Revisiting Structured Storage: A Transactional Record Store

Robert Grimm, Michael M. Swift, and Henry M. Levy
University of Washington
{rgrimm, mikesw, levy}@cs.washington.edu

Abstract

An increasing number of applications, such as electronic mail servers, web servers, and personal information managers, handle large amounts of homogeneous data. This data can be effectively represented as records and manipulated through simple operations, e.g., record reading, writing, and searching. Unfortunately, modern storage systems are inappropriate for the needs of these applications. On one side, file systems store only unstructured data (byte strings) with very limited reliability guarantees. On the other side, relational databases store structured data and provide both concurrency control and transactions; but relational databases are often too slow, complex, and difficult to manage for many applications.

This paper presents a transactional record store that directly addresses the needs of modern applications. The store combines the simplicity and manageability of the file system interface with a select few features for managing record-oriented data. We describe the principles guiding the design of our transactional record store as well as its design. We also present a prototype implementation and its performance evaluation.

1   Introduction

With the success of the Internet, the last few years have seen a proliferation of networked data services, such as electronic mail servers, web servers, and personal information managers. Such applications manage data that are regular in structure and easily represented as records with distinct fields, e.g., contacts, schedules, user preferences, or customer orders. Electronic mail messages or merchandise descriptions are less regular, but can also be organized as a set of well-defined fields, such as sender, subject, or order number. Internet applications also exhibit fairly simple workloads. For a given task, they sequentially access only a small number of data sources, for example, when a user browses merchandise descriptions or when a server processes a customer order. As a result, they perform a relatively simple set of operations: they read, modify, or search small records. Finally, these applications typically require replication for availability or reliability.

In the future, we expect to see an increasing number of such data-centric applications and a wider range of computing devices running them [13, 44]. On the hardware side, non-traditional computing devices, such as palm-sized computers or cell phones, are already in wide-spread use and provide access to a user's contacts and messages. Newer devices such as pads are about to be commercially released as well. On the software side, cluster-based services implement highly available and scalable networked servers [14, 39], while replication schemes specifically designed for a mobile environment automate the synchronization of data between remote and mobile nodes [1, 18, 25, 37].

Despite this diversity of existing and future applications with common data storage needs, the management of persistent storage remains a challenge. Currently, an application has two choices—file systems or databases—each of which fails to meet application needs in one or more dimensions. File systems typically store only unstructured data and provide limited (if any) failure atomicity, making them ill-suited for reliably storing large numbers of homogeneous records. Object stores and relational databases manage objects and records, respectively, and provide concurrency control and reliability through transactions. However, object stores are optimized for maintaining heterogeneous objects and their linked relationships. Relational databases are often far too complex and difficult to manage as they provide considerable functionality, e.g., support for a query language, joins, and batch processing. These functions are often overkill for modern networked data services, and they come at a high

cost.

To address this lack of appropriate storage solutions, we present a transactional record store that combines the simplicity and manageability of the file system interface with select features for managing record-oriented data. Records are stored in tables, and all records in a table share the same field names and types. The records are the rows of the table and the fields are the columns. Tables, in turn, are organized in directories, resulting in a hierarchical name space similar to that of a file system. To simplify replication, the store exposes globally unique identifiers [28] (GUIDs) for individual records as well as tables and directories. To ensure good performance, it uses a simple yet expressive hinting system. Finally, to provide reliability across failures, all operations are atomic and transactions can be used to group several operations into one atomic unit.

Record storage as a system service is not a new idea. For example, IBM's VSAM [34], Compaq's RMS [10], and Palm Computing's Palm OS [6] all provide record storage at the system level. This raises the question of why it is necessary to revisit the topic of structured storage. We believe that record storage as a system service is important for three reasons. First, a relatively new class of applications, as discussed above, requires it. Second, the most common computing platforms do not offer record storage. As a result, many commercial applications either (1) ship with an application-specific solution, thus leading to unnecessary application complexity and a considerable duplication of functionality, or (2) build on top of a relational database, which has its own performance, cost, and complexity implications. Third and most important, building a viable record store is hard, because it is not obvious how to make the right trade-offs between scalability, flexibility, complexity, and performance. The primary contribution of this paper is a thorough exploration of these trade-offs.

The rest of this paper is structured as follows. Section 2 develops the principles guiding the design of our transactional record store and Section 3 presents the actual design. Section 4 describes a prototype implementation and Section 5 reflects on our experiences in building the prototype as well as its performance. Section 6 reviews related work. Finally, Section 7 concludes.

2 Principles

A practical record store for data-centric applications should meet three requirements. First, it should be reliable, i.e., its operations should be atomic in the face of failures. This is particularly crucial given the economic significance of Internet services. Second, the record store should be scalable; it must be implementable across a wide range of computing platforms, from wearable devices to large-scale clusters. Third, it should effectively support application-specific replication, providing both flexibility and performance of replication mechanisms.

Relational databases use tables to store large numbers of records and provide transactions for reliability. Techniques for implementing both efficiently are well known. For this reason, we also base our record store on tables and transactions. However, relational databases provide complex functionality, such as a query language, joins, and batch processing, that are of limited use to networked data services. Furthermore, databases typically implement replication internally [32, 35] and thus lack effective support for application-specific replication schemes. The key issues for the record store are therefore which features to provide and how to structure its various interfaces.

