



A practical scalable distributed B-tree

Marcos K. Aguilera, Wojciech Golab

Information Services and Process Innovation Laboratory

HP Laboratories Palo Alto

HPL-2007-193

December 17, 2007*

distributed data
structures,
distributed
algorithms,
scalability, fault
tolerance

We propose a new algorithm for a practical, fault-tolerant, and scalable B-tree distributed over a set of servers. Our algorithm supports practical features not present in prior work: transactions that allow atomic execution of multiple operations over multiple B-trees, online migration of B-tree nodes between servers, and dynamic addition and removal of servers. Moreover, our algorithm is conceptually simple: we use transactions to manipulate B-tree nodes so that clients need not use complicated concurrency and locking protocols used in prior work. We implemented our approach and show that its performance and scalability are comparable to previous schemes, yet it offers many additional practical features. We believe that our approach is quite general and can be used to implement other distributed data structures easily.

Internal Accession Date Only

Approved for External Publication

A practical scalable distributed B-tree

Marcos K. Aguilera
HP Laboratories

Wojciech Golab
University of Toronto

Abstract

We propose a new algorithm for a practical, fault-tolerant, and scalable B-tree distributed over a set of servers. Our algorithm supports practical features not present in prior work: transactions that allow atomic execution of multiple operations over multiple B-trees, online migration of B-tree nodes between servers, and dynamic addition and removal of servers. Moreover, our algorithm is conceptually simple: we use transactions to manipulate B-tree nodes so that clients need not use complicated concurrency and locking protocols used in prior work. To execute these transactions quickly, we rely on three techniques: (1) We use optimistic concurrency control, so that B-tree nodes are not locked during transaction execution, only during commit. This well-known technique works well because B-trees have little contention on update. (2) We replicate inner nodes at clients. These replicas are lazy, and hence lightweight, and they are very helpful to reduce client-server communication while traversing the B-tree. (3) We replicate version numbers of inner nodes across servers, so that clients can validate their transactions efficiently, without creating bottlenecks at the root node and other upper levels in the tree. We implemented our approach and show that its performance and scalability are comparable to previous schemes, yet it offers many additional practical features. We believe that our approach is quite general and can be used to implement other distributed data structures easily.

1 Introduction

The B-tree is an important data structure used in databases and other systems that need to efficiently maintain large ordered datasets. In this paper, we

propose a new algorithm for *distributed B-trees*, which are B-trees whose nodes are spread over multiple servers on a network (Figure 1). We focus on B+-trees, where key-value pairs are all stored on leaf nodes. Our algorithm is fault-tolerant and has good scalability and performance: clients can execute B-tree operations in one or two network round-trips most of the time. In addition, our algorithm provides a number of practical features that improve on prior schemes [6, 11]:

- *Transactional access.* A client can execute a sequence of B-tree operations on one or more B-trees, and do so atomically. This is important for applications that use B-trees to build other data structures.
- *Online migration of tree nodes.* A client can move one or many tree nodes from one server to another, and this is done transparently while the B-tree remains functional. This enables better load balancing among servers, especially when new servers are added.
- *Dynamic addition and removal of servers.* Servers holding the B-tree can be added and removed transparently while the B-tree remains functional. This feature allows the system to grow and shrink dynamically, and it is also useful for replacing or maintaining servers.

Another advantage of our algorithm is that it avoids the subtle (and error-prone) concurrency and locking protocols in previous schemes [6, 11]. Instead, we use a very simple idea: manipulate B-tree nodes using 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 concurrent operations.

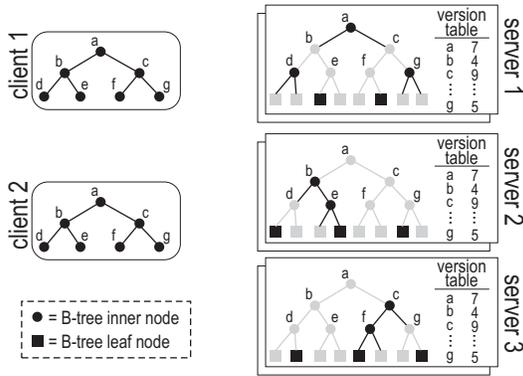


Figure 1: Our distributed B-tree. Nodes are divided among servers (grey indicates absence of node). A version table stores version numbers for inner nodes. Leaf nodes have versions, but these not stored in the version table. Two types of replication are done for performance: (a) lazy replication of inner nodes at clients, and (b) eager replication of the version table at servers. In addition, we optionally use a primary-backup scheme to replicate entire servers for enhanced availability. 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 key challenge we address is how to execute such transactions efficiently. To do so, we rely on the combination of three techniques: (1) optimistic concurrency control, (2) lazy replication of inner nodes at clients, and (3) eager replication of node version information at servers.

With the first technique, optimistic concurrency control [7], a client does not lock tree nodes during transaction execution. Instead, nodes are only locked when the transaction commits, at which point the client checks that the nodes read by the transaction have not changed. To facilitate this check, B-tree nodes have a version number that is incremented every time the node changes. During commit, if the version numbers of nodes read by the transaction match those at the servers, the transaction commits. Otherwise, the client retries the transaction. Optimistic concurrency control works well because there is typically little update contention on nodes of a B-tree.

