

Context-Sensitive Search and Exploration of XML Text

Thomas Baby
thomas@cs.umd.edu

Sudarshan S. Chawathe
chaw@cs.umd.edu

Department of Computer Science
University of Maryland, College Park, MD 20742

Abstract

XML permits documents with arbitrary nested context (tag structure). We investigate how this context may be used to aid the task of searching and exploring XML text. We describe the design and implementation of the Cextor system, which includes a context-sensitive text-search engine. Cextor also includes a novel technique for organizing and exploring very large search results based on context. A distinguishing feature of this technique is that it does not assume search results are of modest size. Rather, it is designed to cope with search results that are potentially the size of the database. We present the results of an experimental evaluation of Cextor on derived data from the Web.

1 Introduction

The ability to easily locate information on the Internet is significantly improving the efficiency of scientific and business activities. Given the size and rapid growth of the Internet, especially in recent years, the design of scalable systems for searching networked documents remains challenging. Nevertheless, the availability of commercial search engines such as Google has considerably eased the task of locating documents that can be accurately described using a few distinguishing terms. For example, it is not difficult to

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

This work was supported by the National Science Foundation with grants IIS-9984296 (CAREER) and IIS-0081860 (ITR).

**Submitted to the 27th VLDB Conference,
Rome, Italy, 2001.**

find information about the `ide-scsi` driver for Linux using Google and the query `linux ide-scsi`. Our task in this example was simplified by our knowledge (or assumption) that relevant documents contain the term `ide-scsi`, which occurs infrequently in the document collection. Unfortunately, this happy circumstance is more an exception than the norm and we must often search for documents that cannot be discriminated this easily. Continuing our example, suppose we are looking for information on monitors that work well with Linux. Several of the obvious Google queries (e.g., `linux monitor`) return a very large (800 thousand) matches. Further, a high proportion of the first few matches are not relevant to monitor hardware, but use the term `monitor` in other contexts (e.g., `network monitor`, `diald monitor`). Successive refinements (e.g., `linux monitor -network`, `linux monitor -network display hardware`) yield progressively more relevant results.

Such refinement requires one to first examine the early search results in order to determine the terms that may help in filtering out irrelevant results. This task is often complicated by the presence of documents that use the same word or phrase in different contexts (e.g., the use of the word `monitor` in our example). Unless one is very careful, relevant documents may be inadvertently eliminated from the result. In our example, the addition of refinement term `-network` (intended to remove documents describing network monitors and not computer displays) results in the elimination of several helpful documents from organizations with the word `network` in their names (e.g., Maximum Linux Network).

The importance of the context in which words appear in a document is well recognized in the Information Retrieval literature, as is the need for effective (efficient and usable) refinement mechanisms. However, most documents on the Web are in HTML format, which is severely limited in its ability to encode meaningful context. While a few fixed contexts (e.g., title, headings) are available, there is no way to define and use more meaningful contexts (e.g., hardware review, price). Further, since HTML mixes content with its presentation, many documents misuse HTML tags

for formatting purposes, resulting in further complications. Therefore, the simple form of context-sensitivity found in some search engines (e.g., `title:review` in AltaVista) results in very limited improvements.

The emergence of XML and related technologies promises to improve the situation by cleanly separating data from its presentation. In particular, XML documents may define and use their own context hierarchies (by nesting user-defined tags). For example, the word Stewart on line 10 of document 1 in Figure 1 is marked with the tag `name`. Start and end tags (e.g., `<writer>` and `</writer>` delimit an *element* that we shall identify with the name of the tag (lines 9–12 of document 1). Elements can be nested (e.g., the above `writer` element has `name` subelements on lines 10 and 11; the `writer` element is, in turn, a subelement of the `show` element beginning on line 7. The context depends on all the ancestors of the element in which a word appears. For example, the context of the `name` element on line 8 of document 1 is different from that of the `name` element on line 10. We distinguish these contexts by using their fully qualified forms: `/guide/theater/show/name` and `/guide/theater/show/writer/name`, respectively.

The ability to define document-specific (more commonly, application- and domain-specific) contexts leads to both opportunities and challenges. On the one hand, proper use of this added power can help alleviate the problems described earlier. On the other hand, the unbridled use of user-defined contexts can result in difficulties in their interpretation. Continuing our example, an XML document containing the fragment `<Monitor>... <Size>18</Size>... </Monitor>` provides a more precise method for locating 18-inch computer monitors compared with what is possible with HTML documents (e.g., a Google search for `monitor 18`). However, while it is tempting to assume the most obvious interpretation of the elements, there is no guarantee that this interpretation is correct. In our example, the XML document could be the configuration file for a network monitoring tool, with the `size` element indicating the size, in bytes, of test packets.

Similar observations have resulted in a flurry of activity on the standardization of XML tags in various communities [?]. Recognizing that complete global standardization for all domains is unlikely, there has also been work on standardized specification of semantics and ontologies and on the integration of such specifications [?]. Such work aims to arrive at an integrated, semantically consistent version of all relevant XML documents (either by standardization or by reasoning with ontologies) and is not the focus of this paper.

In this paper, we adopt a different view: In the near future, there are likely to be many XML documents that do not adhere to the kind of careful se-

mantic specifications that the standardization work demands. Further, even in the long term, a diverse and autonomous environment such as the Web will always contain a significant amount of useful information in documents that are semantically unconstrained or ill formed (perhaps because the generator of such information does not have the motivation or resources to put it in a standard form). Of course, tools for searching XML could always ignore such documents; however, they would then be rather limited in their reach. In order to benefit from the information in such documents, we believe it is important to study the following problem, which is the **focus of this paper**: *How can we improve the effectiveness of XML search without assuming anything other than well-formedness of XML?* (Intuitively, an XML document is well-formed if it satisfies some very simple syntactic constraints, such as proper nesting of elements.) Our work shares this guiding principle with recent work in semistructured data: Structure is considered descriptive, but not prescriptive. Our goal is to make the best use of any available structure (context) without insisting on any particular structure.