To guide our design, we have used three fundamental principles:

1. *Limit global knowledge.* Knowledge should be locally generated and managed whenever possible. The primary purpose of this principle is to ensure an efficient implementation and scalability by limiting the need for global coordination of system state.

2. *Don't hide power.* The abstraction barrier between applications and the record store should not hide expressive power. The primary purpose of this principle is to ensure that the record store provides sufficient functionality.

3. *Separate independent concerns.* Different design aspects, such as data layout, data access, performance, or access control, should be clearly separated by using distinct operations and abstractions. The primary purpose of this principle is to ensure that the record store is easy to use and replicate.

These principles are not absolutes, as clearly useful features may follow one principle but violate others. The three principles thus need to be carefully weighed against the requirements of data-centric applications, as discussed above. When principles conflict, we typically favor limiting global knowledge to aid scalability.

3 Design

In our design, records are stored in tables, which are collections of identically-structured records. Each record in a table thus has the same fields with the same name and type (for a given record, though, not every field needs to store a value). Operations on tables affect only one table. In addition, a table has distinct operations to manage its schema (i.e., fields), its performance hints, its access control information, and its data. These operations are straight-forward and either access or modify a table's meta-data or data. For example, the schema operations allow an application to add and change individual fields, lookup a field by name, and retrieve a list of all of a table's fields.

Tables are organized in directories, which support typical directory operations such as entry lookup or move. Directories facilitate the logical grouping of related tables, and the resulting hierarchical name space provides a convenient and proven interface for managing persistent storage. Adding files as a separate storage abstraction in addition to tables makes it possible to integrate record and byte string storage into a single storage system. To further aid manageability, the store also supports symbolic links in the form of aliases, which can reference other aliases, directories, and tables and are automatically resolved during lookup. We call aliases, directories, and tables *store objects*.

To provide reliability, all store operations are atomic and applications can specify a transaction to group several operations into one atomic unit. To provide consistency, all transactions are fully serializable by default. Transactions combine two separate concerns, atomicity and concurrency control, which violates one of our principles. As a result, applications that only need reliability, such as a single-threaded contact manager running on a cell phone, also pay the overhead for concurrency control. The record store thus includes the option to set a different isolation level for individual transactions.

We have specified our record store in the form of a set of Java interfaces and exceptions. The specification does not rely on features unique to Java; it simply serves as a concise description of the record store's application programming interface (API). The store consists of 15 interfaces, most of which represent simple descriptors such as fields, queries, or hints, with a total of 82 methods. It also contains 16 exceptions that represent specific exceptional conditions and are subclasses of class `StorageException`. For transactions, we rely on Jini's transaction specification [2]. Figure 1 pro-

```
Guid add(Guid g,List fields,List data,Txn t);
  /* Add a new record with GUID g and data for
     fields and return its GUID. If g is null,
     the GUID is automatically generated. */

void write(Guid g,List fields,List data,Txn t);
  /* Write data to fields for the record with
     GUID g. */

List read(Guid g, List fields, Txn t);
  /* Read fields from the record with GUID g
     and return that data. */

Results query(Query q, List values, Txn t);
  /* Instantiate query q with values, perform
     the instantiated query, and return an
     iterator over the results. */

void delete(Guid g, Txn t);
  /* Delete the record with GUID g. */
```

Figure 1: The five methods for accessing a table's records. Records are added, written, read, and deleted one at a time by GUID. In contrast, queries can search records according to application-specific criteria and return more than one result. Applications can group several operations into an atomic unit by passing a transaction through the `Txn` (short for `Transaction`) parameter. All five methods may throw a `StorageException` or a `TransactionException`. GUIDs are explained in detail in 3.1 and queries in 3.2.

vides a flavor of our interfaces by showing the five operations used to access a table's data.

The rest of this section is structured as follows. We describe the motivation for and the use of globally unique identifiers in 3.1, followed by queries (3.2), hints (3.3), and access control (3.4). We conclude this section with a summary of our design in 3.5.

3.1 Globally Unique Identifiers

A primary challenge for implementing an application-specific replication scheme on a structured store is the identification of records and collections of records across node boundaries. In general, database implementations require an internal identifier to uniquely name records [19]; our record store formalizes this identifier and associates globally unique identifiers [28] (GUIDs) with records as well as store objects (i.e., aliases, directories, and tables). GUIDs represent an attractive choice for such an identifier, because the

specification for GUIDs includes an algorithm for generating them autonomously on every node while also guaranteeing that they are globally unique.

GUIDs are associated with store objects and records during creation and are immutable afterwards. Applications can either let the record store create a fresh GUID or explicitly specify the GUID. Typically, applications allow the record store to create the GUID when a store object or record is originally created on a node. They specify the corresponding GUID when a store object or record is propagated to a replica. Following our principle of limiting global knowledge, the record store enforces the uniqueness of GUIDs only within a limited scope. In particular, the store guarantees the uniqueness of GUIDs for records within a single table and the uniqueness of GUIDs for store objects within its local name space.