With the second technique, lazy replication at clients, each client maintains replicas of inner B-tree nodes it has discovered so far. As B-tree nodes tend to have many children (fan-out of 200 or more is typical), inner nodes comprise a small fraction of the data in a B-tree, so replicating inner nodes takes relatively little space. Replicas are updated lazily: clients fetch updates only after they attempt to use a

stale version. This avoids the problem of updating a large number of clients in a short period when inner nodes change.

A transaction frequently needs to check the version number of the root node and other upper-level nodes, and this can create a performance bottleneck at the servers holding these nodes. With the third technique, eager replication of version numbers at servers, we replicate these version numbers across all servers so that they can be validated at any one server. The resulting network traffic is manageable since we replicate version numbers only for inner nodes of the tree, and these typically change infrequently when the tree fanout is large.

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. It is the combination of *all* three techniques that provides efficiency. For example, without lazy replication, 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.

We have implemented our scheme and evaluated it using experiments, which show that performance and scalability are good. We also derive space and time complexities analytically.

To summarize, in this paper we provide a new practical and fault-tolerant algorithm for distributed B-trees. This is the first algorithm that supports transactional access (multiple B-tree operations executed atomically), online node migration, and dynamic addition and removal of servers—features that are important in practice. Our scheme is simpler to implement and more powerful than previous schemes based on rather complicated locking protocols. Moreover, we believe our approach is quite general and can be used to implement other distributed data structures with little effort. We consider the performance of our algorithm analytically and experimentally, and we show that performance and scalability are good.

This paper is organized as follows. Related work is explained in Section 2. We describe the model in Section 3 and the problem in Section 4. In Section 5 we explain the transactions we use and techniques to

make them fast. The B-tree algorithm is presented in Section 6, followed by its analysis in Section 7 and experimental evaluation in Section 8. Section 9 concludes the paper.

2 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, and online addition and removal of servers.

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. The best known concurrent B-tree scheme is called a B-link tree [8, 13], which seeks 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.

Algorithms for distributed B-trees in message-passing systems are proposed in [6, 11]. For managing concurrency, [6, 11] use subtle protocols and locking schemes. These algorithms do not support the execution of multiple operations atomically, node migration, or dynamic server addition and removal. Moreover, experiments show that our algorithm performs similarly to [11] ([6] does not publish performance numbers).

Much work has been done on peer-to-peer data structures, such as distributed hash tables (DHT) (e.g., [15, 12]) and others. Unlike B-trees, hash tables do not support enumeration operations, though some DHT extensions support range queries (e.g., [1]). 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 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 [2]. Other distributed data structures include LH* [10], RP* [9], and their variants. None of these support transactions over multiple operations, and furthermore [2] and LH* lack efficient enumeration.

Work on transactional memory proposes to use transactions to implement concurrent data structures [4, 14, 3]. 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 requires schemes that minimize communication and coordination.

For message-passing systems, distributed B-trees are proposed in [11, 6], without support for transactions or online removal of servers. Replicating B-tree information for performance is proposed in [6], which replicates inner nodes eagerly at many servers. In contrast, we replicate inner nodes lazily at clients for performance, we replicate version numbers at servers for performance, and we (optionally) replicate servers for fault tolerance, not performance.

3 Model

We consider distributed systems with a set $\Pi = \Pi^s \cup \Pi^c$ of processes, where processes in Π^s are called *servers* and processes in Π^c are called *clients* ($\Pi^s \cap \Pi^c = \emptyset$). Intuitively, servers implement a service (i.e., the distributed B-tree) while clients utilize the service. Throughout this paper, *nodes* refer to B-tree nodes rather than hosts or processes.

Processes can communicate with each other by sending messages through bidirectional links. There is a link between every client and server, and between every pair of servers, but there may not be links between pairs of clients (e.g., to model the fact they are not aware of each other). Links are reliable: they do not drop, create, or duplicate messages.

Operation	Description
$\text{Insert}(k, v)$	adds (k, v) to B
$\text{Lookup}(k)$	returns v s.t. $(k, v) \in B$ or \perp (if no such v)
$\text{Delete}(k)$	deletes (k, v) from B for v s.t. $(k, v) \in B$
$\text{getNext}(k)$	returns smallest $k' > k$ s.t. $(k, \bullet) \in B$, or \perp
$\text{getPrev}(k)$	returns largest $k' < k$ s.t. $(k, \bullet) \in B$, or \perp

Figure 2: Operations on a B-tree B .

Processes are subject to crash failures. To deal with crashes, servers have stable storage. A server that crashes subsequently recovers with its stable storage intact. Clients may not have stable storage and may not recover from crashes.

The system is synchronous: message delays and process step delays are bounded. This is intended to model corporate data centers where B-trees are used¹ and it is similar to the perfect failure detection assumption in [11].

4 Problem

We provide an overview of B-trees, define distributed B-trees, explain transactional access, and describe B-tree management operations.

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) such that there is at most one value v associated with each key k . A B-tree supports the standard dictionary operations (Insert, Lookup, Delete) and enumeration operations (getNext, getPrev), described in Figure 2.