To address the above problem, we have designed and implemented the *Cextor* system. *Cextor* implements context-sensitive boolean queries on XML documents. Intuitively, the query `fosse IN /guide/show/name AND NOT fosse IN /guide/show/director/name` matches XML documents containing the word `fosse` in the first context but not in the second. (Details appear in Section 2.) This query language is implemented using some simple and effective extensions to the standard inverted file data structures. Unlike common search engines, the execution of a *Cextor* query results in more than an annotated list of document identifiers. Instead, the matching documents (and matching locations and contexts within them) are organized in an intuitive and efficient data structure, called the *context tree*. Intuitively, the context tree groups the documents in a query result based on the contexts in which they match the query terms. *Cextor* provides three operations for *exploring* the query results through the context tree: *navigation* (expanding and hiding tree nodes), *refinement* (filtering results), and *anchoring* (reorganizing the tree using a new node as root). The context tree and the exploration operations serve as efficient building blocks for expressive interfaces that integrate search and exploration of a large XML document collection. We do *not* assume that the result of a query contains a modest number of documents. Instead, the context tree and the exploratory operations are designed to efficiently operate on query results that are comparable in size to the entire document collection.

We have built a complete system, including a user interface. However, our interest lies primarily

```

1:<guide>
2: <city> New York </city> <state> New York </state>
3: <theater> Ford Center for Performing Arts
4: <address>
5: <street> 213 West 42nd Street </street>
6: </address>
7: <show>
8: <name> 42nd Street </name>
9: <writer>
10: <name> Michael Stewart </name>
11: <name> Mark Bramble </name>
12: </writer>
13: <director> Gower Champion </director>
14: </show>
15: </theater>
16: <theater> Broadhurst Theatre
17: <address>
18: <street> 235 West 44th Street </street>
19: </address>
20: <show>
21: <name> Fosse </name>
22: <director> Ann Reinking </director>
23: </show>
24: </theater>
25:</guide>

```

(a) Document 1

```

1:<guide>
2: <city> New York </city> <state> New York </state>
3: <broadway> <theater>
4: <name> Shubert Theatre </name>
5: <address> 225 West 44th Street </address>
6: <show>
7: <name> Chicago </name>
8: <writer>
9: <name>John Kander</name>
10: <name>Fred Ebb</name>
11: </writer>
12: <director> Bob Fosse </director>
13: </show>
14: </theater>
15: <theater>
16: <name>American Airlines Theatre </name>
17: <address> 227 West 42nd Street </address>
18: <show>
19: <name> Design for Living </name>
20: <playwright> Noel Coward </playwright>
21: <director> Joe Mantello </director>
22: </show>
23: </theater> </broadway>
24:</guide>

```

(b) Document 2

Figure 1: Sample XML Documents

in the data-centric query-and-exploration operations that (through the Cextor application programming interface) enable an expressive user interface, not in the interface itself. Further, since the number of XML documents on the public Web is much smaller than the number of HTML documents, we have tested our system by crawling and indexing HTML, not XML, documents. While using such HTML (converted to XML as XHTML) suffices for testing our ideas, the test system is not as intuitive to use as is one based on XML. Our contribution is not the test system, but the Cextor system that is capable of indexing any XML (or HTML) collection. We have made the Cextor source code publicly available (GNU GPL terms) at <http://www.cs.umd.edu/projects/cextor/>.

In summary, our **primary contributions** in this paper are (1) an index structure for XML that implements context-sensitive boolean queries; (2) an extension to this structure for speeding up XML queries in languages similar to XML-QL; (3) methods for organizing and exploring very large search results; (4) an experimental evaluation of our work; and (5) an implemented system whose source code is publicly available.

2 The Cextor System

In this section, we describe our system for search and exploration of XML documents. We begin with some preliminary definitions followed by a description of the syntax and semantics of our query language. Next, we present the context tree that forms the basis of our the Cextor application programming interface (API). We describe our simple interface based on this API. We then describe the exploration operations introduced

in the previous section. Finally, we discuss the implementation techniques for the indexing and exploratory modules.

2.1 Context and Context Expression

The **context** of an element in a document is the string formed by concatenating, in order, the /-prefixed tags of elements on the path from the document root to the node corresponding to the element. The **context** of a word or phrase in a document is the context of the element containing it. For example, the context of the word “fosse” in line 21 of Document 1 (Figure 1) is the string */guide/theater/show/name*.

A **context expression** is a string that identifies one or more contexts, each of which is said to *match* the context expression. A context expression is formed by concatenating tags, separated by either / or //. The separators / and // specify parent-child and ancestor-descendant nesting relationships, respectively, that must hold between tags in contexts that match the context expression.

Example 2.1 The context expression */guide//show/director* specifies that, in a context matching the context expression, a **show** element must be a descendant of a **guide** element (because of // separating **guide** and **show** tags) that is the document’s root. In addition, the **show** element must have a **director** element as its child (because of / separating **show** and **director** tags). The context */guide/theater/show/director* is a context that matches the context expression. However, the context */guide/city* does not match the context expression.

In this paper, strings representing contexts are typeset using *italic font* (e.g., */guide/theater/show/name*) whereas strings representing context expressions are typeset using *typewriter font* (e.g., */guide//show/director*).

2.2 Query Language

A query consists of one or more *query terms*, combined using the boolean connectives AND, OR, and NOT. A **query term** is either a word or a phrase. It can be optionally **qualified** with a context expression, using keywords *IN* (denoting *containment*) or *DIN* (denoting *direct containment*). The context expression is said to **qualify** the query term.

The context expression that qualifies a query term identifies *interesting* instances of the query term in the document repository. If a query term and the context expression that qualifies it are connected using *DIN* (e.g., *42nd DIN /guide//show*), an instance of the query term in a document is **interesting** if it is contained within an element whose context matches the context expression. If a query term and its context expression are connected using *IN* (e.g., *fosse IN /guide//show*), an instance of the query term in a document is **interesting** if it is contained within an element or within the descendant of an element whose context matches the context expression. The boolean connectives combine constraints in the usual manner.

Example 2.2 Consider the query *fosse DIN /guide//show/director* on the two documents in Figure 1. The instance of the query term “*fosse*” in line 12 of Document 2 is interesting because it has the context */guide/broadway/theater/show/director*, which matches */guide//show/director*. However, the instance of the query term “*fosse*” in line 21 of Document 1 is not interesting because it has the context */guide/theater/show/name*, which does not match */guide//show/director*.

Example 2.3 Consider the query *fosse IN /guide//show* on the two documents in Figure 1. The instance of “*fosse*” in line 12 of Document 2 is interesting because it is contained within a **director** element, whose parent’s context (*/guide/broadway/theater/show*) matches the context expression */guide//show*. The instance of “*fosse*” in line 21 of Document 1 is also interesting because it is contained within a **name** element, whose parent’s context (*/guide/theater/show*) matches the context expression.