For store objects, GUIDs provide an alternative name space: store objects can be looked up (and deleted) by name as well as GUID. This makes it possible to locate replicated store objects across different nodes, even if they have different names on different nodes, as long as the store objects share a common GUID. As a result, store objects can be effectively replicated, even though different nodes may have different policies for organizing the local store.

For records, GUIDs provide the only name space. Every table has a field representing its records' GUIDs; this is comparable to the primary key in a relational database. As shown in Figure 1, records can be added, written, read, and deleted by GUID only. Furthermore, operations manipulate only one record at a time. The simplicity and regularity of these operations simplifies logging for replication. We believe that it does not represent an undue limitation, because many tasks need to access only a small number of records. However, access to records by GUID alone is not sufficient, because most applications have external identifiers, such as user names or book titles, that also need to be searched.

3.2 Queries

Applications search the records of a table using queries, for example, when searching for all mail messages sent by a particular user. A query is not limited to GUID-based record access and it can return more than one record. A query has three parts: (1) a select clause that specifies which records to select, (2) a (possibly empty) list of sort clauses that specifies the sort order for the selected records, and (3) a list of fields that specifies which fields to return

of the selected and sorted records. The select clause consists of one or more subclauses that compare a field to a value. This value may be specified at either of two times: query creation time or query execution time. In the latter case, the record store creates a *query template*, which is instantiated with the actual value at execution time (see Figure 1). Comparisons may be negated and are combined using conjunctions and disjunctions. The result of a query is an iterator over the selected and sorted records.

The challenge in designing a query facility for record storage is to balance expressiveness against implementation complexity and performance. To be consistent with the principle of limiting global knowledge, queries, just like other operations on tables, are restricted to a single table, thus avoiding the complexities associated with supporting joins [19, 33]. Furthermore, to be consistent with the principle of not hiding power, queries support sort clauses and templates. Sort clauses ensure that a query's results are ordered. Applications thus do not need to sort the returned data themselves and the record store can effectively schedule the prefetching of query results. Templates let applications express the structure of common queries, for example, those resulting from users filling out search forms. The record store can thus optimize table layout and indexes for performing these queries well.

3.3 Hints

Application-specific hints have been successfully used to optimize the caching and prefetching behavior of file systems [27, 42], thus suggesting that they can be an effective mechanism for optimizing the performance of record storage as well. For file systems, hints are dynamically issued by applications because they primarily control the dynamic behavior of the file system cache. To perform well, however, a record store not only needs to optimize the management of its cache, but also optimize the on-disk layout of tables (i.e., the on-disk order of records) as well as the generation of indexes (i.e., for which fields to generate which indexes).

Our record store consequently uses sets of hints to characterize dominant access patterns for tables. Hints are explicitly created for a specific table and statically associated with it. We expect them to only change when workloads change. Individual hints describe either an add, write, read, query, or delete operation. They have a name to simplify programmatic access and a weight specifying that hint's relative importance. Hints for add, write, and read operations also specify the fields to be added, writ-

```
Hint createHint(int type,String name,int weight,
                List fields, Query q);
  /* Create a new hint with type, name, and
     weight. fields specifies the fields for
     add, write, and read hints. q specifies
     the query for query hints. */

void setHints(List hints, Txn t);
  /* Set a table's hints. */

List getHints(Txn t);
  /* Get a table's hints. */
```

Figure 2: A table's operations on hints. Hints are created for a specific table and always accessed as a set. All three methods may throw a `StorageException`; `setHints()` and `getHints()` may also throw a `TransactionException`.

ten, or read. Hints for queries specify the query to be performed. Figure 2 illustrates the interface for managing hints.

Based on these hints, the record store can optimize the creation of indexes as well as the physical layout of a table. For example, if the workload specified by the hints is dominated by reads, the record store should create indexes for all fields searched by queries. At the same time, if the workload is more balanced between reads and writes, it should only create indexes for the most frequently searched fields. Finally, if the majority of queries search on a particular field or are sorted by a particular field, it should store the records ordered by that field.

3.4 Access Control

Choosing an appropriate access control model for the record store is difficult. Common file systems, such as those on Unix or Windows NT, typically use a form of access control list (ACL) that is stored with a file's meta-data and maintained by the file system. At the same time, an increasing number of systems base access control on the name of a resource and not on its meta-data. For example, Java security [17], distributed virtual machines [41], and domain and type enforcement [3] rely on central policy descriptions that are based on resource names. Similarly, SPKI [12] uses authorization certificates that specify the name of a resource. It has already been shown that merging file system permission models is difficult [23]. So, settling on any of these models or developing our own is not viable as we want the record store to scale across a wide

Permission	Corresponding Rights
`add`	To add to a directory or to a table.
`write`	To change a directory or a record.
`read`	To read data and meta-data.
`delete`	To delete a store object or record.
`control`	To change a store object's or record's ACL.
`layout`	To change a table's schema.
`hint`	To change a table's performance hints.

Table 1: The permissions used by the record store.

range of computing platforms.

