The XSet XML Search Engine and XBench XML Query Benchmark

by

Ben Yanbin Zhao

B.Sc. (Yale University) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Anthony D. Joseph, Chair
Professor Joe M. Hellerstein

2000

The dissertation of Ben Yanbin Zhao is approved:

Chair Date

Date

University of California at Berkeley

2000

The XSet XML Search Engine and XBench XML Query Benchmark

Abstract

The XSet XML Search Engine and XBench XML Query Benchmark

by

Ben Yanbin Zhao

Master of Science in Computer Science

University of California at Berkeley

Professor Anthony D. Joseph, Chair

Internet-scale distributed applications (such as wide-area service and device discovery and location, user preference management, Domain Name Service) impose interesting requirements on information storage, management, and retrieval. They maintain structured soft-state and pose numerous queries against that state. These applications typically require the implementation of a customized proprietary query engine, often not optimized for performance, and costly in resources. Alternatives include using traditional databases, which can hamper flexibility and extensibility (both of which are critical requirements of Internet-scale applications), or LDAP (Lightweight Directory Access Protocol), which poses composability problems and imposes rigid structure on queries. This paper proposes a different approach, based upon the use of the eXtensible Markup Language (XML) [7] as a data storage language, along with a main memory-based database and search engine. Using XML allows applications to use dynamic, simple, flexible data schemes and to perform simpler, but faster queries. The approach yields a single, common data management platform, XSet. XSet is an easy to use, main memory, hierarchically structured database with incomplete ACID properties. Preliminary measurements show that XSet performance is excellent: insertion time is a small constant value, and query time grows logarithmically with the dataset size. A portable Java-based version of XSet is available for download, both as a standalone application and as a component of the Ninja service infrastructure.

To my father.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Professors Joe Hellerstein, Mike Franklin, and Eric Brewer for their valuable feedback and direction on XSet. Many thanks to the members of the Ninja and Iceberg research groups for motivating and testing XSet for their applications. Thanks to Steve Gribble and Andy Begel for their help in previous versions of this work. Finally, I'd like to thank my advisor Professor Anthony Joseph for supporting and overseeing my ongoing research, and my parents, for supporting me through it all.

1 Introduction

The development of modern distributed applications has led to several interesting information storage, management, and retrieval requirements. In particular, an increasing number of applications are providing novel functionality by incorporating a fast searching component. For the lack of a better term, we call this new class of applications "Query Enabled" applications. These applications often maintain a mix of structured soft-state [8] and durable hard-state, and pose numerous queries against that state. Examples of such applications are service- and device-location and discovery protocols, such as DNS [24] and LDAP [18], and applications which make use of simple and fast query functionality, such as searchable XML-enabled email systems and personal location trackers. The problems with these applications are three-fold: their extensibility is often very limited due to predefined, rigid data schemas; they pay for query power and flexibility with added schema complexity; and many of them offer similar functionality with significantly different implementations, duplicating effort and functionality.

In this paper, we propose to unify this class of applications by using the eXtensible Markup Language (XML) [7] as a data storage language along with a memory-based database and search engine we call XSet. We then define a set of data semantics we propose for these applications, with sufficient semantic guarantees, that maximizes performance and concurrency. Finally, we provide a simple benchmark for evaluating XML query engines, such as XSet.

We chose XML as a description language because it offers numerous benefits including structured extensibility, strong data validation capabilities, powerful expressiveness, and ease of use. XML accentuates structure by making explicit the inherent structure of the data, without imposing a rigid schema. XML also provides flexible validation through Document Type Definitions (DTD). Furthermore, XML tags allow direct reference to data fields, extending expressiveness. Finally, XML is text-based, and offers data encapsulation in a human readable form without high overhead. These properties and a standardization effort make XML a natural choice for our needs.

1.1 Existing Database Models

Given the benefits that XML can bring to information management applications, there is the issue of how to store and query XML documents. At first, a database-based approach would appear to be an appropriate choice. We will argue, however, that for the set of metadata / distributed applications we have introduced, a streamlined minimalistic approach should improve performance.

Relational and Object-Relational Approaches

There are currently two main thrusts of database design: relational and object-relational databases. While relational databases have been extremely popular in existing industrial applications, object-relational databases are becoming increasingly popular for supporting correlated data of different types and sizes, such those popularized by the World Wide Web.

We believe that there are two main reasons why neither database design is well-suited to the search functionality required by distributed applications. The first involves the structure of XML data, which is usually simple, but hierarchically organized. Relational databases are ill-equipped to handle such a structure. Translating hierarchically structured documents into tabular relations is an unnatural and complex mapping. Furthermore, a single query in a deeply nested tree may require repeated table retrievals for each level of the tree. This intuition has been confirmed by recent work [3]. The authors showed that while most queries can be transformed into relational queries, there were exceptions. Certain types of queries cannot be mapped into SQL, while other simple queries on XML were mapped to large numbers of SQL queries, or single queries with numerous joins.

The second and more fundamental argument against using traditional database systems is the strict nature of database consistency. In our class of XML-enabled applications, consistency requirements are generally less strict and more application-specific than those in a traditional database model. For example, while directory applications such as LDAP may support transactions, they generally make little use of such functionality, and treat inserts as independent operations. These relaxed constraints can often be achieved through simpler application-specific algorithms that do not incur the performance penalties associated with strict ACID properties.

Semantics and Performance

Past work in the database community has recognized the evolutionary model of database applications, and their changing semantic requirements [5]. While other approaches to address these changes give limited concessions for increased concurrency, we want to focus instead on the tradeoff between semantics and performance.

Given these arguments against existing approaches in current database research, we decided to develop a new XML storage and query mechanism called *XSet*. From a database perspective, XSet can be described as a memory-resident, hierarchically structured database with support for an incomplete set of the ACID semantics.

1.2 Evaluation

In Section 5, we present detailed performance analysis of a single-node XSet implementation. The goal is to show that, by removing overhead due to transaction support, XSet can provide much better performance. In practice, many industrial databases execute with relaxed runtime semantics, giving up serializability for concurrency. There still remains a significant overhead due to concurrency control and locking overhead. Concurrency is crucial to their performance, since synchronous I/O is a major factor in response time. In a memory-resident database such as XSet, however, most operations do not block on I/O; and therefore, enforcing coarse-grain locking per thread reduces lock contention overhead while minimizing the performance sacrifice. The results in this section highlight the performance benefits of relaxing traditional database semantics by showing that the resulting query processing time is low, and scales logarithmically as the size of the dataset.

Given our implementation of XSet, we want to compare its performance with similar XML query engines. Choosing the metric of evaluation, however, is non-trivial. With the currently ill-defined XML query languages, query engines may return drastically different results for an identical query on two different sets of data. To produce a fair performance comparison that would reflect real application performance, we need to take a closer look at how applications use XML queries, and we produce a set of benchmarks that accurately reflect the result. We present the resulting benchmark we call *XBench*, in Section 6.

1.3 Assumptions and Goals

In designing XSet, we make three assumptions about application workloads and environments: we set the design goal that a single XSet server can handle a reasonably sized data collection, such as a local area directory service; we avoid the problem of updating to conform to new XML standards by assuming that our data model is constrained to a well-defined core set of XML functionality; and we require that XSet servers have large amounts of memory (e.g., 1 to 2 GB, an amount that is readily available in off-the-shelf servers). In a few months, we expect this memory capacity to be available in mid- to high-end workstations. Because of XSet's use of physical memory, it may incur a higher performance penalty when the dataset size scales beyond memory capacity. In Section 9, we propose a cluster model which ameliorates this problem.

Within these constraints, the primary goals of XSet are to support the XML storage, query, and semantic requirements of "Query Enabled" applications, while accommodating a range of semantic constraints and maintaining fast and scalable performance and simplicity.

The rest of the paper is organized as follows: In Section 2, we present XSet's architecture, and then discuss the implications on data semantics in Section 3, the implementation in Section 4, and analyze XSet's performance in Section 5. Next, we explore several motivating applications that use XSet's simple, high performance XML functionality in Section 7 and discuss related work in Section 8. Finally, we discuss future work in Section 9 and conclude in Section 10.

