XSet: A Lightweight Database for Internet Applications

Ben Y. Zhao, Anthony D. Joseph

Computer Science Division

University of California, Berkeley

{ravenben, adj}@cs.berkeley.edu

Abstract

Internet-scale distributed applications (such as wide-area service and device discovery and location, user preference management, the Domain Name Service, and personalized web portal pages) impose interesting requirements on information storage, management, and retrieval. They maintain structured soft-state and pose numerous queries against that state. These "Query Enabled" applications typically require the implementation of a customized, proprietary query engine that is often not optimized for performance and is costly in resources. Alternatives include using traditional databases, which can hamper flexibility, extensibility, and performance, all of which are critical requirements of Internet-scale applications, or a directory-based protocol, such as the Lightweight Directory Access Protocol (LDAP). Directory protocols pose composability problems and impose a rigid structure on queries. This paper proposes a different approach, *XSet*, based upon the use of the eXtensible Markup Language (XML) as a data storage language, along with a high performance, 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. XSet is a Java-based, easy to use, main memory, hierarchically structured database with partial ACID properties. Experimental measurements show that XSet performance is excellent: insertion time is a small constant value, and query time grows logarithmically with the dataset size. Furthermore, XSet significantly outperforms platform-specific LDAP servers and XML-based databases on the LDAP directory benchmark. XSet is available for download, both as a stand-alone application and as a component of the Ninja service infrastructure.

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 search and information retrieval component. This new class of "Query Enabled" applications often maintains a mix of structured soft-state [9] and durable hard-state, and poses numerous queries against that state. Examples of such Query Enabled applications are service- and device-location and discovery protocols, such as DNS [21] and LDAP [16], and applications which make use of simple and fast query functionality, such as personalized web portal pages, 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 a simple solution to these problems by using the eXtensible Markup Language (XML) [8] 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 for these applications 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 flexible data validation through Document Type Definitions (DTD), powerful expressiveness, and ease of use. XML accentuates structure by making explicit the inherent structure of the data, without imposing a rigid schema. 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 current standardization efforts make XML a natural choice for our needs.

XML is also useful because it provides a semi-structured data model. 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, performance optimizations 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. XML 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.

1.1 XSet Applications

In this section, we discuss several Query Enabled applications that use XSet for data management. Some of these applications use XSet to improve performance, while others are new applications that are made possible by XSet. As a whole, they demonstrate how XSet gives applications fast flexible querying capabilities with minimal overhead.

1.1.1 Berkeley Secure Service Discovery Service

Consider an academic campus of the near future, where the majority of the population is networked, and access the local computing resources using portable wireless devices. Users would like to utilize context-aware applications to access a wide range of dynamic data. For instance, a visitor wants to specify and find resources in their immediate surroundings, such as their meeting contacts or video projectors. Similarly, people who enter a building become temporary services, and register their personal preferences and profiles. Other applications such as group pagers can then query the XSet server to locate and reach users.

Using traditional databases to solve this would require a large number of static schemas, ranging from personal location profiles to printer specifications, in addition to constant updating of these schemas as the format of data evolves. The transactional support and consistency guarantees available would be underutilized. Furthermore, these overhead costs would be duplicated per administrative domain, possibly exacerbated by incompatible databases and schemas.

The Service Discovery Service (SDS) [10] is a wide-area soft-state-based directory service application implemented using XSet's querying functionality. "Soft-state" is the notion that all data has a finite lifetime, and systems provide fault-tolerance by supporting periodic refreshment of data, rather than specialized recovery modes. The SDS does not support transactions across queries, and leverages soft-state data to manage update consistency. Current performance analysis shows that XSet query latency comprises a small fraction of the SDS query cost, and given expected optimizations on security and data distribution, the SDS will scale to extremely large datasets and query loads.

1.1.2 Personal Activity Coordinator

Another example of an XSet application is the Personal Activity Coordinator (PAC), which acts as an intelligent cache of the current location and activities of users in the ICEBERG [27] application architecture. 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.

1.1.3 Automatic Path Creator

One of the key components of the Ninja [25] service infrastructure is the Automatic Path Creator (APC), which constructs a dataflow path between multiple Ninja services to compose a larger service. The APC uses an XSet server to store information on known subpaths and known services, and queries against it as part of a graph search algorithm to generate the logical path composition. Here, data is short-lived, and the fast query times of XSet are crucial to constructing paths within a reasonable response time.

