Figure 10.1 yield?

10.4 Evaluation of XML retrieval

INEX The premier venue for research on XML retrieval is the *INEX* (**IN***itiative for the* **E***valuation of* **X***ML retrieval*) program, a collaborative effort that has produced reference collections, sets of queries, and relevance judgments. A yearly INEX meeting is held to present and discuss research results. The INEX 2002 collection consisted of about 12,000 articles from IEEE journals. We give collection statistics in Table 10.2 and show part of the schema of the collection in Figure 10.11. The IEEE journal collection was expanded in 2005. Since 2006, INEX uses the much larger English Wikipedia as a test collection.

Table 10.2 INEX 2002 collection statistics.

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

The relevance of documents is judged by human assessors using the methodology introduced in Section 8.1 (page 140), appropriately modified for structured documents as we will discuss shortly.

Two types of information needs or topics in INEX are content-only (CO) topics and content-and-structure (CAS) topics. *CO topics* are regular keyword queries as in unstructured information retrieval. *CAS topics* have structural constraints in addition to keywords. We already encountered an example of a CAS topic in Figure 10.3. The keywords in this case are summer and holidays, and the structural constraints specify that the keywords occur in a section that in turn is part of an article and that this article has an embedded year attribute with value 2001 or 2002.

CO TOPICS

CAS TOPICS

Because CAS queries have both structural and content criteria, relevance assessments are more complicated than in unstructured retrieval. INEX 2002 defined component coverage and topical relevance as orthogonal dimensions of relevance. The *component coverage* dimension evaluates whether the

COMPONENT COVERAGE

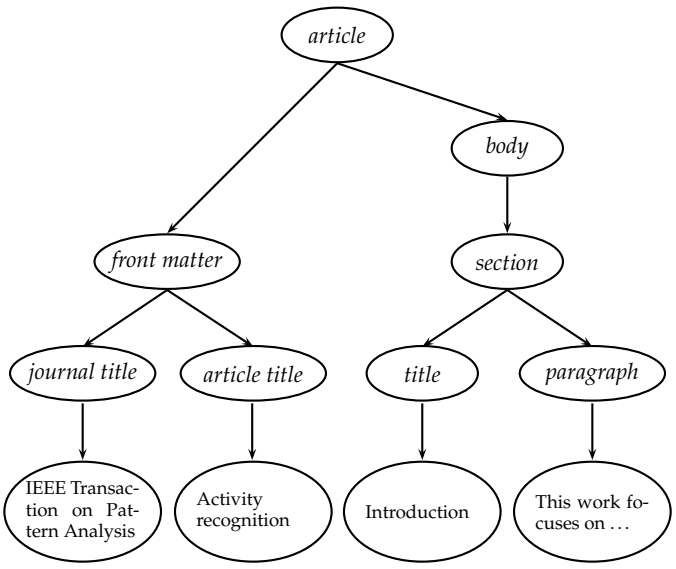

Figure 10.11 Simplified schema of the documents in the INEX collection.

element retrieved is "structurally" correct, that is, 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.
- 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.

TOPICAL RELEVANCE

The *topical relevance* dimension also has four levels: highly relevant (3), fairly relevant (2), marginally relevant (1), and nonrelevant (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 sixteen combinations of coverage and relevance, but many cannot occur. For example, a nonrelevant component cannot have exact coverage, so the combination 3N is not possible.

The relevance–coverage combinations are quantized as follows:

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

This evaluation scheme takes account of the fact that binary relevance judgments, which are standard in unstructured information retrieval (Section 8.5.1, page 153), are not appropriate for XML retrieval. A 2S component provides incomplete information and may be difficult to interpret without more context, but it does answer the query partially. The quantization function $\mathbf{Q}$ does not impose a binary choice relevant/nonrelevant and instead allows us to grade the component as partially relevant.

The number of relevant components in a retrieved set A of components can then be computed as:

$$\#(\text{relevant items retrieved}) = \sum_{c \in A} \mathbf{Q}(rel(c), cov(c)).$$

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 graded as opposed to binary relevance assessments. See the references on focused retrieval in Section 10.6 for further discussion.