2 Architectural Design

In this section, we discuss the use of XML as a type of semistructured data, provide a scenario that motivates the need for XSet, and present the XSet architecture and its components.

2.1 Semistructured Data

The structure and organization of data is often a limiting factor in how it can be used by applications. Data with a fixed, well-defined structure, as in a relational database, allows static typing, consistency checking, and well-defined queries, but can be confining should the data or query model evolve. Free-form data supports all data types and query models, but nothing can be known about the data statically.

Between these extremes is the semistructured data model provided by XML, a model that provides many of the benefits of both extremes. Not only can one reason about (and validate) the data a priori, but the data is also flexible enough to adjust to new data and query models.

2.2 Motivating Scenario

To motivate the functional requirements for XSet, consider an academic or corporate campus of the near future where people migrate between offices and buildings, and their networked personal devices alert the environment to their presence while exporting interfaces for them to access local resources.

Ideally, these users would like to utilize context-aware applications to access a wide range of ever-changing data. For instance, a visitor wants to specify and find resources in their immediate surroundings, such as their meeting contacts, video projectors, and available lecture halls. This application query model works equally well in reverse. People who enter a building become temporary services, and register their personal preferences and profiles with local servers. Other applications, such as group paging or meeting reminders, can then query the XSet server to locate and reach users.

To support these type of applications using only traditional databases, it would be necessary to design a large number of static schemas, ranging from personal location profiles to printer specifications, to lecture room scheduling events. Given the dynamic nature of these resources, constant rewriting of these schemas would be necessary to keep databases up to date. Furthermore, most queries would not benefit from transactional support and consistency guarantees available in most transactional databases. Finally, these costs would be duplicated per administrative domain, and possibly exacerbated by incompatible databases and schemas.

In the XSet world, these problems are solved through the use of the combination of schema standardization, semi-structured schemas, and the simplicity of XSet queries. XSet servers are easily deployed across administrative domains. Servers can benefit from the lack of constraint on schemas, to support a standardized core set of datatypes, while allowing for locally customized tags and objects as they appear. Also, the simplicity and directness of XSet queries reduce the implementation effort necessary to extend the query processor interface to the user.

In our ideal environment: users, given a standard search and browse interface, can specify flexible queries on thin clients which translate the queries to XSet queries and send them to local servers. An example of a common query might look for a color printer on the fourth floor, the nearest set of accessible workstations with a DVD drive, or a lecture hall reserved with the visitor's name. Descriptions of long-lived services are encoded in simple XML schemas, and stored on the server. Short lived services such as roaming projectors and thin client docking stations might not conform to any static schema, and would periodically broadcast their XML descriptions to the XSet server, with a timeout period associated with each description. At regular intervals, a cleaner runs through the server dataset and, following a flexible user policy, filters out any outdated services.

Under the covers, XSet servers parse and index incoming XML documents, optionally validate them with a cached Document Type Definition (DTD), and depending on the intended longevity of the data, provide the appropriate level of durability. Incoming XML queries are parsed, and then processed against the memory-resident index to access the documents. These queries could even embed personal certificates that are matched to access control lists in service descriptions. Such a resource discovery mechanism would handle large volumes of requests efficiently with relaxed semantic data guarantees.

2.3 Overall Design

Figure 1 shows the internals of a single XSet Server. A single server consists of several components: a main-memory component, which we refer to as the SetServer, a disk-based component consisting of a file backing store, a write-ahead log, and a fuzzy checkpointing system. The SetServer includs a XML index and a memory-resident data store.

During registration / insertion, the SetServer receives XML documents via a

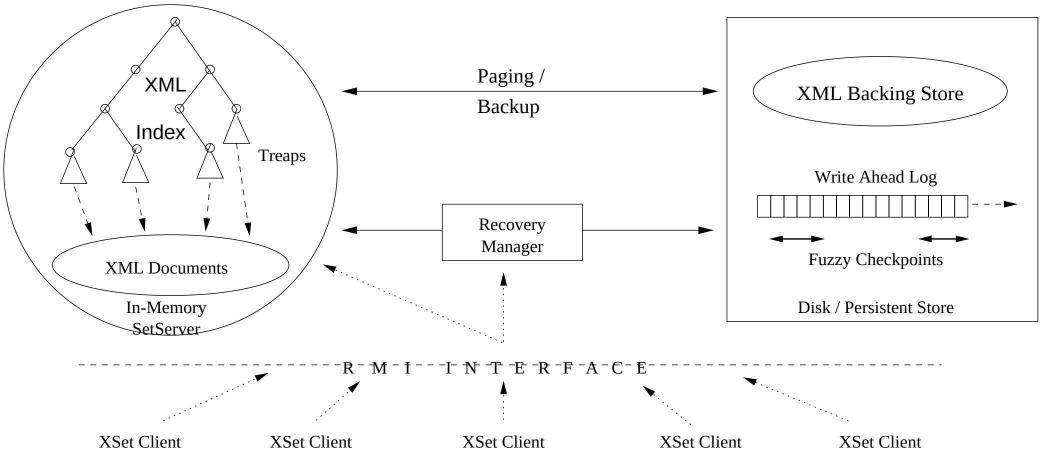


Figure 1: Single XSet Server

JavaRMI interface, adds the documents to the disk backing store, parses the XML, and merges the document structure into the hierarchical tag index structure. In the backing store, documents are assigned a monotonically increasing unique identifier, which can be used in paging and logging operations. Each subtree of the document is merged into the index. For each tag in the index, documents are stored as sets inside a treap [27] (probabilistic self-balancing tree structures), each set indexed by a common tag value. A single document would have its reference indexed into tag treaps, each corresponding to XML tags inside the document. To summarize, tags are the keys used to access the index, and document references are the final data values.

XSet supports both "soft-state" and persistent state. Whereas "soft-state" or short-lived data can be handled by the main memory index and store alone, long-lived data makes use of the XSet durability layer. An up to date copy of the dataset resides on stable storage. Modify operations (inserts and deletes) are logged to a finite-sized log buffer in memory. The buffer is flushed to disk when full, or when an explicit flush operation is issued by the client. XSet also supports fuzzy checkpoints (where data is still available during the checkpointing process), both at regular intervals, and also by explicit client request.

Additionally, XSet exposes functionality to the user for explicitly paging documents in and out of the memory store, providing support for user-designed paging policies.

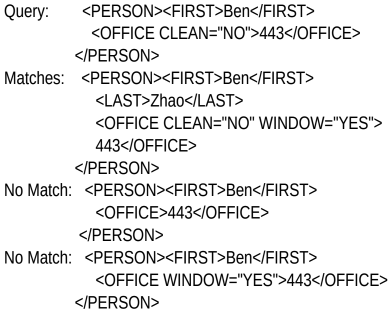
Since many of the target applications deal with soft-state data, XSet also includes an optional data cleaner that incrementally removes stale data at regular intervals.

In the following sections, we discuss the query model and several components in more detail.

2.4 Query Model

To simplify query composition and make query processing fast, we chose a simple XML document subset model with minor functionality extensions.