A B-tree is internally organized as a balanced tree. We focus on the B+-tree, a B-tree variant where key-value pairs are only stored at leaf nodes (Figure 3). Each tree level stores keys in increasing order. The lowest level also stores values associated with keys, while upper levels store pointers. To lookup a key, we start at the root and follow the appropriate pointers to find the proper leaf node. 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

¹Our scheme should be extensible to asynchronous systems by relaxing B-tree consistency, but we have not done so since applications of B-trees, such as databases, require consistency.

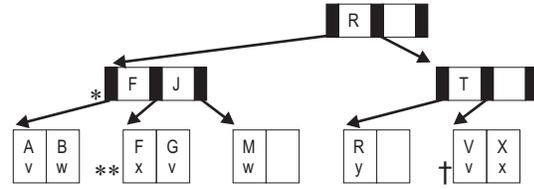


Figure 3: B+-tree: leaf nodes store key-value pairs and inner nodes store keys and pointers. Keys and values are denoted in upper and lower case, respectively. To lookup key G , we start at the root, follow the left pointer as $G < R$, arrive at node $*$, follow the middle pointer as $F < G < J$, and arrive at node $**$ where G is.

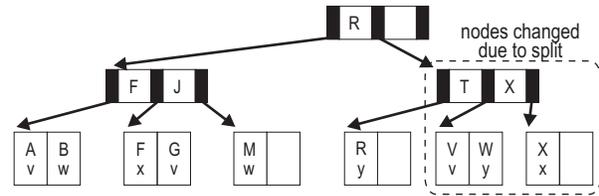


Figure 4: Splitting nodes: inserting (W, y) causes a split of node \dagger in the tree from Figure 3.

update the parent appropriately as illustrated in Figure 4. Updating 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, we require that a node can be placed on any server according to some arbitrary user-supplied function $\text{chooseServer}()$ that is called when a new node is allocated. For example, if a new server is added, $\text{chooseServer}()$ can return the new server until it is as full as others.

A distributed B-tree supports concurrent operations by multiple clients, and its operations should be linearizable [5]. Roughly speaking, linearizability requires each operation to appear to take effect instantaneously at a point in time between the operation invocation and response.

4.3 Transactional access

Transactional access allows clients to perform multiple B-tree operations atomically. Transactions are linearizable: the corresponding sequence of B-tree

Operation	Description
<code>addServer(s)</code>	add new server s
<code>removeServer(s)</code>	remove server s from system
<code>Migrate(x, s)</code>	migrate node x to server s
<code>firstNode(s)</code>	enumerate nodes in server s
<code>nextNode(x, s)</code>	enumerate nodes in server s

Figure 5: Management operations on a B-tree.

operations appears to take effect instantaneously at a point in time between when the transaction begins and when it commits successfully. We expect transactions to be short-lived rather than long-lived (i.e., execute within milliseconds, not hours). Below we show a sample transaction to atomically increment the value associated with key k .

```

BeginTx();
 $v \leftarrow$  Lookup( $k$ );
Insert( $k, v + 1$ );
 $success \leftarrow$  Commit();
EndTx();

```

4.4 Management operations

Management operations include migration of nodes and additions and removals of servers, as shown in Figure 5. Function `addServer` adds a new server that can then be used to allocate B-tree nodes, which is useful for expanding the system. Function `removeServer` is used to remove a server, say, for upgrades or maintenance, or just to shrink the system. The operation first migrates all nodes in a server to other servers, which could take some time, and then removes the server from the system (before some client adds nodes to it). `Migrate` is used to move nodes from one server to another, say to balance load after a new server has been added. To enumerate nodes, `firstNode(s)` returns the first node in server s (in some arbitrary order) and `nextNode(x, s)` returns the node after node x . Other management operations (not shown) include querying current and historical load and space usage of a server. All management operations occur *online*: during their execution the B-tree remains fully operational.

5 Transactions

In our B-tree implementation, clients use transactions to atomically manipulate nodes and other objects. We use a type of optimistic distributed transaction with several techniques to improve performance.

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. Below is a list of objects we actually use.

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

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 [7]: objects are not locked during transaction execution, only during commit, which happens through standard two-phase commit.

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 replicate certain key 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 (inner nodes of the B-tree) change relatively rarely.

Eager replication of version numbers at servers. B-tree operations tend to read upper-level B-tree nodes frequently. For example, every 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. We maintain these replicas synchronized when a transaction commits; this is done as part of two-phase commit. Figure 1 illustrates the various types of replication that we use.

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. We can then commit the transaction in one phase instead of two (explained below). The end result is that most B-tree operations commit in one network round-trip.

Committing transactions. A client commits a transaction with standard two-phase commit, with modifications to deal with objects that have replicated version numbers at servers. If such an object is in the write set, its new version must be updated at all servers. If such an object is in the read set, it can be validated at any server. Validation means to check whether the object is up-to-date, by comparing its version number (or its content if the object has no version numbers) against what is stored at some server. The client chooses the server so as to minimize the number of different servers involved in the transaction; if only one server is involved, the transaction can be committed faster (as explained below).