1.1.4 Personalized Web Portal

Another product developed using on top of the XSet database is SmartPages[1], a customizable web-based information portal. Smartpages retrieves interesting news articles, stock quotes and other dynamic content, stores it inside an XSet database, and presents it upon request. Users specifying their interest using Smartpage queries, and the XML content is transformed using XSL for their specific end-device. XSet could be used to provide similar functionality for other types of customized web portals, such as MyYahoo, MyExcite, and MyNetscape. These portals use user preference databases, based upon LDAP, real, or custom databases, to generate customized content for individual users.

1.2 Existing Database Models

Given the benefits that XML can bring to information management applications, the issue is 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 metadata intensive and distributed applications we have introduced, a streamlined minimalistic approach should improve performance.

[1]More information on SmartPages is available from `http://www.cs.berkeley.edu/~emrek/research/smartpagedemo.html`.

1.2.1 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 commercial 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.

There are two main reasons why neither database design is well-suited to the search functionality required by distributed applications. As discussed previously, the hierarchically organized structure of XML data has important advantages for Query Enabled applications. Unfortunately, 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 [4] that showed that while most queries can be transformed into relational queries, there were exceptions. Certain types of XML queries cannot be mapped into SQL, while other simple queries on XML map to large numbers of SQL queries, or single queries with numerous joins.

The second and more fundamental argument against using relational or object-relational databases, is the strict nature of database consistency and the associated performance penalty. While object-relational databases can efficiently model semi-structured data, they still incur this penalty. For 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 (See Section 4).

1.2.2 Semantics and Performance

Previous work in the database community has recognized the evolutionary model of database applications, and their changing semantic requirements [6]. While other approaches to address these changes give limited concessions for increased concurrency, XSet instead focuses on the tradeoffs between semantics and performance.

From a database perspective, XSet can be described as a memory-resident, hierarchically structured database with support for a partial set of the ACID semantics. In particular, XSet does not support transactions and it provides atomicity at the level of individual operations.

While 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. Such overhead is necessary to mask the effects of synchronous I/Os and long pausing transactions from the user. XSet is a main-memory database without transactional support; and therefore it can use coarse-grain locking per thread to minimize locking overhead with no adverse effects on performance.

We present detailed performance analysis of a single-node XSet implementation in Section 6 demonstrating the superior performance that results from the removal of overhead associated with transaction support. This section highlights 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.

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). Since XSet uses physical memory as its primary storage mechanism, it will incur a performance penalty when the dataset size scales beyond memory capacity. In Section 7, we propose a cluster model that should help ameliorate this problem.

Within these constraints, XSet has these following goals:

- Partial ACID Semantics (no transactions)

- Simplicity, Portability, Composability
- Flexible Extensible Schema Support
- Low Query Latency and High Scalability

The rest of the paper is organized as follows: we first discuss related work in Section 2. Then we present XSet's architecture in Section 3, and discuss its implications on data semantics in Section 4. That is followed by the implementation in Section 5, and XSet's performance analysis in Section 6. Finally, we discuss future work in Section 7 and conclude in Section 8.

2 Related Work

In this section, we discuss related work on relaxed semantics in databases, the LORE [20] semi-structured database, and proposed XML query languages. The discussion also highlights XSet's position on the tradeoff between features and performance: XSet lies on the end of minimal complexity and increased performance, while database systems and other XML repositories tend to choose a fuller feature set (often with the added burden of more complexity and thus lower performance).

2.1 Relaxed Semantics in Databases

Past work in the database community has recognized the need for relaxed ACID semantics [6]. 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 about datatypes can be exploited to safely trade transaction serializability or data consistency for increased concurrency. Farrag and Ozsu analyze in [12] a proposal to utilize semantic information to allow the use of selected nonserializable schedules. They propose the notion of "relatively consistent" (RC) schedules and concurrency mechanisms to produce RC schedules. Similarly, Badrinath and Ramamritham define a "recoverability" predicate, checked using a table of predefined conflicts between well-defined operations [5]. Since utilizing 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 [28], Wong and Agrawal define the notion of bounded inconsistency, where users can accept datatype-specific ranges of inconsistency in order to increase commutativity of operations for increased concurrency. Additionally, there have been efforts such as [17] which offer increased concurrency without breaking the bounds of traditional serializability under certain conditions [3].