XSet queries are themselves well-formed XML documents, with optional embedded query instructions for the query processor. XSet queries exploit the flexibility of XML tag structure by using the subset tag model, where satisfiability of the query is defined as whether an XML document's tag structure is a strict superset of that of the query document. Tags that are not explicitly stated in the XSet query are assumed to be "wildcards" that can match any XML tag value or subtree. In query processing, collections of document references that match each search constraint undergo global intersection to return the result set. Some simple query examples are shown in Figure 2.



```
Query:      <PERSON><FIRST>Ben</FIRST>
              <OFFICE CLEAN="NO">443</OFFICE>
            </PERSON>
Matches:    <PERSON><FIRST>Ben</FIRST>
               <LAST>Zhao</LAST>
               <OFFICE CLEAN="NO" WINDOW="YES">
               443</OFFICE>
            </PERSON>
No Match:   <PERSON><FIRST>Ben</FIRST>
               <OFFICE>443</OFFICE>
             </PERSON>
No Match:   <PERSON><FIRST>Ben</FIRST>
               <OFFICE WINDOW="YES">443</OFFICE>
            </PERSON>
```

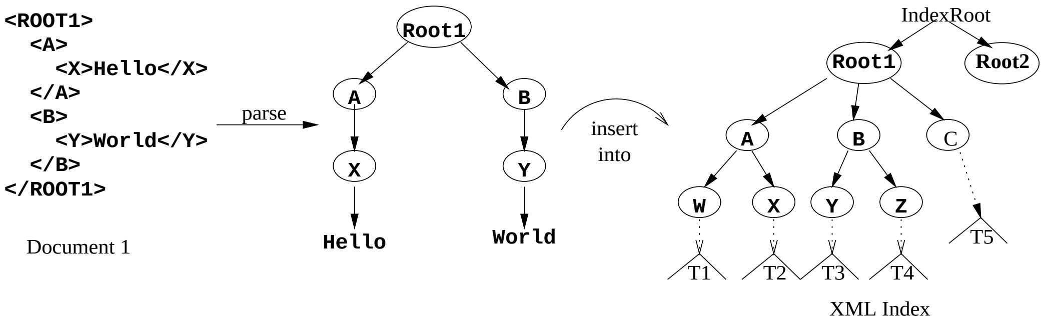
Figure 2: Sample Queries

Special query instructions passed to the XSet query processor are encoded inside the query as non-standard XML attributes, and removed by XSet prior to processing the query. For example, a constraint that searches for an integer value in tag `DOC` between 10 and 20 would look like: `<DOC GT="10" LT="20" KEY_T="INTEGER">RANGEQ</DOC>`

In addition to the XSet query model, several XML query languages have been proposed and implemented, including XML–QL [10], LOREL [1], and XQL [25]. Compared to XSet, these languages chose different points on the simplicity vs. functionality tradeoff scale. On the same scale, XSet has the least complex query model, and supports a much smaller set of queries. XSet queries can be characterized as a subset of the XQL language, represented as a XML document. As a query model, XSet queries also resemble the associative matching aspects of Linda Tuplespaces [12]. Linda differs from XSet in that it is a distributed communication mechanism, rather than a standalone query engine.

2.5 Tag Index

The tag index is a simple, hierarchical indexing structure. It can be characterized as a dynamic structural summary of the documents in the dataset.



```
<ROOT1>
  <A>
    <X>Hello</X>
  </A>
  <B>
    <Y>World</Y>
  </B>
</ROOT1>
```

Figure 3: Simple Indexing Example

When a document is indexed, its tag hierarchy is merged with the overall XSet tag index, and each tag value from the document becomes the document index key for the corresponding tag Treap in the main index. Figure 3 shows an indexing example of a short document. In this case, the document reference would be inserted into Treap T2 with “Hello” as the key, and Treap T3 with “World” as the key. Additionally, references to documents also keep any attributes and their values attached to the relevant tag, so that they can be checked against queries with attribute constraints.

The key distinction between this index scheme and some other XML indices [23, 10] is the notion of contextual semantics. We believe that the semantics associated with a tag value are only valid given the exact context in which the tag appears. For example, the same tag for `PHONENUMBER` can have entirely different meanings whether it appears inside the sequence of tags `PERSON -> HOME -> ADDRESS` or `BUSINESS -> CONTACTINFO -> SHIPPING -> ADDRESS`. For that reason, tags are defined uniquely by a combination of context and tag name, and cannot be indexed purely on their tag names. This type of contextual semantics is similar to path-based queries in LORE [23], except the root node end of the path is fixed.

2.6 Document Paging

Both advantageous and perhaps limiting to the XSet model is its dependence on large amounts of physical memory. The memory overhead per document can be as large as 2kb, which can be significant for the smallest of documents. One solution is to remove from memory (page out) less frequently referenced documents, keep their indexing information in memory, and read them back from the backing store on disk (page in) on a on-demand basis. XSet provides such a flexible paging mechanism, while leaving policy decisions to the application writer.

Document objects in XSet export a paging interface which can be invoked by the user to exploit application specific paging information. When documents are paged out to disk, their indexing information remains in memory, and the document is paged in lazily if its is found to be a part of a solution set.

Simple paging algorithms such as LRU, random, and MRU can be implemented easily using this approach. Additionally, more complex algorithms, ones which better exploit XML tag structure, can also be used. For example, one potential policy in a directory service could partition services by type, and apply a prioritized LRU algorithm, giving priority to more dynamic service data, such as the current location of a professor, while paging out more static data, such as his or her telephone number.

2.7 Durability Mechanisms

Two related components provide the persistence and failure recovery functionality for long-lived data in XSet. The in-memory SetServer interacts directly with the backing store on disk. It ensures durability by adding the document to the backing store, and also pages documents out to disk as needed to free up memory. The recovery manager (RM), exposes a useful set of recovery API calls to both internal XSet components and the external application interface. These calls give explicit control over all durability mechanisms, including the use and compaction of the write-ahead/redo log, when and how often the fuzzy checkpointing mechanism is called, and the use of the in-memory log buffers.

The redo log records log entries both before the beginning and after the end of each operation. Each entry records the type operation it is and unique identifiers of the document(s) operated on. During recovery, this allows large numbers of logged operations to be aggregated efficiently into a single patch, and applied to a checkpointed index. When a logging operation discovers a full log buffer, it flushes the buffer synchronously before proceeding. Further details of the fuzzy checkpointing and redo log optimizations are discussed in Section 3.3.

As an application component, XSet focuses on providing the mechanisms on top of which a wide range of policies can be implemented. This is reflected in the in-memory log buffer, which is regularly flushed to disk to ensure durability of operations. Varying the buffer size controls the tradeoff between performance (reflected in frequency of I/O operations) and durability. Similarly, there is no preset algorithm for determining when the document pager is run, and what order documents are evicted from memory. Finally, we leave it to the application writer to define an algorithm that determines when and how often to checkpoint.

2.8 Cleaner Mechanism

XSet also includes an optional data cleaner component for soft-state data management. Applications that periodically refresh their data can have the cleaner run at regular intervals with user specified policies to incrementally clean out the XML dataset. For example, transient user location data could be invalidated after 5 minutes, while printer description documents could have a lifetime of 5 days. This allows an administrator to provide customized soft guarantees on the freshness of the dataset contents.

3 Semantic Guarantees

In this section, we define the data semantics provided by XSet. We assert several assumptions regarding the nature of data used by "Query Enabled" applications (see Section 1), general access patterns on this data, and use them to motivate a data model that focuses on performance and simplicity.

3.1 Partial ACID Semantics

To help the reader better gauge the relative semantics of XSet and typical databases, we discuss XSet's semantics in terms of ACID [16] terminology, where ACID stands for Atomicity, Consistency, Isolation, and Durability. As explained below, XSet does not support the notion of transactions, and the semantic list below follow the context of a transaction-free model.

From the discussions of semantics in previous subsections, we summarize these points, which are further explored in following sections:

- *Atomicity:* Atomicity is provided on the granularity of single operations.
- *Consistency:* Consistency is guaranteed. No inconsistency can occur during normal operations, since only one thread is allowed into the database at one time.
- *Isolation:* Isolation is not provided in the context of transactions, but single operations are isolated via the lock mechanism.
- *Durability:* XSet provides full durability and recovery across failures, by providing a simple and efficient combination of write-ahead logging and fuzzy checkpointing.

3.2 Applications Semantics

As stated above, XSet provides different data semantics from those provided by typical database systems. While XSet is a database providing full durability, it is motivated by applications which gather soft-state data, and pose large numbers of queries against it. The queries are generally self-contained, and single queries produce useful results. Directory services are an example which exemplifies this class of applications. We optimized the XSet design towards certain properties of data used in these applications, such as immutability and short lifetimes. Applications using data that break these assumptions, however, can still benefit from the overall XSet model.