In two-phase commit, the first phase locks and validates objects used in the transaction, and the second phase commits the changes if the first phase is successful. More precisely, in the first phase, the client sends the read and write sets to servers. A server tries to read-lock and write-lock objects in the read and write sets, and validates the objects in the read set against the objects stored at the server. Servers reply with an OK if the objects are successfully locked

and validations are successful. Servers do not block on locks; they simply return not-OK to the client. In the second phase, the client tells servers to commit if all replies are OK, or abort otherwise. In the latter case, the client restarts the transaction (possibly after some random exponential back-off, to deal with contention).

If a transaction only involves one server then it can commit with one-phase commit: a client can simply send a single message to a server with the read and write sets. The server validates the read set and, if successful, writes the objects in the write set.

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 objects in the read set. If some object is stale, this check enables clients to abort doomed transactions early. A read-only transaction (e.g., performing only Lookup) for which the last read was validated as just described, can be committed without accessing any server.

Fault tolerance. Our algorithm tolerates crash failures of clients and servers. A server uses stable storage to recover from crashes without losing the contents of the B-tree. Because we use transactions, fault tolerance comes for free: we just need to ensure that transactions are atomic, and this is provided by two-phase commit. For example, if a client crashes outside the two-phase commit protocol, there is no recovery to do. If a client crashes during two-phase commit, the recovery comprises determining the outcome of the transaction (by querying the servers involved in the transaction) and informing all servers of this outcome.

We also optionally provide another form of fault tolerance: active primary-backup replication of servers (not to confuse with the other forms of replication discussed above). Roughly speaking, the backup server is a shadow replica of the primary at all times: everything is replicated, including all B-tree nodes. This well-known technique enables the system to fail over to the backup if the primary server crashes, without having to wait for the primary server to recover. We use transactions as natural points in time to synchronize the backup. Because primary-backup replication is expensive in terms of resources, this mechanism is optional, according to the user's

preferences. Note that even without primary-backup replication, data are not lost on crashes since servers can use stable storage to recover.

Transaction interface. The table below shows the interface to transactions, where n denotes an object ID and val is an object value.

operation	description
BeginTx()	clear read and write sets
Read(n)	read object n locally or from server and add (n, val) to read set
Write(n, val)	add (n, val) to write set
Commit()	execute two-phase commit
Abort()	abort transaction
IsAborted()	check if transaction has aborted
EndTx()	garbage collect transaction structures

Roughly speaking, Read and Write add objects to the read and write sets, while Commit executes two-phase commit as described above. Abort marks the transaction as prematurely aborted so that subsequent calls to Commit fails. IsAborted indicates whether the transaction has aborted.

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.

We show Insert in detail on page 7; other operations are similar, and are included in Appendix B. InsertHelper is a recursive helper function for inserting, and Search returns the insertion index for a key within a node. The overall algorithm is very similar to that of a centralized B-tree; an asterisk indicates lines where there are differences:

- We use the transactional Read and Write to read and write B-tree nodes.

Field	Description
isLeaf	Boolean indicating node type
depth	distance of node from root
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 6: Fields of a tree node

Function Insert($key, value$)

Input: $key, value$ – key/value pair to insert
Output: true iff key was not already in the tree

```

*  $rootNum \leftarrow getRoot()$ 
*  $(ret, modified, root) \leftarrow InsertHelper(rootNum, key, value, -1)$ 
if IsAborted then return  $\perp$ 
if  $root$  has too many keys then
    | split  $root$  into children  $child$  and  $child'$ , and
    | new root  $root$ 
*   |  $c \leftarrow Alloc(); c' \leftarrow Alloc()$ 
*   | Write( $rootNum, root$ )
*   | Write( $c, child$ ); Write( $c', child'$ )
else if  $modified$  then
* | Write( $rootNum, root$ )
end
return  $ret$ 

```

- We use a special getRoot helper function to find the B-tree root.
- We use our Alloc and Free to allocate and free B-tree nodes transactionally.
- We perform various safety checks to prevent the client from crashing when its local replicas of objects are inconsistent with each another.

We now explain these differences in more detail.

Reading and writing B-tree nodes (Read, Write). Reading and writing B-tree nodes simply entails calling the transactional Read and Write functions, which were explained in Section 5.

Finding the root (getRoot). 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, Free). We need to allocate B-tree nodes transactionally to

Function Search(*node*, *key*)

Input: *node* – node to be searched
Input: *key* – search key
if *node.numKeys* = 0 **then**
| **return** \perp
else
| **return** index of the largest key in
| *node.keys*[1..*numKeys*] that does not exceed
| *key*, or else 0 if no such key
end

Function InsertHelper(*n*, *key*, *value*, *d*)

Input: *n* – node ID
Input: *key*, *value* – key/value pair to insert
Input: *d* – depth of previous node visited
Output: tuple (*ret*, *modified*, *node*) where
ret = true iff *key* was not already in
the tree, *modified* = true iff node *n*
has changed, and *node* is node *n*