The result of a query consists of a set of documents and a set of contexts. We define the **document set** of a query term as the set of documents that have at least one interesting instance of the query term. The set of documents in the result is formed by combining the document sets of the query terms using union, intersection, and difference, corresponding to OR, AND,

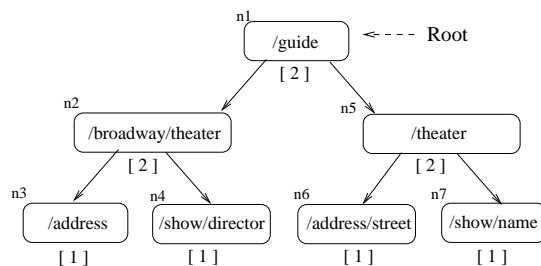


Figure 2: **Example of a Context Tree.**

and NOT, respectively. The **context set** of a document is the set containing the contexts of all interesting query term instances that are present in the document. The set of contexts in the result of a query is the union of the context sets of documents in the result. We call this set of contexts the **span** of the query.

Example 2.4 Consider the Cextor query (*42nd IN /guide//theater/address*) AND (*fosse IN /guide//show*) on the two documents in Figure 1. Document 1 contains one interesting instance of “*42nd*” (line 5, context */guide/theater/address/street*). Document 2 also contains one interesting instance of “*42nd*” (line 17, context */guide/broadway/theater/address*). The document set of the query term “*42nd*” contains both documents 1 and 2. The document set of the query term “*fosse*” also contains both the documents. The set of documents in the result of the query is the intersection (corresponding to AND) of the document sets of “*42nd*” and “*fosse*.” Document 1 contains two interesting query term instances (“*42nd*” in line 5 and “*fosse*” in line 21). Its context set contains the contexts of these interesting instances: */guide/theater/address/street* and */guide/theater/show/name*. The context set of Document 2 contains contexts */guide/broadway/theater/address* and */guide/broadway/theater/show/director*. The span of the query is the union of the context sets of documents 1 and 2. It contains four contexts: */guide/theater/address/street*, */guide/theater/show/name*, */guide/broadway/theater/address*, and */guide/broadway/theater/show/director*.

2.3 Result Presentation and Exploration

Cextor presents the result of a query as a rooted, labeled tree, called the **context tree**, which represents the span of the query. The context tree is a trie that is built using strings of the alphabet of tags [Knu00]. Each context in the span maps to a root-leaf path in the tree. The string formed by concatenating the node labels along any root-leaf path is a context in the span. Contexts that share a prefix map to paths that share nodes in the context tree. (If there is no prefix common to all paths, the context tree has a dummy root with the empty string as its label.)

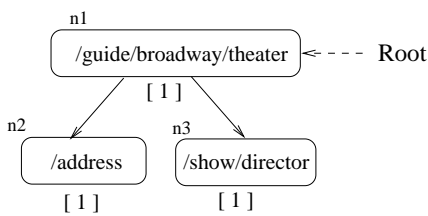


Figure 3: **Context Tree Refinement.**

Example 2.5 Figure 2 illustrates the context tree that is presented as output of the query (42nd IN /guide//theater/address) AND (fosse IN /guide//show) on the two documents in Figure 1. The context corresponding to the root-leaf path $n_1 - n_5 - n_7$ is /guide/theater/show/name, obtained by concatenating the labels of nodes n_1 , n_5 , and n_7 .

We define the **path set** of a node in the context tree to be the set of contexts whose paths include that node. We define the **document set** of a node in the context tree to be the set of result documents whose context sets have a non-empty intersection with the path set of the node. The number next to a node is the size of the node’s document set, which can be viewed by clicking the node.

Since contexts that share a common prefix map to paths that share nodes in the context tree, the context tree represents the span of a query more compactly than a linear list. However, if the span is large, its context tree can be too large to display within the available screenspace. In Cextor, we limit the number of displayed nodes based on three parameters derived from a user specified screensize: their depths, the number of contexts in their path sets, and the number of documents in their document sets. By clicking on a node at which the tree is truncated (such a node is identifiable by its color), the user may expand the subtree that is rooted at that node. In this expanded view of the subtree, the user can also see the path that leads to the root of the subtree. Other nodes can also be clicked to turn on or off the display of the subtrees rooted at them.

In spite of the ability to selectively control the display of subtrees, navigation of the context tree to explore the query result can be cumbersome if the span of the query is very large. We introduce two operations—*refinement* and *anchoring*—to address this deficiency.

Refinement: Often, a user may not know the tags used in a document corpus and may be unable to specify context expressions. Therefore, the initial query result may be too large (low precision). However, after viewing the query’s context tree either in its entirety or in its truncated form, the user may gather enough information about tags to be able to specify more precise context expressions for one or more terms in the query.

The refinement operation takes as input context expressions for one or more query terms. It uses the context expression input for a query term to further constrain contexts of interesting instances of the query term. Based on the new set of interesting instances of a query term, it updates the document set of the query term. It uses the new document sets to update the documents in the result and the span of the query. The output of refinement is the context tree built using the new span.

Example 2.6 Consider the query (42nd IN /guide//theater/address) AND (fosse IN /guide//show) on the documents in Figure 1. Figure 2 illustrates the initial context tree displayed after execution of the query. The user may choose to refine the term “fosse” using the context expression /guide//show/director. The result of refinement is the same as the result of the query (42nd IN /guide//theater/address) AND (fosse IN /guide//show/director), and the corresponding context tree is shown in Figure 3.

Anchoring: Among contexts in the span of a query (Section 2.2), a user may sometimes be interested only in those that include a specific tag (e.g., **theater**). Using simple pattern matching, contexts that do not include the tag can be eliminated from the span. However, the remaining contexts may have the tag at different depths (e.g., /guide/broadway/theater/address and /guide/theater/address/street). Even if all contexts have the tag at the same depth, they may not have a common prefix that includes the tag. As a consequence, the context tree built using these contexts has the tag scattered across multiple nodes, making it difficult to visualize the nesting of relevant tags. For example, in the context tree built using the span of the query (42nd IN /guide//theater/address) AND (fosse IN /guide//show) on the two documents of Figure 1, the **theater** tag appears on the labels of nodes n_2 and n_5 (Figure 2). Although in this toy example it is not difficult to visualize tag nesting, the problem is a serious one in a typical context tree containing hundreds of nodes.