In contrast, our approach in XSet can be viewed as an extreme version of those proposed by [2, 5, 12]. 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. Additionally, the simplifying assumption of independent operations removes the need for transactions along with any associated overhead.

2.2 LORE

LORE [20] is a database management system for semi-structured data developed at Stanford University. LORE's internal *Dataguides*, which are evolutionary representations of DTDs, are similar to XSet's merging tag index. While LORE and XSet are similar in basic functionality, LORE supports a much richer feature set, including full database semantics, multiple indexing methods, cost-based query optimization, concurrent user support, and logging and recovery. LORE supports LOREL [1], a query language for XML which follows the general SQL syntax and adds extensive XML-specific functionality such as similarity searches. Compared to XSet, LORE's much richer functionality set makes it too complex for the low latency, soft consistency information management applications XSet targets.

2.3 Proposed XML Query Languages

Whereas XSet chooses a simple query model with a small set of core query functionality, several XML query language development efforts are underway to provide more robust and powerful query models. XML-QL from AT&T research labs [11] 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, adding extensions to support XML-specific 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 datasets, but its high complexity level and high overhead would make it too complex for our needs.

XML Query Language (XQL) [22] is a similar query language effort from Microsoft. It abandons the SQL syntax in favor of a natural XML approach composed of paths constructed from tag hierarchies. It also supports a greater query functionality, and accepts the associated complexity in query construction and processing.

We believe that XML-QL, LOREL and XQL are far more complex than is necessary for XSet's target applications. In fact, we can characterize the XSet query model as a very limited subset of XQL, using a intuitive syntax. As a query model, XSet queries also resemble the associative matching aspects of Linda Tuplespaces [13]. Linda differs in that it is a communication mechanism with a simple query interface, limited to flat data tuples.

The distinctions between XSet and these existing systems will become clear as we discuss the XSet architecture (Sections 3 and 4) and present performance comparisons between LORE and XSet (Section 6).

3 Architectural Design

In this section, we discuss the design choices and their ramifications for the XSet architecture: the query model, the tag index, document paging, and the durability and cleaner mechanisms.

3.1 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 includes an XML index and a memory-resident data store.

During registration/insertion, the SetServer receives XML documents via a 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, each document is assigned a monotonically increasing