For our store, we chose to specify a standard interface to an external access controller that implements the platform-specific access control model. The access controller interface supports both ACL-based and name-based access control and is invoked by the record store on all operations. For tables, it can provide access control at the granularity of the entire table, individual records, as well as individual fields. In order to keep the access controller interface simple, it uses seven permissions to represent the individual record store operations, as shown in Table 1.

The record store manages the ACLs associated with store objects as well as records for ACL-based access control. It stores their internal, binary representation with its own meta-data and lets applications access their external object representation. Furthermore, it uses the access controller to convert between the two representations. Newly created store objects automatically inherit a copy of their parent directory's ACL. Similarly, newly added records are protected by their table's ACL. The complete interface of the access controller is shown in Figure 3.

3.5 Summary

In our design, records are stored in tables, and tables, in turn, are organized in directories, forming a hierarchical name space similar to that of file systems. To ensure the scalability of the record store, all operations on tables affect only a singe table. To ensure its reliability, all operations are atomic and applications can use transactions to group several operations into one atomic unit. To simplify replication, all records are associated with GUIDs and accessed by GUID, one record at a time. Applications use queries to search records by other criteria. Furthermore, applications can provide hints, so that the record store can optimize table access and layout for the applications' workload. Finally, access

```
public interface AccessController {

boolean usesAcls();
  /* Return true if the access controller
     uses ACLs. */

void check(String name, Guid g, int perm,
           byte[] acl);
  /* Check that the caller has permissions
     perm for the store object with name
     and GUID g. */

void check(String path, Guid g1, Guid g2,
           List fields, int perm, byte[] acl);
  /* Check that the caller has permissions
     perm for fields of the record with
     GUID g2 in the table with name and
     GUID g1. */

Acl convert(byte[] acl);
  /* Convert the binary representation of
     acl into its object representation. */

byte[] convert(Acl acl);
  /* Convert the object representation of
     acl into its binary representation. */
}
```

Figure 3: The interface to the access controller. The `name` argument for both `check()` methods is the fully qualified name that does not contain any aliases for the corresponding store object. If the access controller uses ACLs, the record store passes the ACL protecting a store object or record to the corresponding `check()` method. Both `check()` methods throw a `SecurityException` if the check fails.

control is delegated to an external access controller, which can perform checks based on ACLs or names.

4 Prototype Implementation

The primary goal for our prototype implementation is to provide a platform for validating that our design (1) meets the needs of modern data-centric applications and (2) effectively supports application-specific replication. We therefore decided to implement our prototype using a relational database as the backing store instead of providing a native implementation. This may seem like a strange decision, given our assertion that relational databases are too complex; however, for our prototype the relational database acts simply as a reliable persis-

tent store with support for transactions. Our implementation is written in Java and uses JDBC [45] to access the underlying database. It consists of 16 classes and about 8,200 lines of well-documented code.

Our implementation maps the record store into the database as follows. It uses a separate database table to store each record store table. Additionally, it uses three database tables to store meta-data, one for the hierarchical name space, one for the field descriptors of all record store tables, and one for the hints of all record store tables.

To minimize any performance overhead caused by using the database, the implementation makes extensive use of caching. Transactions are an implicit property of the database connections used by JDBC to access a database. Our implementation thus maintains a pool of connections and maps the explicit transactions used in the record store API to the corresponding database connections. Furthermore, it uses prepared statements for all operations on record store tables and caches them for future reuse. Finally, it caches the Java objects representing store objects.

5 Experiences

In order to gain experience with our record-storage interface, we implemented several benchmark programs and measured their execution on the record store. For comparison, we also implemented and measured several of these tests using straight JDBC. Our benchmark programs are:

1. A micro-benchmark that creates a simple dictionary table mapping integer keys to string values.

2. An application implementing a portion of the TPC-W benchmark for e-commerce [43] that searches a database for all the books by a given author.

3. An application, also based on the TPC-W benchmark, that implements a user database supporting account additions, logons, and account updating after an order.

4. A simple mail server that supports the functions necessary for responding to IMAP4 requests [11]. We layered the mail server on top of a replication module that intercepts calls to the record store and copies data to a peer machine.

Writing these programs allowed us to gauge the usefulness of the record store's API, as well as to discover flaws. Overall, using the record store's interface is easier than using JDBC, mostly due to its clean design. The two most useful features turned out to be the automatic connection management, which simplifies multi-threaded programming, and explicit transaction support, which proved simpler than associating transactions with connections. Furthermore, our experiences showed that the interface is both sufficiently powerful to write a real application and simple enough to implement replication on top of it. The major drawback of the API turned out to be its verbosity: to perform operations that take a single line of SQL code requires several lines to build the corresponding record store data structures. However, the API encourages re-use of these data structures, so the complexity is centralized. The programs also demonstrated that the performance of the record store, even when layered on top of JDBC, is good enough to be used seriously.

5.1 Experimental Setup

We had three goals in evaluating our prototype implementation. The first was to make sure that our implementation did not have a major impact on performance relative to that of JDBC. The second was to demonstrate that the record store performs well for the workloads it targets, such as e-commerce or electronic mail. The third was to show that the interface can effectively support an application-specific replication mechanism. All experiments use Sun's HotSpot Server virtual machine, version 2.0 RC2, and a commercial relational database as the underlying storage layer. They were performed on commodity PCs with a 350 MHz Pentium III processor, 128 MByte of RAM, and two IDE hard disks, which are connected by a 100 Mbps switched Ethernet. We report the average of ten trials for each experiment.

