A Practical Scalable Distributed B-Tree

Marcos K. Aguilera*
Microsoft Research Silicon Valley
Mountain View, CA
USA

Wojciech Golab*
University of Toronto
Toronto, ON
Canada

Mehul A. Shah
HP Laboratories
Palo Alto, CA
USA

ABSTRACT

Internet applications increasingly rely on scalable data structures that must support high throughput and store huge amounts of data. These data structures can be hard to implement efficiently. Recent proposals have overcome this problem by giving up on generality and implementing specialized interfaces and functionality (e.g., Dynamo [4]). We present the design of a more general and flexible solution: a fault-tolerant and scalable distributed B-tree. In addition to the usual B-tree operations, our B-tree provides some important practical features: transactions for atomically executing several operations in one or more B-trees, online migration of B-tree nodes between servers for load-balancing, and dynamic addition and removal of servers for supporting incremental growth of the system.

Our design is conceptually simple. Rather than using complex concurrency and locking protocols, we use distributed transactions to make changes to B-tree nodes. We show how to extend the B-tree and keep additional information so that these transactions execute quickly and efficiently. Our design relies on an underlying distributed data sharing service, Sinfonia [1], which provides fault tolerance and a light-weight distributed atomic primitive. We use this primitive to commit our transactions. We implemented our B-tree and show that it performs comparably to an existing open-source B-tree and that it scales to hundreds of machines. We believe that our approach is general and can be used to implement other distributed data structures easily.

1. INTRODUCTION

Internet applications increasingly rely on scalable data structures that must support high throughput and store huge amounts of data. Examples of such data structures include Amazon's Dynamo [4] and Google's BigTable [3]. They support applications that manage customer shopping carts, analyze website traffic patterns, personalize search results, and serve photos and videos. They span a large number of machines (hundreds or thousands), and store an unprecedented amount of data (tens of Petabytes) for a huge customer base (hundreds of millions of users) that can generate high retrieve and update rates.

Besides massive scalability, three other requirements drive the design of these data structures: low cost, fault tolerance, and manageability. Low cost calls for cheap commodity hardware and precludes the use of expensive business software. Fault tolerance is important for continuous business operation. And manageability is necessary because human time is precious and unmanageable systems can lead to catastrophic human errors. These requirements are not easy to meet, and as a result recent proposals compromise on generality and opt for approaches tailored for a given use. In particular, most deployed solutions are limited to simple, hash-table-like, lookup and update interfaces with specialized semantics [4].

In this paper, we present a more general and flexible data structure, a *distributed B-tree* that is highly scalable, low cost, fault-tolerant, and manageable. We focus on a B-tree whose nodes are spread over multiple servers in a local-area network[1]. Our B-tree is a B+tree, where leaf nodes hold key-value pairs and inner nodes only hold key-pointer pairs. A B-tree supports the usual dictionary operations (Insert, Lookup, Update, Delete), as well as ordered traversal (GetNext, GetPrev). In addition, our distributed B-tree provides some practical features that are absent in previous designs (such as [9, 14]):

- *Transactional access.* An application can execute several operations on one or more B-trees, and do so atomically. Transactional access greatly simplifies the development of higher-level applications.
- *Online migration of tree nodes.* We can move tree nodes transparently from one server to another existing or newly-added server, while the B-tree continues to service requests. This feature helps in performing online management activities necessary for continuous business operation. It is useful for replacing, adding, or maintaining servers, and thus it enables smooth incremental growth and periodic technology refresh. It also allows for load balancing among servers to accommodate changing workloads or imbalances that arise as servers come and go.

1.1 Motivating use cases

Here are some concrete use cases and examples of how applications might benefit from our B-tree:

*Work developed while author was at HP Laboratories.

[1]Throughout the paper, the term *node* refers to nodes of the B-tree, unless explicitly qualified, as in "memory nodes."

VLDB '08, August 24-30, 2008, Auckland, New Zealand

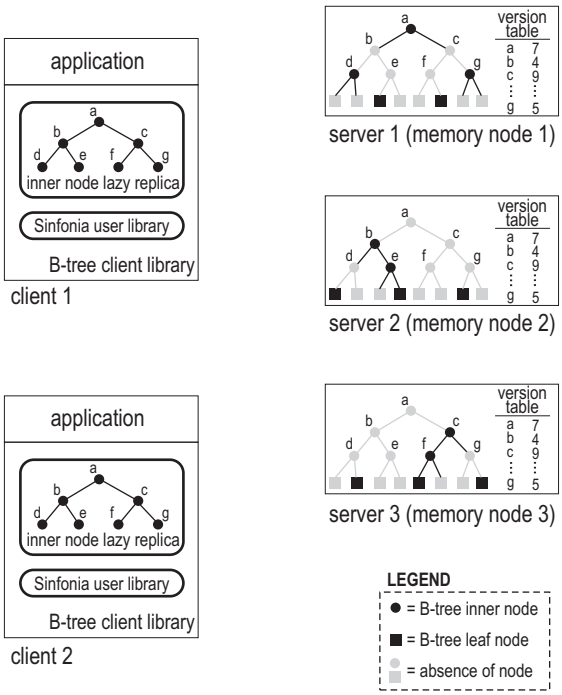

Figure 1: Our distributed B-tree. Nodes are divided among servers (grey indicates absence of node). A version table stores version numbers for inner nodes. Leaf nodes have versions, but these are not stored in the version table. Two types of replication are done for performance: (a) lazy replication of inner nodes at clients, and (b) eager replication of the version table at servers. Note that a realistic B-tree will have a much greater fan-out than shown. With a fan-out of 200, inner nodes represent $\approx 0.5\%$ of all nodes.

- *The back-end of a multi-player game.* Multi-player games have thousands of players who generate high aggregate request rates, and latency can be critical. These systems keep persistent state for the players, such as their inventory and statistics. Our B-tree could keep this persistent state: transactional access can implement atomic multi-object updates that ensure state consistency, while range queries are useful for searching. For instance, transactional access ensures that a player's item does not appear in two places simultaneously, and range queries can be used to search for items in a player's inventory.
- *Keeping metadata in a cluster file system.* In a file system, metadata refers to the attributes of files, the list of free blocks, and the contents of directories. Metadata access is often a bottleneck in cluster file systems such as Lustre or Hadoop's HDFS [21, 22]. Our B-tree could hold the metadata and alleviate this bottleneck. Transactional access is useful for implementing atomic operations, like rename, which involves atomically inserting and deleting a key-value. Ordered traversal is useful for enumerating files in a directory. And distribution provides scalable performance.
- *Secondary indices.* Many applications need to keep more than one index on data. For example, an e-auction site may support search of auction data by time, bid price, and item name, where each of these attributes has an index. We can keep each index in a separate B-tree, and use transactional access to keep indices mutually consistent.

One might be able to use database systems instead of distributed B-trees for these applications, but distributed B-trees are easier to scale, are more streamlined, have a smaller footprint, and are easier to integrate inside an application. In general, a distributed B-tree is a more basic building block than a database system. In fact, one could imagine using the former to build the latter.

1.2 Challenges and contribution

Our B-tree is implemented on top of Sinfonia, a low-level fault-tolerant distributed data sharing service. As shown in Figure 1, our B-tree comprises two main components: a B-tree client library that is linked in with the application, and a set of servers that store the B-tree nodes. Sinfonia transparently makes servers fault-tolerant.

