Technical Report RT/24/2013

STI-BT: A Scalable Transactional Index

Nuno Diegues
INESC-ID/IST
ndiegues@gsd.inesc-id.pt

Paolo Romano
INESC-ID/IST
romano@inesc-id.pt

September 2013

Abstract

In this article we present STI-BT, a highly scalable, transactional index for Distributed Key-Value (DKV) stores. STI-BT is organized as a distributed B+Tree and adopts an innovative design that allows to achieve high efficiency in large-scale, elastic DKV stores. We have implemented STI-BT on top of a mainstream open-source DKV store and deployed it on a public cloud infra-structure. Our extensive experimental study reveals the efficiency of our solution with demonstrable scalability in a cluster of 100 commodity machines, and speed ups with respect to state of the art solutions of up to 5.4x.

Keywords: Data locality, Distribution, B$^+$Tree, Strong consistency

Technical Report RT/24/2013[1]

STI-BT: A Scalable Transactional Index

Nuno Diegues   Paolo Romano
INESC-ID/IST   INESC-ID/IST
ndiegues@gsd.inesc-id.pt   romano@inesc-id.pt

1. Introduction

The ever growing need for computational power and storage capacity has fostered research of alternative solutions to classic monolithic relational databases. In this context, Distributed Key-Value (DKV) stores (such as Dynamo [8]) became quite popular due to their scalability, fault-tolerance and elasticity. On the down side, developing applications using these DKV stores is far from trivial, due to two main factors: the adoption of weak consistency models and of simplistic primitives to access data.

Concerning data consistency, the inherent complexity of building applications on top of weakly consistent systems has motivated a flurry of works offering strongly consistent transactional semantics in large scale platforms [4, 6, 22, 23, 25, 28].

On the other hand, the lack of indexing support for non-primary data attributes forces programmers to either implement ad-hoc indexing strategies at the application level, or to rely on asynchronous indexing solutions [3, 19]. Neither approach is desirable, as the former is costly and error-prone, whereas the latter ensures only weak consistency.

In this paper we tackle this issue by introducing STI-BT, a transactional index designed to match the scalability and elasticity requirements of DKV stores. STI-BT is organized as a distributed B$^+$Tree and adopts a unique design that leverages on the following mechanisms in a synergic way:

▷ *Data placement and data-driven transaction migration:* a unique feature of STI-BT is that accesses to any indexed data item require normally only one remote access, regardless of the size of the index and of the number of machines in the system. This is achieved via the exploitation of data placement and transaction migration techniques, which are jointly leveraged to maximize data locality. STI-BT partitions the index into sub-trees that are distributed to different machines via lightweight data placement techniques based on custom consistent hashing [16] schemes. Transaction migration is used to relocate the execution of an operation to a remote machine when the data required is not available locally. This results in a drastic minimization of communication, enhancing the efficiency and scalability of the index.

▷ *Hybrid replication*: STI-BT adapts the number of copies of the tree nodes according to their depth in the tree: top levels are fully replicated, whereas lower levels are partially replicated over a small set of machines. This brings a number of advantages. The nodes in the top of a B$^+$Tree, despite representing a minimum fraction of the tree, account for a large part of the read accesses. By fully replicating them, STI-BT avoids the load unbalance that would otherwise occur if they were replicated only in a small set of machines. Also, top level nodes have the lowest probability of being updated, which makes the cost of maintaining them fully replicated negligible even in large clusters. Conversely, using partial

[1] This work was supported by national funds through FCT (Fundação para a Ciência e Tecnologia) under project PEst-OE/EEI/LA0021/2013 and by projects Cloud-TM (257784), specSTM (PTDC/EIA-EIA/122785/2010), and GreenTM (EXPL/EEI-ESS/0361/2013).

replication for the lower levels (the majority of the data) ensures fault-tolerance, maximizes efficiency of storage, and keeps the cost of updating the index bounded at any scale of the system.

▷ *Elastic scaling:* The ability to adjust the resources employed to the actual demand is one of the most attractive features of DKV stores. For this reason, STI-BT was tailored to ensure efficiency in the presence of shifts of the platform's scale. It reacts to changes by autonomously adjusting the boundaries of the fully replicated part, with the intent of minimizing it, and redistributing the index across the cluster.