5.2 Micro-Benchmarks

The first micro-benchmark tests the overhead introduced by the record store interface. In this test, a table with two fields, an integer key and a string value, is populated with data, then queried by integer key, and finally updated by GUID. The size of the string was varied between 10 bytes and 7,000 bytes, near the maximum buffer size of JDBC. The throughput results in Figure 4 show that the performance difference is never greater than 16%, which
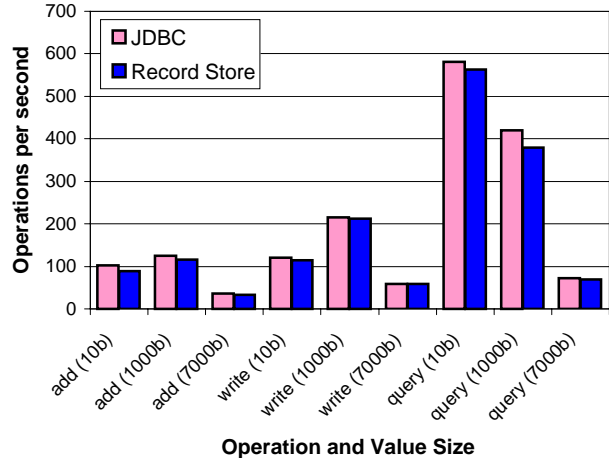

Figure 4: Performance comparison of JDBC and the record store for adding, writing, and querying 10, 1000, and 7000 byte values. Bars represent throughput in operations per second.

occurs for small data items that are stored in the buffer cache of the database. This represents the worst-case performance, because the overhead of the record store interface is amortized over reading just ten bytes of data. For larger data values, the overhead drops to less than 6%. For 1,000 byte values, the performance on writes is better than for 10 byte values due to the underlying database implementation. Nonetheless, this test demonstrates that the overhead introduced by the record store is relatively small and does not severely compromise the performance of JDBC.

The next test simulates "Search Author Web Interaction" database operations from the TPC-W e-commerce benchmark [43]. In this test, two tables, an author table and a book table, are populated with a set of 5,000 author names and 30,000 book titles that are randomly generated using a tool provided with the benchmark specification. Each book record contains a field identifying the book's author by author ID. The test program picks a random author name and then queries for books by those authors whose names start with the same letters. Because titles and authors are stored in separate tables, the implementation on top of JDBC uses a join operation on the author ID for both tables. In the record store interface, however, it must be implemented by performing a nested loop join: first the program queries for the author, to find the author ID, and then the book table is queried to find books by that author. We ran the test searching for one, ten, and fifty books.
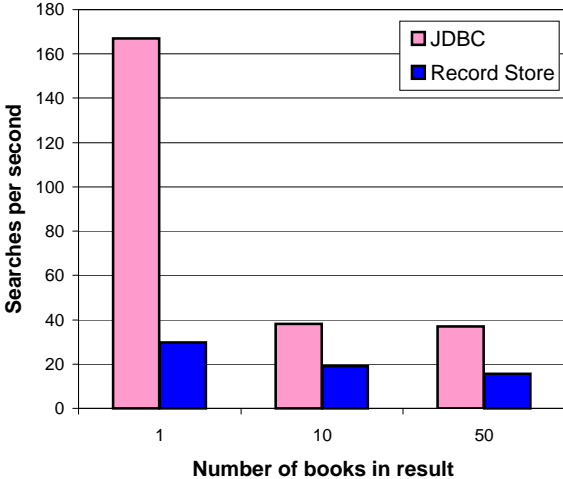
The results, shown in Figure 5, illustrate that

Figure 5: Performance comparison of JDBC and the record store for searching a database of books by their author. Bars represent throughput in operations per second when 1, 10, and 50 books are requested.
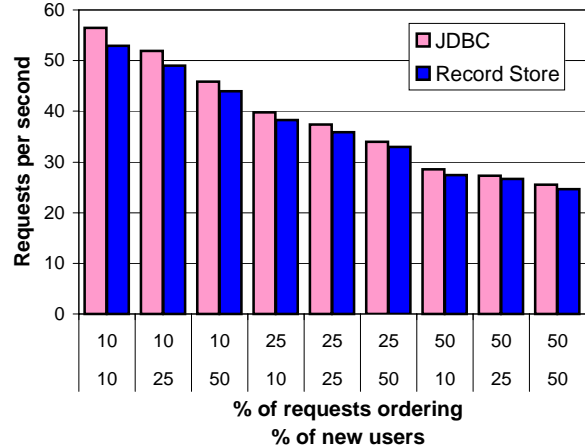

Figure 6: Performance comparison of JDBC and the record store for simulating user sessions on an e-commerce site. The portion of sessions resulting in the creation of new user accounts varies from 10% to 50%. Similarly, the portion of sessions resulting in an order varies from 10% to 50%. Bars represent throughput in requests per second.

there is a significant performance penalty for using an interface without relational operators. The results for searching for a single book are somewhat anomalous. In this case, the data set is small enough to fit into the database's buffer cache. However, due to a bug in JDBC, the record store has to iterate through all the results for each query. Consequently, the JDBC implementation is able to satisfy searches for a single book out of memory, while the record store is forced to go to disk. The results for searching for 10 and 50 books better demonstrate the penalty of not supporting relations. The throughput for 10 results is about 50% of the throughput of JDBC, while for 50 results it is 45% as fast. The difference can be accounted for by the record store implementation issuing separate queries for each author until it has a sufficient number of results. Overall, this test demonstrates that for common Web applications, such as searching a database and returning a small number of results, the record store performs well enough that the common order-of-magnitude differences in database performance do not occur [20].

