A Practical Scalable Distributed B-Tree

CS 848 Paper Presentation

Marcos K. Aguilera, Wojciech Golab,
Mehul A. Shah

PVLDB '08

March 8, 2010

Presenter: Evguenia (Elmi) Eflov

Presentation Outline

Distributed (key,value) Storage

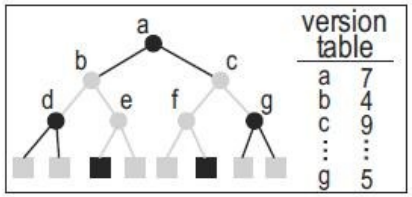
The paper presents three motivating examples:

- **The back-end of a multiplayer game.** Multiplayer games need to store and manage data for thousands of players while providing low latency access and very high data consistency
- **Metadata storage for a cluster file system.** Metadata access is often the bottleneck in such systems. Metadata changes, for example, file renaming or relocation, in cluster file systems need to be atomic
- **Secondary indexes.** A lot of application require more than one index on a set of data to guarantee fast access based on different conditions. As in the two previous examples, data changes need to be atomic

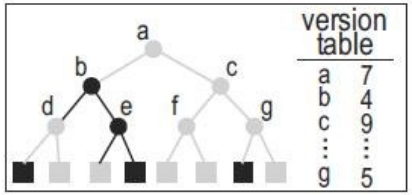
B+tree

- B-tree is a tree data structure that stores values sorted by key and allows updates and lookups in amortized logarithmic time
- B+tree is a form of B-tree where inner nodes of the tree store keys and pointers, and leaf nodes store key-value pairs

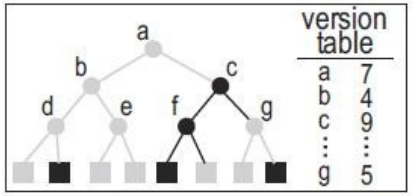
Distributed B+tree



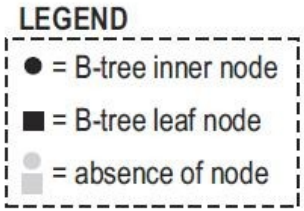
server 1 (memory node 1)



server 2 (memory node 2)



server 3 (memory node 3)



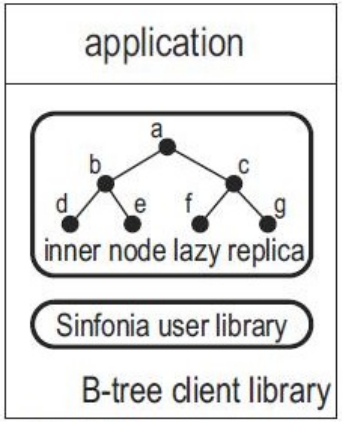
Sinfonia

- Sinfonia is a distributed data storage service that provides ACID properties for the application data
- Sinfonia provides a data manipulation primitive, a minitransaction
- Minitransaction ...
  - Consists of 3 (possibly empty) sets of operations
  - Operations are comparisons, reads, and writes
  - Reads and writes are performed only if all of the comparisons are successful
  - Is performed as part of a two-phase commit
  - Some varieties can be performed in a single phase

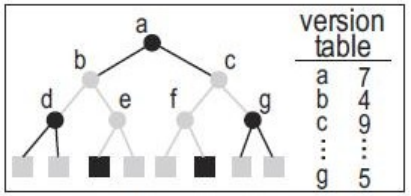
Assumptions

- The B-tree operates in a data center environment. This guarantees high bandwidth, low latency connections between client and server machines
- Individual machines can fail without causing the system to stall, but network partitions will stall the system
- B-tree is not going to grow or shrink rapidly

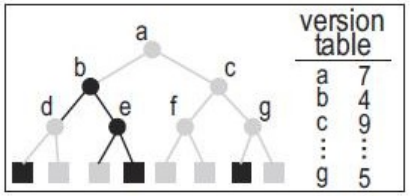
Design of the B-tree



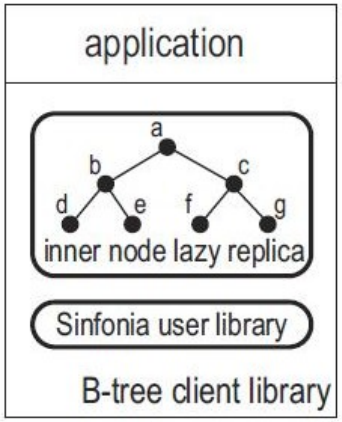
client 1



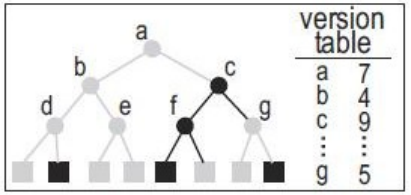
server 1 (memory node 1)