* *node* \leftarrow Read(*n*)
* **if** *node* = $\perp \vee$ *node.depth* \leq *d* **then**
* | Abort (); **return** \perp
end
i \leftarrow Search(*node*, *key*)
if *node.isLeaf* **then**
| **if** *i* \neq 0 \wedge *node.keys*[*i*] = *key* **then**
| | *node.values*[*i*] \leftarrow *value*
| | **return** (false, true, *node*)
else
| insert *key* and *value* into *node*
| **return** (true, true, *node*)
end
else
| *c* \leftarrow *node.children*[*i*]
* (*ret*, *modified*, *child*) \leftarrow InsertHelper(*c*,
* *key*, *value*, *node.depth*)
* **if** IsAborted **then** **return** \perp
* **if** *child* has too many keys **then**
* | split *child* into *child* and *child'*, update
* | *node* as needed
* | *c'* \leftarrow Alloc()
* | Write(*c*, *child*); Write(*c'*, *child'*)
* | **return** (*ret*, true, *node*)
* **else if** *modified* **then**
* | Write(*c*, *child*)
end
return (*ret*, false, *node*)
end

avoid races (double allocations) and memory leaks (when a client crashes). 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). Then, the client selects a free entry in the bit vector of that server and marks it as used. The client then adds the chosen bit vector entry (not the node itself) to the transaction's write set and read set. The reason for adding it to the read set is to ensure that the corresponding node is still free at the time just before the transaction commits.

Node deallocation is similar to node allocation: a client sets the appropriate bit vector entry and adds this object to a transaction's read and write set.

Servers can be removed from the system, and a client should not allocate B-tree nodes on a server that is removed or being removed. For this purpose, the server list in the metadata object includes a *transient* tag for servers where allocation is prohibited. The metadata object is included in a transaction's read set, to ensure the server list is consistent on commit.

Safety checks. Clients may have inconsistent data in their lazy replicas of tree nodes, and so, to avoid crashing, clients need to perform some safety checks:

- array index out of bounds
- null pointer or divide by zero
- object read has unexpected type
- infinite cycles while traversing tree

We detect the last condition by recording the distance of a node from the root in the *depth* field, and ensuring that the value of this field increases monotonically during a traversal of the tree. If any of the consistency checks fail, the transaction is aborted and restarted.

Furthermore, when a client reads a node from a server (in Read), we piggyback a request to validate the transaction's read set. 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, to learn 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 updates in the transaction’s write set. Thus, a client can execute multiple operations, and then commit all of them together.

6.4 Management operations

We now describe how to migrate tree nodes, add servers, and remove servers. These are executed on-line (i.e., the B-tree remains fully operational).

Migration. 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.

Server addition. Adding a server (operation `addServer(s)`) entails populating the version table of s and making it available for future node additions and migrations. To do so, we first initialize the version number table with special “don’t know” versions that are smaller than any real version (e.g., -1). Then, we update the metadata object (using a transaction) to add s to the server list and tag s as transient. This writes the new metadata object to all servers including s itself. A transient server does not participate in node allocation, but continues to receive updates of version numbers. Next, we populate the version number table of s by reading version numbers of inner nodes from another server and taking the max with what s might already have. This reading need not be done using a transaction, since concurrent updates of version numbers will be delivered to s and correctly handled by applying `max`.

Finally, we update the metadata object (using a transaction) to tag s as not transient. Now, server s can be populated with nodes using the `Migrate` operation, or by waiting for new nodes to be gradually allocated on s .

Server removal. Removing a server (operation `removeServer(s)`) entails migrating away nodes on s , while ensuring that clients eventually stop using s . To do so, we first update the metadata object (using a transaction) to tag server s as transient. This prevents clients from adding new nodes on server s , but s continues to respond to reads and validations of its nodes. Next, if the directory service points to s (see “Initialization” in Section 5), we change it to point to a different server. This ensures that once s is eliminated, new clients can still find a functional server. We then use `Migrate` to migrate all nodes of server s to different servers according to some placement policy (in our implementation, placement is done randomly). Then, we update the metadata object (using a transaction) by removing s from the server list. (This transaction also updates the replica of the metadata object in server s .) After this is done, a client that still thinks that s is in the system will have its transactions aborted, since this client has a stale metadata object and every transaction validates the metadata object on commit. Finally, we terminate the server process at s .

7 Analysis

In this section we analyze the complexity of our B-tree in terms of the variables shown below.

Variable	Meaning
S	number of servers
N	number of key-value pairs in the tree
Z_k, Z_v	size of each key and value, respectively
D	size of a tree node

Note that the number of leaf nodes in the tree is $\Theta(N(Z_k + Z_v)/D)$, and the number of internal nodes is at most the number of leaf nodes. The branching factor is $\Theta(D/Z_k)$, and so the height of the tree is $\Theta(\log_{D/Z_k} N(Z_k + Z_v)/D)$.

Space complexity. Space is nearly evenly distributed across servers. Total space needed is $\Theta(N(Z_k + Z_v))$ for tree nodes, $O(S^2)$ for metadata objects containing the root node ID and server list,

and $O(N(Z_k + Z_v)S/D)$ for version number tables and node allocator bit vectors. Thus, the space complexity is $O(N(Z_k + Z_v)(1 + S/D) + S^2)$.