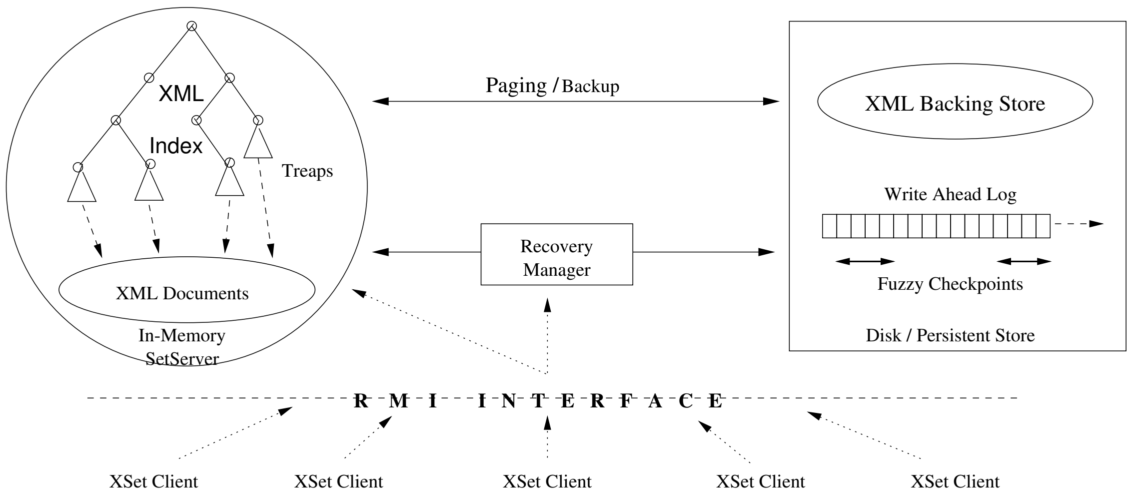

Figure 1: Single XSet Server

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 [24] (a probabilistic self-balancing tree structure), each set indexed by a common tag value. A single document has 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. Checkpointing provides both log truncation and fast recovery. Fuzzy checkpointing is especially important to Query Enabled applications, since data must be available at all times, even during the course of a checkpoint operation. 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 short-lived 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.

```
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

3.2 Query Model

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.

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>

3.3 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.

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 attributes and their values attached to the relevant tag, so that they can be checked against queries with attribute constraints.

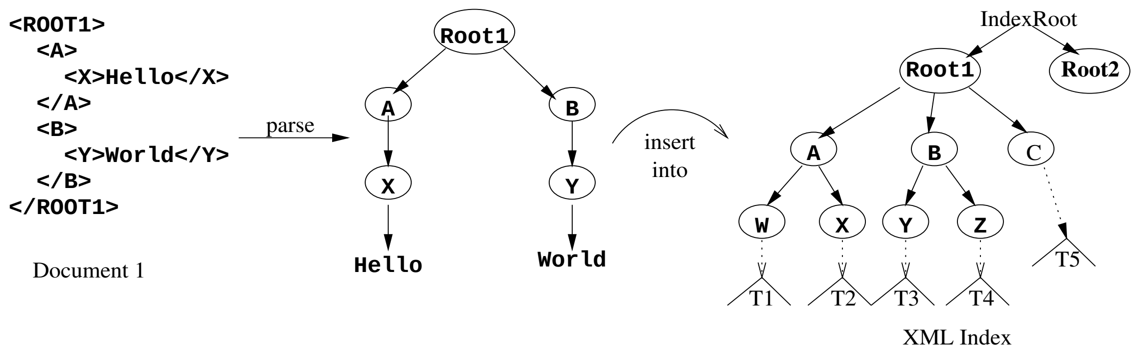

Figure 3: Simple Indexing Example

The key distinction between this index scheme and some other XML indices [20, 11] 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 if it appears inside the sequence of tags `PERSON -> HOME -> ADDRESS` versus `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 [20], except the root node end of the path is fixed.

3.4 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 2 kbytes, which can be significant for large collections of small 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 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 that better exploit XML tag structure can also be used. For example, one policy for a service discovery service would be to partition the data on service types, and apply an LRU algorithm to the services, resulting in the least frequently queried services being paged out of memory.

3.5 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 of operation and the unique identifiers of the affected document(s). During recovery, this approach 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 4.3.

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 are no preset algorithms for determining when the document pager is run, or in 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.

3.6 Data Cleaning Mechanism

XSet also includes an optional data cleaning 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 lifetimes of 5 days. This allows an administrator to provide customized soft guarantees on the freshness of the dataset contents.

4 Semantic Guarantees

In this section, we define the data semantics provided by XSet. We explain several assumptions regarding the nature of data used by Query Enabled applications and the general access patterns on this data. We then use these assumptions as the basis for a data model that focuses on performance and simplicity.

4.1 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 [15] terminology, where ACID stands for Atomicity, Consistency, Isolation, and Durability. As explained below, XSet does not support transactions, and the list below falls under that context.

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 recovery across failures, by providing a simple and efficient combination of write-ahead logging and fuzzy checkpointing. Durability of data is flexible, and we provide the mechanism to support different positions in the durability and performance tradeoff.

4.2 Applications Semantics

As stated above, XSet provides different data semantics from those provided by typical database systems. While XSet provides 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 exemplify this class of applications. We optimized the XSet design towards certain properties of data used in these applications, such as immutability and transient data. Applications using data that break these assumptions, however, still benefit from the overall XSet model.