The anchoring operation takes a tag and a context tree as inputs. First, it removes from the tree contexts that do not include the tag. Next, it aligns the remaining contexts at the positions of the tag in them. It splits each context into two parts: an *outer context* and an *inner context*. The **outer context** of a context is the context prefix that ends at the position of the input tag in the context. The **inner context** of a context is the context suffix that begins at the position of the input tag in the context. For example, if the input tag is **theater**, the outer context and the inner context of /guide/broadway/theater/address are /guide/broadway/theater and /theater/address, respectively. We reverse each outer context by flipping

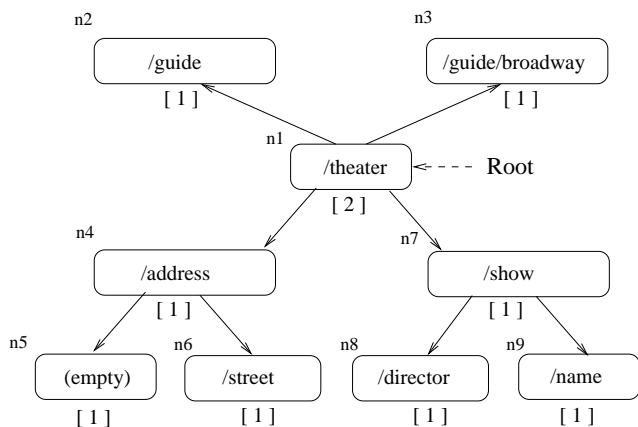


Figure 4: **Context Tree Anchoring.**

the order of tags in it. For example, the reverse of */guide/broadway/theater* is */theater/broadway/guide*. We use all inner contexts and all outer contexts to build two context trees: an *inner tree* and an *outer tree*. The **inner tree** is the context tree built using the set of all inner contexts. The **outer tree** is the context tree built using the set of all reversed outer contexts. We reverse the labels of nodes of the outer tree, and collapse the roots of the outer tree and the inner tree, to generate the result of anchoring. We display the result with the outer tree drawn upside down.

Example 2.7 Figure 4 illustrates the tree that is the result of anchoring the tree in Figure 2 using the **theater** tag. The outer contexts are */guide/theater* and */guide/broadway/theater*. The outer tree nodes have labels */guide*, */theater*, and */broadway/guide*. The outer tree is inverted, its node labels are reversed (See nodes n_1 , n_2 , and n_3 in Figure 4), and its root is collapsed with the root of the inner tree, which is displayed without modification.

2.4 Implementation

Figure 5 illustrates the high-level architecture of Cextor. Cextor consists of three main subsystems. The gathering and cleaning subsystem is responsible for crawling the Web, downloading all HTML documents at depth below a given value (parameter, set to 40 in our experiments), and cleaning them using the *Tidy* software to heuristically remove faulty HTML and insert closing tags to convert HTML to XHTML. The querying subsystem is responsible for building the inverted file indexes described earlier and using them to implement the Cextor query language. We describe it more detail below. responsible for implementing the context tree along with the API operations of navigation, refinement, and anchoring.

The Context Index The context index includes a *dictionary* that contains all words in the document repository, except those that occur only as tags. The *context list* for a word in the dictionary is a sorted list

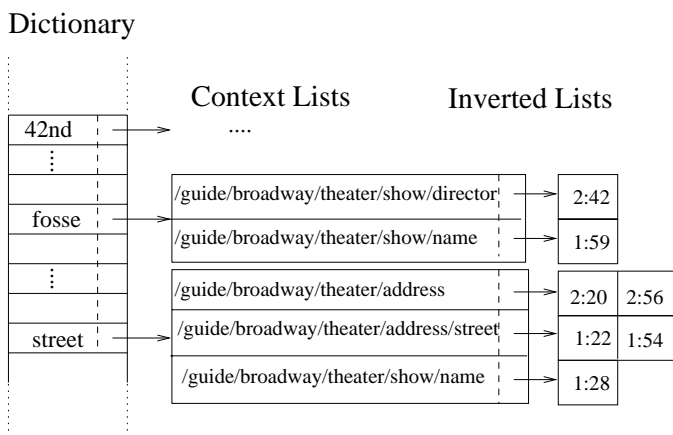


Figure 6: **The Context Index.**

containing the contexts of all instances of the word in the document collection. The dictionary contains a pointer to the context list for each word in it. For each context, the context list contains a pointer to an *inverted list*, which is a sorted list of *postings*. Each posting is a pair of integers: the identifier of a document containing an instance of the word within that context and the offset of the instance within the document.

Example 2.8 In Figure 6, we show a portion of the context index for the two documents in Figure 1. The word “street” occurs in Document 1 (lines 5, 7, and 17) and Document 2 (lines 6 and 16). Its context list has three contexts: */guide/broadway/theater/address*, */guide/broadway/theater/address/street*, and */guide/broadway/theater/show/name*, and its postings are grouped into three lists based on these contexts.

Index File Construction The Indexer module parses the repository in phases, where each phase involves the construction of main memory structures that are written to disk at the end of the phase. During a phase, it builds in main memory a trie containing words encountered in that phase [Knu00]. The instances of a word that are encountered in a phase are called the **phase instances** of the word. For each word in the trie, the Indexer module builds a sorted list of the contexts of its phase instances. We call this list the **context list** of the word. For each context in the list, the Indexer module builds a sorted list of the locations of the phase instances that have that context. We call this list the **location list** of the context and the word.

Dictionary Creation The index file contains all context lists and all inverted lists that comprise the context index (Section 2.4) for the document repository. The Indexer module constructs the dictionary by performing a scan of the index file. It inserts into the dictionary each word present in the index file, along with the offset in the index file of the start of its context list. In order to better test our system, we do