The main difficulty in building a scalable distributed B-tree is to perform consistent concurrent updates to its nodes while allowing high concurrency. Unlike previous schemes that use subtle (and error-prone) concurrency and locking protocols [9, 14], we take a simpler approach. We update B-tree nodes spread across the servers using distributed transactions. For example, an Insert operation may have to split a B-tree node, which requires modifying the node (stored on one server) and its parent (stored possibly on a different server); clients use transactions to perform such modifications atomically, without having to worry about concurrency or locking protocols. Sinfonia provides a light-weight distributed atomic primitive, called a *minitransaction*, which we use to implement our transactions (see Section 2).

A key challenge we address is how to execute such transactions efficiently. A poor design can incur many network round-trips or limit concurrent access. Our solution relies on a combination of three techniques. (1) We combine optimistic concurrency control and minitransactions to implement our transactions. (2) Our transactions use eagerly replicated version numbers associated with each B-tree node to check if the node has been updated. We replicate these version numbers across all servers and keep them consistent. (3) We lazily replicate B-tree inner nodes at clients, so that clients can speculatively navigate the inner B-tree without incurring network delays.

With these techniques, a client executes B-tree operations in one or two network round-trips most of the time, and no server is a performance bottleneck. We have implemented our scheme and evaluated it using experiments. The B-tree performs well compared to an open-source B-tree implementation for small configurations. Moreover, we show that it can scale almost linearly to hundreds of machines for read and update workloads.

This paper is organized as follows. We describe the basic approach in Section 2. In Section 3, we explain the assumptions upon which we rely. We then explain the features of our B-tree in Section 4. In Section 5, we describe the transactions we use and techniques to make them fast. The B-tree algorithm is presented in Section 6, followed by its experimental evaluation in Section 7. We discuss some extensions in Section 8. Related work is explained in Section 9. Section 10 concludes the paper.

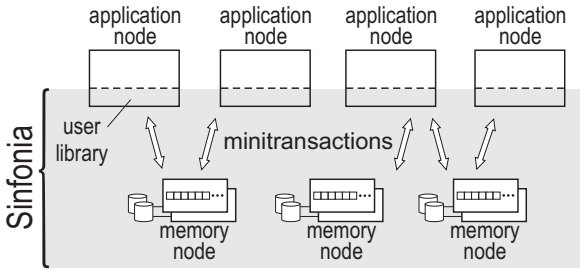

Figure 2: Sinfonia service, on top of which we build our distributed B-tree.

2. BASIC APPROACH

In this Section, we give an overview of our B-tree design. We give further technical details in later sections. Our B-tree is built on top of the Sinfonia service, and so we first describe Sinfonia and its features in Section 2.1. Then, in Section 2.2, we outline our design and justify the decisions we made.

2.1 Sinfonia overview

Sinfonia is a data sharing service that, like our B-tree, is composed of a client library and a set of servers called *memory nodes* (see Figure 2). Each memory node exports a linear address space without any structure imposed by Sinfonia. Sinfonia ensures the memory nodes are fault-tolerant, offering several levels of protection. Sinfonia also offers a powerful primitive, a *minitransaction*, that can perform conditional atomic updates to many locations at many servers. A minitransaction is a generalization of an atomic compare-and-swap operation. While the compare-and-swap operation performs one comparison and one conditional update on one address, a minitransaction can perform many comparisons and updates, where each update is conditioned on all comparisons (i.e., all updates occur atomically only if all comparisons succeed). Comparisons and updates can be on different memory locations and memory nodes. In addition, minitransactions can read data from memory nodes and return it to the application.

More precisely, a minitransaction comprises a set of compare items, a set of read items, and a set of write items. Each item specifies an address range on a memory node; compare items and write items also have data. All items must be specified *before* a minitransaction starts executing. When executed, a minitransaction compares the locations specified by the compare items against the specified data. If all comparisons succeed, the minitransaction returns the contents of the locations specified by the read items and updates the locations specified by the write items with the specified data.

Sinfonia uses a variant of two-phase commit to execute and commit minitransactions in two phases. Memory nodes lock the minitransaction items only for the duration of the two-phase commit. Sinfonia immediately aborts the minitransaction if it cannot acquire the locks, and the client must retry. Thus, unlike general transactions, a minitransaction is short-lived, lasting one or two network round-trips. More details on minitransactions and commit protocols are in the Sinfonia paper [1].

Minitransactions are not optimistic, but they can be used to implement general optimistic transactions, which we use for our B-tree operations. Briefly, each transaction (not minitransaction) maintains a read and write set of objects that it touches. Each object has a version number that is incremented on updates. We commit a transaction using a minitransaction to (1) verify that (version numbers of) objects in the read set are unchanged and, (2) if so, update the objects in the write set. In other words, a minitransaction performs the final commit of optimistic transactions.

It is worth noting that, even though we built optimistic transactions using Sinfonia, we could have implemented them from scratch. Thus, our B-tree design is not limited to running on top of Sinfonia. However, the performance and scalability numbers reported here depend on the lightweight minitransactions of Sinfonia.

2.2 Design outline

Figure 1 shows how our B-tree is distributed across servers (which are Sinfonia memory nodes). Each B-tree node is stored on a single server, though clients may keep replicas of these nodes (possibly old versions thereof). The true state of the B-tree nodes is on the servers and the clients store no permanent state. Nodes are partitioned across servers according to some data placement policy. For example, larger servers may get more nodes than others. Our current implementation allocates nodes randomly among servers.

Our B-tree operations are implemented as natural extensions of centralized B-tree algorithms wrapped in optimistic transactions. As a client traverses the tree, it retrieves nodes from the servers as needed, and adds those nodes to the transaction's read set. If the client wants to change a node, say due to a key-value insertion or a node split, the client locally buffers the change and adds the changed node to the transaction's write set. To commit a transaction, the client executes a Sinfonia minitransaction, which (a) validates that the nodes in the read set are unchanged, by checking that their version numbers match what is stored at the servers, and (b) if so, atomically performs the updates in the write set. As we explain, a consequence of wrapping B-tree operations in transactions is that we can easily provide features such as online node migration and multi-operation transactions.

Since clients use optimistic concurrency control, they do not lock the nodes during the transaction, thereby allowing high concurrency. Instead, nodes are only locked during commit— and this is done by Sinfonia as part of minitransaction execution—which is short-lived. If a minitransaction aborts, a client retries its operation. Optimistic concurrency control works well because, unless the B-tree is growing or shrinking dramatically, there is typically little update contention on B-tree nodes.

Using optimistic concurrency control alone, however, is not enough. A transaction frequently needs to check the version number of the root node and other upper-level nodes, because tree traversals will frequently involve these nodes. This can create a performance bottleneck at the servers holding these nodes. To avoid such hot-spots, we replicate the node version numbers across all servers, so that they can be validated at any server (see Figure 1). Only version numbers are replicated, not entire tree nodes, and only for inner nodes, not leaf nodes. We keep the version number replicas consistent, by updating the version number at all servers when an inner node changes, and this is done as part of the transaction. Because inner nodes change infre-

quently when the tree fanout is large, the resulting network traffic is manageable.