The final micro-benchmark implements the "Buy Request Web Interaction" from TPC-W. This test uses a table of user accounts and a separate table of addresses. During each request session, a user either logs on to an existing account or creates a new user account and address. Following that, the user may exit without ordering, in which case the last-logon time of the account is updated, or make an order, which causes the whole account to be read and the account balance to be updated. This bench-

mark is intended to highlight mixed read/write performance. In order to explore the sensitivity of the record store to the mix of read and write operations, we varied the percentage of new users and the percentage of users making an order. The results, shown in Figure 6, illustrate that the performance of the record store causes a negligible 3% to 7% loss in throughput, which drops when performing more disk-intensive write operations. Thus, this test demonstrates that on a realistic workload with both read and write operations, the record store interface adds negligible overhead to JDBC. It indicates that a native implementation could perform equally well or better.

5.3 Replication

As a final test, we implemented a simple mail server program on top of our record store and then inserted a replication layer underneath the mail server. The replication layer implements the replication protocol used in the Porcupine cluster mail server [39], which was designed for efficient multi-master replication among a small number of peers. The protocol uses a last-writer-wins strategy for resolving update conflicts, which causes every update to rewrite the object with its new contents. This is appropriate for a mail server, since mail messages are typically only created and deleted, but not modified. The replication code is written as a layer that intercepts requests to the record store and logs updates while

otherwise passing through all requests. The replication layer uses a background thread to read objects referenced in the replication log and to copy them to the replication peers.

To test the mail server, we created a client program that generates requests to either send mail or to read mail for a particular user. The size of messages sent is chosen according to the size distribution used in [39], and the users for reading mail are chosen in a randomized round-robin fashion. Clients randomly choose to either send mail or read mail with equal probability. To avoid the overhead of parsing mail protocols in our tests, we use a simplified RPC mechanism that sends serialized Java objects over a TCP connection. For our experiments, we use two machines for running the clients and one or two machines for running the server, depending on whether data is replicated or not.

Figure 7 shows the results for one mail server without replication handling requests from one, two, and four clients as well as for two servers with replication handling requests from the same number of clients. The results demonstrate that, while replication causes a 20% performance drop for a single client, replication increases the scalability of the overall mail system for two and four clients. This increased scalability has two reasons. First, when updates arrive through replication, the user need not be authenticated and her mailbox need not be located. Second, updates arrive in batches, so there is less overhead than when processing individual requests. This experiment shows that application-specific replication can readily be implemented on top of the record store and can be used to increase application scalability when client communication is relatively expensive.

5.4 Summary

Overall, we believe that the record store's API considerably simplifies the implementation of data-centric applications. Furthermore, even with an implementation on top of a relational database, the record store introduces only a small performance overhead for many applications. Finally, it provides an effective platform for application-specific replication, because it only exposes a small number of simple operations that modify data and that must be captured by the replication layer.
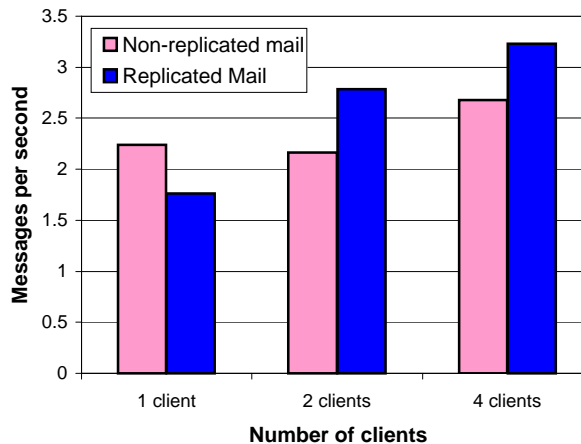

Figure 7: Performance comparison of a mail server with and without replication. Clients randomly either send mail or retrieve mail for one of 1,000 users. The non-replicated test uses a single server and multiple clients while the replicated test splits clients between two servers that replicate all data. Bars represent throughput of the mail system in messages per second.

6 Related Work

While storage systems cover a wide range of design points in the space of possible storage solutions, three aspects stand out. The first aspect is how data is structured, the second is how the storage system ensures reliability, and the third is the abstraction level provided by the storage system. Traditionally, storage systems store either unstructured or structured data. On one side, file systems and recoverable virtual memory (RVM) manage basically unstructured data. On the other side, record stores, object stores, tuple spaces, and relational databases manage either records or objects. Semi-structured data, notably XML [7], is just emerging as an alternative to both unstructured and structured data. At the same time, efficiently storing and querying semi-structured data is still a topic of active research [46].

