Context-Sensitive Search and Exploration of XML Text

Thomas Baby
thomas@cs.umd.edu
Department of Computer Science
University of Maryland, College Park
MD 20742

Sudarshan S. Chawathe
chaw@cs.umd.edu
Institute for Advanced Computer Studies
University of Maryland, College Park
MD 20742

CS-TR-#4223
UMIACS-TR-#2001-12

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 and 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 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 very large (800,000) 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 in 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 in lines 10 and 11; the writer element is, in turn, a subelement of the *show* element beginning in 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 in line 8 of Document 1 is different from that of the name element in 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

```
<guide>
  <city> New York </city> <state> New York </state>
  <theater> Ford Center for Performing Arts
    <address>
      <street> 213 West 42nd Street </street>
    </address>
    <show>
      <name> 42nd Street </name>
      <writer>
        <name> Michael Stewart </name>
        <name> Mark Bramble </name>
      </writer>
      <director> Gower Champion </director>
    </show>
  </theater>
  <theater> Broadhurst Theatre
    <address>
      <street> 235 West 44th Street </street>
    </address>
    <show>
      <name> Fosse </name>
      <director> Ann Reinking </director>
    </show>
  </theater>
</guide>
```

(a) Document 1

```
<guide>
  <city> New York </city> <state> New York </state>
  <broadway> <theater>
    <name> Shubert Theatre </name>
    <address> 225 West 44th Street </address>
    <show>
      <name> Chicago </name>
      <writer>
        <name>John Kander</name>
        <name>Fred Ebb</name>
      </writer>
      <director> Bob Fosse </director>
    </show>
  </theater>
  <theater>
    <name>American Airlines Theatre </name>
    <address> 227 West 42nd Street </address>
    <show>
      <name> Design for Living </name>
      <playwright> Noel Coward </playwright>
      <director> Joe Mantello </director>
    </show>
  </theater> </broadway>
</guide>
```

(b) Document 2

Figure 1: Sample XML Documents

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 semantic specifications that the standardization work demands. Further, even in the long term, a diverse and autonomous environment such as the Web will always a 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 context but not in the second. (Details appear in Section 2.) This query language is implemented using some simple and effective extensions to the traditional 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 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. (For example, we do not expect to use the interface suggested by the screenshot in Figure 2 for purposes other than validation and experimentation.) 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 Document Model

In this paper, we adopt a simplified view of XML documents. Each document has a single element, called the root, within which all other elements are nested (e.g., the `guide` element in Document 2 of Figure 1). We view each document as a rooted, ordered tree, where nodes represent elements and edges represent nesting of elements. Each node in the tree is labeled with the tag of the element it represents. We further simplify the document model by treating an element's attribute as its subelement, with the attribute name as tag and the attribute value as content[1].

2.2 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*.

[1]This simplified model overlooks several distinctions between subelements and attributes (e.g., restriction on attribute names and textual context). However, we believe our model is effective for XML search and exploration (as distinct from XML data processing).