Finally, to further reduce network traffic, clients replicate the B-tree inner nodes as they discover them (see Figure 1). With large B-tree fan-out (say, 200 or more is typical), inner nodes comprise a small fraction of the data in a B-tree, and so replicating inner nodes takes relatively little space. Given the increasingly larger memory sizes today, it is not infeasible to replicate the entire B-tree upper-levels of a moderate sized tree on a single machine. Thus, a typical lookup operation will only need one network round trip to the server to fetch the leaf (and simultaneously validate the version numbers of nodes on the path from root to leaf). Node replicas at clients are updated lazily: clients fetch new versions only after they attempt to use a stale version. This avoids the problem of updating a large number of clients in a short period of time when inner nodes change.

As we show, with these techniques, a client executes B-tree operations in one or two network round-trips most of the time, and no server is a performance bottleneck. Note that it is the combination of *all* three techniques that provides efficiency. For example, without lazy replication of tree nodes, clients require multiple network round-trips to just traverse the B-tree. Without optimistic concurrency control, clients require additional network round-trips to lock nodes. Without eager replication of version numbers, the server holding the root node becomes a performance bottleneck.

3. ASSUMPTIONS

We now briefly explain assumptions that we made on the environment and workload.

3.1 Environment

Our B-tree is designed to operate within a data center environment. A data center is a site with many reasonably well-connected machines. It can have from tens to thousands of machines running tens to hundreds of applications. We assume that processes communicate through links that usually have high bandwidth, low latencies, and small latency variations. Links are fairly stable and made reliable via TCP. Network partitions are rare, and while they occur it is acceptable to stall the system, because most likely the data center is unusable anyways.

Machines are subject to crash failures. Servers have stable storage to deal with crashes. A server that crashes subsequently recovers with its stable storage intact. Clients may not have stable storage and may not recover from crashes. (For this reason, no permanent B-tree state should be stored on clients.) Sinfonia, which provides fault tolerance for our B-tree, operates in this environment and transparently handles crash failures of clients and servers.

3.2 Workload

We designed our B-tree to handle read and update workloads that do not rapidly grow or shrink the tree. This assumption is common in database systems and reasonable in our setting. For example, an application that maintains user profiles need not worry about user populations growing on the order of 1000s per second. For initializing the B-tree, one can apply offline B-tree bulk loading techniques (e.g., [19]).

4. DISTRIBUTED B-TREE FEATURES

We provide an overview of B-trees, define distributed B-trees, and explain some additional features that our distributed B-trees have, such as transactional access and migration.

4.1 B-tree overview

We provide a brief overview of B-trees; details can be found in a data structures textbook. A B-tree stores a set of key-value pairs (k, v). A B-tree supports the standard dictionary operations (Insert, Update, Lookup, Delete) and enumeration operations (GetNext, GetPrev), described in Figure 3. Note that in some interfaces, Update and Insert are combined into a single Insert operation with various flags; we could have implemented this easily.

Operation	Description
Lookup(k)	return v s.t. $(k, v) \in B$, or error if none
Update(k, v)	if $(k, *) \in B$ then replace it with (k, v) else error
Insert(k, v)	add (k, v) to B if no $(k, *) \in B$, else Update(k, v)
Delete(k)	delete (k, v) from B for v s.t. $(k, v) \in B$, or error if none
GetNext(k)	return smallest $k' > k$ s.t. $(k', *) \in B$, or error if none
GetPrev(k)	return largest $k' < k$ s.t. $(k', *) \in B$, or error if none

Figure 3: Operations on a B-tree B.

A B-tree is internally organized as a balanced tree. We focus on the B+tree, a B-tree variant where all key-value pairs are stored at leaf nodes (Figure 4); inner nodes store only key-pointer pairs. Each tree level stores keys in increasing order. To lookup a key, we start at the root and follow the appropriate pointers to find the proper leaf node. Updating the value of a key entails looking up the key and, if found, changing the value associated with it at the leaf. To insert a pair (k, v), we lookup the leaf node where k would be, and place (k, v) there if there is an empty slot. Otherwise, we *split* the leaf node into two nodes and modify the parent appropriately as illustrated in Figure 5. Modifying the parent might require splitting it as well, recursively (not shown). Deleting a key entails doing the inverse operation, merging nodes when a node is less than half full. The enumeration operations (GetNext and GetPrev) are almost identical to Lookup.

4.2 Distributed B-tree

A distributed B-tree has its nodes spread over multiple servers. For flexibility, tree nodes can be placed on any server according to some arbitrary user-supplied function chooseServer() that is called when a new node is allocated. For example, if a new server is added, chooseServer() can return the new server until it is as full as others.

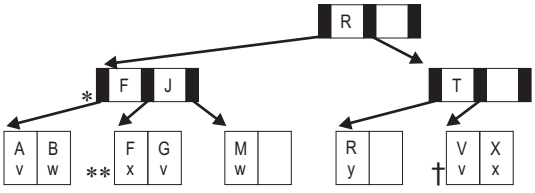

Figure 4: B+tree: leafs store key-value pairs and inner nodes store keys and pointers. Keys and values are denoted in upper and lower case, respectively. To lookup key G, we start at the root, follow the left pointer as $G<R$, arrive at node $*$, follow the middle pointer as $F<G<J$, and arrive at node $**$ where we find G.

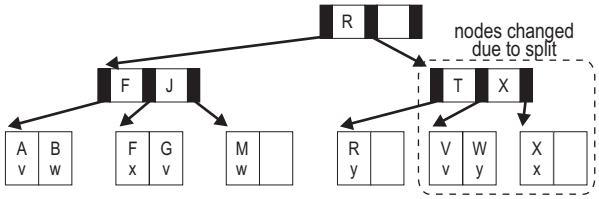

Figure 5: Splitting nodes: inserting (W, y) causes a split of node † in the tree from Figure 4.

4.3 Transactional access

Since our B-tree supports concurrent operations by multiple clients, and since each such operation may touch multiple tree nodes, we use transactions to prevent such operations from interfering with one another. Moreover, transactional access allows clients to perform multiple B-tree operations atomically, even on multiple B-trees.

Transactions are serializable: they appear to execute in isolation, without intermingling operations of different transactions. We expect transactions to be short-lived rather than long-lived (i.e., execute within milliseconds, not hours). Figure 6 below shows a sample transaction.

```
txn ← BeginTx();
v ← Lookup(txn, k);
Update(txn, k, v + 1);
success ← Commit(txn);
EndTx(txn);
```

Figure 6: Transaction to atomically increment the value associated with key k.

4.4 Migration

On-line migration, or simply migration, refers to the ability to move B-tree nodes from one server to another, and do so transparently while allowing other B-tree operations to execute. The basic migration mechanism is offered through an operation Migrate(x, s), which migrates B-tree node x to server s, where x is a node reference represented as a pair $\langle server, offset \rangle$. This is a low-level function since it refers to actual nodes of the B-tree, and it is unlikely to be invoked directly by users. Rather, users perform migration through higher-level *migration tasks*, which determine what to migrate and where to migrate to, based on a specific goal. These migration tasks, shown in Figure 7, are tied to common management operations: remove a server from the system (Migrate-away), add a new server (Populate), replace a server (Move), balance storage (Even-out-storage), and balance load (Even-out-load).

Migrate task	Description
Migrate-away	migrate all nodes at server x to other servers.
Populate	migrate some nodes from other servers to server x.
Move	migrate all nodes from server x to server y.
Even-out-storage	migrate some nodes from more full to less full servers.
Even-out-load	migrate some nodes from more busy to less busy servers.