File systems, while nearly ubiquitous, store only unstructured data, which considerably complicates concurrent updates to the same file as well as replication. Furthermore, while several efforts have explored providing failure atomicity for file systems [9, 16, 22, 31, 38] as well as the underlying disk system [8, 21], most file systems limit failure atomicity to their meta-data, if they provide it at all.

RVM [30, 40] represents a fault-tolerant alternative for managing application state by providing transactional guarantees for regions of virtual memory. However, since memory, like files, is inherently

unstructured, RVM suffers from similar problems. As data is directly mapped into an application's address space, it considerably complicates the effective sharing between applications as well as replication.

Record stores, such as IBM's VSAM [34] and Compaq's RMS [10], provide a record-oriented storage API and include support for indexes. Because these record stores expose the on-disk layout of records and lack any high-level mechanisms for ensuring atomicity, they are more suitable as the underlying storage layer for our record store than as a general storage abstraction for applications. Palm Computing's Palm OS [6] does not distinguish between main memory and persistent storage. Its record storage is limited to providing a possibly sorted list of records and thus represents an even lower level of abstraction.

Object stores, such as Thor [29], provide a persistent heap of objects. By preserving the structure of application objects, they let applications safely share data. By using transactions, they provide both concurrency control and reliability across failures. However, object stores are optimized for storing heterogeneous objects and for maintaining the relationships between them, and not for storing large collections of homogeneous records.

Tuple spaces, such as JavaSpaces [15] and T Spaces [47], are emerging as a new kind of network service. A tuple space stores objects and supports three basic operations: write (to add an object), read (to return a copy of an object that matches a template), and take (to remove and return an object that matches a template). While tuple spaces support collections of homogeneous objects and use transactions for reliability, their limited interface is not well suited for data-centric applications that frequently modify data.

Relational databases [19] are specifically designed to store large collections of records, to provide concurrency control and reliability through the use of transactions, and to support sophisticated queries to access the data. They are widely used as the underlying store for server applications. Furthermore, embedded databases are increasingly used as a storage substrate for resource-limited devices, such as personal digital assistants [36]. The level of abstraction provided by databases is much higher than that of our record store because of the support for relations, a query language [24], and replication, resulting in a system that is overly complex and requires significant management efforts.

Our record store, like other record stores and relational databases, manages record-oriented data. The store provides operations that are atomic across failures, and applications can use transactions to group several operations into an atomic unit. Compared to the other systems described, we designed our record store to provide a simple interface specifically tuned to the needs of modern networked data services. In particular, our interface is (1) simpler than that of other record stores, because it provides a higher level of abstraction and hides the on-disk layout of data, (2) simpler than that of relational databases, because it does not support many of their advanced features, and (3) cleaner than the interfaces of other systems, because it clearly separates different concerns and provides separate operations and abstractions to represent them. Finally, our record store is the only system specifically designed to support application-specific replication.

7 Conclusions

A new generation of networked data services has appeared, due in part to the success of the Internet. These applications store and retrieve relatively simple data objects, but have high demands for availability and reliability, which requires replication. Neither file systems nor databases provide a good match for these Internet applications.

In this paper, we have presented a transactional record store that better meets the requirements of modern data-centric applications. The design of our record store is based on three principles: limit global knowledge, don't hide power, and separate independent concerns. The store combines the manageability of the file system interface with select features for managing record-oriented data. Records are stored in tables, which are organized in a hierarchical name space. To simplify replication, the store exposes globally unique identifiers for individual records as well as the objects in its name space. To ensure good performance, the store uses a simple yet expressive hinting system. Finally, to provide reliability across failures, all operations are atomic and applications can use transactions to group several operations into a single atomic unit.

An implementation of our record store on top of a relational database shows negligible overhead over direct database access for workloads dominated by reads and writes and a reasonable overhead for workloads dominated by relational operations. Furthermore, the implementation demonstrates that the record store is an effective platform for implementing application-specific replication.

We are considering two future extensions to our record store. First, we wish to support references

as a basic type in addition to the existing numeric, string, and binary types. Applications can already reference specific records by using a pair of GUIDs, one for the record's table and the other for the record itself. The principle of not hiding power suggests that this type of reference should be formalized, especially since references can provide information on which data to prefetch [5]. Support for references, however, raises the question of whether to ensure their integrity. On one side, relational databases provide referential integrity between primary and foreign keys and thus help applications maintain consistency between related records. On the other side, referential integrity clearly violates the principle of limiting global knowledge. Consequently, we plan to investigate how networked data services may utilize references in order to better understand their requirements.

Second, the emergence of tuple spaces as a new kind of network service raises the question of how to effectively implement them. Tuple spaces lend themselves towards storage in a table, but supporting multiple versions and subclasses of objects makes the mapping non-trivial [4, 26]. Consequently, we plan to determine the minimal feature set necessary to implement tuple spaces directly within the record store and how to integrate tuple and record storage.

Acknowledgments

We thank Brian Bershad for his input in early discussions of our project. We also thank David Ely and Suzanne Swift for their comments on earlier versions of this paper.