The first simplification XSet makes is in its approach to concurrency. In database systems where the majority of data is stored on disk, disk I/O cost dominates query latency. Concurrency is necessary to maximize utilization of resources. XSet, however, is a main memory database, where memory access latency is the dominating latency factor. As a result, threads spend few cycles waiting for memory I/O; and increasing concurrency does not greatly benefit latency, since the cost of a context switch is comparable to a memory fetch operation. Also important in this consideration is the absence of transactions in XSet. Transactional databases use concurrency to eliminate waiting on user latency between operations in a transaction. This is no longer a concern in XSet. The XSet design reflects this shift of focus off of concurrency control, by placing a global lock on the server, and only allowing a single thread to enter at any time. This guarantees single operation consistency trivially.

A second optimization derives from the types of documents XSet serves. Whereas traditional databases operate on large numbers of small records in a single database, XSet targets large numbers of small descriptive XML documents, the whole of which make up the XML database. These documents can describe large numbers of different objects such as services, preferences, people and locations, and tend to be compact XML documents with limited number of attributes. XSet optimizes for this type of small records by using a "replace-only" update model, where any changes to a document are made by replacing the existing document with a new version. Documents become immutable. We show in the next section how by using this model, we greatly reduce the complexity of logging and recovery.

Finally, the majority of "Query Enabled" applications use an access model consisting of single queries. The notion of transactions, while useful in certain contexts, is not used enough to justify the additional complexity and performance overhead. Instead, we choose the single operation as the granularity of operation. Furthermore, because data is immutable, and operations are independent, modify operations in XSet become idempotent; that is, single operations can be repeated in order without fear of making the database inconsistent.

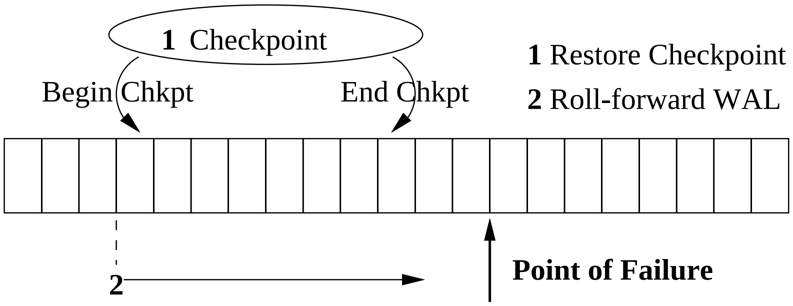


Figure 4: Recovery Mechanism

3.3 Fast Recovery

As a result of XSet data semantics, recovery of failures is simple and efficient. In addition to a standard write-ahead log, XSet includes a fuzzy checkpointing mechanism. Because of the idempotent nature of modify operations in XSet, a fuzzy checkpoint can be taken any time without extensive use of locks. The begin_checkpoint and end_checkpoint operations are both logged, and the Log Sequence Number (LSN) of the begin_checkpoint operation is stored with the checkpoint. While the checkpoint itself is inconsistent, it is easily brought up-to-date by rolling forward all log entries after the begin_checkpoint operation.

Figure 4 is a simple illustration that demonstrates how recovery occurs after a system failure. We assume that persistent storage, such as disk, survives major failures by using mechanisms, such as replication or mirroring. After a system failure removes the memory contents of the server, the recovery process follows two steps: First, the system restores a memory image using the fuzzy checkpoint. Then, the system takes the Log Sequence Number (LSN) of the begin_checkpoint operation, and applies the redo log starting at that LSN. Because operations are idempotent, any inconsistencies in the fuzzy checkpoint will be made consistent through the redo log.

As previously mentioned, log entries contain three fields, the “begin” or “end” of an operation, the type of operation, and unique identifiers for documents it operates on. An additional optimization made possible by the immutable data abstraction is that during recovery, the log can be traversed to generate a compact, simulated mapping of "live" documents at the time of failure, each reference by their identifier. We use this mapping as a single patch, and apply it to the in-memory document store, bringing it up-to-date in one single operation. This guarantees that only documents present in memory at time of failure are loaded, and frequent insert/delete operations in the log will not impact recovery time.

4 Implementation and Status

XSet has undergone several major modifications in both design and implementation and a distribution is now publicly available in two forms: a stand-alone application[1], and as an application written using the Ninja distributed services framework [29]. XSet has also been integrated or is being integrated in to several applications (see Section 7). In this section, we discuss XSet's implementation details.

4.1 Implementation Platform

For portability and ease of implementation, we chose Java [15] as the programming language. As a result, the stand-alone version of XSet is small (5000 lines) and runs without modification on several OS platforms.

The third major revision of XSet has been implemented on top of the Ninja distributed services architecture. The Ninja operating environment strives to provide services with fault-tolerance, load balancing and fast communication. The two versions of XSet are mostly identical in implementation.

XSet uses the XML4J parser from IBM Research Labs to parse XML. Since XSet

[1] The XSet distribution is available at: `http://www.cs.berkeley.edu/~ravenben/xset`.

uses the DOM API [17], the implementation is largely parser independent, and minimal changes can be made to integrate XSet with alternative parsers.

4.2 Persistent Datastore

For simplicity, XSet currently uses the filesystem as its persistent backing store. Flushed log buffers are appended to a single log file, the head of which is truncated after each successful checkpoint operation. This allows XSet to be easily portable, while leveraging large amounts of research in file system fault-tolerance and recovery. Furthermore, XSet's I/O interface can easily be modified to operate on top of an alternative backing store, such as a MMAP interface or a log-based file system [26].

4.3 Treaps

Treaps are probabilistically self-balancing trees that achieve $O(Log_2(n))$ time for all operations [27]. As the data structures for indexing documents by their tag values, they were chosen for their research value rather than performance. While a data structure with a larger branch factor such as a B-tree would reduce the tree traversal time, the choice of treaps gave us a chance to explore novel properties of a cartesian tree (trees using two indexing keys).

While treap performance characteristics are similar to other binary trees such as T-trees and red-black or AVL trees, treaps have the advantage of preserving heap order on a secondary key. In the naive case, this secondary key is a pseudo-random "priority" generated at insertion time, used to provide self balancing qualities. In practice, this secondary key can be further manipulated by the treap structure during accesses to implement specific heap order policies. One example of an useful policy is to increment the priority with a small randomized number during each access, and then rotate the treap to maintain heap order if necessary. The net result of such a policy is that the values accessed most often tend to “rise” in the heap, providing shorter trips down from the root node and exploiting temporal locality for improved performance. This property could prove especially useful when considering cache versus main-memory performance on future systems [21].

Treaps have also been shown to be extremely efficient for parallel algorithms on ordered sets [6]. Using treaps allows us to investigate these fast parallel algorithms in distributed and parallel versions of XSet.

5 Performance

In this section, we evaluate the performance and scalability under different workloads of the current XSet implementation. Our hypothesis is that given XSet's relaxed consistency constraints relative to conventional database systems, XSet should yield very high performance.

5.1 Experimental Background

We performed XSet experiments on two platforms: Linux 2.0.36 with Blackdown.org's port of Sun's JDK 1.1.7B (Intel Pentium II 350 Mhz with 128MB of memory), and Windows NT 4.0 Terminal Server with Sun Microsystems' JDK 1.1.7B (Intel Pentium II Xeon 450Mhz with 1 GB of memory). All measurements were taken with Just-In-Time compilers enabled. The TYA JIT[2] was used on Linux, and Symantec's JIT[3] was used on Windows NT.

5.2 Performance Components

There are three performance components in XSet's normal operation: validation, indexing, and queries. *Validation* is the important one-time process of certifying that an

[2] The TYA JIT compiler is available at `http://www.dragon1.net/software/tya/`.

[3] The Symantec JIT compiler is included in all Sun Microsystems Java Development Kits (JDK) after 1.1.6.