Figure 7: Migration tasks on a B-tree.

5. TRANSACTIONS

In our B-tree implementation, clients use a type of distributed, optimistic transaction to atomically manipulate tree nodes and other objects. We first give an overview of our transaction interface and then describe the optimizations we make.

5.1 Transaction interface

A transaction manipulates *objects* stored at servers. Each object is a fixed-length data structure, such as a B-tree node or a bit vector. Objects are either static or allocated from a static pool at each server, where each object in the pool has a flag indicating whether it is free. We chose to implement allocation and deallocation ourselves, so that these can be transactional. Objects include an *ID*, which uniquely identifies the object and the server where it is stored, and possibly a *version number*, which is incremented whenever the object changes. Figure 8 below lists the types of objects used in our implementation.

Object	Description
tree node	stores keys and pointers or values
bit vector	used by node allocator
metadata	tracks server list and root node ID

Figure 8: Transactional objects.

Transactions have a *read set* and a *write set*, with their usual semantics: they are empty when a transaction starts, and as the transaction reads and writes objects, these objects are added to the read and write sets, respectively. Our transactions use optimistic concurrency control [10]: objects are not locked during transaction execution, only during commit, which is implemented with Sinfonia's minitransactions.

The interface to transactions is shown below in Figure 9, where n denotes an object ID and val is an object value. All transactional operations other than BeginTx() take as an argument a *transaction handle*, which is created by calling BeginTx(). Roughly speaking, Read and Write operations add objects to the read and write sets, while Commit attempts to commit the transaction. Abort marks the transaction as prematurely aborted so that subsequent calls to Commit fail. IsAborted checks whether the transaction has aborted.

Operation	Description
BeginTx()	clear read and write sets, return transaction handle
Read(txn, n)	read object n locally or from server and add (n, val) to read set
Write(txn, n, val)	add (n, val) to write set
Commit(txn)	try to commit transaction
Abort(txn)	abort transaction
IsAborted(txn)	check if transaction has aborted
EndTx(txn)	garbage collect transaction structures

Figure 9: Interface to transactions.

5.2 Lazy replication of objects at clients

The transactions we use tend to have a small write set with objects at a single server, and a large read set with objects at many servers. For example, a B-tree Insert typically writes only to a leaf node and reads many nodes from root to leaf. To optimize for this type of transaction, clients store local replicas of certain frequently accessed objects. These replicas speed up transaction execution because a client can read replicated objects locally. Not every object is replicated: only inner B-tree nodes and metadata objects (which record information such as the list of servers in the system), so that the size of replicated data is small.

Replicas are updated lazily: when a transaction aborts because it used out-of-date objects, the client discards any local copies of such objects, and fetches fresh copies the next time the objects are accessed. The use of lazy replicas may cause a transaction to abort even without any overlap with other transactions, in case a replica is out-of-date when the transaction begins. We did not find this to be a problem for B-trees since the objects we replicate lazily (e.g., inner nodes of the B-tree) change relatively rarely.

5.3 Eager replication of version numbers at servers

B-tree operations tend to read upper-level B-tree nodes frequently. For example, every tree traversal begins at the root node. As explained above, these nodes are replicated at clients, but when a transaction commits, they still need to be validated against servers. The servers holding these popular nodes can become a performance bottleneck. To prevent this problem, we replicate the version numbers of all inner nodes of the B-tree at all servers, so that a transaction can validate them at any server. This replication scheme is illustrated in Figure 1. Similarly, we also replicate the version numbers of metadata objects. The version replicas are kept consistent by updating them as part of the transaction that changes the associated objects.

Another benefit of replicated version numbers is to optimize transactions that read many inner nodes but only modify one leaf node, such as when one inserts a key-value at a non-full leaf, which is a common operation. Such transactions typically need to contact only one server, namely the one storing the leaf node, since this server can validate all inner nodes. In such cases, it is possible to commit a transaction in only one phase instead of two (as explained below). Thus, by replicating version numbers and keeping the replicas consistent, most B-tree operations commit in only one network round-trip.

5.4 Committing transactions

We use Sinfonia's minitransactions to commit our transactions, taking advantage of minitransaction compare items to validate all objects read during the transaction. As described above, when possible, compare items reference version numbers that are all in the same server to minimize the number of servers involved the transaction. If only one server is involved, the transaction can be committed faster (as explained below).

We use minitransaction write items to update objects in the transaction's write set. If the updated objects are inner nodes or metadata objects, then all replicas of their version numbers are also updated, using additional write items in the minitransaction.

Sinfonia uses a variant of two phase commit to atomically execute the minitransaction. Roughly speaking, the first phase compares and locks the minitransaction items and the second phase performs the updates if the first phase is successful. An important optimization is when the minitransaction involves only a single server. In this case, Sinfonia uses one-phase rather than two-phase commit, saving a network round-trip [1].

5.5 Fault tolerance

Sinfonia handles failures that are typical of a single datacenter environment (see Section 3.1). Sinfonia offers multiple levels of fault tolerance for the servers. It offers logging, primary-backup replication, or both, depending upon the performance and protection needs of the application. Primary-backup replication provides fast fail-over with little downtime but at the cost of twice the hardware and bandwidth. Logging uses fewer additional resources but comes at the cost of higher runtime overhead and slower recovery. The tradeoffs among the various Sinfonia fault tolerance modes are quantified in the Sinfonia paper [1].

Sinfonia transparently handles the failure of clients and servers for us. When clients fail, they may not recover, and so we store permanent state only on the servers. Sinfonia cleans up any outstanding minitransactions left by a client that fails.

By leveraging Sinfonia, we are able to cleanly separate concerns of fault tolerance and concurrency control. These two concerns are intimately intertwined in most database implementations, which increases system complexity. In contrast, we do not have to worry about recovery mechanisms when a failure occurs, or about locking protocols to prevent the corruption of data structures and ensure consistency. The result is a design that is simpler and orthogonal to the fault tolerance that is provided. For instance, Sinfonia can be configured to replicate servers and/or log updates to disk synchronously, and these choices can be changed by just flipping a switch, without changing the design of our distributed B-tree in any way.

5.6 Other optimizations

If a transaction reads an object for which it does not have a local replica, the client must request the object from the server storing it. When this happens, we piggyback a request to check the version numbers of other objects currently in the read set. If some object is stale, this check enables clients to abort doomed transactions early. Moreover, a read-only transaction (e.g., which is performed by a Lookup operation) whose last read was validated as just described, can be subsequently committed without any further communication with servers.

6. DETAILS OF DISTRIBUTED B-TREE ALGORITHM

We now explain how we use transactions to implement our distributed B-tree.

6.1 Dictionary and enumeration operations

The dictionary and enumeration operations of a B-tree (Lookup, Insert, etc) all have a similar structure: they initially traverse the B-tree to get to the leaf node where the given key should be. Then, if the operation involves changing data (i.e., Insert, Delete), one or more nodes close to the leaf node are changed. We use a transaction to make these modifications.

Figure 10 shows the detailed fields of a B-tree node, which we use in our algorithm. Each node has a flag indicating if the node is a leaf, a height within the tree, and a number of key-value pairs (for leaf nodes) or key-pointer pairs (for inner nodes).

Figure 11 shows the code for Insert in detail. Other operations are analogous. The overall algorithm is very similar to that of a centralized B-tree; the symbol $\star$ in the pseudo-code indicates lines where there are differences.