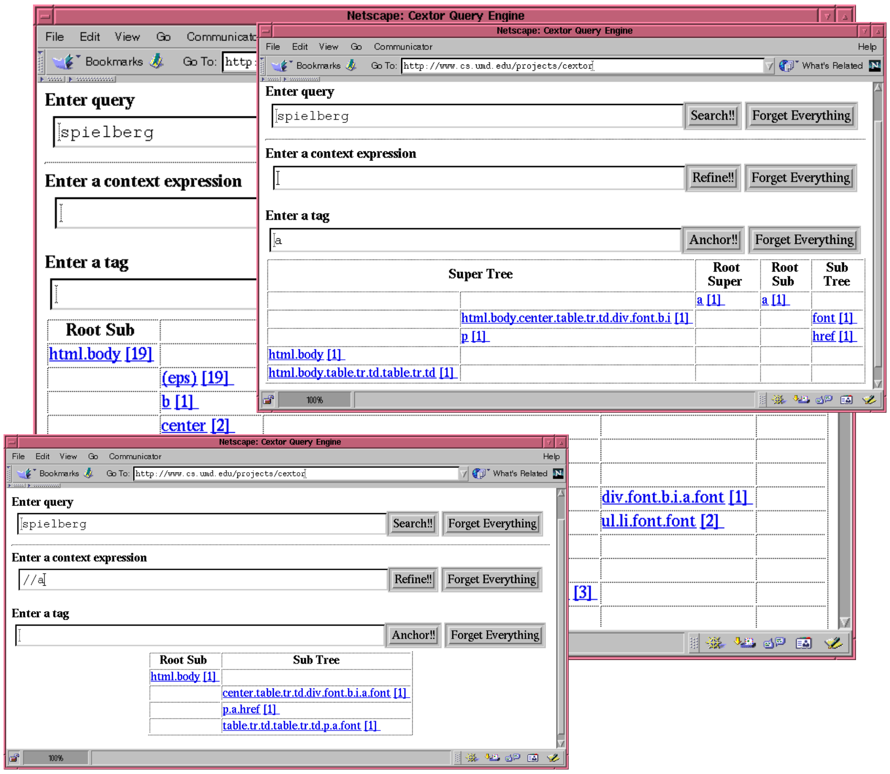


Figure 2: **Screenshot of our Search Engine in action.**

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 / seperating `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`).

2.3 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 and the context expression that qualifies it 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, 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*.

2.4 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.)

Example 2.5 Figure 3 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. The center figure in Figure 2 is a screenshot of the context tree output by our system for the query `spielberg`. The tree is displayed as a table, with the label of a node displayed at a column and row given by the node's depth and preorder number, respectively.

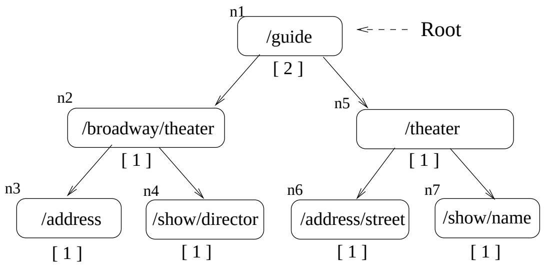


Figure 3: **Example of a Context Tree.**

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.

Inspite 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.

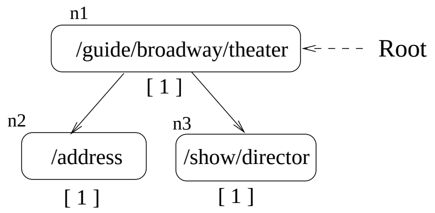


Figure 4: **Context Tree Refinement.**

Example 2.6 Consider the query `(42nd IN /guide//theater/address) AND (fosse IN /guide//show)` on the documents in Figure 1. Figure 3 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 4. The left figure in Figure 2 is a screenshot of the output of our system on refining, using the context expression `//a`, the initial context tree for the query `spielberg`.

Anchoring: Among contexts in the span of a query (Section 2.3), 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 3). 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 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

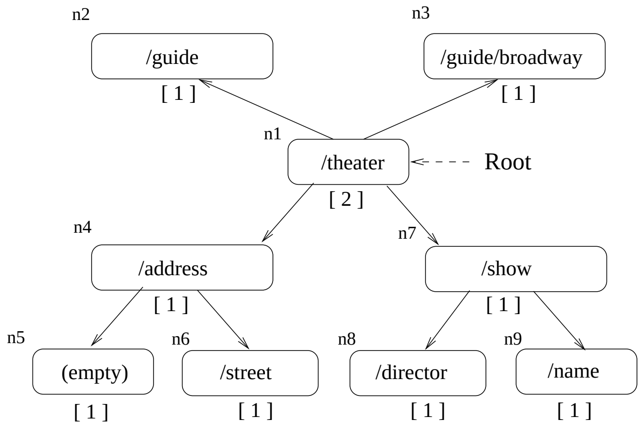


Figure 5: **Context Tree Anchoring.**

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 5 illustrates the tree that is the result of anchoring the tree in Figure 3 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 5), and its root is collapsed with the root of the inner tree, which is displayed without modification. The right figure in Figure 2 is a screenshot of the output of our system on anchoring, using the `a` tag, the context tree for the query `spielberg`. Columns in increasing order are displayed from left to right for the inner tree table and from right to left for the outer tree table.