▷ *Concurrency enhancing mechanisms:* STI-BT combines a number of mechanisms to minimize data contention. STI-BT is built on top of GMU [22], a recent distributed multi-versioning scheme that executes read-only transactions in a wait-free fashion, hence allowing for executing lookup operations highly efficiently. To maximize scalability also in conflict-prone workloads, we used algorithms to navigate and manipulate the index that exploit concurrency-enhancing programming abstractions [10, 14] and drastically reduce the conflicts among concurrent index operations.

We have built STI-BT on top of Infinispan, a mainstream open-source transactional DKV store from Red Hat. We conducted an extensive experimental study, deploying STI-BT in a large scale public cloud infrastructure (up to 100 machines) using benchmarks representative of OLTP workloads. The results highlight the high scalability of the proposed solution, which can achieve up to 5.4× speedups over state of the art solutions.

The rest of the paper is organized as follows. In Section 2. we discuss related work. Section 3. introduces background concepts and assumptions. Section 4. overviews our solution, of which we provide a more detailed description through Sections 5.-8.. Finally, we present our evaluation in Section 9., and conclude in Section 10..

2. Related Work

The development of B$^+$Trees for indexing data constitutes a large body of work. One traditional usage targets centralized systems with persistent storage in disk [13, 18], whereas others have targeted distributed environments [20] (although without allowing atomic accesses to data items).

A Scalable BTree was proposed in [1] to index large-scale data transactionally. Accesses to the tree are governed by Sinfonia [2], a distributed abstraction of shared memory with serializable transactions. However the main drawback is the focus on queries alone, for instance by fully replicating every tree node (except leafs). Thus, its performance lacks scalability under mixed workloads where the data is updated.

Another design of a distributed B$^+$Tree was proposed in Minuet [26] (also on top of Sinfonia). Similarly to STI-BT, Minuet also exploits multi-versioning to enhance concurrency between transactions. Yet, Minuet handles multi-versioning externally to Sinfonia, by using a centralized snapshot identifier that is incremented whenever a read-only transaction requires a fresh snapshot. As we shall discuss in the next Section, our solution uses a scalable distributed multi-versioning scheme [22], which provides significant benefits for read-only transactions. Furthermore, Minuet distributes the tree nodes randomly across the Sinfonia cluster. In STI-BT, we exploit the structure of the tree to co-locate tree nodes and maximize data locality. Finally, we extend their usage of Dirty Reads (initially promulgated in [14]) to cope with Delayed Actions [10] for reduced concurrency conflicts in transactions.

The solution for large-scale data indexing presented in Global Index [30] also proposes to use tree structures, albeit in a different way. Every machine maintains the local data indexed in a local B$^+$Tree and chooses only a subset of nodes to publish — publishing means adding the tree nodes to the Global Index, which is a BTree built using an peer-to-peer overlay network called BATON [15]. The published nodes of some machine reflect an over-estimation of the data indexed at that machine, with the objective to reduce the frequency of expensive global synchronization upon updates of the index. The downside of this design is that, due to the inaccuracy of the published information, queries typically contact unnecessary machines, which can hinder performance. STI-BT's design, conversely, ensures that at most a remote machine is contacted to process an index operation, except for the rare case of concurrent rebalances affecting regions of the tree accessed by the operation. Another fundamental difference is that Global Index only ensures eventual consistency across replicas, whereas STI-BT ensures strong consistency.

Other work has focused on indexing multi-dimensional data. The key idea of Global Index has been adapted to this purpose [29]: each machine indexes its local data in an R-tree to account for the multi-dimensions. Once again, the locality-efficient design contributions of STI-BT could be used in the distributed index that is built on top of these R-Trees. Other approaches to this topic rely also on distributed data-structures (such as the SkipTree [3]), on space-filling curves (as on Squid [24]), or on hyperspace hashing (as on HyperDex [12]). Many of these techniques have been developed in the context of Peer-to-Peer (P2P) network overlays — we refer to [31] for a recent survey on this topic. Traditionally, these systems were designed for environments with limited synchrony and high churn, as typical of large networks over the Internet. As such, they provide weak or no consistency guarantees at all, much less transaction abstractions that allow to atomically access and modify multiple data items. In contrast, STI-BT ensures strong consistency (without compromising scalability), and is designed to operate in clusters of cloud machines, which are more stable than typical P2P systems.