Field	Description
isLeaf	Boolean indicating node type
height	distance of node from leaf level
numKeys	number of keys stored
keys[1..numKeys]	sorted array of keys
values[1..numKeys]	values for keys in leaf node
children[0..numKeys]	child pointers in inner node

Figure 10: Fields of a B-tree node.

We start by explaining insertion in a centralized B-tree (readers familiar with the algorithm can just skim this paragraph). To illustrate the steps involved, we reference the corresponding lines in Figure 11. Execution begins in function Insert(k, v), where (k, v) is the key-value pair to insert (ignore for now the argument txn in Figure 11). Insert invokes a recursive function InsertHelper passing the root node ID as parameter (line 3). This function actually does insertion; after reading the specified node (line 16), it examines the keys keys[1..numKeys] stored in the node (which, in the first invocation, is the root node) to determine which child corresponds to k (line 20). This is done by calling Search to find the largest index i such that keys$[i] \leq k$. After i is determined, there are two cases:

- Case 1 (lines 21–29). *The node is a leaf node.* Then index i is the place where (k, v) should be inserted. If keys$[i]$ already has v, then we simply store v in values$[i]$, the value associated with keys$[i]$ (lines 23–24). Otherwise, we add (k, v) to the node (lines 27–28). This may cause the node to have more key-value pairs than its maximum. This case is handled after we return from InsertHelper, by splitting the node into two nodes and updating the parent node (lines 5–10 when the parent is the root node, and lines 34–39 otherwise).

- Case 2 (lines 30–40). *The node is an inner node.* Then we follow the pointer associated with index i, arriving at a child node, and recursively call InsertHelper on this child node (lines 31–32). After returning from InsertHelper, the child node may have grown by one beyond its maximum size. In this case, we split the child into two children and update the parent to reflect the split (lines 34–39). Updating the parent requires inserting a new key-pointer pair at the parent. We then return from InsertHelper. The caller will then check if the parent needs to be split, and so on.

The above description is for a centralized algorithm. The distributed B-tree algorithm requires some simple changes (the changed lines are indicated by ⋆ marks in Figure 11):

- Accesses to the B-tree occur within a transaction.
- We use the transactional Read and Write to read and write B-tree nodes and add them to the transaction.
- We use a special getRoot helper function to find the B-tree root (by way of the metadata object).
- We use our Alloc and Free functions to allocate and free B-tree nodes transactionally.
- We use $\perp$ as a special return value to indicate that the current transaction has aborted, and we check the status of the transaction before using any return value.
- We perform various safety checks to prevent the client from crashing or hanging when its local replicas of objects are inconsistent with each other.

We now explain these changes in more detail.

```
Function Insert(txn, key, value)                                   ⋆
    Input: txn – transaction handle                                ⋆
    Input: key, value – key/value pair to insert
    Output: true iff key was not already in the tree
 1  rootNum ← getRoot(txn)                                         ⋆
 2  if IsAborted(txn) then return ⊥                                ⋆
 3  (ret, root) ← InsertHelper(txn, rootNum, key, value, ∞)        ⋆
 4  if IsAborted(txn) then return ⊥                                ⋆
 5  if root has too many keys then
 6      split root into children child and child′, and new root
        root
 7      c ← Alloc(txn); c′ ← Alloc(txn)                            ⋆
 8      if IsAborted(txn) then return ⊥                            ⋆
 9      Write(txn, rootNum, root)                                  ⋆
10      Write(txn, c, child); Write(txn, c′, child′)               ⋆
11  return ret

Function Search(node, key)
    Input: node – node to be searched
    Input: key – search key
12  if node.numKeys = 0 then
13      return 0
14  else
15      return index of the largest key in node.keys[1..numKeys]
        that does not exceed key, or else 0 if no such key

Function InsertHelper(txn, n, key, value, h)                       ⋆
    Input: txn – transaction handle                                ⋆
    Input: n – node ID
    Input: key, value – key/value pair to insert
    Input: h – height of previous node visited
    Output: tuple (ret, node) where ret = true iff key was not
            already in the tree, and node is node n
16  node ← Read(txn, n)                                            ⋆
17  if IsAborted(txn) then return (⊥, ⊥)                           ⋆
18  if node.height ≥ h then                                        ⋆
19      Abort(txn); return (⊥, ⊥)                                  ⋆
20  i ← Search(node, key)
21  if node.isLeaf then
22      if i ≠ 0 ∧ node.keys[i] = key then
23          node.values[i] ← value
24          Write(txn, n, node)                                    ⋆
25          return (false, node)
26      else
27          insert key and value into node
28          Write(txn, n, node)                                    ⋆
29          return (true, node)
30  else
31      c ← node.children[i]
32      (ret, child) ← InsertHelper(txn, c, key, value,
        node.height)                                               ⋆
33      if IsAborted(txn) then return (⊥, ⊥)                       ⋆
34      if child has too many keys then
35          split child into child and child′, update node as
            needed
36          c′ ← Alloc(txn)                                        ⋆
37          if IsAborted(txn) then return (⊥, ⊥)                   ⋆
38          Write(txn, c, child); Write(txn, c′, child′)           ⋆
39          Write(txn, n, node)                                    ⋆
40      return (ret, node)
```

Figure 11: Algorithm for Insert.

Transaction context. B-tree operations are executed in the context of a transaction, indicated by a transaction handle txn. Clients obtain txn by calling BeginTx, as explained in Section 5, and pass this handle when they call B-tree operations. After executing B-tree operations, clients can commit the underlying transactions.

Reading and writing B-tree nodes: Read **and** Write (lines 9, 10, 16, 24, 28, 38, 39). Reading and writing B-tree nodes simply entails calling the Read and Write functions explained in Section 5. These functions perform the read and buffer the write, respectively, updating the transaction's read and write sets.

Finding the root: getRoot (line 1). The B-tree root might change as it undergoes migration or splits. Since the root is where all tree traversals start, we need an efficient way to locate it. To do so, we keep some metadata about the B-tree in a special metadata object, which includes the ID of the root node and a list of all current servers. We include the metadata object, which is replicated at all servers for efficiency, in a transaction's read set to ensure the root is valid.

Node allocation and deallocation: Alloc **and** Free (lines 7, 36). We need to allocate B-tree nodes transactionally to avoid races (double allocations) and memory leaks (when a transaction aborts). To do so, we use a simple allocation scheme: at each server, there is a static pool of nodes and a bit vector indicating whether a node in the pool is free. Clients keep a lazy replica of each server's bit vector. To allocate a new node, the client first decides on a server to host the node, by calling chooseServer() (our implementation simply returns a random server, but more elaborate schemes are possible). Next, the client selects a free entry in the bit vector of that server, and adds that entry (not the node itself) to the transaction's read set, marks it as used, and finally adds the updated entry to the write set.

Node deallocation is analogous to node allocation, except that a bit vector entry is marked as free instead of used. Note that Alloc and Free must increment the version number of the node being allocated or freed, which is necessary to invalidate lazy replicas stored at clients.