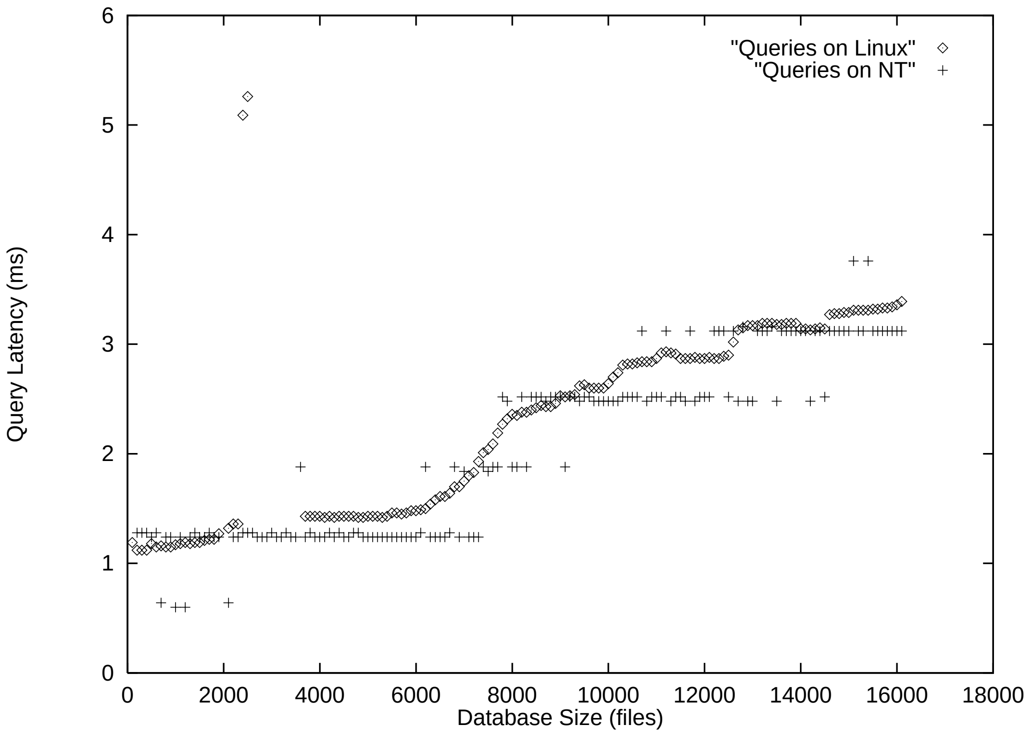


Figure 5: Query time versus dataset size (Linux and NT)

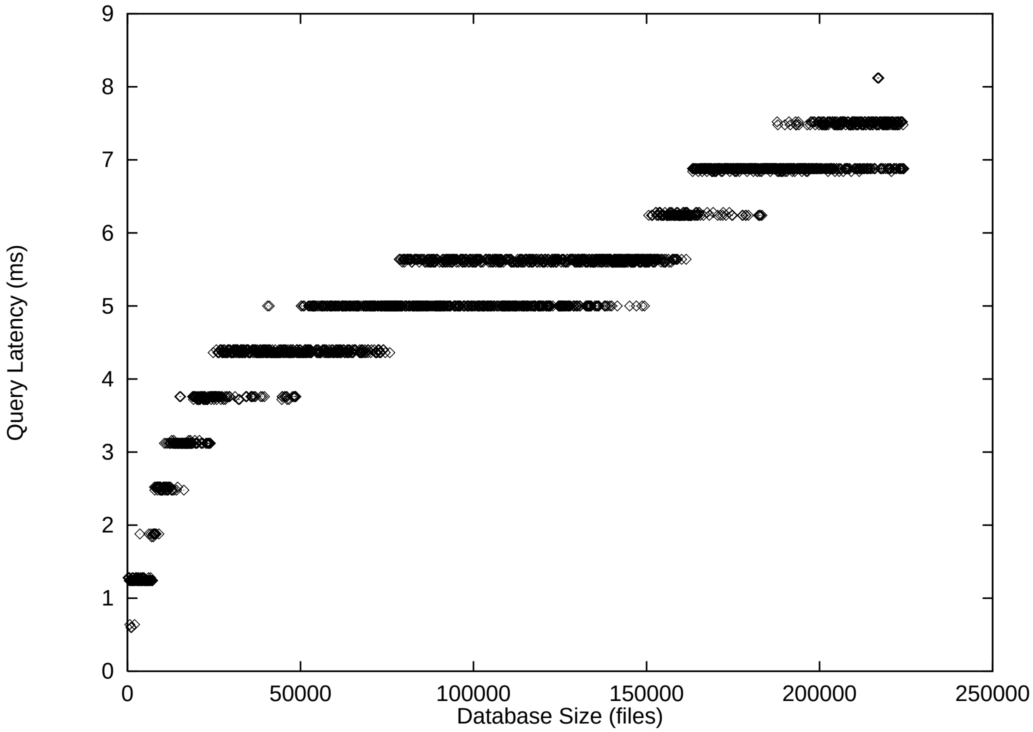
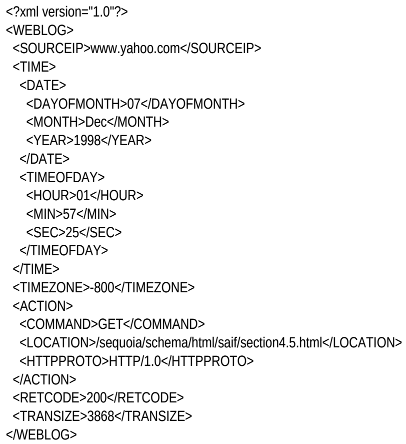


Figure 6: Query time versus dataset size (NT)

XML document conforms to an external DTD. *Indexing* is the process of adding a new XML document to an existing XSet index. Indexing performance affects both first time insertions of documents and per document recovery time after a crash. Finally, *query processing* is the latency involved in servicing a query.

5.3 Experimental Results

For the large data set, we converted an HTTP web server access log into small (slightly less than 1KB) XML files, where each file encoded the information for one HTTP request. The resulting tree has a depth of 3 levels with an average branch factor of 5 at each tag. We did not use the complete dataset for all of the experiments. Because of different host memory sizes, the experiments on Linux used 16100 files, while the experiments on NT used 224330 files. Figure 7 shows a sample XML database file.



```
<?xml version="1.0"?>
<WEBLOG>
 <SOURCEIP>www.yahoo.com</SOURCEIP>
 <TIME>
  <DATE>
   <DAYOFMONTH>07</DAYOFMONTH>
   <MONTH>Dec</MONTH>
   <YEAR>1998</YEAR>
  </DATE>
  <TIMEOFDAY>
   <HOUR>01</HOUR>
   <MIN>57</MIN>
   <SEC>25</SEC>
  </TIMEOFDAY>
 </TIME>
 <TIMEZONE>-800</TIMEZONE>
 <ACTION>
  <COMMAND>GET</COMMAND>
  <LOCATION>/sequoia/schema/html/saif/section4.5.html</LOCATION>
  <HTTPPROTO>HTTP/1.0</HTTPPROTO>
 </ACTION>
 <RETCODE>200</RETCODE>
 <TRANSIZE>3868</TRANSIZE>
</WEBLOG>
```

Figure 7: A sample XML database file

While this is a large dataset, it is not an optimal choice. Most data about each HTTP access is unique, so queries performing exact matches only return small result sets. Also, we found that accesses by the same IP address tend to be grouped closely in the indexing sequence, resulting in IP address locality in the storage treaps. To circumvent this problem in our measurements, the queries in the query set are based upon a collection of IP addresses evenly distributed in the database. We then averaged the results across the query set.

Figures 5 and 6 shows the time to perform a query as a function of the number of documents in the dataset. The results show that, as expected, the query time grows logarithmically with the dataset size (i.e., approximately 1.28 ms at 200 documents, 3.12 for 16,000, and 6.88 ms for 224,000 documents). From the figure, one can also observe when a query takes XSet to an additional level of a treap. A surprising result is that the Linux times closely match those for Windows NT, even though they have different processor speeds. It is likely that the similarity is a result of XSet being memory bound on queries Note that the gap in Linux times between 2000 and 4000 files is due to outliers (they are not visible because they are off the scale of the graph). The outliers are due to the Java VM performing garbage collection during the measurement.