Communication complexity. In common cases, a Lookup requires one network round-trip to read a leaf node and simultaneously validate inner nodes. getNext and getPrev are similar to Lookup. An Insert or Delete require two network round-trips, one to read a leaf node and one to commit the change to the leaf node. In the worst case, a tree operation requires an unbounded number of network round-trips as transactions repeatedly abort due to contention. In practice, contention on a B-tree is rare and is managed successfully using an exponential back-off scheme. The worst case without contention occurs when a client’s replicas of tree nodes are all stale. Then, the number of round-trips for an operation is proportional to the height of the tree as a client needs to read each tree level in sequence.

The number of network messages exchanged per network round-trip is constant, except when an Insert or Delete operation changes at least one inner node. In that case the client generates $\Theta(S)$ messages on commit in order to update version number tables. However, the number of messages per round-trip amortized over multiple B-tree operations is still constant provided that the fanout of the tree is $\Omega(S)$.

8 Evaluation

We implemented our scheme using C++ and evaluated its performance on a cluster of 24 1GHz Pentium III machines with SCSI hard disks connected by a Gb Ethernet switch. Each client and server ran on their own machine. In each experiment, the tree was pre-populated with 100,000 elements. Then each client ran three batches of Insert, Lookup, and Delete operations, respectively and in this order, where each batch had 10,000 operations. The tree node size was 4 KB, keys had 10 bytes, and values had 8 bytes, for a total of 180 and 220 keys per inner and leaf nodes, respectively. For Insert operations, keys were generated uniformly at random from a space of 10^9 elements. For Lookup and Delete operations, keys were chosen from those previously inserted. Clients started each batch simultaneously (to within one second), with possibly stale replicas

of inner nodes. Servers wrote updates to disk synchronously and used primary-backup replication.

Figure 7 shows aggregate throughput as we vary the system size. For the lower curve, we vary the number of clients but keep the number of servers at two. For the higher curve, we vary the number of clients and servers together. Error bars show one standard deviation. As can be seen, throughput tends to level off when the number of servers is constant, while throughput increases almost linearly when the number of servers increase with the number of clients. Insert and Delete operations were 3-7 times slower than Lookup, since Lookup does not write to disk. These numbers are similar to the ones reported in [11]. We also did experiments without disk logging and primary-backup replication, and performance improved roughly two-fold.

The number of network round-trips per tree operation, averaged over 10,000 operations, was 2.0-2.2 for Insert, 1.000-1.001 for Lookup, and 2.1-2.6 for Delete. These averages are close to the corresponding lower bounds of two, one, and two network round-trips for these operations, respectively. For example, Insert requires one round-trip to fetch a leaf node (since leaf nodes are not replicated at clients) and one round-trip to write it. Operation latency (under minimal load) was 2.5 ms for Insert, 1.8 ms for Lookup, and 2.7 ms for Delete. These numbers are reasonable, being close to disk access times.

9 Conclusion

We presented a conceptually simple method to implement a distributed B-tree, by using distributed transactions to manipulate the B-tree nodes. Our approach has some features that are important in practice, namely (1) being able to atomically execute multiple B-tree operations, (2) migrating B-tree nodes, and (3) dynamically adding and removing servers. A key challenge addressed by our scheme is how to efficiently use transactions to manipulate data structures. To do so, we proposed three techniques: optimistic concurrency control, lazy replication of inner nodes at clients, and eager replication of node versions at servers. These techniques together allow clients to execute common B-tree operations very efficiently, in one or two network round-

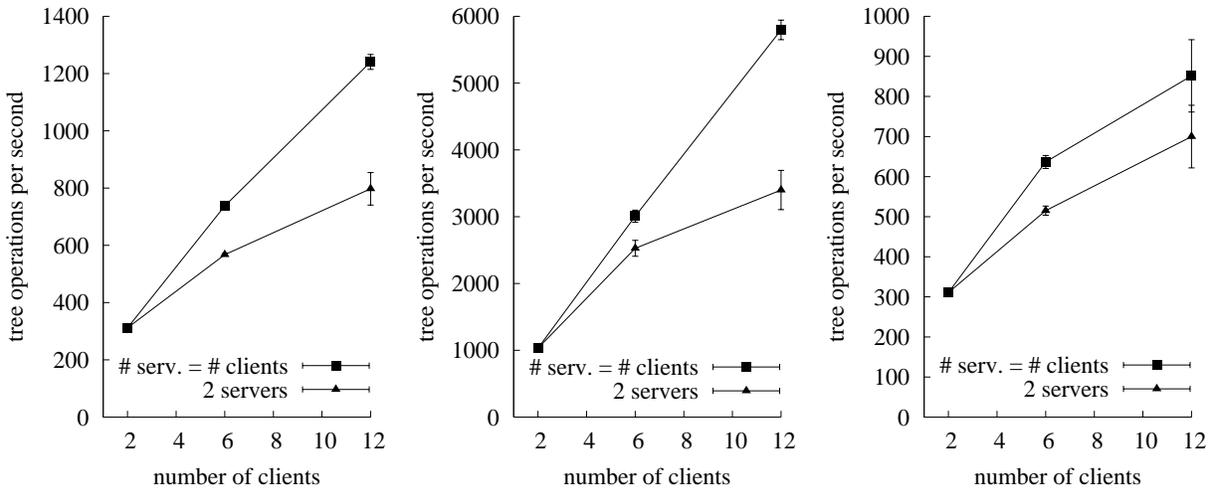


Figure 7: Aggregate throughput for Insert (left), Lookup (middle), and Delete (right) operations.

trips most of the time. An implementation and experiments confirm the efficiency of our scheme. We believe our approach would be useful to implement other distributed data structures as well.