Safety checks (line 18). Clients may have inconsistent data in their lazy replicas of tree nodes, and so, to avoid crashing or hanging, clients must detect the following conditions:

- object read has unexpected type
- infinite cycles while traversing B-tree

Failure to catch the first condition may result in a variety of anomalies, including invalid memory access, divide by zero, and infinite (or extremely long) loops. The correct way to check the type of an object is implementation-specific, and we do not show this in Figure 11. But roughly speaking, each object has a type identifier and when the code accesses an object, it checks if the type identifier matches what it expects. If it does not, the transaction is aborted (the caller can then restart the transaction).

We detect the second condition by recording the distance of a node from the leaf level in the height field, and ensuring that the value of this field decreases monotonically during a traversal of the tree (line 18). If it does not, the transaction is aborted (line 19). (In our code, note that leaf nodes have a height of 0.)

In addition to the consistency checks, we validate the transaction's read set whenever a client reads a node from a server. This does not introduce additional network roundtrips because we can piggyback comparison items in the same minitransaction that is issued by Read. If this validation fails (as indicated by Read returning $\perp$), the transaction can abort early.

6.2 Initialization

When a client begins executing, it uses a directory service to find one of the B-tree servers and contacts this server to read the metadata object. This object, in turn, specifies the ID of the root node and the list of servers.

6.3 Transactional access

With our approach, it is straightforward to combine multiple B-tree operations in the same transaction: the code for B-tree operations (e.g., Insert or Delete) does not actually begin or commit the transaction, it just accumulates changes to a transaction's read and write set. Thus, a client can execute multiple operations by passing in the same transaction handle, and then committing the transaction afterwards.

6.4 Migration of nodes

To migrate a node x to server s (operation Migrate(x, s)), we need to destroy node object x and create a new node object at server s. To do so, a client executes a transaction that reads node x, deallocates x using Free, allocates a new node x' at server s using Alloc, writes node x', and replaces the pointer to x with a pointer to x' in x's parent. If x is the root node, then the transaction also updates the metadata object with the new root.

We use Migrate() to implement the higher-level migration tasks, such as the Migrate-away task that migrates all nodes at a specific server s_0 to other servers. This choice is the simplest and most modular, but not necessarily the most efficient: for example, it might be faster to migrate several nodes in the same transaction. However, we were not trying to optimize the rate of migration, since migration happens online and can be done in the background provided that the migration rate is not awfully slow.

For example, for the Migrate-away task, we perform a depth-first traversal of the B-tree nodes and, whenever we find a node at server s_0 (from which we are migrating away), we invoke Migrate to move it to another random server. These actions are executed by a special client machine called the *migration client*, which is different from any of the servers and clients.

There are two caveats in implementing Migrate-away. First, the goal is to have an empty server s_0 at the end, so we must ensure that clients do not place new B-tree nodes at server s_0. Thus, we need a mechanism to notify clients about s_0. To do so, each server records the set of *active* servers — those where allocations are allowed — and clients validate node allocations against this set. This validation is done as part of the transaction that allocates the node: if the validation fails, the transaction aborts.

Second, as the migration client traverses the B-tree, it is possible that B-tree nodes vanish (another client deletes a key that causes a node merge), including the node that the migration client is currently visiting. It is not a problem if the migration client tries to migrate a vanished node (the

transaction of the migration operation will abort since the node changed). The problem is that the migration client will not know how to continue its depth-first traversal. One possibility is to restart from the root, but this is much too inefficient. Our solution is that the migration client remembers the largest key k at the last leaf node that it visited. If the node it is visiting vanishes, the migration client finds k and continues the traversal from there. This method is not always foolproof: it is theoretically possible that k gets removed, other keys get added, and k gets placed at a completely different place in the B-tree. As a result, the migration client may need additional traversals of the B-tree if, when it completes its current traversal, nodes are still allocated at server s_0. This situation should occur infrequently in practice.

The other migration tasks are implemented similarly to Migrate-away, via a depth-first traversal, with differences only with respect to how the source and destination servers are selected, and when the migration terminates. For example, Move and Populate always picks the same target server to which to migrate a node (instead of the random server chosen in Migrate-away). In addition, Populate picks the source servers in a round-robin fashion and migration terminates when the destination server is satisfactorily full. Even-out-storage picks source and target servers based on their storage utilization, while Even-out-load picks servers based on their load.

7. EVALUATION

We now evaluate the performance and scalability of our B-tree and of its migration capabilities. Our testing infrastructure had 188 machines on 6 racks connected by Gigabit Ethernet switches. Intra-rack bisection bandwidth was ≈14 Gbps, while inter-rack bisection bandwidth was ≈6.5 Gbps. Each machine had two 2.8GHz Intel Xeon CPUs, 4GB of main memory, and two 10000RPM SCSI disks with 36GB each. Machines ran Red Hat Enterprise Linux AS 4 with kernel version 2.6.9. Sinfonia was configured to log mini-transactions to disk synchronously.

7.1 Workload

We considered a B-tree with 10-byte keys and 8-byte values. Each B-tree node had 4 KB and leaf nodes held 220 key-value pairs while inner nodes held 180 key-pointer pairs, where each pointer had 12 bytes: 4 bytes specify a server and 8 bytes specify an offset within the server.

In all experiments, workloads were generated at client machines separated from B-tree server machines. Each server offered 32MB of available memory for the B-tree. All client machines accessed the same B-tree. Clients kept a local lazy replica of all inner nodes of the B-tree but none of the leaf nodes. Each client machine had 4 parallel threads, each with one outstanding request issued as soon as the previous completed. We considered four workloads:

- *Insert.* New keys are generated uniformly at random from a space of 10^9 elements and inserted into the B-tree.
- *Lookup.* The previously inserted keys are looked up, in the same random order in which they were inserted.
- *Update.* The values for previously inserted keys are modified.

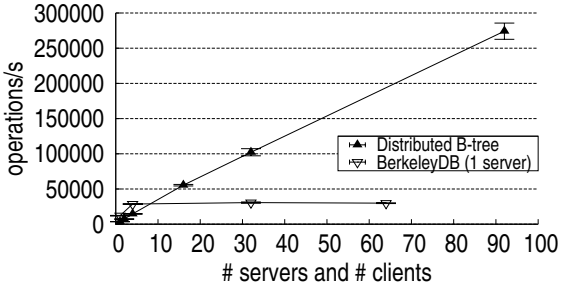

Figure 12: Aggregate throughput, lookup workload.

- *Mixed.* This is a mixture of 60% lookups and 40% updates randomly interspersed, for randomly chosen keys that were previously inserted.

Before the insert workload, the B-tree was pre-populated with 40,000 elements rather than starting with an empty B-tree.

7.2 Scalability experiments

In these experiments, we measured the throughput of B-tree operations for each of the workloads as we increased the system size. In each experimental run and for each workload, each client thread issued 2048 requests. For the insertion workload, for example, a single client issued 8192 requests total (recall each client had 4 threads).

We also ran our workloads on a centralized system running an efficient open-source B-tree implementation. This provides a base comparison for small configurations of our distributed B-tree. (For larger configurations, we observe how much performance improves as we increase the system size.) For the centralized system, we use the open-source B-tree of BerkeleyDB 4.5. We built a multi-threaded RPC server of our own to process client requests. Our RPC server accepts multiple client connections and issues the operations to BerkeleyDB as they arrive. The BerkeleyDB instance had a cache of 512MB, so that the entire B-tree could fit in memory, a setup that favors BerkeleyDB. A larger tree would have forced BerkeleyDB to perform random I/Os while our distributed B-tree could stay resident in the servers' main memory.