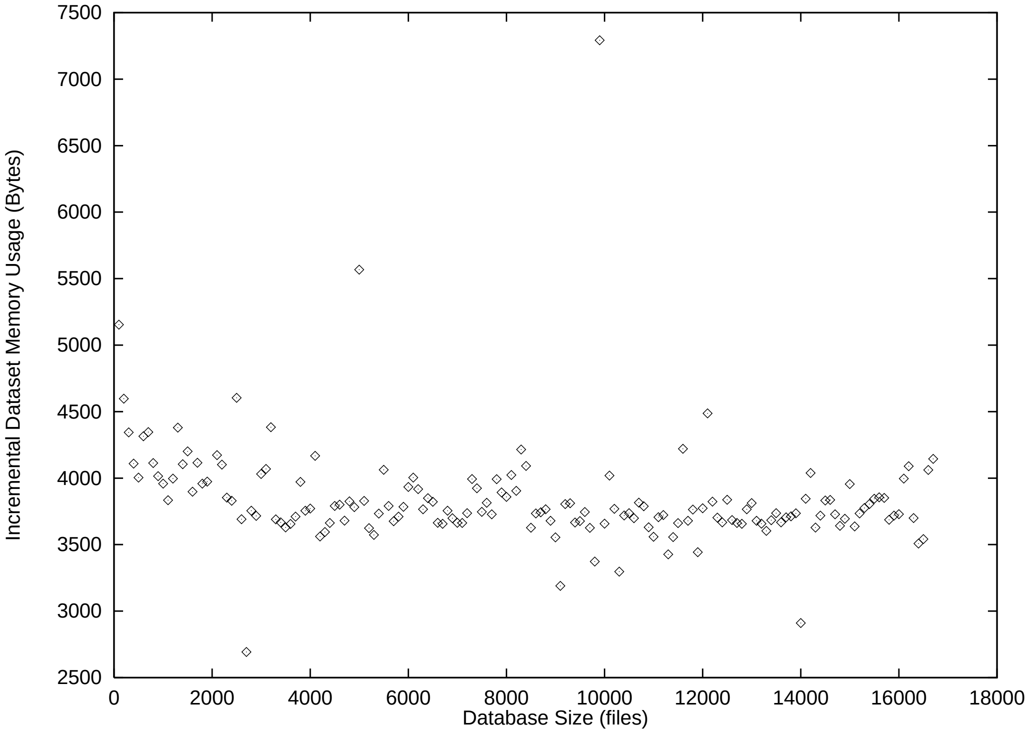


Figure 8: Incremental dataset memory usage versus dataset size (Linux)

There is an interesting performance artifact observed in comparing the NT and Linux performance measurements. While the Linux latency numbers increases smoothly, the NT measurements show a "staircase" effect, where measurements jump across discrete latency levels. We conjecture that this is due to a difference between the memory allocation policies of the NT and Linux Java JVM implementations.

The second set of experiments measured the incremental increase in the size of the dataset as documents are indexed (see Figure 8). The results show that the average increase is 3800 bytes (400 byte standard deviation). While this number is significantly larger than the document size (slightly less than 1KB), we believe it is mostly due to constant factors (e.g., the data structures used to store documents and index information). We expect that incremental growth will be linear in the size of documents and *not* a multiplicative factor.

We also measured the variance in query times based upon the number of terms in the query. However, because the dataset has too few terms, we did not get statistically significant results.

6 XBench

Having taken a look at the XSet performance results, we would like to put the numbers in perspective with those of other XML databases. This calls for a suite of tests that provide a realistic evaluation of XML query performance with respect to a variety of workloads. In addition, the tests should focus in on the types of query operations that applications will most likely utilize.

6.1 Functionality Space

In order to design a benchmark indicative of real application performance, we need to first analyze the functionality provided by these query engines, and how they are utilized in current XML applications. Their highly variable structure means that query performance on different types of XML documents can vary greatly, making the task of identifying XML querying patterns even more important.

Because of the immaturity of the research area, the use of XML in various application spaces is still being explored. Currently, the use of XML is focused in two main areas, metadata encoding, and as an access method to large scale databases. These two modes of operation focus on different aspects of XML access, and would result in drastically different performance.

XML Metadata

XML's flexibility, low overhead, and readability make it ideal for encoding metadata. In applications such as resource discovery, searchable E-mail clients, and meta-indices for filesystems, XML provides a valuable way to find and access metadata quickly.

Due to the dearth of deployed XML applications in the real world, establishing the nature of a typical XML search workload is not a simple task. From initial experiences with XSet and its applications, we have some knowledge about some common access patterns on XML metadata. In our applications, we found that almost all of the queries involve simple searches across a large number of small XML files, with the target being a single XML file or specific tag values selected from the target XML files. Queries are generally short, with string matching being the norm. Documents are generally modified by replacement, so updates are uncommon. Overall, queries dominated updates and index operations in our applications.

Bulk Data Manipulation

At the other end of the scale, a different set of applications use XML to encode relatively large documents with size on the order of databases. In this case, the majority of manipulation and searching occurs inside a single document, on the granularity of tags and subtrees.

One class of these applications deals primarily with data extraction and presentation. XML documents can be used as the canonical data format for persistent documents, which are modified in XML, and can be presented, transformed, or searched for in a variety of ways. The key operations involved are searching for tags and subtrees, relationships between

Another application class uses XML as an intermediate schema type for transforming between and querying across heterogeneous databases. Commercial products such as ObjectDesign Inc.'s Excelon support queries against an XML cache generated from heterogeneous database backends. Research efforts such as XML-QL [10] seek to use XML to extract large volumes of data from legacy databases for inclusion into new ones.

In general, operations on large-scale XML documents vary dramatically from those

Document Set	*MaxDepth*	*MaxBreadth*	*DocSize*	*NumDocs*
Bulk-data				
Shakespeare Major Plays	5	20	200K	37
Heart of Darkness Chapters	3	140	40K	4
Metadata				
Printer Descr.	3	11	1K	20
Book Catalog Entries	3	8	1K	4080
Web Log Entries	4	8	1K	154,500,000

Table 1: XBench Dataset

of XML metadata applications. Because of the inherent relational nature of the data, queries on these large documents are very similar to SQL queries [10]. Queries may include multiple joins, selection, and aggregation operations. In addition to complex queries, operations on these bulk XML documents will include incremental updates to the data. As a result, these XML datastores have high consistency and transactional requirements.

6.2 Experimental Datasets

Given the two XML application classes we outlined above, we want to design a benchmark that includes performance evaluation in both modes of operation. We have created a selection of XML documents to support both metadata and bulk XML workloads.

The bulk document portion of the dataset includes XML versions of large literary works, including the complete works of William Shakespeare and chapter by chapter XML representations of Joseph Conrad's *Heart of Darkness.* The metadata portion of the dataset includes metadata from network services (printers), bookstore catalog entries, and web access log entries. Table 1 summarizes the general properties of the dataset.

In addition to these metadata files, an XML data generator has been developed. It takes in arguments on multiple characteristics of the XML tree, such as number of files, breadth, and enumerated values versus random values inside tags. We believe this tool will be helpful in generating customized workloads to predict query performance in new applications.

6.3 Workloads

There are two workloads in XBench, simulating sample operations on bulk XML documents and metadata documents. There are seven performance tests in each workload.

In the bulk data workload, the performance tests focus on SQL style queries and a combination of selection, join, and aggregation operators. A sample query on the Shakespearean play dataset is: Find all scenes of each play, where the last line is spoken by the character who has the most lines in the play.

In the metadata workload, performance tests are geared towards finding a subset of the documents which match a set of given criteria. For example, a search on the book catalog dataset is: Find all books written by R. Allen Wyke in 1997. Because the aggregate nature of metadata XML files, several of the tests in this workload focus on scalability of performance to large numbers of XML files.