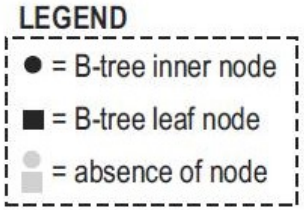
server 2 (memory node 2)



client 2



server 3 (memory node 3)



Design of the B-tree

- Each server in the system stores some number of inner and leaf nodes of the B-tree
- Each server in the system stores the version table of all the inner nodes of the B-tree
- Each client caches all inner nodes of the B-tree, and uses this cache while executing a transaction
- During a transaction, the client composes a set of reads and writes required
- At commit time, Sinfonia's minitransaction is used to perform the B-tree operations required on the server data
- Comparisons are added by the B-tree client library to guarantee data consistency

B-tree Operations Efficiency

To make the B-tree efficient, the following three techniques are used:

- Clients use optimistic concurrency control, which works well unless the B-tree is rapidly shrinking or growing
- Since version numbers of the inner nodes are stored at each server, inner node versions can be checked at any server in the system, for example, at the server where a leaf node being accessed is stored
- Inner B-tree nodes are lazily replicated by clients - nodes that a particular client does not access may be stale or not present on the client

Standard B-tree Operations

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.

Migration Operations

The distributed B-tree supports the following additional operations:

- *Migrate*(x, s) - migrates node B-tree node x to server s
- The following operations for multi-node migration:

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.

Why are transactions required?

- In order to guarantee data consistency, each data manipulation on the B-tree has to be performed atomically, for example, renaming of a file in the cluster file system or transferring an item and payment for the item between characters in the computer game
- While a minitransaction provided by Sinfonia is sufficient to perform the necessary B-tree node manipulations, it is tedious of the user of the B-tree to code in terms of the minitransaction
- The B-tree provides transaction interface as a way for the user to define all the necessary *Read* and *Write* operations within a transaction, while adding necessary comparisons to guarantee data consistency

Transaction Interface

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.

Extensions

The following extensions are suggested to enhance the existing implementation

- **Enhanced migration tasks** - migration tasks that help the system adapt to seasonal variations or balance the load aggressively can be implemented
- **Dealing with hot-spots** - migration task to migrate popular keys to different servers can be implemented, including migration of the keys that are currently stored in the same node

(continued)

Extensions

- **Varying the replication factor of inner nodes** - replicating version numbers of the inner nodes on lower levels of the tree less aggressively could decrease the cost of modifying those nodes
- **Finer-grained concurrency control to avoid false sharing** - if concurrency control operated on keys (or small groups of keys) rather than nodes, the number of conflicts could be decreased

Experimental Setup

- 10-byte keys, 8-byte values, 12-byte pointers - 4 bytes specify server, 8 bytes - offset within the server
- 4 KB nodes, with leaf nodes storing 220 key-value pairs and inner nodes storing 180 key-pointer pairs
- Same number of servers and clients
- Each client has 4 parallel threads, each thread issues a new request as soon as the current request is completed
- Key space consists of 10^9 elements, with keys chosen uniformly, at random for each operation

Workloads

The following workloads are used in both scalability and migration experiments

- Insert
- Lookup
- Update - values for existing keys are updated
- Mixed - 60% lookups and 40% updates
- "Before the insert workload, the B-tree was pre-populated with 40,000 elements rather than starting with an empty B-tree."
- Were the experiments performed in the order they are presented in?