2.5 A Typical User Session

A Cextor session starts with a user posing a query. The result of the query is presented as the initial context tree built using the span of the query. The user may explore the span by performing a series of refinement and anchoring operations. The result of each exploratory operation is again presented as a tree. Depending on the screensize available for display, the tree produced by any of the three operations (i.e., querying, refinement, and anchoring) may be truncated. If a tree is truncated, the user may click on a node at which the tree is truncated to expand it. The result of an expansion operation is also presented as a tree. After a series of refinements, anchorings, and expansions, the user may view the document set of a node in the output tree by clicking the node.

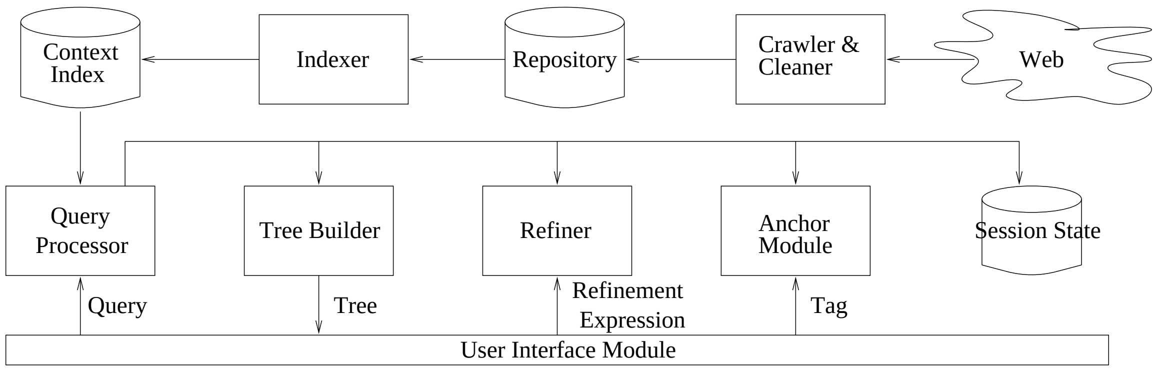


Figure 6: **Cextor Architecture.**

2.6 Implementation

Figure 6 illustrates the high-level architecture of Cextor. The Crawler&Cleaner module 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 Indexer module builds the context index in two steps: *index file construction* and *dictionary creation*.
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 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 7, 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 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.