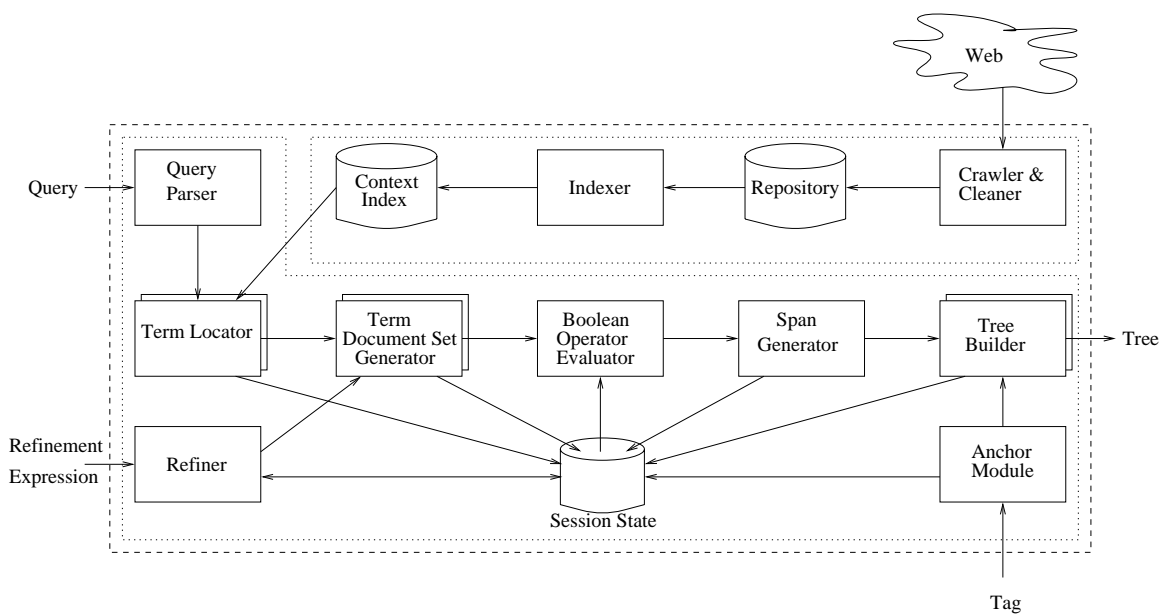


Figure 5: **Cextor Architecture.**

not eliminate stop-words or perform stemming during this step. The presence of all words also allows us to answer queries (e.g., “The Who”) that are composed of stop words (e.g., “the” and “who”).

We have implemented the dictionary as an external hash table using a file, in which buckets (each with a fixed number of slots) are written contiguously in increasing order of their bucket numbers. When a bucket is full (all its slots are occupied), we rehash (using a new hash function) the contents of all buckets, distributing the contents of each bucket between the original bucket and a newly created one. We append all newly created buckets to the file. We load dictionary values in batches, each batch sorted according the buckets of the values. Our hashing scheme is likely to create a larger hash table than that created by linear hashing or extendible hashing for identical insertions. However, our scheme leads to a simpler implementation. (e.g., In linear hashing, one has to worry about the position of the *bucket pointer* while bulk loading the hash table.)

3 The Augmented Index

XML query languages like XML-QL permit more sophisticated querying than is possible using Cextor [FSW⁺99]. In Figure 7, we illustrate a query expressed using a syntax that is quite similar to that of XML-QL. The WHERE clause specifies constraints on elements and the CONSTRUCT clause uses elements satisfying the constraints to build the query result. The “*” following IN in the WHERE clause indicates that the query has to be evaluated over all documents in the collection. The query in the figure asks for all elements with context */guide/broadway/theater*,

```

WHERE <guide> <broadway> <theater> $t </> </> </>
IN   ‘*’,
<address> $a </>
<show> fosse </> IN $t
CONSTRUCT <result> <theater> $t </> </>

```

Figure 7: **Sample query.**

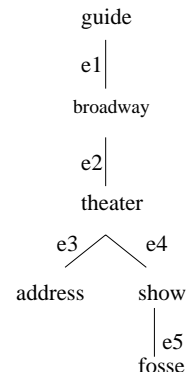


Figure 8: **Tree Pattern.**

and having an **address** subelement and a **show** subelement. In addition, the **show** subelement must contain the word “fosse.” The context index cannot be used to locate subelements of an element (e.g., **theater**), and so cannot be used to evaluate such queries. We present an enhancement to the context index that can be used to speed up evaluation of such queries. We call this enhanced index an **augmented index**.

The constraints on elements expressed in the WHERE clause of a query in many XML query lan-

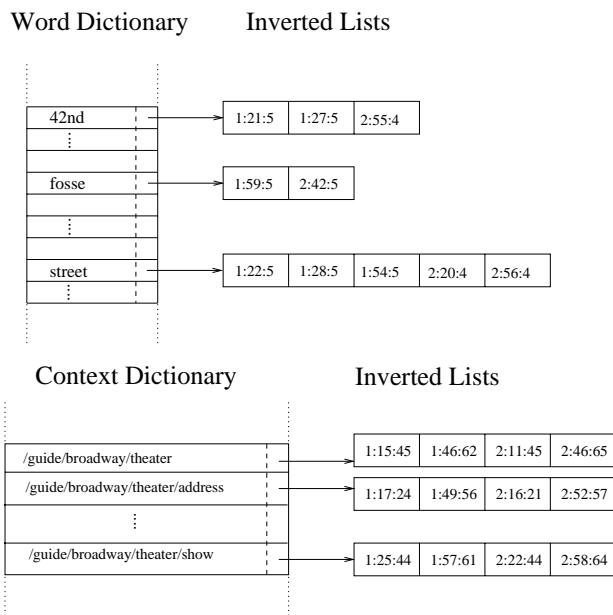


Figure 9: **Augmented Index.**

guages can be viewed as a *tree pattern*. Each node in the tree pattern represents a tag or a term in the WHERE clause and each edge represents direct containment or containment. We use a single edge to represent direct containment and a double edge to represent containment. Evaluation of the WHERE clause can be viewed as finding trees (i.e., XML documents) that match the tree pattern. Figure 8 illustrates the tree pattern for the query in Figure 7. The edge e_1 in Figure 8 specifies that a `broadway` element has to be a subelement of a `guide` subelement.

The augmented index includes two dictionaries: a *word dictionary* and a *context dictionary*. The **word dictionary** contains all words in the document repository, excluding those that occur only as tags or as names of attributes. For each word, it stores a pointer to a sorted list of *word postings*. A **word posting** consists of three integers: the identifier of a document containing an instance of the word, the offset of the instance in the document, and the depth of the context of the instance. The **context dictionary** contains all contexts that occur in some document of the repository. For each context, it stores a pointer to a sorted list of *context postings*. A **context posting** consists of three integers: the identifier of a document containing an element with that context, and the offsets start and end of the element in the document.

Example 3.1 Figure 9 illustrates a portion of the augmented index for the two documents in Figure 1. We describe one way in which the query suggested by the tree pattern in Figure 8 can be evaluated using the augmented index. First, we find all `theater` elements with the context `/guide/broadway/theater` and containing the

`address` subelement by merging the inverted lists for the contexts `/guide/broadway/theater/address` and `/guide/broadway/theater`. While merging, we use offsets in postings with identical document identifiers to locate `theater` elements having an `address` subelement, and we output the postings of `theater` elements that qualify. This merge completes the evaluation of edge e_3 in Figure 8. We evaluate edges e_4 and e_5 in a similar manner. Note that edges e_1 and e_2 need not be evaluated since the constraints they represent are subsumed by the context `/guide/broadway/theater`. If we modify the tree pattern by replacing edge e_2 with a containment operator, during evaluation, we use the dictionary to first find all contexts matching the context expression `/guide/broadway//theater`. We compute the union of the inverted lists for the contexts that match, and merge it with the inverted list for the context `/guide/broadway/theater/address`. The rest of the evaluation is unaffected.