Table 10.3 INEX 2002 results of the vector space model in Section 10.3 for CAS queries and the quantization function **Q**.

algorithm	average precision
SIMNOMERGE	0.242
SIMMERGE	0.271

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 Section 8.5.1 (page 153). This problem is worse in XML retrieval because of the problem of multiple nested elements occurring in a search result as we discussed on page 185. Much of the recent focus at INEX has been on developing algorithms and evaluation measures that return nonredundant results lists and evaluate them properly. See the references in Section 10.6.

Table 10.3 shows two INEX 2002 runs of the vector space system we described in Section 10.3. The better run is the SIMMERGE run, which incorporates few structural constraints and mostly relies on keyword matching. SIMMERGE's median average precision (where the median is with respect to average precision numbers over topics) is only 0.147. Effectiveness in XML retrieval is often lower than in unstructured retrieval because XML retrieval is harder. Instead of just finding a document, we have to find the subpart of a document that is most relevant to the query. Also, XML retrieval effectiveness – when evaluated as described here – can be lower than unstructured retrieval effectiveness on a standard evaluation because graded judgments lower measured performance. Consider a system that returns a document with graded relevance 0.6 and binary relevance 1 at the top of the retrieved list. Then, interpolated precision at 0.00 recall (cf. page 145) is 1.0 on a binary evaluation, but can be as low as 0.6 on a graded evaluation.

Table 10.3 gives us a sense of the typical performance of XML retrieval, but it does not compare structured with unstructured retrieval. Table 10.4 directly shows the effect of using structure in retrieval. The results are for a language-model–based system (cf. Chapter 12) that is evaluated on a subset of CAS topics from INEX 2003 and 2004. The evaluation metric is precision

Table 10.4 A comparison of content-only and full-structure search in INEX 2003/2004.

	content only	full structure	improvement
precision at 5	0.2000	0.3265	63.3%
precision at 10	0.1820	0.2531	39.1%
precision at 20	0.1700	0.1796	5.6%
precision at 30	0.1527	0.1531	0.3%

at k as defined in Chapter 8 (page 148). The discretization function used for the evaluation maps highly relevant elements (roughly corresponding to the 3E elements defined for **Q**) to 1 and all other elements to 0. The content-only system treats queries and documents as unstructured bags of words. The full-structure model ranks elements that satisfy structural constraints higher than elements that do not. For instance, for the query in Figure 10.3 an element that contains the phrase summer holidays in a *section* will be rated higher than one that contains it in an *abstract*.

The table shows that structure helps to increase precision at the top of the results list. There is a large increase of precision at $k = 5$ and at $k = 10$. There is almost no improvement at $k = 30$. These results demonstrate the benefits of structured retrieval. Structured retrieval imposes additional constraints on what to return and documents that pass the structural filter are more likely to be relevant. Recall may suffer because some relevant documents will be filtered out, but for precision-oriented tasks structured retrieval is superior.

10.5 Text-centric versus data-centric XML retrieval

In the type of structured retrieval we cover in this chapter, XML structure serves as a framework within which we match the text of the query with the text of the XML documents. This exemplifies a system that is optimized for *text-centric XML*. Although both text and structure are important, we give higher priority to text. We do this by adapting unstructured retrieval methods to handling additional structural constraints. The premise of our approach is that XML document retrieval is characterized by (i) long text fields (e.g., sections of a document), (ii) inexact matching, and (iii) relevance-ranked results. Relational databases do not deal well with this use case.

TEXT-CENTRIC XML

In contrast, *data-centric XML* mainly encodes numerical and nontext attribute-value data. When querying data-centric XML, we want to impose exact match conditions in most cases. This puts the emphasis on the structural aspects of XML documents and queries. An example is:

DATA-CENTRIC XML

> Find employees whose salary is the same this month as it was 12 months ago.

This query requires no ranking. It is purely structural and an exact matching of the salaries in the two time periods is probably sufficient to meet the user's information need.