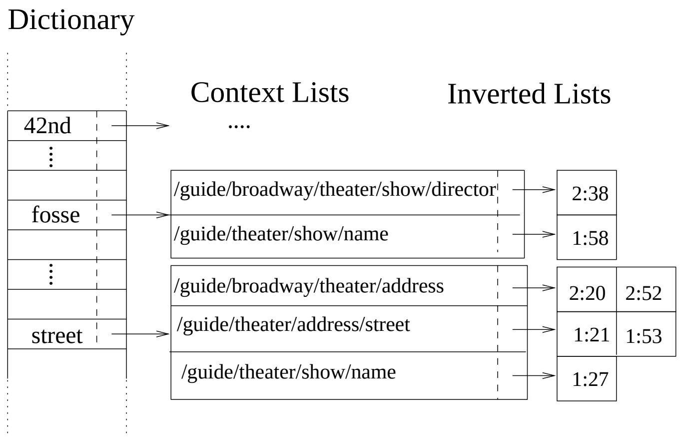


Figure 7: **Context Index.**

When the amount of main memory occupied by the trie, the context lists, and the location lists created during a phase exceeds a certain threshold (tunable parameter in our system) or if no more documents need to be parsed, the Indexer writes the trie, the context lists, and the location lists to a new file on disk. We call this file a *run*. The Indexer writes words in the trie in depth-first order, appending after each word the contents of its context list. After writing all contexts for a word, it appends the contents of all location lists for the word. It writes location lists in the lexicographic order of their contexts. Along with each context for a word, it stores a forward pointer (i.e., relative file offset) to the start of its location list, in order to be able to easily find the instances of the word occur within that context. The time taken to create a run is linear in the size of the run, which is linear in the combined size of the trie, the context lists, and the location lists.

After all documents are parsed, one or more runs exist on disk. If multiple runs exist, the Indexer module merges them to create a single file, called the *index file*, using the standard algorithm for merging sorted lists.

Dictionary Creation The index file contains all context lists and all inverted lists that comprise the context index (Section 2.6) 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 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

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

Figure 8: **Sample query.**

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.)

The four remaining modules—Query Processor, Tree Builder, Refiner, and the Anchor Module—are quite straightforward. For example, the Query Processor uses offsets in postings to locate query terms. For each query term, it constructs a list of contexts of interesting instances of the query term, and for each context in the list, it constructs a list of documents that have interesting query term instances in that context. A distinguishing feature of our system is that it writes temporary structures created by each of these four modules to disk so that subsequent operations in the same session can make use of them. For example, the contexts of interesting instances of a query term are saved as session state by the Query Processor so that a subsequent refinement using the query term can operate on them. Some operations (e.g., refinement) cannot be supported without such state information whereas others (e.g., node expansion) use it for efficiency.

3 The Augmented Index

XML query languages like XML-QL permit more sophisticated querying than is possible using Cextor [FSW+99]. In Figure 8, 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*, 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 languages 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

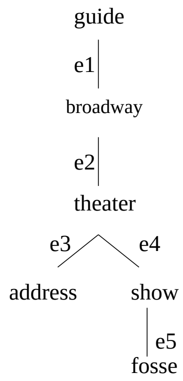


Figure 9: **Tree Pattern.**

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 9 illustrates the tree pattern for the query in Figure 8. The edge e_1 in Figure 9 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 dictonary** 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 10 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 9 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 9. 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

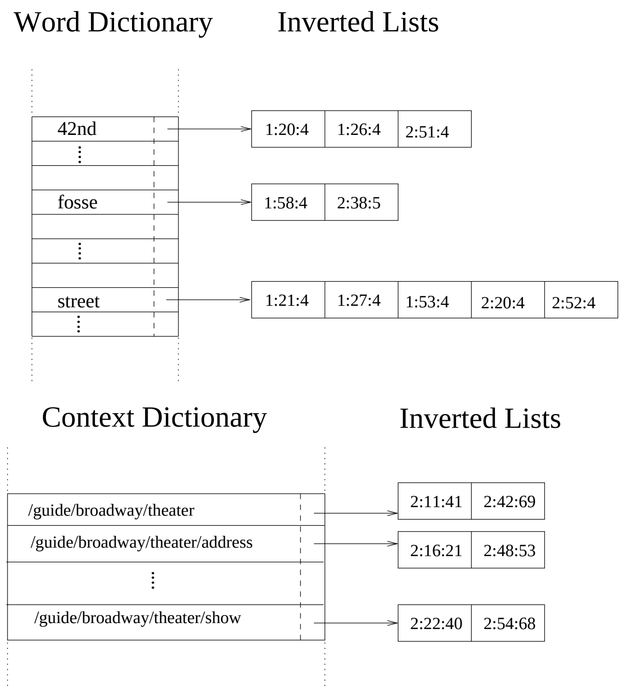


Figure 10: **Augmented Index.**

/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 connected by an edge 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 9), one or more edges need not be evaluated (See 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 Cextor 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 2.5. The document repository built by crawling the `umd.edu` domain contained about 210,000 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 2.6. We present our experimental results in three sets. The first set of