Our augmented index is similar in spirit to the index used by the Niagara system [NDM⁺00]. The Niagara index consists of two dictionaries: a *word dictionary* and a *tag dictionary*. The word dictionary contains all words in the document repository, except those that occur only as tags or as names of attributes. For each word, the word dictionary stores a pointer to a sorted list of postings. Each posting is a pair of integers: the identifier of a document containing an instance of the word and the offset of the instance within the document. The tag dictionary contains all tags in the document collection. For each tag, the tag dictionary stores a pointer to a sorted list of tag postings. Each tag posting consists of three integers: the identifier of a document containing an element with the tag, and the offsets start and end of the element in the document. Matching a tree pattern to documents using the Niagara index involves merging two lists for each edge in the tree pattern. The lists that are merged are the inverted lists corresponding to the tags and words an edge connects in the tree pattern.

The relative performance of the augmented index and the Niagara index depends on the type of query. If the Niagara index is used to match a tree pattern having containment operators, one does not have to perform a union, as is necessary with the augmented index (See example evaluation using the augmented index). The inverted lists for the tags and words connected by an edge representing a containment operator can be merged to locate relevant elements. However, since the Niagara index does not store depth information, it cannot be used to match tree patterns that have direct containment operators. A depth-enhanced Niagara index needs to have the depth (an integer) stored with each tag posting. If the augmented index is used to match a tree pattern that consists of a chain of direct containment operators ($e_1 - e_2 - e_3$ in Figure 8), one or more edges need not be evaluated (See

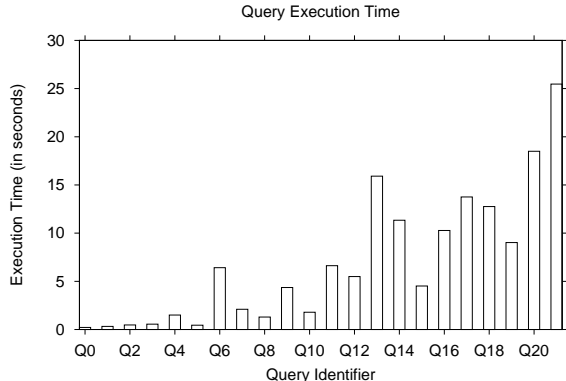


Figure 10: **Query Execution Time.**

example).

The Niagara index can be used to match tree patterns to XML documents, if the patterns do not have direct containment operators. A context expression is a simple instance of a tree pattern. Therefore, the Niagara index can be used to find documents with interesting query term instances of query terms in a query, provided that the context expressions in the query do not involve direct containment operators. However, since the index does not store contexts, it cannot return contexts that match a context expression. Therefore, it cannot be used to explore documents that a search returns.

4 Experimental Results

Except for the augmented index, we have implemented the Cextor system as described in Section ???. The document repository built by crawling the `umd.edu` domain contained about 210 thousand HTML files amounting to 10 GBytes of data. After cleaning the files, we parsed them using a SAX-based parser. We built contexts using tags contained even in documents that could not be cleaned by *Tidy*, generating very deep contexts. (We observed a maximum depth of 200.) We evaluated our system on a Sun Ultra 5 workstation with a 270 MHz Sparc processor and 128 MBytes of RAM, and running Solaris version 3.6. Our experimental results are grouped into three sets. The first set of experiments evaluates the context index. The second set evaluates our algorithm for context tree construction. In the third set, we study some properties of our corpus.

4.1 Index Construction and Query Processing

We used 22 queries (Table 1), chosen to cover a wide range of result sizes, to study the time to execute queries in Cextor. Figure 10 shows for each query, its execution time, which includes the time to compute (1) the query’s span, (2) the set of documents in its result, and (3) the association between contexts and documents in the result (i.e., what documents have

QID	Query
Q0	thesaurus
Q1	catholic
Q2	workstation
Q3	germany
Q4	sport basket ball
Q5	china
Q6	“graduate school” rank
Q7	joint appointment
Q8	database
Q9	service “parking permit”
Q10	system
Q11	theory group
Q12	that
Q13	“computer science” faculty
Q14	computer science
Q15	this
Q16	research thomas
Q17	health center
Q18	a
Q19	edu
Q20	and
Q21	the

Table 1: **Sample Queries.**

interesting query term instances that have a specific context?). The execution time shown for each query does not include the time to construct the context tree using the span.

Queries having a single word (*Q12, Q15, Q18, Q19, Q20, Q21*) take time roughly proportional to the number of instances of the word. For queries with multiple query terms where each query term is a single word (*Q14, Q16, Q17*), their execution times depend on three factors: (1) the number of query terms, (2) the total number of query term instances that are interesting, and (3) the skew in the number of instances of the different query terms. For example, *Q17* (727811 instances) takes more time than either *Q14* (157609 instances) or *Q16* (172957 instances) because it selects a larger number of interesting query instances. By the same argument, one would expect that *Q16* take more time than *Q14*. However, the words “computer” and “science” have about the same number of interesting instances (84,976 for “computer” and 72,633 for “science”), but the words “research” and “thomas” have a disproportionate number of instances (161,361 for “research” and 11,596 for “thomas”). During evaluation of the OR, merge of these unequally sized lists for the query terms in *Q16* takes less time than the merge of the roughly equal sized lists for the query terms in *Q14*. Query *Q13* has a moderate number (109135) of interesting query term instances, but it takes more time compared to queries with similar number of interesting query term instances. This high execution time is because the evaluation of *Q13* involves merging lists for “computer” and “science”, which are both very common words in our corpus (gathered from a

Run Size (No. of Word Instances)	1000000
No. of Runs	173
Run Generation Time	49 min. 56 sec.
Run Merge Time	35 hrs. 40 min. 13
Dictionary Creation Time	27 min. 26 sec.