The first simplification XSet makes is 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 resource utilization and increase throughput. XSet, however, is a main memory database, where memory access latency is the dominating 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. While transactions are useful in certain contexts, they introduce significant additional complexity and performance overhead. Thus, we choose the single operation as the granularity of operation. Furthermore, because data is immutable, and operations are independent, modify operations in XSet are idempotent; that is, single operations can be repeated in order without making the database inconsistent.

4.3 Fast Recovery

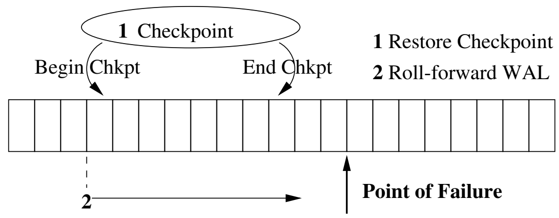

Figure 4: Recovery Mechanism

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 (the loss of the server's memory contents), the recovery process follows two steps: First, the system restores a memory image using the fuzzy checkpoint. Then, the system applies the redo log starting at the LSN of the begin_checkpoint operation. 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.

5 Implementation and Status

XSet has undergone several major design and implementation modifications and a distribution is now publicly available in two forms: a stand-alone application (`http://www.cs.berkeley.edu/~ravenben/xset`), and as an application written using the Ninja distributed services framework [25]. XSet has also been integrated or is being integrated in to several applications (see Section 1.1). In this section, we discuss specific details about the current XSet implementation.

5.1 Implementation Platform

For portability and ease of implementation, we chose Java [14] 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. To parse XML, XSet uses the DOM API [26] for parser indepdence and the XML4J parser from IBM Research Labs.

5.2 Persistent Datastore

For simplicity, XSet currently uses the file system as its persistent backing store. 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 memory-mapped interface or a log-based file system [23].

5.3 Treaps

Treaps are probabilistically self-balancing trees that achieve $O(Log_2(n))$ time for all operations [24]. 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 trees with dual indices for each object (Cartesian trees).

Treap performance characteristics are similar to other binary trees such as T-trees and red-black or AVL trees, and treaps have the advantage of preserving heap order on a secondary key. This secondary key usually is a pseudo-random "priority" generated at insertion time and 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. For example, incrementing the priority with a small randomized number during each access and then rotating 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 [19].

Treaps have also been shown to be extremely efficient for parallel algorithms on ordered sets [7]. Using treaps allows us to investigate these fast parallel algorithms in distributed and parallel versions of XSet.

6 Performance

In this section, we evaluate the performance and scalability of the current XSet implementation, and compare its performance under a LDAP Benchmark to that of Lore [20] and SLAPD [18], a widely-used, publicly available stand-alone LDAP server. Both Lore and SLAPD are implemented in C. We first show that XSet compares favorably against existing systems, then show that XSet scales well as the size of the dataset increases[2]. For experiments, we used Linux and Windows NT platforms: Linux 2.2.13 with IBM's JDK 1.1.8 for Linux (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).

6.1 Comparative Query Performance

Given the lack of an existing XML database benchmark, we gauge the performance of XSet by comparing its query latency with Lore and SLAPD using an existing directory benchmark, DirectoryMark 1.1[3]. We

[2]Note that we did not compare XSet performance against commercial LDAP servers as such a comparison would not appropriate. These servers are not composable components and they represent 1000's of person-hours of hand optimization and are written in platform-specific languages.

[3]DirectoryMark is available from Mindcraft at http://www.mindcraft.com/directorymark/ .

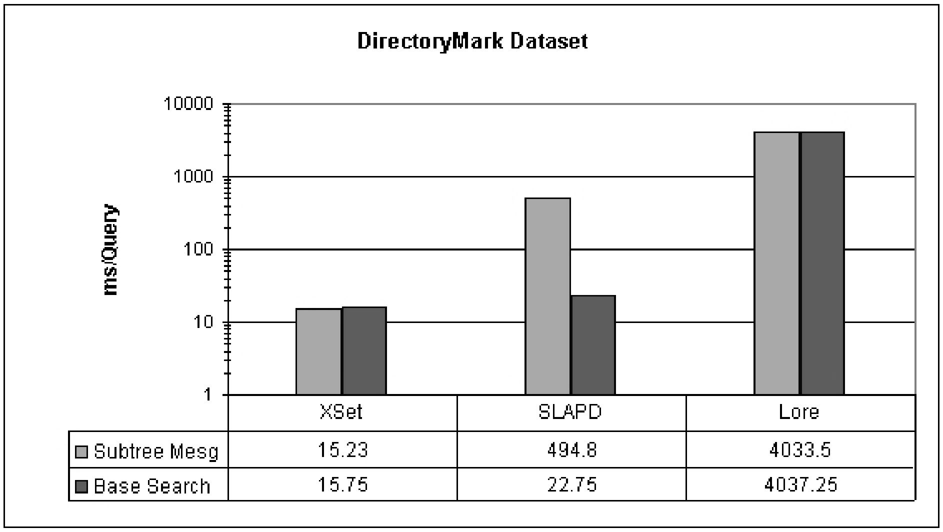

Figure 5: Query Performance under LDAP DirectoryMark 1.1