We first discuss the results for the lookup, update, and mixed workloads, shown in Figures 12, 13, and 14. These graphs show the aggregate throughput as we increased the number of servers from 1 to 92. For our distributed B-tree, we used the same number of clients as servers for each experiment, thus growing the workload as we increased the number of servers. For the BerkeleyDB experiment, the x-axis indicates the number of clients, each with 4 threads issuing requests.

These graphs show that our B-tree scales linearly with system size. For these experiments, our servers were nearly saturated. For example, an increase to 16 threads per machine increased total throughput of lookups by at most 10%. For updates, the bottleneck was Sinfonia's logging subsystem. A closer examination of the update graph shows that the system improves slightly super-linearly. This improvement is a result of better coalescing of transactions in batch commits to the log as the workload increases. Since the mixed workload is a combination of the previous two, it

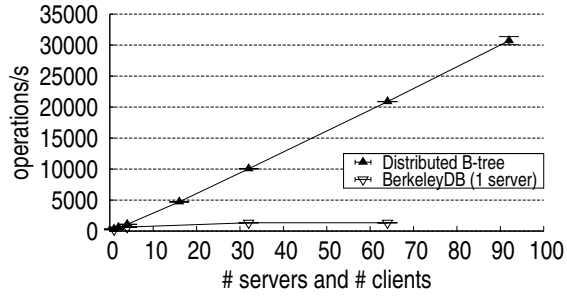

Figure 13: Aggregate throughput, update workload.

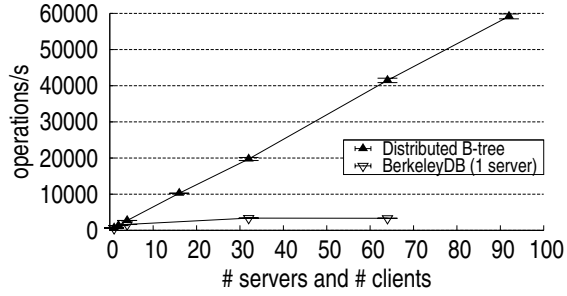

Figure 14: Aggregate throughput, mixed workload.

falls squarely between the two.

As expected, the BerkeleyDB instance saturates at a much lower value. However, BerkeleyDB, a code base optimized for over 15 years, does outperform our B-tree at small system sizes. The cross over is at 8 servers for lookups and at about 4 servers for updates. For lookups, part of the difference is because our clients fetch entire nodes when using our B-tree, but only fetch an 8 byte value from BerkeleyDB.

Finally, Figure 15 shows the performance of inserts compared to BerkeleyDB. Since our design was not intended for a rapidly growing B-tree, we see that its performance is only slightly better than a single BerkeleyDB instance. At smaller sizes, logging is the bottleneck for our B-tree. As we increase system size, contention becomes the bottleneck because as inner nodes split, concurrent inserts need to be retried. (The point in Figure 15 for 92 clients and servers is for a single thread rather than four threads.) In cases when lots of inserts must be done, one might consider using offline distributed bulk-insertion techniques to speed up the insertions (e.g., [19]). For BerkeleyDB the bottleneck is logging, and, as we increase the workload, better coalescing of batch commits improve the throughput.

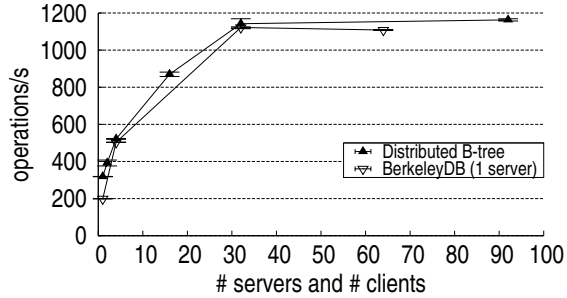

Figure 15: Aggregate throughput, insert workload.

These experiments show two important characteristics. First, at small scale our B-tree is competitive with a well-developed single-node system, BerkeleyDB. Second, except for the insert workload, our B-tree scales linearly, which is the best that one can expect from scaling a system.

7.3 Migration experiments

In these experiments, we measured the performance of the B-tree while nodes were being migrated in the background. Migration was performed by a *migration client* separate from other clients.

Figure 16 below shows aggregate throughput for a system with 16 clients and 16 servers, with and without migration running in the background. We used the Move migration task shown in Figure 7, which migrates all nodes from a given server to another server. In cases where the migration task finished before the experiment completed, we restarted the task in the other direction, as many times as needed.

Workload	Throughput without migration (operations/s)	Throughput with migration (operations/s)
Insert	870 ± 12	842 ± 32
Lookup	55422 ± 1097	55613 ± 1243
Update	4706 ± 18	4662 ± 19
Mixed	10273 ± 70	10988 ± 109

Figure 16: Effect of Move task on B-tree performance.

As can be seen, the throughput without and with migration is the same for every workload (for the mixed workload, performance with migration is better than without — this is an experimental fluctuation). In other words, migration imposes negligible overhead on each of the workloads. This holds for two reasons: the workloads were very active and our migration scheme backs off aggressively when it detects contention, choosing to work opportunistically when the system is less busy. The rate of migration in a completely idle system was 55.3 ± 2.7 nodes/s, or around 10000 key-value pairs/s. In contrast, in the above experiments with the various workloads, migration backed off to a rate of less than 5 nodes/s.

We also ran the above experiments with the Migrate-away task, and the results were similar: no noticeable overhead when migration occurs. We have not implemented the Populate, Even-out-storage, or Even-out-load tasks, but we expect them to behave identically, as migration backs off and does not interfere.

In our experiments, there were many servers and they were not fully saturated, and so there was little performance difference when the number of active servers decreased by one. When there are only a few servers or all servers are saturated, we expect migration to have a greater impact.

8. EXTENSIONS

Enhanced migration tasks. We implemented only simple migration tasks. A production system might benefit from migration based on expected load (due to expected seasonal variations) or in reaction to flash crowds. It might be interesting to support a migration scheme that operates more ag-

gressively when the system is loaded, to remove load quickly from overloaded nodes — this is the opposite of what we do now, where the migration backs off when the system is busy.

Dealing with hot-spots. The system may be subject to *hot-spots*, caused by a small number of popular keys that are accessed at a disproportionately high rate. This would create a performance bottleneck at the servers holding these keys, impairing scalability. Migration can help alleviate this problem, by migrating nodes so that the popular keys are spread over the servers in the system. This does not work directly if the popular keys are all stored in a *single* B-tree node, since migration granularity is a node. However, in this case we can use a simple trick: first, we spread the popular keys apart over different B-tree nodes by artificially inserting "dummy" key-value pairs between them. Then we spread the different nodes across servers through node migration.

Varying the replication factor of inner nodes. Currently, our implementation replicates the version number of every inner node at every server. Thus, operations that modify inner nodes (such as an Insert that causes a node split, or a Delete that causes a node merge) are fairly expensive because they cause a read-write transaction that involves every server. We could improve the performance of these operations by replicating version numbers less aggressively. For example, higher levels of the B-tree tend to change less frequently and so they could be replicated over more servers than lower levels.