References

- [1] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Proc. P2P'02*, Sept. 2002.
- [2] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. OSDI'00*, Oct. 2000.
- [3] 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.
- [4] 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.
- [5] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *Trans. on Programming Languages and Syst.*, 12(3):463–492, July 1990.
- [6] 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.
- [7] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.
- [8] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.
- [9] 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.
- [10] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - a scalable, distributed data structure. *ACM Trans. Database Syst.*, 21(4):480–525, 1996.
- [11] 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.
- [12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM 2001*, Aug. 2001.
- [13] Y. Sagiv. Concurrent operations on B-trees with overtaking. In *Proc. PODS'85*, pages 28–37, Mar. 1985.
- [14] N. Shavit and D. Touitou. Software transactional memory. In *Proc. PODC'95*, pages 204–213, Aug. 1995.
- [15] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM 2001*, Aug. 2001.

A Implementations of transactional primitives

In this section, we present implementations of the transactional primitives discussed in Section 5. The pseudo-code below is expressed in terms of the following static variables: `status` is the status of a transaction (one of pending, committed, aborted); `readSet` and `writeSet` are the read and write sets, respectively; and `localReplicas` is the set of local object replicas. The latter three sets are initially empty.

We represent a set of objects (e.g., `localReplicas`) as a set of pairs of the form (ID, object). If S is such a set then we denote by $S[i]$ the element with ID i (or \perp if there is no such element in S). In the pseudo-code we use $S[i]$ on the left side of an assignment to indicate the addition to S of an object with ID i , replacing any prior such object in S .

The functions `ValidateFetch` and `ValidateUpdate` called by `Read`, `Write`, and `Commit` provide atomic access to one or more objects at one or more servers. The pseudo-code gives their high-level specification independent of the message-passing protocols that implement them. `ValidateFetch(V, i)` atomically validates objects in set V and fetches the object with ID i . Similarly, `ValidateUpdate(V, U)` atomically validates objects in set V and writes objects in set U .

In reality, `ValidateFetch(V, i)` is implemented with a simple two-phase protocol (not shown), where the first phase contacts servers to validate objects in V and the second phase contacts a server to fetch object i . If all objects in V can be validated at a server where i is stored, we can combine both phases into a single phase (i.e., the server validates objects in V and responds with the value of object i , both in the same phase).

`ValidateUpdate(V, U)` is implemented using two-phase commit (not shown) with the modifications described in Section 5: (1) if an object in U has its version replicated at servers then two-phase commit updates this version at all servers, and (2) if all objects in V can be validated at a single server, and all updates in U occur at this server (in particular, no objects in U have their version replicated at servers), then we can combine both phases of the protocol (one-phase commit).

Function BeginTx

```
status ← pending
```

Function EndTx

```
if status = pending then Abort ()
readSet ← ∅
writeSet ← ∅
```

Function Write(*n*, *obj*)

```
Input: i – object ID
Input: obj – object
writeSet[i] ← obj
return OK
```

Function Read(*i*)

```
Input: i – object ID
Output: object with ID i, or else ⊥ if
        transaction aborted

if writeSet[i] ≠ ⊥ then
  | return writeSet[i]
else if localReplicas[i] ≠ ⊥ then
  | readSet[i] ← localReplicas[i]
  | return localReplicas[i]
else
  Ret ← ValidateFetch(readSet, i)
  if Ret = ⊥ then
    | Abort ()
    | return ⊥
  else
    readSet[i] ← Ret
    if Ret is an inner tree node then
      | localReplicas[i] ← Ret
    end
    return Ret
end
end
```

Function Abort

```
if status = pending then
  | status ← aborted
  | foreach (i, obj) ∈ readSet do
    | localReplicas[i] ← ⊥
  end
end
return OK
```

Function IsAborted

```
Output: true if status is aborted, otherwise
        false
return status = aborted
```

Function Commit

```
Output: true if readSet successfully validated,
        otherwise false

if status ≠ pending then
  | return false
end
Ret ← ValidateUpdate(readSet, writeSet)
if Ret = ⊥ then
  | Abort ()
  | return false
else
  status ← committed
  foreach (i, obj) ∈ writeSet do
    | if obj is an inner tree node then
      | localReplicas[i] ← obj
    end
  end
  return true
end
```

Function ValidateFetch(*V*, *i*)

```
Input: V – objects to validate
Input: i – ID of object to fetch
Output: if objects in V are up-to-date then
        return object i, otherwise return ⊥

Atomically execute the following:
foreach (j, obj) ∈ V do
  | if object j at server differs from obj then
    | return ⊥
  end
end
Ret ← value of object i at server
return Ret
```

Function ValidateUpdate(*V*, *U*)

```
Input: V – objects to validate
Input: U – objects to update
Output: if objects in V are up to date then
        return OK, otherwise return ⊥

Atomically execute the following:
foreach (i, obj) ∈ V do
  | if object i at server differs from obj then
    | return ⊥
  end
end
foreach (i, obj) ∈ U do
  | write obj to object i at server
end
return OK
```

B Transactional B-tree pseudo-code

In this section, we present the implementations for the Lookup and Delete operations, whereas Insert was already discussed in Section 6. We omit implementations of `getPrev` and `getNext` as these are very similar to Lookup.