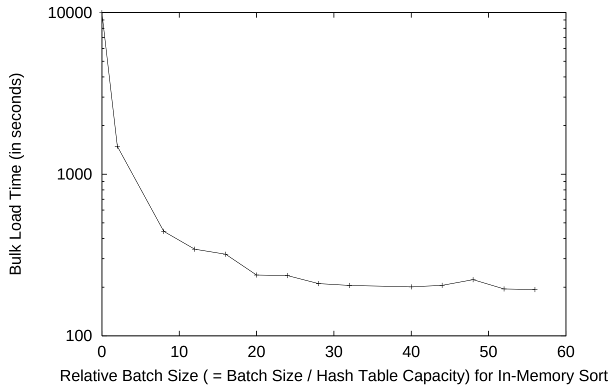


Figure 11: **Bulk Load of Dictionary.**

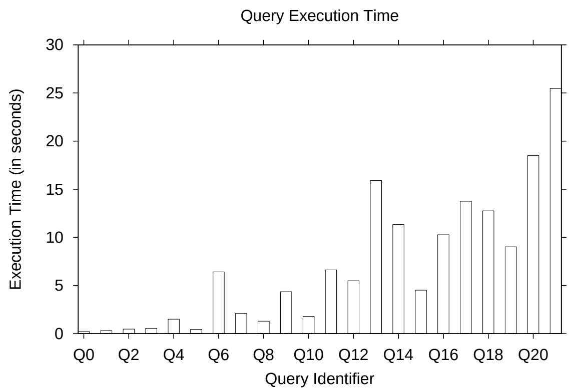


Figure 12: **Query Execution Time.**

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. (The query ids are in increasing span size.) Figure 12 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 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.

QID	Query	Size of Span	Context-Doc Pairs	Docs.	Query Term Instances.
Q0	thesaurus	78	184	137	252
Q1	catholic	217	2395	2265	3202
Q2	workstation	290	1897	1404	3215
Q3	germany	348	2541	2074	4040
Q4	sport OR basketball	406	5503	5009	8880
Q5	china	442	3209	2424	9296
Q6	"graduate school" OR rank	582	5759	4961	10880
Q7	joint OR appointment	780	7954	6792	13127
Q8	database	1176	15829	8662	35355
Q9	service OR "parking permit"	1780	23192	17894	47867
Q10	system	2237	40360	28386	105428
Q11	theory OR group	2417	42814	28372	95414
Q12	that	2991	102605	77059	936730
Q13	"computer science" OR faculty	3140	52879	30787	109135
Q14	computer OR science	3541	57476	36096	157609
Q15	this	3861	132477	92673	529866
Q16	research OR thomas	4355	72134	38588	172957
Q17	health OR center	5136	194492	89590	727811
Q18	a	8564	216293	119190	2149533
Q19	edu	8736	385141	100530	1280713
Q20	and	11062	311998	133396	3215259
Q21	the	11655	338066	140307	5830851

Table 1: **Sample Queries.**

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, $Q17$ ($727,811$ instances) takes more time than either $Q14$ (157609 instances) or $Q16$ ($172,957$ 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, merging of these unequally sized lists for the query terms in $Q16$ takes less time than the merging of the roughly equal sized lists for the query terms in $Q14$. Query $Q13$ has a moderate number ($109,135$) 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

Run Size (No. of Word Instances)	1,000,000
No. of Runs	173
Run Generation Time	49 min. 56 sec.
Run Merge Time	35 hrs. 40 min. 13 sec.
Dictionary Creation Time	27 min. 26 sec.

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

"science", which are both very common words in our corpus (gathered from a university domain).

The execution times for some of the queries ($Q13$, $Q20$, $Q21$) 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 $Q21$, the number of context-document pairs ($33,806$) in its result is more than twice the number of documents ($140,307$) in the result.