Finer-grained concurrency control to avoid false sharing. We treat B-tree nodes as indivisible objects in our transactions, and this sometimes generates false sharing. For example, suppose that an Insert causes a leaf node to split, which requires modifying (a small part of) the parent node. Because transactions operate on a whole node at a time, this will create a conflict with any other B-tree operation that reads or writes this parent node, causing an aborted transaction, even if the other operation accesses different key-pointer pairs of the parent node. It should be possible to avoid this issue by implementing concurrency control at a finer level than a node.

9. RELATED WORK

As far as we know, this is the first work to provide a distributed data structure that efficiently and consistently supports dictionary and enumeration operations, execution of multiple operations atomically in a single instance and across multiple instances, as well as online migration of nodes.

Most prior work on concurrent B-trees focuses on shared memory systems, in which the B-tree is stored in a single memory space and multiple threads or processors coordinate access through the judicious use of locks. There are efficient concurrency control schemes for B-link trees [11, 17], which seek to reduce lock usage for efficiency. A B-link tree is a B-tree where each tree node is augmented with a pointer to its right sibling. This pointer allows Lookup operations to execute without acquiring any locks, while Insert and Delete operations acquire locks on a small number of nodes. Intuitively, the additional pointer allows a process to recover from temporary inconsistencies. ARIES/KVL is a lock-based method popular in commercial database systems because it accommodates range scans and integrates well with the recovery subsystem [15].

Algorithms for distributed B-trees in message-passing systems are proposed in [9, 14]. These algorithms use subtle protocols and locking schemes. Moreover, they do not support the execution of multiple operations atomically, node migration, or dynamic server addition and removal. The work in [14] considered only a fairly small system of up to 8 machines, and the work in [9] did not publish experimental results. Replicating B-tree information for performance is proposed in [9], which replicates inner nodes eagerly at many servers. In contrast, we replicate inner nodes lazily at clients for performance, and we replicate version numbers at servers for performance. The work in [19] presents offline bulk insertion techniques for data that is range-partitioned over a cluster. Their method balances insertion throughput with the cost of re-partitioning the existing data. We could complement our B-tree design with similar techniques.

Much work has been done on peer-to-peer data structures, such as distributed hash tables (DHT) (e.g., [20, 16]) and others. Unlike B-trees, hash tables do not support enumeration operations, though some DHT extensions support range queries (e.g., [2]). Peer-to-peer data structures work with little synchrony and high churn (i.e., nodes coming and going frequently), characteristic of the Internet, but tend to provide weak or no consistency guarantees. Applications of peer-to-peer systems include file sharing. In contrast, our work on the B-tree focuses on providing strong consistency in more controlled environments, such as data centers and computing clusters. Applications of B-trees include database systems.

A high-performance DHT that provides strong consistency in computing clusters was proposed in [5]. Other related distributed data structures include LH* [13], RP* [12], and their variants. None of these support transactions over multiple operations, and furthermore LH* and the DHT in [5] lack efficient enumeration.

Work on transactional memory proposes to use transactions to implement concurrent data structures [7, 18, 6]. That work focuses on shared memory multiprocessors, where processes communicate cheaply via a shared address space, and the challenge is how to execute memory transactions efficiently. This is done either in hardware, by extending the cache coherence protocol, or in software, by carefully coordinating overlapping transactions. In contrast, we consider message-passing systems where process communication is expensive (compared to shared memory), which calls for schemes that minimize communication and coordination.

Amazon's Dynamo [4] is a key-value store kept as a hash table using consistent hashing, without support for transactions or ordered key traversal. It provides weak consistency when the network partitions, to trade off consistency for availability. In contrast, our B-tree favors strong consistency and is designed for a single data-center environment.

BigTable [3] (and its variants, HyperTable and HBase) is a distributed scalable store that provides a "table-like" structure with rows and columns. There is a single key, which is one of the columns, and the system supports ordered traversal over the rows by the key. Within in each row, applications can fetch a subset of the columns. Internally, BigTable uses a tree-based structure similar to a B-tree, to organize the set of servers that store data. We tend to think of BigTable as a higher-level building block than a B-tree.

BigTable and its variants, however, only provide single-row atomicity: they do not support atomicity over multiple operations within the same table or across tables. There is also no support for consistent secondary indices, and this is a concern for some Google applications [3]. HyperTable and HBase have a single master that, without additional fault tolerance mechanisms [8, 22], are a single point of failure. Our B-tree, on the other hand, relies on Sinfonia for fault tolerance, which does not have a single point of failure.

Sinfonia [1] is a data sharing service that provides applications with unstructured address spaces on which to keep data. Applications are responsible for organizing and structuring their data on top of the address spaces. This is a lower-level service than a B-tree, which keeps key-value pairs. Our work builds upon the Sinfonia service.

10. CONCLUSION

We presented the design of a scalable distributed B-tree, based on distributed transactions for manipulating the B-tree nodes. Our approach has some features that are important in practice, namely (1) atomic execution of multiple B-tree operations, and (2) migration of B-tree nodes to enable important management operations, such as dynamic addition and removal of servers and load balancing. Our scheme relies on three key techniques to minimize network round-trips and achieve high concurrency: optimistic concurrency control, eager replication of node version numbers at servers, and lazy replication of inner nodes at clients. These techniques together allow clients to execute common B-tree operations efficiently, in one or two network round-trips most of the time. Our scheme has been implemented and evaluated. We showed that it scales to hundreds of machines and, at a small scale, its performance is comparable to BerkeleyDB. We believe that our approach is quite general and can be easily carried over to implement other distributed data structures.

Acknowledgements. We would like to thank Theodore Johnson for his insightful suggestions, and the anonymous reviewers for many helpful comments.

11. REFERENCES

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. SOSP'07*, pages 159–174, Oct. 2007.

[2] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Proc. P2P'02*, pages 33–40, Sept. 2002.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. SOSP'06*, pages 205–218, Nov. 2006.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP'07*, pages 205–220, Oct. 2007.

[5] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proc. OSDI'00*, pages 319–332, Oct. 2000.

[6] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proc. PODC'03*, pages 92–101, July 2003.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. ISCA'93*, pages 289–300, May 1993.

[8] HyperTable. HyperTable: An Open Source, High Performance, Scalabe Database, 2008. Online: http://hypertable.org/.

[9] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the B-link tree. In *Proc. IPPS'92*, pages 319–324, Mar. 1992. A longer version appears as MIT Tech Report MIT/LCS/TR-530, Feb. 1992.

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

[11] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, Dec. 1981.

[12] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proc. VLDB'94*, pages 342–353, Sept. 1994.

[13] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - a scalable, distributed data structure. *ACM Trans. Database Syst.*, 21(4):480–525, Dec. 1996.

[14] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. OSDI'04*, pages 105–120, Dec. 2004.

[15] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In *Proc. VLDB'90*, pages 392–405, Aug. 1990.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. SIGCOMM'01*, pages 161–172, Aug. 2001.

[17] Y. Sagiv. Concurrent operations on B-trees with overtaking. In *Proc. PODS'85*, pages 28–37, Mar. 1985.

[18] N. Shavit and D. Touitou. Software transactional memory. In *Proc. PODC'95*, pages 204–213, Aug. 1995.

[19] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *Proc. SIGMOD'08*, pages 765–778, June 2008.

[20] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. SIGCOMM'01*, pages 149–160, Aug. 2001.

[21] Sun Microsystems. Lustre, 2008. Online: http://lustre.org/.

[22] The Apache Software Foundation. Hadoop, 2008. Online: http://hadoop.apache.org/.