3. Background and Assumptions

Like other modern cloud data platforms [2, 6], Infinispan (our underlying DKV store) also supports transactional strong consistency. In particular, Infinispan integrates the GMU protocol [22], a strongly consistent transactional replication protocol that relies on a fully decentralized multi-versioning scheme. Unlike classic solutions, GMU does not rely on a global logical clock, which may represent a contention point and impair scalability. Conversely, GMU determines version's visibility and transaction serialization order by means of vector clocks, which are updated *genuinely* (i.e., contacting only the machines involved in the execution of the transaction) and hence in a highly scalable fashion. Also, GMU never aborts read-only transactions and spares them from the costs of distributed validation, maximizing the efficiency of read-intensive workloads.

Concerning data placement, Infinispan relies on consistent hashing [16] to determine the placement of data (similarly to Dynamo [8], for instance). Consistent hashing is particularly attractive in large scale, elastic stores, as it avoids reliance on external lookup services, and allows to minimize the keys to be redistributed upon changes of the system's scale. However, deterministic random hash functions to map keys identifiers to machines lead to poor data locality [21, 27]. Hence, DKV stores using consistent hashing (like Infinispan) typically allow programmers to provide custom functions. As we will discuss in Section 5., STI-BT relies heavily on novel, custom data placement strategies layered on top of consistent hashing to enhance data locality.

Finally, STI-BT relies on the structure of a B$^+$Tree, where leaf nodes are connected to allow for in-order traversal and inner nodes do not contain values held by the index. We highlight Table 1 describing the symbols in our terminology.

Table 1. Description of terminology used through the paper.

symbol	description
α	arity of a tree node
$\mathcal{C}$	cut-off level
$\mathcal{M}$	memory available in a machine
$\mathcal{N}$	number of machines in the cluster
$\mathcal{K}$	degree of partial replication

4. Design Rationale and Overview

One of the main goals of STI-BT's design is to maximize data locality, i.e., to minimize the number of remote data accesses required to execute any index operation. This is a property of paramount importance not only to ensure efficiency, but also scalability. In fact, in solutions based on simplistic random data placement strategies, the probability of accessing data stored locally is inversely proportional to the number of machines. Hence, as the system scales, the network traffic generated by remote accesses grows accordingly, severely hindering scalability.

Data locality may be achieved using full replication, but that would constrain scalability from a twofold perspective. First, the cost of propagating updates (to all machines) grows with the size of the cluster. Second, it prevents scaling out the storage capacity of the system by adding machines.

We also highlight that partial replication techniques pose challenges to load balancing. Consider a simple approach in which each tree node is replicated across $\mathcal{K} = f + 1$ machines, to tolerate up to f faults in the cluster. Since the likelihood of accessing a tree node is inversely proportional to its depth in the tree, the machines maintaining the topmost tree nodes will receive a larger flow of remote data accesses, hence becoming bottlenecks of the system.

To cope with the issues mentioned above, STI-BT divides the B$^+$Tree in two parts: the topmost $\mathcal{C} - 1$ levels are fully replicated, whereas the bottom part, containing the $\mathcal{C}^{th}$ level downwards, is partially replicated. Here, $\mathcal{C}$ represents the *cut-off level*, i.e., the depth level of the tree where the nodes are no longer fully replicated. As we will discuss, $\mathcal{C}$ is a dynamic value, which is adjusted when the scale of the platform is altered. Hence, $\mathcal{C}$ is stored in a fully replicated key of the underlying key-value store, which ensures that its value can be known by any machine with transactional consistency. In addition to this, the bottom part of the tree is organized in co-located sub-trees. In Fig. 1 we can see the index distributed among 3 machines and the cut-off (we use $\mathcal{K} = 1$ in the example).

The main advantage of this design is that, traversing down the tree at any given machine can, at most, incur in one remote access to another machine. If a traversal at machine M_1 reaches level $\mathcal{C}$, and requires data replicated at machine M_3, we take advantage of the co-location within each sub-tree to forward the execution flow of the transaction from M_1 to M_3. The B$^+$Tree supports range searches by traversing down to the leaf holding the initial value of the interval of the search, and then following the pointers to the next leaf. Hence, we try to replicate neighbour sub-trees in the same machine, which also helps to minimize the communication required for range searches. This locality-aware design results in a uniform load of the machines if the popularity of accesses is uniform across the indexed data.