Recall from Section 2.6 that our implementation of the dictionary is a hash table that rehashes all buckets when a bucket is full. We studied the time taken to bulk load $208,867$ strings (URLs) to a hash table file on disk for different sizes of the batch used to write them to disk. We inserted the strings in random order to a hash table that was large enough ($40,36$ buckets, each with 100 slots) to contain all strings without the need to rehash during the bulk load. We varied the size of the batch from 1 (write to disk immediately after each insertion) to $208,867$ (write all strings in one access to file on disk). We show the time taken (processing + I/O time) to write all strings to the file on disk.

Figure 11 shows that for small batch sizes, the execution time is dominated by I/O time due to large seeks in the file on disk, as entries in the same batch belong to buckets that are widely separated in the file. However, for larger batch sizes, almost each bucket in the file is accessed during the flush of each batch so that there is no more gain in performance. Performance levels off at about 25% of the capacity of the hash table (total number of slots in the hash table). The CPU component of the execution time remained almost even (at about 14 seconds) for all batch sizes. At the largest batchsize ($208,867$ strings), the file is accessed once, sequentially. As a result, the I/O time for bulk load is low and the CPU component is a non-trivial fraction (about 7%) of the total execution time. We observed a similar behavior while bulk loading the dictionary as well.

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 $11,655$ contexts). Figure 13 shows that the time to construct a context tree is linear in the number of contexts it represents.

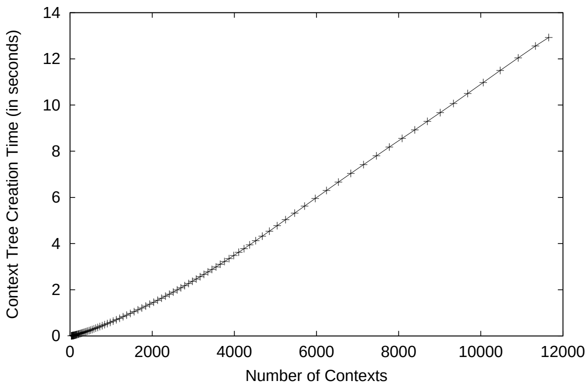
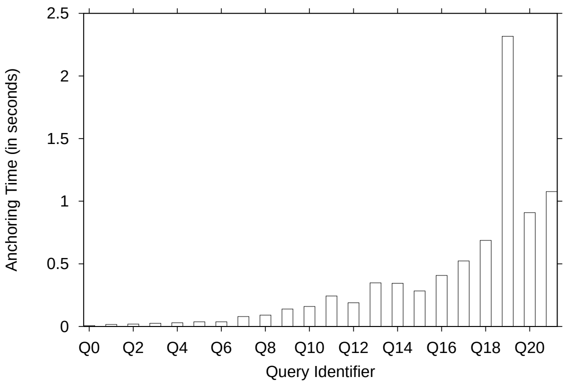
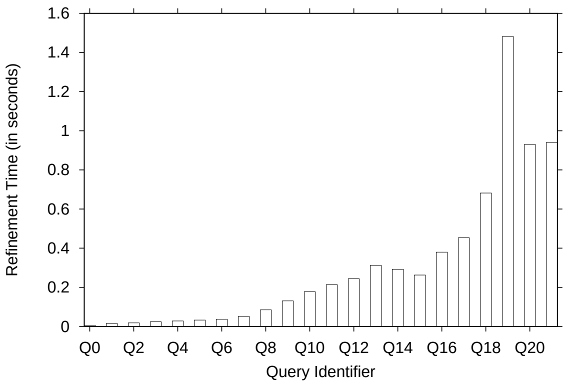