implemented an LDAP front-end to XSet that supported BASE, SINGLE, and SUB tree exact match queries. We then implemented a similar front-end for Lore (Diet 5.0 for Linux) and configured SLAPD to support similar queries. Running each LDAP server on our Linux machine (2.2.13, PII 350Mhz, 128MB RAM), we tested two main components of DirectoryMark, Subtree matches and Base-level searches, using its small dataset configuration, and present our results below.

In Figure 5, we show the query latency results of the DirectoryMark 1.1 data on a logarithmic scale. We see that on the Subtree messaging test, XSet outperforms SLAPD by more than an order of magnitude, and Lore by more than two orders of magnitude. In the Base level search test, XSet again outperforms Lore by two orders of magnitude, and SLAPD by 31%.

Note that we have not included the cost of transforming data between LDAP's native binary formats and XML (approximately 200 milliseconds/query). While generalized tools for this conversion exist in Java, their string manipulation performance is poor, thus we believe that given time, we can implement their functionality much more efficiently in C, and reduce the conversion overhead to an estimated 4-5 milliseconds/query. While this overhead is not shown in our charts, the impact of its addition will have little impact except for the Base level search, where the performance of XSet with conversion will be similar to that of SLAPD. The LORE test incurred a high cost as a result of the Java interface to the standalone LORE server (The C-API for Linux was not functional at the time of our measurements). To be fair to LORE, we did not include this prohibitively high data translation cost (¿ 1 minute) in our final results.

NOTE to reviewer(s): We expect to have an efficient ANS.1 to XML translator implemented by summer; and should this paper by accepted, we will include the translation overhead in the results for the final paper. We will also be able to include more detailed measurements and analysis.

When we analyze the differences in performance between XSet, SLAPD, and LORE, the key factors to consider are language of implementation, memory- versus disk-based storage, and complexity due to functionality. While XSet is implemented in Java, both SLAPD and LORE are implemented in C. Despite this fact, XSet shows a considerable performance advantage, the majority of which can be attributed to its operation as a main-memory database, whereas both SLAPD and LORE are disk-based. Finally, the high query latency for LORE can be attributed to complexity associated with supporting additional functionality such as proximity searches.

6.2 XSet Performance and Scalability

While we have shown XSet to compare favorably against existing databases in LDAP applications, a second key goal is to show that XSet performance scales well with dataset size. XSet performance consists of three components: validation, insert/delete, and queries. *Validation* is the one-time process of certifying that the XML document conforms to an external DTD, and takes roughly 4 ms/query (NT). *Insert/delete* costs impact operations that modify the index, and also log roll-back during database recovery. Finally, *query processing* is the latency involved in servicing a query.

In our measurements, we find that index/delete latency is independent of dataset size and approximately 2 ms for small documents (NT). We also found that query latency scales logarithmically with dataset size, as shown in Figure 6. As the dataset increases in size, the query latency is dominated by the cost of memory accesses incurred through traversing the in-memory index. Since the treap index structure is probabilistically self-balancing, each search down an index tree involves $O(Log_2(S))$ memory fetches, where S is the number of entries in the database. In fact, we expect that by replacing the treap in the index with a higher-branching data structure such as the B-tree, we can further reduce the rate of latency increase to $O(Log_n(S))$, where n is the B-tree branch factor. The latency does stray from the logarithmic line, however, when the dataset

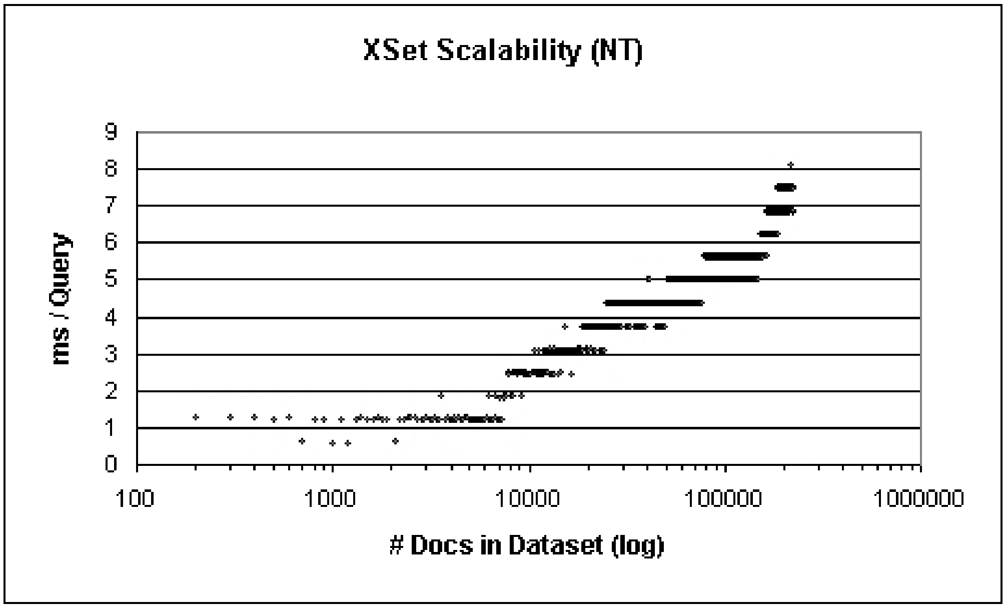

Figure 6: Query time versus dataset size (NT Server)