STI-BT integrates also a set of mechanisms to minimize the likelihood of contention among concurrent operations on the index. The key idea is to address the two possible sources of contention in a B$^+$Tree— structural changes of the tree topology and modifications of the node's contents — by exploiting the

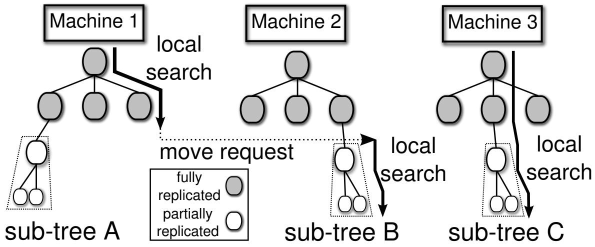


Figure 1. Example of communication flow in STI-BT.

commutativity of the various operations supported by the index. This allows achieving high efficiency even in challenging update-intensive scenarios.

Another innovative algorithmic aspect of STI-BT is the management of the cut-off level $\mathcal{C}$. This is governed by two contradicting forces: (1) we aim to maintain the fully replicated part as small as possible, to ensure that rebalances are less likely to update inner nodes that are fully replicated; and (2) we need the cut-off level to be deep enough so that it contains enough tree nodes at that level to serve as sub-tree roots to load-balance between all the machines in the cluster. Based on these considerations, STI-BT reacts to the elastic scaling of the underlying DKV by adapting the number of fully replicated nodes in the distributed B$^+$Tree.

5. Maximizing Data Locality

We now discuss in more detail how to achieve the co-location and forwarding of execution flow. We begin with the example in Fig. 1 with the flow of two index operations, one in M_1 and the other in M_3, each corresponding to a transaction being processed at each machine.

The transaction at M_1 accesses the index to obtain a data element in the sub-tree B. This sub-tree is stored at M_2, for which reason the traversal cannot be processed only with the data stored at M_1. To minimize communication, when the traversal reaches the level of depth $\mathcal{C}$ we move the request to M_2 and continue the traversal there. The design of STI-BTensures that this can happen only once while traversing the tree, which is optimal except when the request can be processed only with local data (as for the transaction at M_3).

Underlying this solution is the ability to co-locate data. For this, we still rely on the underlying consistent hashing from the DKV store, but extend it by means of a custom hashing scheme. To this end, we encode two parts in the identifier of each key k (that maps to a tree node), i.e., $k = \langle k_u, k_{cl} \rangle$: a unique identifier ($k_u$), which identifies the node of the B$^+$Tree, and a co-locality identifier (k_{cl}), which is used to ensure co-location of different unique identifiers. STI-BT hashes only k_{cl} when performing machines lookup for a key k, and uses k_u when conducting local queries within a given machine. As a result, two different keys, sharing the co-location identifier k_{cl}, are hashed to the same machine. We exploit this by assigning the same k_{cl} to all keys used to maintain the contents of a given sub-tree, which results in co-locating its tree nodes on a set of $\mathcal{K}$ machines determined via consistent hashing.

We also exploit the underlying DKV store's consistent hashing to govern the execution flow of transactions. To better present this idea, we shall rely on Algorithm 1. For simplicity, we omit the management of the DKV store when possible. Also, for the moment it suffices to consider that each tree node is mapped into a single key/value in the DKV store — we extend that in Section 7.. Finally, we use a generic function portraying the role of operations accessing the tree (such as an insertion or range query), with focus on the common part of traversing down the tree.

Algorithm 1 Execution flow.

```
 1: struct TreeNode
 2:   bool isLeaf                                  ▷ if false, then right/left siblings are null
 3:   List subNodes                                                     ▷ sorted children nodes
 4:   TreeNode* parent, rightSibling, leftSibling

 5: ▷ called in the context of application's transactions
 6: function ACCESS(Transaction tx, B$^+$Tree tree, Obj o)
 7:   TreeNode node ← tree.getGlobalRoot()
 8:   while $node.k_u$.isFullyRepl()
 9:     node ← node.getSubNodes().chooseChild(o)
10:   if ownerOf($node.k_{cl}$) = localMachine
11:     return LOCALACCESS(node, o)
12:   else
13:     long[] vecClock ← tx.getSnapshotVC()
14:     send REQUEST[vecClock, LOCALACCESS(node, o)]
            to ownerOf($node.k_{cl}$)
15:     receive REPLY[vecClock, Obj result]
16:     tx.setSnapshotVC(vecClock)
17:     tx.addParticipant(ownerOf($node.k_{cl}$))
18:     return result

19: when receive REQUEST[long[] vc, Task function]
20:   Transaction tx ← startTx()
21:   tx.setSnapshotVC(vc)
22:   Obj result ← trigger function
23:   vc ← tx.getSnapshotVC()
24:   tx.suspend()
25:   send REPLY[vc, result]

26: function LOCALACCESS(TreeNode node, Object obj)
27:   ▷ conduct the local operation; rebalance if needed
```