Function Lookup(*key*)

Input: *key* – search key

Output: value corresponding to *key*, or \perp is no such value

* *rootNum* \leftarrow `getRoot()`

* **return** `LookupHelper(rootNum, key, -1)`

Function LookupHelper(*n*, *key*, *d*)

Input: *n* – node ID

Input: *key* – search key

Input: *d* – depth of the last node visited

* *node* \leftarrow `Read(n)`

* **if** *node* = \perp \vee *node*.depth \leq *d* **then**

* | `Abort()`; **return** \perp

end

i \leftarrow `Search(node, key)`

if *node*.isLeaf **then**

if *i* \neq 0 \wedge *node*.keys[*i*] = *key* **then**

 | **return** *node*.values[*i*]

else

 | **return** \perp

end

else

n \leftarrow *node*.children[*i*]

* | **return** `LookupHelper(n, key, node.depth)`

end

Function Delete(*key*)

Input: *key* – key to delete

Output: value corresponding to *key*, or \perp if not found

* *rootNum* \leftarrow `getRoot()`

* (*ret*, *modified*, *root*) \leftarrow `DeleteHelper(rootNum, key, -1)`

if *root* has exactly one child – an internal node with ID *c* **then**

 | *child* \leftarrow `Read(c)`

* | `Write(rootNum, child)`

* | `Free(c)`

else if *modified* **then**

* | `Write(rootNum, root)`

end

return *ret*

Function DeleteHelper(n, key, d)

Input: n – node ID**Input:** key – key to delete**Input:** d – depth of previous node visited**Output:** tuple ($ret, modified, node$) where ret is the value for key (or \perp if no such key),
 $modified = \text{true}$ iff node n has changed, and $node$ is node n

```
*  $node \leftarrow \text{Read}(n)$ 
* if  $node = \perp \vee node.depth \leq d$  then
* | Abort (); return  $\perp$ 
end
 $i \leftarrow \text{Search}(node, key)$ 
if  $node.isLeaf$  then
| if  $i \neq 0 \wedge node.keys[i] = key$  then
| |  $ret \leftarrow node.values[i]$ 
| | remove  $key$  from  $node$ 
| | return ( $ret, \text{true}, node$ )
| else
| | return ( $\perp, \text{false}, node$ )
| end
else
 $c \leftarrow node.children[i]$ 
* ( $ret, modified, child$ )  $\leftarrow \text{DeleteHelper}(c, key, value, node.depth)$ 
if  $child$  has too few keys then
| find node ID  $c'$  of a sibling of  $c$  in  $node$ 
*  $child' \leftarrow \text{Read}(c')$ 
* if  $child' = \perp \vee child'.depth \leq d$  then
* | Abort (); return  $\perp$ 
| end
| if  $child'$  has enough keys then
| | move some elements from  $child'$  to  $child$ , update  $node$  as needed
* | Write( $c, child$ )
* | Write( $c', child'$ )
| else
| | move all elements from  $child$  to  $child'$ , update  $node$  as needed
* | Free( $c$ )
* | Write( $c', child'$ )
| end
| return ( $ret, \text{true}, node$ )
else if  $modified$  then
* | Write( $c, child$ )
| return ( $ret, \text{false}, node$ )
end
end
end
```

C Implementation of node allocator

In this section we present the implementations of the Alloc and Free functions, which are use to allocate and deallocate tree nodes.

Function Alloc

Output: node ID of allocated node, or \perp if out of memory

$s \leftarrow \text{chooseServer}()$

$i \leftarrow$ object ID for allocator bit vector on server s

$vect \leftarrow \text{Read}(i)$

if $\exists j : vect[j] = 0$ **then**

$vect[j] \leftarrow 1$

$i \leftarrow$ object ID for entry j of allocator bit vector on server s

 Write($i, vect[j]$)

return node ID corresponding to s and j

else

return \perp

end

Function Free(n)

Input: n – node ID of allocated node

Output: OK on success, \perp if node was not allocated previously

$(s, j) \leftarrow$ server address and bit vector index corresponding to node n

$i \leftarrow$ object ID for allocator bit vector on server s

$vect \leftarrow \text{Read}(i)$

if $vect[j] = 0$ **then**

 Abort

return \perp

else

$vect[j] \leftarrow 0$

$i \leftarrow$ object ID for entry j of allocator bit vector on server s

 Write($i, vect[j]$)

return OK

end