References

[1] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 161–172, Tucson, Arizona, May 1997.

[2] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[3] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 66–77, Oakland, California, May 1995.

[4] P. A. Bernstein, B. Harry, P. Sanders, D. Shutt, and J. Zander. The Microsoft repository. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 3–12, Athens, Greece, Aug. 1997.

[5] P. A. Bernstein, S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 327–338, Edinburgh Scotland, Sept. 1999.

[6] C. Bey, E. Freeman, D. Mulder, and J. Ostrem. Palm OS SDK reference. Technical report, Palm Computing, Inc., Santa Clara, California, Jan. 2000.

[7] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts, Feb. 1998.

[8] C. Choa, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, Hewlett Packard, Nov. 1992.

[9] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of 1992 Winter USENIX Conference*, pages 43–60, San Francisco, California, Jan. 1992.

[10] Compaq. OpenVMS record management services. Reference Manual AA–PV6RC–TK, Compaq Computer Corporation, Houston, Texas, Jan. 1999.

[11] M. R. Crispin. Internet message access protocol—version 4. RFC 1730, Internet Engineering Task Force, Dec. 1994.

[12] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, Internet Engineering Task Force, Sept. 1999.

[13] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges: Data-centric networking for invisible computing. In *Proceedings of the Fifth ACM/IEEE International Conference on Mobile Computing and Networking*, pages 256–262, Seattle, Washington, Aug. 1999.

[14] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 78–91, Saint-Malo, France, Oct. 1997.

[15] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[16] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the*

1st USENIX Symposium on Operating Systems Design and Implementation, pages 49–60, Monterey, California, Nov. 1994.

[17] L. Gong. *Inside Java Platform Security—Architecture, API Design, and Implementation.* Addison-Wesley, June 1999.

[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[20] J. Greene. Microsoft ignites new war with Oracle at Comdex show. *Seattle Times*, page D5, 1998. 17 November 1998.

[21] R. Grimm, W. C. Hsieh, W. de Jonge, and M. F. Kaashoek. Atomic recovery units: Failure atomicity for logical disks. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*, pages 26–36, Hong Kong, May 1996.

[22] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 155–162, Austin, Texas, Nov. 1987.

[23] D. Hitz, B. Allison, A. Borr, R. Hawley, and M. Muhlestein. Merging NT and UNIX filesystem permissions. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 87–95, Seattle, Washington, Aug. 1998.

[24] ISO/IEC. Information technology—database languages—SQL. ISO/IEC Standard 9075, International Standards Organization/International Electrotechnical Commission, Geneva, Switzerland, 1999.

[25] P. J. Keleher. Decentralized replicated-object protocols. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 143–151, Atlanta, Georgia, May 1999.

[26] A. M. Keller, R. Jensen, and S. Agarwal. Persistence software: Bridging object-oriented programming and relational databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 523–528, Washington, DC, May 1993.

[27] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 19–34, Seattle, Washington, Oct. 1996.

[28] P. J. Leach and R. Salz. UUIDs and GUIDs. Internet Draft draft-leach-uuids-guids-01.txt, Internet Engineering Task Force, Feb. 1998.

[29] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In R. Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 230–257, Lisbon, Portugal, June 1999. Springer-Verlag.

[30] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 92–101, Saint-Malo, France, Oct. 1997.

[31] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the FREENIX Track, 1999 USENIX Annual Technical Conference*, pages 1–17, Monterey, California, June 1999.

[32] Microsoft. Replication for SQL Server 7.0. White paper, Microsoft Corporation, Redmond, Washington, Dec. 1998. `http://www.microsoft.com/SQL/deployadmin/replication.htm`.

[33] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, Mar. 1992.

[34] B. Musteata and R. Lesser. *VSAM Techniques: System Concepts and Programming Procedures.* QED Information Sciences, 1987.

[35] Oracle. Oracle8*i* advanced replication. Technical white paper, Oracle Corporation, Redwood Shores, California, Feb. 1999. `http://www.oracle.com/database/documents/adv_replication_twp.pdf`.

[36] S. Ortiz, Jr. Embedded databases come out of hiding. *IEEE Computer*, 33(3):16–19, Mar. 2000.

[37] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 288–301, Saint-Malo, France, Oct. 1997.

[38] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[39] Y. Saito, B. N. Bershad, and H. Levy. Manageability, availability and performance in Porcupine: A highly scalable Internet mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island Resort, South Carolina, Dec. 1999.

[40] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *Proceedings of the*

14th ACM Symposium on Operating Systems Principles, pages 146–160, Asheville, North Carolina, Dec. 1993.

[41] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 202–216, Kiawah Island Resort, South Carolina, Dec. 1999.

[42] D. C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 252–263, Saint-Malo, France, Oct. 1997.

[43] TPC. TPC benchmark W. Specification 1.0.1, Transaction Processing Performance Council, San Jose, California, Feb. 2000.

[44] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.

[45] S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC API Tutorial and Reference*. Addison-Wesley, second edition, June 1999.

[46] J. Widom. Data management for XML: Research directions. *IEEE Data Engineering Bulletin*, 22(3):44–52, Sept. 1999.

[47] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.