We begin by considering a traversal in machine M_1 at a given tree node, as shown by the generic function in line 6. The traversal goes down the tree nodes as long as those data items are fully replicated (verified through the meta-data of the key that allows to access the tree node). When the next child tree node to traverse has a key that is partially replicated, one of two things can happen: (1) the key is locally replicated (the condition in line 10 is true), so the sub-tree is owned by M_1 and the operation is finished locally by calling function LOCALACCESS; or (2) the sub-tree is stored elsewhere, and so we decide to move the flow of execution (line 13 onwards). In the latter case, which can only occur at depth level $\mathcal{C}$ in our STI-BT, we create a task with the arguments of the access being performed in the tree. We then use the consistent hashing function on the k_{cl} identifier of the child's key to obtain the set of machines that replicate that data, and send the task to a random node in that set, which in our example is M_2 (line 14).

In fact, due to the transactional context under which these executions occur, we must send additional meta-data: the transaction that is executing at M_1 has necessarily performed some reads that restrict the snapshot of data that is visible to the transaction. This information is encoded via a vector clock in GMU (see Section 3.). Thus we retrieve the current transaction's vector clock (line 13) and send it to M_2, which answers back with a possibly updated vector clock reflecting any additional reads executed during execution at M_2. In practice, M_2 starts a new transaction that is forced to observe the snapshot used by transaction at M_1, thus guaranteeing consistency. No further meta-data is required at M_2 from M_1: namely, we avoid moving the buffered write-set along with the transaction. This is because, before moving to M_2, the traversal at M_1 would only have read tree nodes and necessarily not written to any. On top of this, the remote execution will only read contents of the B$^+$Tree (and not other data in the store), thus avoiding the possibility of a read-after-write. An exception to this occurs when multiple queries are invoked over the same tree in the course of a single transaction; in the event that these repeated invocations access the same parts of the tree, they will thus contact some previously contacted machines (say M). In such case, we note that the part of the write-set that is relevant is already available at M, because it was created there during the previous remote execution(s).

After the execution returns to the origin machine of the transaction (for instance M_1), the transactional context is also updated (in line 17) to contain a reference to the transaction that was issued at a remote machine (for instance M_2). This remote transaction was suspended before the execution flow returned to the origin machine, and is thus pending a final decision. We extended the underlying DKV store to consider these additional participants in the distributed commit procedure. This means that M_2 will be seen as a participant to the distributed transaction coordinated by M_1, just as if M_1 had invoked some remote accesses to M_2.

Finally, we abstracted away the details of the usual implementation of a B$^+$Tree in the generic function LOCALACCESS. Note that modifications, such as insertion or removal, may require rebalances due to splits or merges. In such case, the rebalance operation goes back up and modifies the inner nodes as required. The likelihood of changing an inner node is proportional to its depth. Thus, it is normally unlikely that a rebalance reaches the top part of our STI-BT, which is fully replicated and incurs larger update costs.

6. Load Balancing Sub-Trees

The algorithm described so far took advantage of the existence of sub-trees with co-located data in the DKV store. Due to that design, a machine replicating more sub-trees ends up receiving more requests from remote nodes. On top of this, memory constraints may also apply (we assume that each machine has limited $\mathcal{M}$ memory capacity). For these reasons, STI-BT integrates a load balancing scheme aimed to homogenize resource consumption across machines.

As the B$^+$Tree changes, rebalances occur and inner tree nodes are changed. In particular, at the $\mathcal{C}$ depth level, inner nodes may also change — we call the nodes at this level sub-tree roots because each one is a root of a sub-tree whose contents are all co-located. Conceptually, each machine has a list of sub-tree roots that is used to reason on the data of STI-BT that is stored there. We ensure that these lists (one per machine, per STI-BT) have balanced lengths in order to balance consumption of memory and processor (assuming that the popularity of indexed data is uniform).