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 because most text documents have some form of interesting structure – paragraphs, sections,

footnotes, and so on. Examples include assembly manuals, issues of journals, Shakespeare's collected works, and newswire articles.

Data-centric approaches are commonly used for data collections with complex structures that mainly contain nontext data. A text-centric retrieval engine will have a hard time with proteomic data in bioinformatics or with the representation of a city map that (together with street names and other textual descriptions) forms a navigational database.

Two other types of queries that are difficult to handle in a text-centric structured retrieval model are joins and ordering constraints. The query for employees with unchanged salary requires a join. The following query imposes an ordering constraint:

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

This query relies on the ordering of elements in XML – in this case the ordering of chapter elements underneath the book node. There are powerful query languages for XML that can handle numerical attributes, joins, and ordering constraints. The best known 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. Due to its complexity, it is challenging to implement an XQuery-based ranked retrieval system with the performance characteristics that users have come to expect in information retrieval. This is currently one of the most active areas of research in XML retrieval.

Relational databases are better equipped to handle many structural constraints, particularly joins (but ordering is also difficult in a database framework – the tuples of a relation in the relational calculus are not ordered). For this reason, most data-centric XML retrieval systems are extensions of relational databases (see the references in Section 10.6). If text fields are short, exact matching meets user needs and retrieval results in form of unordered sets are acceptable, then using a relational database for XML retrieval is appropriate.

? **Exercise 10.4** Find a reasonably sized XML document collection (or a collection using a markup language different from XML like HTML) on the web and download it. Jon Bosak's XML edition of Shakespeare and of various religious works at www.ibiblio.org/bosak/ or the first 10,000 documents of the Wikipedia are good choices. Create three CAS topics of the type shown in Figure 10.3 that you would expect to do better than analogous CO topics. Explain why an XML retrieval system would be able to exploit the XML structure of the documents to achieve better retrieval results on the topics than an unstructured retrieval system.

Exercise 10.5 For the collection and the topics in Exercise 10.4, (i) are there pairs of elements e_1 and e_2, with e_2 a subelement of e_1 such that both

answer one of the topics? Find one case each where (ii) e_1 (iii) e_2 is the better answer to the query.

Exercise 10.6 Implement the (i) SimMerge (ii) SimNoMerge algorithm in Section 10.3 and run it for the collection and the topics in Exercise 10.4. (iii) Evaluate the results by assigning binary relevance judgments to the first five documents of the three retrieved lists for each algorithm. Which algorithm performs better?

Exercise 10.7 Are all of the elements in Exercise 10.4 appropriate to be returned as hits to a user or are there elements (as in the example `<b>definitely</b>` on page 185) that you would exclude?

Exercise 10.8 We discussed the tradeoff between accuracy of results and dimensionality of the vector space on page 189. Give an example of an information need that we can answer correctly if we index all lexicalized subtrees, but can not answer if we only index structural terms.

Exercise 10.9 If we index all structural terms, what is the size of the index as a function of text size?

Exercise 10.10 If we index all lexicalized subtrees, what is the size of the index as a function of text size?

Exercise 10.11 Give an example of a query–document pair for which SimNoMerge(q, d) is larger than 1.0.

10.6 References and further reading

There are many good introductions to XML, including (Harold and Means 2004). Table 10.1 is inspired by a similar table in (van Rijsbergen 1979). Section 10.4 follows the overview of INEX 2002 by Gövert and Kazai (2003), published in the proceedings of the meeting (Fuhr et al. 2003a). The proceedings of the four following INEX meetings were published as Fuhr et al. (2003b), Fuhr et al. (2005), Fuhr et al. (2006), and Fuhr et al. (2007). An up-to-date overview article is Fuhr and Lalmas (2007). The results in Table 10.4 are from (Kamps et al. 2006). Chu-Carroll et al. (2006) also present evidence that XML queries increase precision compared with unstructured queries. Instead of coverage and relevance, INEX now evaluates on the related but different dimensions of exhaustivity and specificity (Lalmas and Tombros 2007). Trotman et al. (2006) relate the tasks investigated at INEX to real-world uses of structured retrieval such as structured book search on Internet bookstore sites.