Table 2: **Context Index Creation Statistics.**

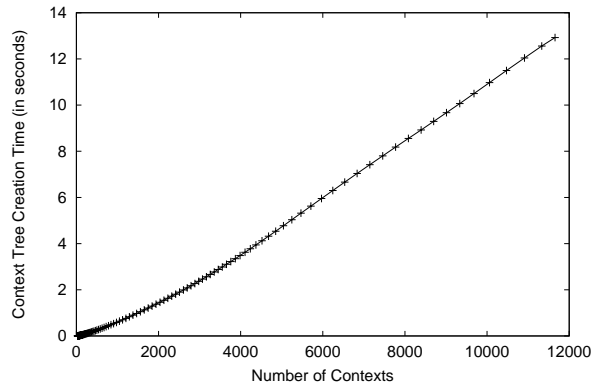


Figure 11: **Context Tree Creation Time.**

university domain).

The execution times for some of the queries (Q_{13} , Q_{20} , Q_{21}) are quite high. These high execution times are due to the fact that we need to carry context information along with each document, through all stages of query evaluation, in order to support operations such as refinement and anchoring. As a result, the context-enhanced list returned by the context index is larger than the inverted list returned by a traditional inverted file. For example, for query Q_{21} , the number of context-document pairs (33806) in its result is more than twice the number of documents (140307) in the result.

In Table 2, we present the execution times of different phases in the creation of the context index.

4.2 Exploration of Query Results

We studied the time to construct a context tree by executing queries that covered a wide range of span sizes (from 28 contexts to 11655 contexts). Figure 11 shows that the time to construct a context tree is linear in the number of contexts it represents.

We studied the time to anchor a context tree by using the `a` tag to anchor the initial context tree output by each of the 22 queries ($Q_0 - Q_{21}$). Figure 12 plots the time to (1) read the span of the query from a file (old session state), (2) find contexts in the span that contain the `a` tag, (3) write the remaining contexts (new span) to a new file (new session state), and (4) split each remaining context into an outer context and an inner context. We do not show the time to construct the output tree. The anchoring time is dominated by the I/O times to read and write ses-

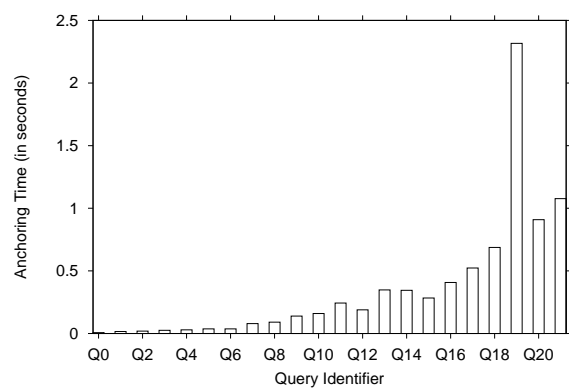


Figure 12: **Anchoring Time.**

Property	Avg.	Max.
Context Depth	7.06	200
Context Length (bytes)	29.29	995
Contexts per Word	5.99	12298
Instances of a Word	146.80	5830831
Documents Containing a Word	44.64	190871
Words in the Collection	1102478	
Contexts in the Collection	106016	
Documents in the Collection	209020	

Table 3: **Some Properties of our Corpus.**

sion state information, which are proportional to the number of contexts in the input (old span) and output (new span), respectively. Anchoring the context tree of Query Q_{19} took the longest time because it had the highest number of contexts in the input (8736) and output (5623) combined.

Using the context expression `//a`, we studied the time to refine the initial context tree output after evaluation of each of the 22 queries. We used the context expression to constrain the contexts of interesting instances of the first query term in each query (the first query term in `health center` is “health”). We observed that the times were I/O dominated, similar to what we observed for anchoring. The refinement of each query took less than one second, and query Q_{19} took the longest.

4.3 Data Statistics

Table 3 summarizes some properties of our document collection.

It is well known that frequencies of words in text documents follow the Zipf distribution [BYRN99]. As expected, we observed the same behavior in our corpus. However, we found it interesting to study the distribution of the number of distinct contexts across words. Figure 13, plotted using logarithmic scales on both axes, illustrates that this distribution follows the Zipf distribution, if we ignore a few words of high rank.

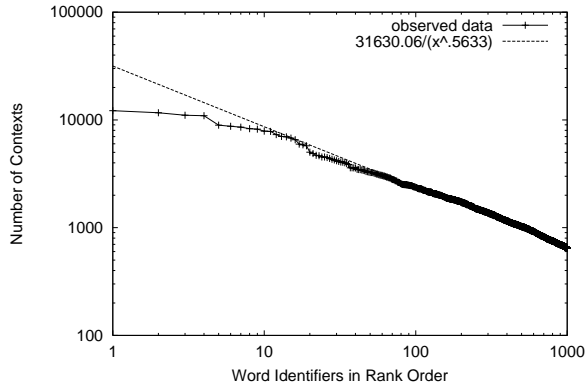


Figure 13: Contexts across Words.

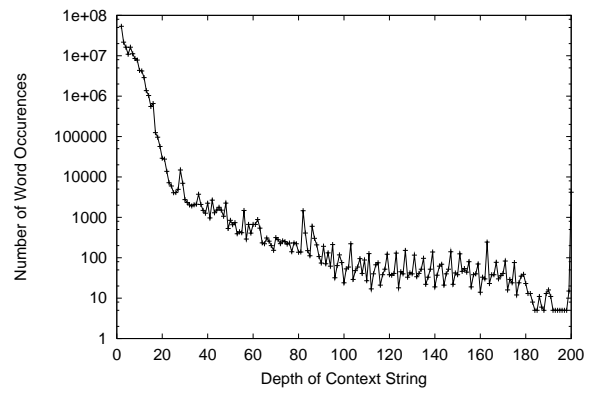


Figure 15: Word Instances across Levels.

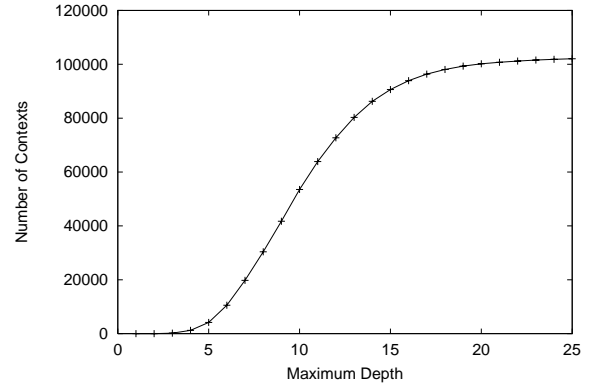


Figure 16: Contexts Depths.