6.4 Benchmark Results

Because XSet supports XML querying focused on the metadata model, we only ran the metadata portion of XBench on XSet. The results are summarized below in Table 2. All tests were run on an Intel Pentium II 350 Mhz machine with 128 MB of memory, running Linux 2.0.36 with JDK1.1.7B and the Tya JIT compiler.

The tests in the metadata workload focus on scalability tests in indexing and queries, as well as multiple constraint queries. Indexing times indicate the full time taken to index a dataset. While these results lack a basis for comparison, they are useful as a baseline performance measure for future versions of XSet, as well as future XML query engines.

We had hoped to perform the XBench tests on additional XML databases such as

Test	*Time in ms*
LargeDBIndex	79950
SmallDBIndex	5780
LargeDBSimpleQ	6.88
MediumDBSimpleQ	5.6
SmallDBSimpleQ	1.24
MultConstQA	41.22
MultConstQB	146.3

Table 2: XBench on XSet

LORE [23] or ODI's Excelon [19]. LORE is a XML query engine which handles metadata queries similarly to XSet, but provides a much richer set of funcitonality. Unfortunately, the current release of LORE is not optimized for XML data, and the optimized version was not ready in time for the benchmark to be performed and included in this paper.

7 Applications

In this section, we discuss several XML-enabled applications that are based upon XSet. Some of these applications use XSet to simplify existing implementations, while others are new applications that are made possible by XSet.

7.1 Service Discovery Service

The Service Discovery Service (SDS) [9] provided the original motivation for the design of XSet. The SDS is a wide-area soft-state-based directory service that responds to client queries about distributed services. Service descriptions are completely independent, so no notion of transactions is necessary. Clients use XSet's flexible query model to formulate powerful service description queries.

The SDS currently incorporates XSet as a component and performance analysis of the SDS system shows that XML queries are only a small component in the overall service discovery latency. Because of the soft-state nature of the data, XSet's cleaner component is used to prune outdated service descriptions from the dataset.

7.2 Personal Activity Coordinator

Another example of an XSet application is the Personal Activity Coordinator (PAC), an application written as part of the ICEBERG [30] application architecture which acts as an intelligent cache of the current location and activities of ICEBERG users. Other ICEBERG applications query the PAC in order to determine the ideal contact point for incoming communication. The current implementation of the PAC uses an internal XSet server to store location- and application-specific information and services application queries.

7.3 Automatic Path Creator

One of the key components of the Ninja [29] service infrastructure is the Automatic Path Creator (APC), a component that constructs a dataflow path between multiple Ninja services to compose a larger service. Inside the APC, an XSet server stores information on known subpaths and known services, and queries against it as part of a graph search algorithm to genereate the logical path composition. Here, data stores are short-lived, and the fast query times of XSet are crucial to constructing paths within a reasonable response time.

7.4 FS-ML: secondary file index

By focusing on performance, XSet is able to integrate XML searching functionality into low-level applications, where performance is paramount. A meta-index on an existing file system is yet another example of such an application. By encapsulating a file inode with its metadata, a user can find a file efficiently by searching on any property associated with it, while the disk layout can still be optimized by the underlying file system. By making

FS-ML a part of the operating system, file uniqueness can be detached from a directory hierarchy, and instead linked to a distinctive subset of XML tagged file properties.

The FS-ML file meta-index would use a soft-state model to cache the most frequently accessed files, so that user queries could be fulfilled immediately while the rest of the index is paged in. Furthermore, XML Linking [22] functionality can express relationships between individual files, and the extensibility of XML allows file searching to naturally extend across the network with the use of additional qualifier tags, e.g.

`<FILESERVER>PLEIAIDES</FILESERVER>`.

Recent work on the HAC file system [14] discusses a system that closely resembles FS-ML in design and functionality. Burra Gopal and Udi Manber suggest user queries to a semantic directory system as an alternative file access model. This is directly analogous to a flexible XSet query on an index of XML-based file metadata. Furthermore, they also mention the notion of mapping remote files into the semantic directory using the notion of "name spaces." The FS-ML fileserver/network tag references offer an extensible superset of this functionality. While the FS-ML idea is untested and untried, it has potentially several advantages over HAC, including modularity, ease of implementation, code reuse, and file linking. The Semantic File System [13] also offers similar functionality to FS-ML.

7.5 Context-based E-Mail Searching

XML searching can lend new functionality to E-mail clients. If E-mail messages are stored with XML tagged metadata, then E-mail clients could use XSet's flexible interface to search E-mail messages.

A scalable, modular E-mail client is now under development using the Ninja distributed services infrastructure. Discussions are underway to integrate XSet inside to provide fast E-mail searching functionality, as well as forming virtual "folders" on the fly through XML searches. Furthermore, E-mail messages can be described as XML documents, so that enhanced E-mail clients can embed and search for specific XML tags using XSet.

8 Related Work

In this section, we discuss several XML storage and query efforts in industry and academia, including: object-oriented XML databases, several proposed XML query languages, and the LORE DBMS. The discussions highlight some of the key tradeoffs between features and performance: XSet lies on the end of less functionality (and thus less complexity) and more speed, while database systems and other XML repositories tend to choose a fuller feature set (with the added burden of more complexity and thus lower performance on smaller, simple workloads).

8.1 Object-Oriented XML Databases

Most of the industry-based implementations of XML-stores are object-oriented database systems that support XML as a native datatype. Two OODB systems that exemplify the industry XML effort are eXcelon from Object Design Inc. [19] and Poet XML Repository from Poet Software [28]. While they diverge slightly in their goals (eXcelon for translation of heterogeneous database backends and Poet for Electronic Data Interchange), both of them provide ACID semantics, which imposes additional overhead on performance and concurrency.

8.2 Relaxed Semantics in Databases

Past work in the database community has recognized the changing semantic requirements of database applications [5]. Several approaches have been taken in the context of full ACID database systems to maximize concurrency by taking advantage of these weaker semantic needs.

Some of these efforts have focused on how semantic information on datatypes can be exploited to safely trade serializability or consistency for increased concurrency. Farrag and Ozsu analyze in [11] a proposal to utilize semantic information to allow selected nonserializable schedules, and also propose the notion of "relatively consistent" (RC) schedules, and concurrency mechanisms to produce RC schedules. Similarly, Badrinath and Ramamritham defined a "recoverability" predicate which is checked using a conflict table of predefined conflicts between well-defined operations [4]. Since utilizing the semantic information incurs a high overhead, Agrawal et. al propose that users intervene to make consistency assertions on abstract data types, which are then used to define new correctness criterion [2]. In [31], Wong and Agrawal define the notion of bounded inconsistency, where users can accept datatype-specific ranges of inconsistency in order to increase commutavity of operations for increased concurrency.

Additionally, there have been efforts such as [20] which offer increased concurrency without breaking the bounds of traditionally serializability.

In contrast, our approach in XSet can be viewed as an extreme version of those proposed by [2], [4] and [11]. Because these efforts are generalized for different datatypes, they require semantic information on new datatypes in order to maintain levels of serializability. XSet, on the other hand, targets XML as its datatype, and can exploit its well-known structure for further optimization. Furthermore, the simplifying assumption of independent operations removes the need for transactions along with any associated overhead.

8.3 Proposed XML Query Languages

Whereas XSet chooses an extremely simple query model with a small set of core query functionality, several XML query language development efforts are underway to provide much more robust and powerful query models.

XML-QL from AT&T research labs [10] is an effort to standardize an XML query mechanism for large volume data extraction and transformation. As a query language, XML-QL tries to stay true to the SQL syntax, making choice extensions for XML functionality. Unlike XSet's focus on single query latency, the XML-QL design focuses on features and very complex queries. As a result, an XML-QL implementation is geared towards supporting extremely large transactions across large portions of the dataset, but its high complexity level and high overhead would make it too complex for our needs.