Results of the Scalability Experiments

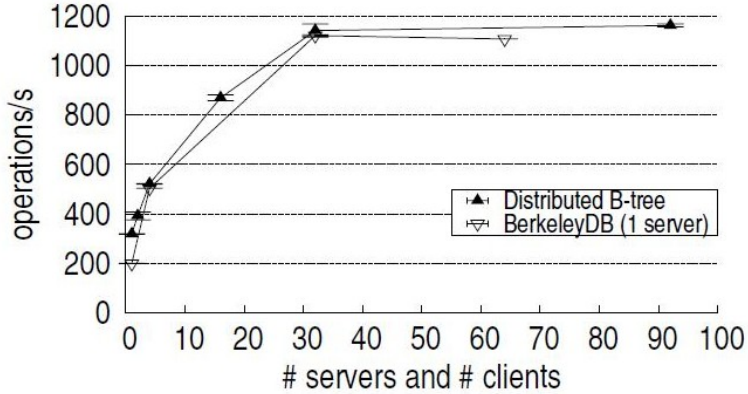


Figure 15: Aggregate throughput, insert workload.

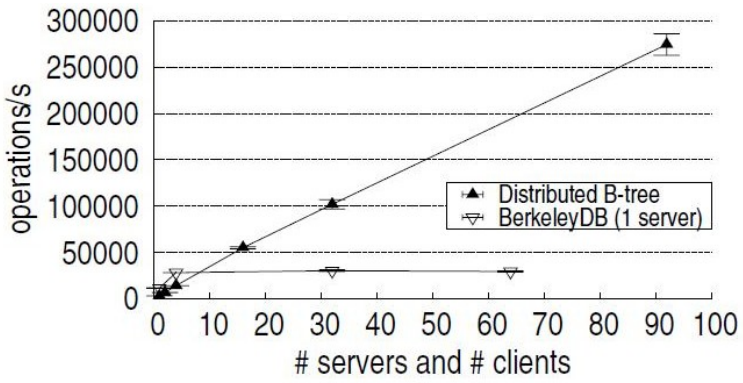


Figure 12: Aggregate throughput, lookup workload.

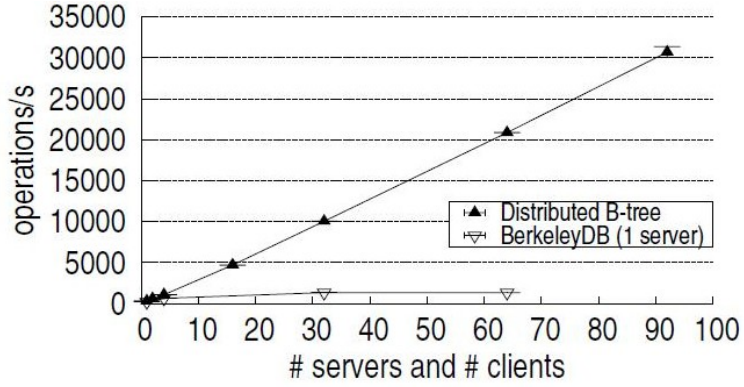


Figure 13: Aggregate throughput, update workload.

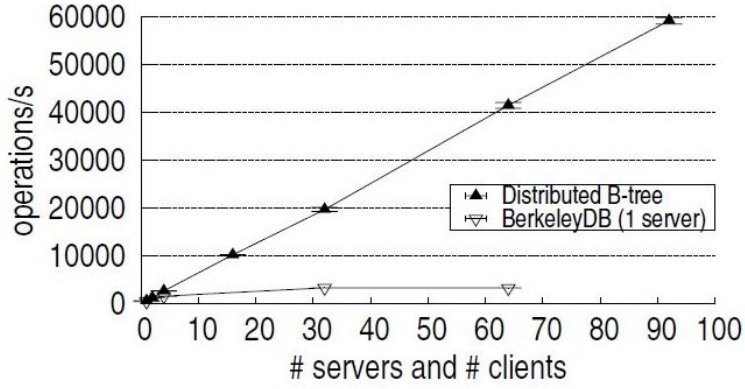


Figure 14: Aggregate throughput, mixed workload.

Results of the Migration Experiments

For migration experiments, *Move* task was performed by a *migration* client while the rest of the setup for the corresponding experiment was executed

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.

Migration rate was 55.3 ± 2.7 nodes/s (around 10000 key-value pairs/s) on an idle system, and less than 5 nodes/s when executed with other tasks.

Some Discussion Questions

- Are experiments representative of the workload of the motivating examples?
- Would larger transactions have different scalability?
- Can co-locating of the lower level inner nodes and the corresponding leaf nodes increase the throughput of the system?