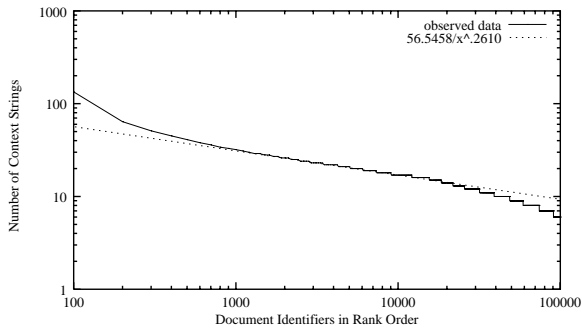


Figure 14: Contexts across Documents.

The dotted line in the figure represents the Zipf curve $31630.06/(x^{5.633})$. We also studied the distribution of the number of distinct contexts in a document. Figure 14 illustrates that this distribution also follows the Zipf distribution. The straight line in the figure represents the Zipf curve $56.55/(x^{2.610})$.

When indexing an HTML corpus, one may choose to limit the depths of the contexts stored in the index, since many deep contexts are a result of missing end tags in documents. Even if documents are well-formed, one may choose to limit the depth to avoid displaying deep contexts in the context tree. We studied the distribution of words across various levels of the document hierarchy. Figure 15, which plots the number of word instances (excluding tags) whose contexts have a certain depth, shows that most of the word instances are located at depth 2 (due to `/html/body`). It also shows that there are fewer than 10,000 words in the repository for all depth greater than 20. We also studied the number of contexts that have depth below a certain value, and found that most contexts have very low (< 10) depths (Figure 16).

5 Related Work

Several index structures have been developed by the Information Retrieval community for search over full text documents [BYRN99]. They include signature files [FC84], inverted files [SM83] and suffix arrays [MM90]. The traditional inverted file stores the postings for each word in a document collection, but does not store the contexts within which the word occurs. The context index augments the traditional inverted file to store the contexts of the occurrences of each word in a document collection.

Some search engines (e.g., Google, HotBot) allow searching for words attached to specific tags (e.g., the title) in a document. They extend the inverted file to include *meta-words* that encode information about tags. For example, the word president attached to the tag `title` in a document is treated as an occurrence of the meta-word “title:president,” and its posting is stored in the inverted list for “title:president.” If the word president is attached to another tag, say `location`, the posting for this occurrence is stored in the inverted list for “location:president.” The meta-words “title:president” and “location:president” are inserted into a dictionary. This approach can be extended to encode words in arbitrarily deep contexts. However, each word appears in several meta-words, as many as there are tags attached to the word. In our toy example, the string “president” occurs twice in the dictionary: once as “title:president” and once as “location:president.” The context index stores each word only once in the dictionary, and thus scales better when the number of contexts for a word is large.

Recent work on querying XML may be classified into two broad and complementary categories based on the type of XML data they study: The first category adopts a data-centric view in which XML encodes a database that may be structured or semistructured. Query languages in this category (e.g., Lorel, WebOQL, XML-QL) resemble OQL and other database query languages [FK99, STZ⁺99, YA94, MAG⁺97, DFS99]. The second category adopts a document-centric view in which XML is standardized syntax for structured documents such as technical reports, legal briefs, and equipment manuals. Query languages in this category resemble those used in information retrieval (e.g., boolean queries, vector space queries). Our work in this paper falls in the second category and our query language is an extension of the boolean query model.

Zhao and Joseph propose an index structure for fast search and retrieval of XML documents from moderately sized data collections such as a local area directory service [Zha00]. Their index structure is designed for main memory, and does not scale well to a large document collection such as the Web. Our approach has more in common with the Niagara system, which allows querying of documents on the Web [NDM⁺00].

A comparison between the indexing schemes used in our system and the scheme used in Niagara is given in Section 3. Schemes similar to the Niagara indexing scheme have also been used to index structured documents [Nav95, SM00].

6 Conclusion

In this paper, we addressed the following problem: How can we use the rich context information inherent in the tag structure of XML documents to improve search and exploration? We motivated the need for methods that improve XML search without assuming anything beyond well-formedness of XML documents. We stressed the need for an exploratory interface that enables users unfamiliar with the corpus to discover its structure and content. Our main contributions are (1) methods for context-sensitive search in XML (2) extensions with applications to query processing in XML-QL; (3) methods for exploring very large search results; (4) an experimental evaluation; and (5) an implemented system whose source code is publicly available. All the methods described in this paper, except the augmented index, have been fully implemented.

We are currently incorporating the augmented index into Cextor. We are also working on further improving the efficiency of index construction by evaluating alternate encoding techniques and implementations on a distributed architecture. We are studying methods to improve the scalability of context trees. Although we did not focus on the user interface itself in this paper, we are working on an innovative, Java-based user interface that uses zooming and other ideas to concisely present a large number of objects (such as large query results). Finally, we are planning a full-scale deployment of a search engine based on Cextor

References

- [BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
- [DFS99] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, 1999.
- [FC84] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM TOIS*, 2(4):267–288, 1984.
- [FK99] D. Florescu and D. Kossman. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
- [FSW⁺99] M. F. Fernandez, J. Simon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu,

- and J. Widom. XML query languages: Experiences and exemplars, 1999. Available at <http://www-db.research.belllabs.com/user/simeon/xquery.ps>.
- [Knu00] D. E. Knuth. *The Art of Computer Programming, Vol. 3*. Addison-Wesley, 2000.
- [MAG⁺97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.
- [MM90] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *SODA*, 1990.
- [Nav95] G. Navarro. A language for queries on structure and contents of textual databases. Master’s thesis, Univ. of Chile, 1995.
- [NDM⁺00] J. Naughton, D. Dewitt, D. Maier, J. Chen, J. Kang, N. Prakash, J. Shanmugasundaram, R. Ramamurthy, Y. Wang, R. Chen, L. Galanis, Q. Luo, F. Tian, C. Zhang, B. Jackson, A. Gupta, and K. Tufte. The Niagara internet query system, 2000. Available at <http://www.cs.wisc.edu/niagara/Publications.html>.
- [SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
- [SM00] T. Schlieder and H. Meuss. Result ranking for structured queries against XML documents. In *First DELOS Workshop on Information Seeking, Querying and Searching in Digital Libraries*, 2000.
- [STZ⁺99] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, 1999.
- [YA94] T. W. Yan and J. Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *VLDB*, 1994.
- [Zha00] B. Y. Zhao. The Xset XML search engine and Xbench XML query benchmark. Tech. Report UCB/CSD-00-1112, Univ. of California, Berkeley, 2000.