XML Query Language (XQL) [25] is a similar query language effort from Microsoft. It's similar to XSet in that it abandons the SQL syntax in favor of a natural XML approach composed of paths constructed from tag hierarchies. Unlike XSet, however, it supports a very complex syntax, accepting complexity in query construction and processing for greater functionality. As with XML-QL, we believe XQL is far more complex than is necessary for XSet's target applications.

8.4 LORE

LORE [23] is a database management system for semistructured data developed at Stanford University. While LORE and XSet are similar in basic functionality, LORE supports a much greater feature set, as well as support for full database semantics, with multiple indexing methods, cost-based query optimization, concurrent user support, and logging and recovery. LORE supports LOREL [1], a query language for XML with similar descriptive power as XML-QL. Compared to XSet, LORE's much richer functionality set makes it too complex for the low latency, soft consistency information management applications XSet targets.

9 Future Work

Because of XSet's widespread applicability, there are several avenues for future expansion. The main limitation to XSet's scalability is the dependence on main memory, and that can be solved by building a clustered version of XSet, where single XSet servers communicate to dynamically partition incoming data. In addition, significant improvements can be made on the XBench suite, to provide a more complete benchmark more indicative of real application performance.

9.1 Clustered XSet

Despite the increasing availability and capacity of memory chips, main memory still remains the only obstacle between XSet and large scale datasets. Our solution is to build an interserver communication layer which allows servers to join a XSet server group, and dynamically repartition the data as necessary to provide scalability. For an overloaded server handling queries on heterogeneous datatypes, the naive solution is to partition data by its document type or DTD. This will likely not solve the real problem, however, since large uniform datasets will still present scalability problems for single XSet servers.

The data partitioning of homogeneous data can be done in two ways:

- *Broadcast Query Model.* One of the solutions to data partitioning is to decouple the data partitioning from the query processing. Queries are forwarded to all available servers and responses are gathered before aggregation and return to the client. Incoming documents are hashed to a set size unique string using a good one-way algorithm such as SHA-1. A global mapping which partitions the hashed space evenly among servers can be used to distribute data.

  In this model where queries are sent to all servers, there may be several disadvantages. First, due to the random clustering, queries that want to search the entire dataset need to block until all nodes in the cluster respond, increasing the response time to that of the slowest node. Second, node failures pose a serious problem, and must be recognized promptly to minimize impact on query latency. Finally, broadcasting queries to each node in a cluster will not scale well as the number of nodes increases.

This issue might be solved by the increasing bandwidth available on System Area Networks, or the use of broadcast and snooping protocols.

- *Introspective Partitioning.* To solve the issues in the broadcast query model, we want to examine how data can be intelligently partitioned, and queries selectively routed to nodes with plausible return values. Any such design would have to deal with the fact that incoming queries can query on any tag, meaning that no single tag can be used as the server partition index.

  To facilitate a solution, we propose this hypothesis:

  For XML or other data queried by name-value pairs, when used in a specific application, the frequency distribution of queries is not uniform across its tags or named attributes; Furthermore, in specialized applications, the majority of queries will be on a small minority of the tags, which we can call "key tags."

  To test this hypothesis, we can design an introspective data partitioning algorithm. We can first initialize the server using the Broadcast Query Model, and process queries for some period of time. During the initial runs, an introspective daemon can monitor all incoming queries, and get a "rough" estimation of the most commonly used query tags, which can be then used as a "primary key" for data partitioning. After the first data partitions are created, queries are continuously monitored, and data can be repartitioned at different granularity levels according to changes in query patterns, data size, and server load. During repartitioning phases, the cluster can either revert back to the broadcast model, or move data in incremental stages, keeping partition maps consistent to maintain query routing.

9.2 XBench Improvements

As the realm of XML applications solidifies, it will become more clear how XML data will be accessed. With some trace data and usage characteristics from real world applications, XBench workloads can be tuned to better predict application performance. It would also be useful to better understand XSet's relative performance advantages, by running XBench on the major XML databases as they mature.

10 Conclusion

In this paper, we have shown how using XML as a data storage language, combined with a main memory database and search engine, provides the class of Internet-scale applications with easily extensible, yet validatable data schemes. We simplify the query language, which enables applications to perform simpler and faster queries.

We also avoid the problems associated with duplicate code development by providing a common data management platform. The key to this final goal is XSet's flexibility in how data is structured, queried, and managed.

The performance results clearly show the benefits of relaxing consistency requirements and using data structures that are better tailored to the datasets' inherent structure — query time scales logarithmically with dataset size.

Finally, we offer XBench, a rudimentary benchmark for measuring XML query performance. XBench offers workloads for testing both bulk documents and small XML metadata documents.

A portable version XSet is available publicly and XSet is being used by several large-scale distributed applications. We are continuing to refine the architecture based upon our experiences and others. Future versions of XSet will address incremental scalability and dynamic data partitioning

Bibliography

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[2] D. Agrawal, A. El Abbadi, and A. K. Singh. Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems*, 18(3):460–486, September 1993.

[3] ayavel Shanmugasundaram, H. Gang, Kristin Tufte, Chun Zhang, David DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 1999 VLDB Conference*. ACM SIGMOD, September 1999.

[4] B. Badrinath and K. Ramamritham. Semantics based concurrency control: Beyond commutativity. *ACM Transactions of Database Systems*, 17(1):163–199, March 1992.

[5] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[6] Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June - July 1998. ACM.

[7] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language

(XML). W3C Proposed Recommendation, December 1997. `http://www.w3.org/TR/PR-xml-971208`.

[8] D. D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM '88 Symposium*, pages 106–114, Stanford, CA, August 1988. ACM.

[9] Steven Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An architecture for a secure service discovery service. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking*, Seattle, WA, August 1999. ACM.

[10] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, August 1998. `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

[11] A. Farrag and M. Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[12] David Gelernter. Generative communication in Linda. In *Transactions on Programming Languages and Systems*, volume 7, pages 80–112. ACM, January 1985.

[13] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[14] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 265–278, New Orleans, Louisiana, February 1999. ACM.

[15] James Gosling and Henry McGilton. The Java language environment, a white paper. `http://java.sun.com/docs/white/langenv/`, May 1996.

[16] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154, September 1981.

[17] W3C DOM Working Group. Document Object Model, December 1998. `http://www.w3c.org/DOM/`.

[18] Timothy A. Howes. The Lightweight Directory Access Protocol: X.500 Lite. Technical Report 95-8, Center for Information Technology Integration, U. Mich., July 1995.

[19] Object Design Inc. An XML data server for building enterprise web applications.

[20] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[21] Josh MacDonald and Ben Y. Zhao. T-treap and cache performance of indexing data structures. http://www.cs.berkeley.edu/ ravenben/research/CS252/252Paper.pdf, December 1999.

[22] Eve Maler and Steve DeRose. XML Linking Language (XLink), March 1998. `http://www.w3.org/TR/1998/WD-xlink-19980303`.

[23] 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, September 1997.

[24] P. V. Mockapetris and K. Dunlap. Development of the domain name system. In *Proceedings of SIGCOMM '88*. ACM, August 1988.

[25] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). In *QL '98 - The Query Languages Workshop*. W3C, December 1998. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[26] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, February 1992.

[27] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[28] Poet Software. XML - The foundation for the future. `http://www.poet.com/xml.html`.

[29] The Ninja Team. The Ninja Project. `http://ninja.cs.berkeley.edu`.

[30] Helen J. Wang, Bhaskaran Raman, Chen nee Chuah, Rahul Biswas, Ramakrishna Gummadi, Barbara Hohlt, Xia Hong, Emre Kiciman, Zhuoqing Mao, Jimmy S. Shih, Lakmi Subramanian, Ben Y. Zhao, Anthony D. Joseph, and Randy H. Katz. Iceberg: An internet-core network architecture for integrated communications. *IEEE Personal Communications*, 2000. Submitted for publication.

[31] M. H. Wong and D. Agrawal. Tolerating bounded inconsistency for increasing concurrency in database systems. In *Proceedings of the 11th Symposium on Principles of Database Systems*. ACM SIGMOD, June 1992.