Figure 13: **Context Tree Creation Time.**



(a) Anchoring Time



(b) Refinement Time

Figure 14: **Execution Time of Exploration Operations**

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.**

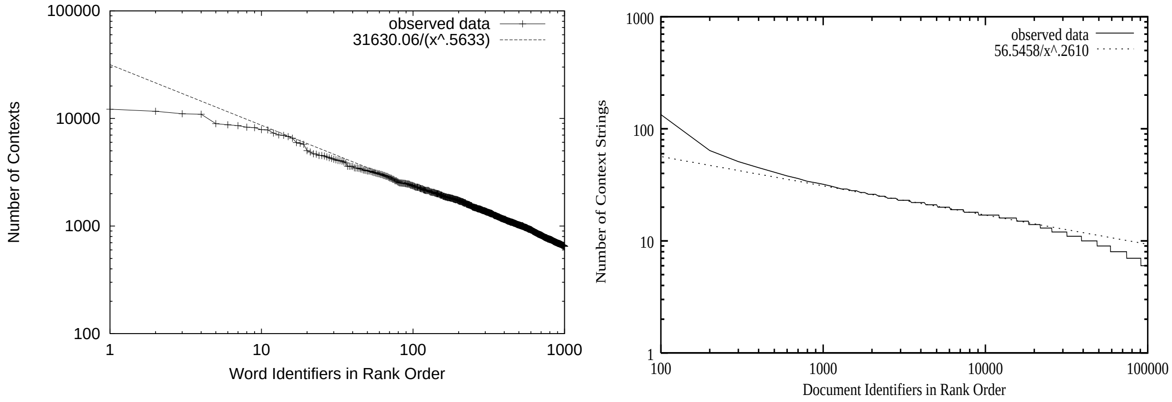
We studied the time to anchor a context tree by using the `a` tag to anchor the initial context tree output by each of the 22 queries ($Q0 - Q21$) in Table 1. Figure 14(a) plots the time to (1) read the span of the query from a file (old session state), (2) find contexts in the span that contain the `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 session 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 $Q19$ took the longest time because it had the highest number of total contexts in the input (8736 contexts) and output (5623 contexts) 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 in Table 1. 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”). In Figure 14(b), we show for each, the time, which includes the time to (1) contexts of all interesting instances of the first term of the query, (2) find contexts that match `//a`, and (3) write the matching contexts to a new file (as session state). It does not show the time to recompute the span and result documents based on the new set of interesting instances. The times were I/O dominated, similar to what we observed for anchoring.

4.3 Data Statistics

Table 3 summarizes some properties of our corpus.

It is well known that frquencies 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 15(a), plotted using logarithmic scales on both axes, illustrates that this distribution follows the Zipf distribution, if we ignore a few words of high rank. The dotted line in the figure represents the Zipf curve $31630.06/(x^{.5633})$. We also studied the distribution of the number of distinct contexts in a docu-



(a) Contexts across Words

(b) Contexts across Documents

Figure 15: **Distribution of Contexts**

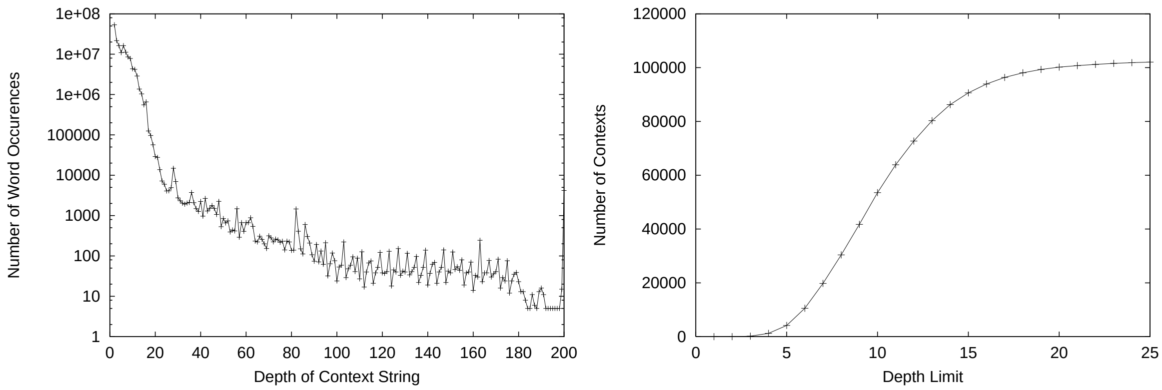
ment. Figure 15(b) illustrates that this distribution also follows the Zipf distribution. The straight line in the figure represents the Zipf curve $56.55/(x^{.2610})$.

