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

1. Background
   - Problem
   - Distributed B+tree
   - Sinfonia

2. Distributed B-tree Implementation
   - Assumptions
   - Design of the B-tree
   - B-tree Operations
   - Transactions
   - Extensions

3. Experimental Results
   - Workload
   - Results

4. Discussion
   - Questions
1 Background
- Problem
  - Distributed B+tree
  - Sinfonia

2 Distributed B-tree Implementation
- Assumptions
- Design of the B-tree
- B-tree Operations
- Transactions
- Extensions

3 Experimental Results
- Workload
- Results

4 Discussion
- Questions
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.
1 Background
   - Problem
   - Distributed B+tree
   - Sinfonia

2 Distributed B-tree Implementation
   - Assumptions
   - Design of the B-tree
   - B-tree Operations
   - Transactions
   - Extensions

3 Experimental Results
   - Workload
   - Results

4 Discussion
   - Questions
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)

LEGEND

- = B-tree inner node
■ = B-tree leaf node
= absence of node
1 Background
   - Problem
   - Distributed B+tree
   - Sinfonia

2 Distributed B-tree Implementation
   - Assumptions
   - Design of the B-tree
   - B-tree Operations
   - Transactions
   - Extensions

3 Experimental Results
   - Workload
   - Results

4 Discussion
   - Questions
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
1. Background
   - Problem
   - Distributed B+tree
   - Sinfonia

2. Distributed B-tree Implementation
   - Assumptions
   - Design of the B-tree
   - B-tree Operations
   - Transactions
   - Extensions

3. Experimental Results
   - Workload
   - Results

4. Discussion
   - Questions
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.
1 Background
   ■ Problem
   ■ Distributed B+tree
   ■ Sinfonia

2 Distributed B-tree Implementation
   ■ Assumptions
   ■ Design of the B-tree
   ■ B-tree Operations
   ■ Transactions
   ■ Extensions

3 Experimental Results
   ■ Workload
   ■ Results

4 Discussion
   ■ Questions
Design of the B-tree

CLIENT 1
- Application
- Inner node: lazy replica
- Sinfonia user library
- B-tree client library

SERVER 1 (MEMORY NODE 1)
- Version table:
  - a: 7
  - b: 4
  - c: 9
  - g: 5

SERVER 2 (MEMORY NODE 2)
- Version table:
  - a: 7
  - b: 4
  - c: 9
  - g: 5

SERVER 3 (MEMORY NODE 3)
- Version table:
  - a: 7
  - b: 4
  - c: 9
  - g: 5

LEGEND
- ● = B-tree inner node
- ■ = B-tree leaf node
- ○ = absence of node
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.
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.
1. **Background**
   - Problem
   - Distributed B-tree
   - Sinfonia

2. **Distributed B-tree Implementation**
   - Assumptions
   - Design of the B-tree
   - B-tree Operations
   - Transactions
   - Extensions

3. **Experimental Results**
   - Workload
   - Results

4. **Discussion**
   - Questions
### Standard B-tree Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup (k)</td>
<td>return (v) s.t. ((k, v) \in B), or error if none</td>
</tr>
<tr>
<td>Update ((k, v))</td>
<td>if ((k, \ast) \in B) then replace it with ((k, v)) else error</td>
</tr>
<tr>
<td>Insert ((k, v))</td>
<td>add ((k, v)) to (B) if no ((k, \ast) \in B), else Update ((k, v))</td>
</tr>
<tr>
<td>Delete (k)</td>
<td>delete ((k, v)) from (B) for (v) s.t. ((k, v) \in B), or error if none</td>
</tr>
<tr>
<td>GetNext (k)</td>
<td>return smallest (k' &gt; k) s.t. ((k', \ast) \in B), or error if none</td>
</tr>
<tr>
<td>GetPrev (k)</td>
<td>return largest (k' &lt; k) s.t. ((k', \ast) \in B), or error if none</td>
</tr>
</tbody>
</table>

**Figure 3:** Operations on a B-tree \(B\).
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:

<table>
<thead>
<tr>
<th>Migrate task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migrate-away</td>
<td>migrate all nodes at server (x) to other servers.</td>
</tr>
<tr>
<td>Populate</td>
<td>migrate some nodes from other servers to server (x).</td>
</tr>
<tr>
<td>Move</td>
<td>migrate all nodes from server (x) to server (y).</td>
</tr>
<tr>
<td>Even-out-storage</td>
<td>migrate some nodes from more full to less full servers.</td>
</tr>
<tr>
<td>Even-out-load</td>
<td>migrate some nodes from more busy to less busy servers.</td>
</tr>
</tbody>
</table>

*Figure 7: Migration tasks on a B-tree.*
1 Background
   ■ Problem
   ■ Distributed B+tree
   ■ Sinfonia

2 Distributed B-tree Implementation
   ■ Assumptions
   ■ Design of the B-tree
   ■ B-tree Operations
   ■ Transactions
   ■ Extensions

3 Experimental Results
   ■ Workload
   ■ Results

4 Discussion
   ■ Questions
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.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginTx()</td>
<td>clear read and write sets, return transaction handle</td>
</tr>
<tr>
<td>Read(txn, n)</td>
<td>read object ( n ) locally or from server and add ( (n, val) ) to read set</td>
</tr>
<tr>
<td>Write(txn, n, val)</td>
<td>add ( (n, val) ) to write set</td>
</tr>
<tr>
<td>Commit(txn)</td>
<td>try to commit transaction</td>
</tr>
<tr>
<td>Abort(txn)</td>
<td>abort transaction</td>
</tr>
<tr>
<td>IsAborted(txn)</td>
<td>check if transaction has aborted</td>
</tr>
<tr>
<td>EndTx(txn)</td>
<td>garbage collect transaction structures</td>
</tr>
</tbody>
</table>

**Figure 9: Interface to transactions.**
1 Background
   - Problem
   - Distributed B+tree
   - Sinfonia

2 Distributed B-tree Implementation
   - Assumptions
   - Design of the B-tree
   - B-tree Operations
   - Transactions
   - Extensions

3 Experimental Results
   - Workload
   - Results

4 Discussion
   - Questions
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)
• **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.
1 Background
   - Problem
   - Distributed B+tree
   - Sinfonia

2 Distributed B-tree Implementation
   - Assumptions
   - Design of the B-tree
   - B-tree Operations
   - Transactions
   - Extensions

3 Experimental Results
   - Workload
   - Results

4 Discussion
   - Questions
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
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?
1 Background
- Problem
- Distributed B+tree
- Sinfonia

2 Distributed B-tree Implementation
- Assumptions
- Design of the B-tree
- B-tree Operations
- Transactions
- Extensions

3 Experimental Results
- Workload
- Results

4 Discussion
- Questions
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.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Throughput without migration (operations/s)</th>
<th>Throughput with migration (operations/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>870 ± 12</td>
<td>842 ± 32</td>
</tr>
<tr>
<td>Lookup</td>
<td>55422 ± 1097</td>
<td>55613 ± 1243</td>
</tr>
<tr>
<td>Update</td>
<td>4706 ± 18</td>
<td>4662 ± 19</td>
</tr>
<tr>
<td>Mixed</td>
<td>10273 ± 70</td>
<td>10988 ± 109</td>
</tr>
</tbody>
</table>

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.
1 Background
   - Problem
   - Distributed B+tree
   - Sinfonia

2 Distributed B-tree Implementation
   - Assumptions
   - Design of the B-tree
   - B-tree Operations
   - Transactions
   - Extensions

3 Experimental Results
   - Workload
   - Results

4 Discussion
   - Questions
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?