To cope with load-balancing, each machine periodically triggers a routine to assess the number of sub-trees it currently owns. This procedure is executed under a transaction and it is re-attempted in case the transaction aborts due to a concurrency conflict.

The procedure, triggered at a given machine M_i, starts by assessing how many sub-trees the machine is responsible for. For this, it uses an array of all lists of sub-tree roots — one list per machine. It then computes the average size for all machines to assess the balance of the tree. We use a small tolerance value δ to avoid repeated migration of sub-trees between the same machines. If M_i does not have an excess of sub-trees with respect to the average, then the procedure concludes. Otherwise, it means we can enhance the load-balancing by having M_i offer some of its sub-trees to under-loaded machines: we change the keys of the tree nodes (being migrated) to the new k_{cl} (that maps to a new machine, according to the consistent hashing).

So far we assumed for simplicity that a single, coarse-grained transaction encapsulated the whole migration procedure. Instead, for instance, it is possible to remove a root from a list of sub-tree roots and insert it in the other list in different transactions, because only the machine who owns a given root is responsible to migrate it else where — no two machines will race to migrate a given sub-tree. Hence, when changing the tree nodes to replicate them elsewhere, one can use a transaction to encapsulate the changes of each tree node. This minimizes the likelihood of conflicts and, in such events, the work to be repeated. The only cost of this optimization is that concurrent data accesses may have to traverse a partially collocated subtree. Consequently, in such rare event, STI-BT performs two remote executions when traversing down the tree.

7. Minimizing data Contention

In order to minimize data contention among index operations, STI-BT relies on mechanisms aimed to avoid structural conflicts — i.e., allow traversals of the tree to be executed in parallel with tree rebalances, and to allow intra-node concurrency — i.e., concurrent updates of different key/values in a tree node. To implement these mechanisms, we leverage on two programming abstractions proposed to enhance concurrency in transactional systems: Dirty Reads [14] and Delayed Actions [10]. Dirty Reads instruct the underlying concurrency control to avoid validating a read operation executed by the transaction. Delayed Actions allow for postponing the execution of conflict-prone code portions until the transaction's commit phase, where they can be executed in a sequential and conflict-free fashion.

STI-BT uses Dirty Reads when traversing down inner nodes of the B$^+$Tree, and when navigating horizontally through the elements of a node. This allows for exploiting the commutativity among (update) operations that target different data items [17], allowing them to be successfully executed in parallel and avoiding unnecessary aborts (that would be triggered if plain reads were used). This technique brings additional benefits beyond minimizing data contention. Since the topmost part of STI-BT is fully replicated, if it did not use Dirty Reads to access fully replicated tree nodes, committing an update transaction would demand involving all machines in the cluster (to ensure consensus in validating the accesses to such nodes). The usage of Dirty Reads to access fully replicated nodes allows removing the corresponding keys from the transaction's read-set, significantly reducing network communication and the cost of maintaining the topmost part fully replicated.

Delayed Actions are used to avoid contention hotspots associated with the manipulation of the counters that maintain the number of elements stored by each tree node. Whenever an element is inserted to/removed from a node, the corresponding counter needs to be updated within the same transaction, and can become a contention point in conflict-prone workloads. To avoid this issue, we read this counter (to determine whether it is necessary to merge or split the node) using Dirty Reads, and update its value using a Delayed Action. As the latter takes place at commit time in an atomic step, this prevents that the update of a node's counter can ever cause the abort of the encompassing transaction. On the down side, this approach can lead to "missed" concurrent updates to a node's counter. This does not affect correctness but can cause an unbalance of the tree, which we detect and fix using a background thread in each machine that periodically checks the local sub-trees and rebalances them, if necessary. A detailed description of STI-BT's concurrency mechanisms has to be omitted for space constraints, but can be found in our technical report [11].