```
<?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/4.5.html</LOCATION>
  <HTTPPROTO>HTTP/1.0</HTTPPROTO>
 </ACTION>
 <RETCODE>200</RETCODE>
 <TRANSIZE>3868</TRANSIZE>
</WEBLOG>
```


Figure 7: A sample XML database file

reaches 180,000 documents. We believe this is due to the Java garbage collection mechanism running more frequently once a high memory utilization threshold (90%) has been reached.

To create this dataset, we converted an HTTP web server access log into small (approx. 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. Figure 7 shows a sample XML database file.

Unfortunately, this dataset is not an optimal choice. Most data about each HTTP access is unique, so queries performing exact matches only return small result sets. Also, 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.

7 Future Work

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.

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 inter-server 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. For an overloaded server containing documents of one type, we plan to use an introspective heuristic to find the optimal keys for partitioning. The heuristic watches the query stream and determines the "primary" keys at regular intervals. The result is a dynamic system that adjusts to changes in query patterns and number of nodes available.

8 Conclusion

In this paper, we have shown how the XSet datatbase leverages XML to provide efficient, high performance storage and query functionality for semi-structured data. By simplifying the query language, XSet enables applications to perform simpler and faster queries.

XSet enables these performance improvements by relaxing the strict ACID semantics of traditional databases. In particular, XSet does not support transactions and only provides atomicity at the granularity of individual operations. The performance results clearly show the benefits of this choice and the use of data structures that are better tailored to the applications' data: query time scales logarithmically with dataset size.

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 also address incremental scalability and dynamic data partitioning.

References

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

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

[3] Agrawal, R., Carey, M. J., and Livny, M. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems 12*, 4 (1987), 609–654.

[4] ayavel Shanmugasundaram, Gang, H., Tufte, K., Zhang, C., DeWitt, D., and Naughton, J. F. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 1999 VLDB Conference* (September 1999), ACM SIGMOD.

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

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

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

[8] Bray, T., Paoli, J., and Sperberg-McQueen, C. M. Extensible Markup Language (XML). W3C Proposed Recommendation, December 1997. `http://www.w3.org/TR/PR-xml-971208`.

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

[10] Czerwinski, S., Zhao, B. Y., Hodes, T., Joseph, A. D., and Katz, R. 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.

[11] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. XML-QL: A Query Language for XML, August 1998. `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

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

[13] Gelernter, D. Generative communication in Linda. In *Transactions on Programming Languages and Systems* (January 1985), vol. 7, ACM, pp. 80–112.

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

[15] Gray, J. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases* (September 1981), pp. 144–154.

[16] Howes, T. A. The Lightweight Directory Access Protocol: X.500 Lite. Tech. Rep. 95-8, Center for Information Technology Integration, U. Mich., July 1995.

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

[18] LDAP Group, Univ. of Michigan. The SLAPD distribution. Available at `http://www.umich.edu/~dirsvcs/ldap`.

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

[20] McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J. Lore: A database management system for semistructured data. *SIGMOD Record 26*, 3 (September 1997), 54–66.

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

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

[23] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* (February 1992).

[24] Seidel, R., and Aragon, C. R. Randomized search trees. *Algorithmica 16* (1996), 464–497.

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

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

[27] Wang, H. J., Raman, B., Chuah, C. N., Biswas, R., Gummadi, R., Hohlt, B., Hong, X., Kiciman, E., Mao, Z., Shih, J. S., Subramanian, L., Zhao, B. Y., Joseph, A. D., and Katz, R. H. Iceberg: An internet-core network architecture for integrated communications. *IEEE Personal Communications* (2000). Special Issue on IP-based Mobile Telecommunication Networks.

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