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 16(a), 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(b)).

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.

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 [STZ+99, MAG+97]. The second category adopts a document-centric view in which XML



(a) Word Instances across Levels

(b) Context Depths

Figure 16: **Words and Contexts across Depths.**

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) [BYRN99]. Our work in this paper falls in the second category and our query language is an extension of the boolean query model. A distinguishing feature of our work is the postprocessing (context tree creation and exploration) performed on query results.

XSet is an index structure for fast search and retrieval of XML documents from moderately sized data collections such as a local area directory service [Zha00]. The main memory data structures used by XSet do not scale to data that does not fit in main memory. Our approach has more in common with the Niagara system [NDM+00]. A detailed comparison of our indexing methods and those in Niagara appears in Section 3. Schemes similar to the Niagara indexing scheme have also been used to index structured documents [Nav95, SM00].

Our work on exploring search results is related to a large body of work in the Human-Computer Interaction field. Due to space contraints, we mention only two systems that share some of our goals. The DLITE system [CK+97] provides an interactive workspace for querying documents and organizing search results. The Cat-a-Cone interface uses the Information Visualizer [CRM96] to present a three-dimensional view of category hierarchies and the documents within them. The main difference between these systems and Cextor is that they focus on the user interface issues (good use of visual cues, interactivity, etc.) for moderate sized data, while while Cextor focuses on data-centric operations on very large data. Cextor can complement systems such as DLITE by providing them with the ability to efficiently manage large amounts of data.

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 exploratary 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] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, Harlow, England, 1999.

[CK+97] S. B. Cousins, S. P. Ketchpel, et al. The digital library integrated task environment (DLITE). In *Proceedings of the ACM International Conference on Digital Libraries*, pages 142–151, Philadelphia, Pennsylvania, July 1997.

[CRM96] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 111–117, Zurich, Switzerland, April 1996.

[FC84] Christos Faloutsos and Stavros Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, 1984.

[FSW+99] Mary F. Fernandez, Jérôme Simon, Philip Wadler, Sophie Cluet, Alin Deutsch, Daniela Florescu, Alon Levy, David Maier, Jason McHugh, Jonathan Robie, Dan Suciu, and Jennifer Widom. XML query languages: Experiences and exemplars, 1999. Available at `http://www-db.research.belllabs.c`
`om/user/simeon/xquery.ps`.

[Knu00] Donald E. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, Reading, Massachusetts, 2000.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[MM90] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 319–327, 1990.

[Nav95] Gonzalo Navarro. A language for queries on structure and contents of textual databases. Master's thesis, University of Chile, 1995.

[NDM+00] Jeffrey Naughton, David Dewitt, David Maier, Jianjun Chen, Jaewoo Kang, Ñaveen Prakash, Jayavel Shanmugasundaram, Ravishankar Ramamurthy, Yuan Wang, R̃ushan Chen, Leonidas Galanis, Qiong Luo, Feng Tian, Chun Zhang, Bruce Jackson, Ãnurag Gupta, and Kristin Tufte. The Niagara internet query system, 2000. Available at `http://www.cs.wisc.edu/niagara/Pu blications.html`.

[SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[SM00] Torsten Schlieder and Holger 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] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the Twenty Fifth International Conference on Very Large Databases (VLDB)*, pages 302–314, 1999.

[Zha00] Ben Y. Zhao. The Xset XML search engine and XBench XML query benchmark. Technical Report UCB/CSD-00-1112, University of California, Berkeley, 2000.