8. Elastic Scaling

In cloud environments, the provisioning process of a DKV store is typically governed by an autonomic manager, which takes into account a number of factors (e.g., current load, energy consumption, utilization of computational and storage resources) and aims to ensure Quality-of-Service levels while minimizing costs [7,9]. Regardless of the reasons that may determine a change in the machines of the cluster, STI-BT reacts by autonomously reconfiguring its structure (in particular, its cut-off level $\mathcal{C}$) to ensure optimal efficiency at any scale. If the cluster size changes, it is possible that the current $\mathcal{C}$ is not deep enough to contain enough sub-tree roots (at least one per machine). Conversely, upon a scale down, the sub-trees may exceed the actual need: this is also undesirable, as the shallower the cut-off, the less likely it is for an update to affect the fully replicated part.

Before presenting the details of the algorithm used to reconfigure $\mathcal{C}$, we first introduce an example to explain the high-level idea. Consider $\mathcal{C} = 2$ and $\alpha = 4$; then we have $\alpha^{\mathcal{C}} = 16$ sub-trees available (assuming $\mathcal{K} = 1$). Suppose that the cluster scales up and brings in a 17^{th} machine; then we cannot assign a sub-tree to the joining machine with the current $\mathcal{C}$. This motivates for deepening $\mathcal{C}$ (i.e., increment it) to create further sub-trees. Hence, by fully replicating $\alpha^{\mathcal{C}+1}$ inner nodes, we obtain $\alpha^{\mathcal{C}+1} = 64$ total sub-trees. Yet, we do not need to be so aggressive: in fact, we do not need so many new sub-trees as we only brought in one new machine. The penalty here is that we are increasing considerably (and unnecessarily) the fully replicated part of the tree in a situation where we only acquired little new resources. By considering only one sub-tree root node in the fully replicated part of the index, we can turn its α children into new sub-tree roots, which can be migrated to new machines joining the DKV store. Hence, we adapt $\mathcal{C}$ using a finer-grained, more efficient strategy: we fully replicate the minimum sub-tree root nodes at the current $\mathcal{C}$ to create as many additional roots as the joining machines.

Algorithm 2 describes this procedure, which is triggered whenever a change of the DKV's scale is detected. For ease of presentation, the pseudo-code considers that the cluster size increases (or decreases) one machine at a time. As explained above, $\mathcal{C}$ needs to be adapted only if STI-BT has an insufficient (line 33) or excessive (line 38) number of sub-trees with respect to the new cluster size $\mathcal{N}$.

To manage the cut-off, we maintain some meta-data in the DKV store to ensure its coherency across machines. This meta-data is used to decide which sub-tree root will be moved to/from the fully replicated part (in lines 34 and 39). We use a round-robin strategy to pick a sub-tree from a different machine each time this procedure is executed, and use meta-data r to keep track of the current round — the rounds count how many sub-trees of the current $\mathcal{C}$ have been lowered to $\mathcal{C} + 1$. Hence why we only update $\mathcal{C}$ sometimes: r consecutive growths (or shrinks) must occur before the full level is considered changed and the cut-off is changed.

The function ADJUSTCUTOFF is responsible for applying the cut-off change in the area of the tree corresponding to one sub-tree. When the objective is to `lower` the cut-off (due to the scale of the cluster increasing), this entails two things: (1) to make the current sub-tree root fully replicated, conceptually moving the cut-off to the next level in that part of the tree (line 48); and (2) to update the list of sub-tree roots that is used for load-balancing (lines 49-50). Lines 52-55 conduct a symmetric `raise` procedure. These nodes belong to the top part of the tree and, since we use Dirty Reads to avoid validating reads issued on inner nodes, it is very unlikely for this procedure to incur any conflict.

Finally, we assess the impact of $\mathcal{C}$ on memory efficiency. Since the nodes above $\mathcal{C}$ are fully replicated, the more machines there are, the larger the portion of memory each one has to allocate to hold the fully replicated nodes. This is an additional motivation for minimizing $\mathcal{C}$. We evaluate the memory capacity TC of STI-BT on a cluster of size $\mathcal{N}$: The equation above subtracts FR fully replicated nodes (each holding α keys of the DKV store) from the capacity of each machine. We can then evaluate the memory efficiency of STI-BT, noted η, as the ratio of its actual capacity to that of an ideal system whose total capacity scales perfectly with the number of machines (i.e., $TC = \mathcal{N} \times \mathcal{M}$): This analysis highlights the efficiency of STI-BT in large scale deployments (containing hundreds or thousands of