The structured document retrieval principle is due to Chiaramella et al. (1996). Figure 10.5 is from (Fuhr and Großjohann 2004). Rahm and Bernstein (2001) give a survey of automatic schema matching that is applicable to XML.

The vector-space–based XML retrieval method in Section 10.3 is essentially IBM Haifa's JuruXML system as presented by Mass et al. (2003) and Carmel et al. (2003). Schlieder and Meuss (2002) and Grabs and Schek (2002) describe similar approaches. Carmel et al. (2003) represent queries as *XML fragments*. The trees that represent XML queries in this chapter are all XML fragments, but XML fragments also permit the operators $+$, $-$, and *phrase* on content nodes.

XML FRAGMENT

We chose to present the vector space model for XML retrieval because it is simple and a natural extension of the unstructured vector space model in Chapter 6. But many other unstructured retrieval methods have been applied to XML retrieval with at least as much success as the vector space model. These methods include language models (cf. Chapter 12; e.g., Kamps et al. (2004), List et al. (2005), Ogilvie and Callan (2005)), systems that use a relational database as a backend (Mihajlović et al. 2005; Theobald et al. 2005, 2008), probabilistic weighting (Lu et al. 2007), and fusion (Larson 2005). There is currently no consensus as to what the best approach to XML retrieval is.

Most early work on XML retrieval accomplished relevance ranking by focusing on individual terms, including their structural contexts, in query and document. As in unstructured IR, there is a trend in more recent work to model relevance ranking as combining evidence from disparate measurements about the query, the document, and their match. The combination function can be tuned manually (Arvola et al. 2005; Sigurbjörnsson et al. 2004) or trained using machine learning methods (Vittaut and Gallinari (2006), cf. Section 15.4.1, page 314).

An active area of XML retrieval research is *focused retrieval* (Trotman et al. 2007), which aims to avoid returning nested elements that share one or more common subelements (cf. discussion in Section 10.2, page 185). There is evidence that users dislike redundancy caused by nested elements (Betsi et al. 2006). Focused retrieval requires evaluation measures that penalize redundant results lists (Kazai and Lalmas 2006; Lalmas et al. 2007). Trotman and Geva (2006) argue that XML retrieval is a form of *passage retrieval*. In passage retrieval (Salton et al. 1993; Hearst and Plaunt 1993; Zobel et al. 1995; Hearst 1997; Kaszkiel and Zobel 1997), the retrieval system returns short passages instead of documents in response to a user query. Although element boundaries in XML documents are cues for identifying good segment boundaries between passages, the most relevant passage often does not coincide with an XML element.

FOCUSED RETRIEVAL

PASSAGE RETRIEVAL

In the last several years, the query format at INEX has been the NEXI standard proposed by Trotman and Sigurbjörnsson (2004). Figure 10.3 is from their paper. O'Keefe and Trotman (2004) give evidence that users cannot reliably distinguish the child and descendant axes. This justifies only permitting descendant axes in NEXI (and XML fragments). These structural constraints were only treated as "hints" in recent INEXes. Assessors can judge an

element highly relevant, even though it violates one of the structural constraints specified in a NEXI query.

An alternative to structured query languages like NEXI is a more sophisticated user interface for query formulation (Tannier and Geva 2005; van Zwol et al. 2006; Woodley and Geva 2006).

A broad overview of XML retrieval that covers database as well as IR approaches is given by Amer-Yahia and Lalmas (2006) and an extensive reference list on the topic can be found in (Amer-Yahia et al. 2005). Chapter 6 of Grossman and Frieder 2004 is a good introduction to structured text retrieval from a database perspective. The proposed standard for XQuery is available at www.w3.org/TR/xquery/ including an extension for full-text queries (Amer-Yahia et al. 2006): www.w3.org/TR/xquery-full-text/. Work that has looked at combining the relational database and the unstructured information retrieval approaches includes (Fuhr and Rölleke 1997); (Navarro and Baeza-Yates 1997); (Cohen 1998); and (Chaudhuri et al. 2006).