Algorithm 2 Scaling the cluster.

```
▷ triggered after the join/leave of one machine
function MANAGECUTOFF(B+Tree tree, int $\mathcal{N}$)
  ‹int, int› ‹$\mathcal{C}$, $r$› ← DKV.getCutoffInfo()
  int $m_{id}$ ← getMachineForRound($r$)                          ▷ round-robin
  int $numSubTrees$ ← totalSize(getRootsLists($tree$))
  if $\mathcal{N} > numSubTrees$                                  ▷ lower $\mathcal{C}$ level
    ADJUSTCUTOFF($tree$, $m_{id}$, lower)
    if $r = \alpha^{\mathcal{C}}$                                 ▷ lowered fully the current $\mathcal{C}$ level
      $\mathcal{C} \leftarrow \mathcal{C} + 1$; $r \leftarrow 0$
    else $r \leftarrow r + 1$
  else if $\mathcal{N} < (numSubTrees - \alpha + 1)$              ▷ raise $\mathcal{C}$ level
    ADJUSTCUTOFF($tree$, $m_{id}$, raise)
    if $r = 0$                                                    ▷ raised fully the current $\mathcal{C}$ level
      $\mathcal{C} \leftarrow \mathcal{C} - 1$; $r \leftarrow \mathcal{K} \times \alpha^{\mathcal{C}}$
    else $r \leftarrow r - 1$
  DKV.setCutoffInfo($\mathcal{C}, r$)                             ▷ update meta-data

function ADJUSTCUTOFF(B+Tree tree, int $m_{id}$, $op$)
  List $subTrees$ ← getAllRootsLists($tree$)[$m_{id}$]
  TreeNode $subTreeRoot$ ← $subTrees$.pickSubTree()
  if $op$ = lower
    $subTreeRoot.k_u$.setFullRepl(true)
    $mySubTrees$.remove($subTreeRoot$)
    $mySubTrees$.add($subTreeRoot$.getChildren())
  else
    $subTreeRoot.k_u$.setFullRepl(false)
    TreeNode $parent$ ← $subTreeRoot$.getParent()
    $subTrees$.remove($parent$.getSubNodes())
    $subTrees$.add($parent$)
```

$$TC = \mathcal{N} \times (\mathcal{M} - \alpha \times FR) \quad \text{where } FR = \sum_{i=0}^{\mathcal{C}-1} \alpha^i = \tfrac{\alpha^{\mathcal{C}}-1}{\alpha-1}$$

$$\eta = 1 - \tfrac{\alpha(\mathcal{N}-1)}{(\alpha-1)\mathcal{M}}$$

servers). In fact, even in such scenarios, it is realistic to assume that the number of keys held by each machine is much larger than the number of machines (i.e., $\mathcal{M} \gg \mathcal{N}$), yielding a memory efficiency very close to 1 for any, non-minimal value of α.

9. Experimental Evaluation

We developed a prototype of STI-BT on top of Infinispan, a popular in-memory transactional DKV developed by Red Hat. Each experiment uses $\mathcal{K} = 2$ for fault-tolerance, and the reported results represent the average over 10 runs. We use geometric mean for averages over normalized results. All tests were executed using $\alpha = 25$, unless specified otherwise. We conducted our tests on FutureGrid, a large scale public cloud infrastructure, from which we acquired a pool of up to 100 virtual machines equipped with 2 physical cores, 4GB of RAM and interconnected via InfiniBand. We also present experiments using a single many-core machine with 48 AMD Opteron cores at 2.1Ghz and 128GB RAM.

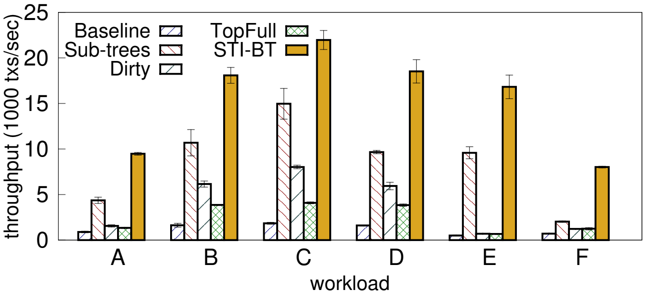


Figure 2. YCSB workloads with 60 machines.

9.1. YCSB Workloads

YCSB [5] is a popular benchmark for DKV stores, whose workloads comprehend single key operations (read, insert and modify) as well as range scans and read-modify-write operations, emulating data access patterns of real applications (typically skewed). We used strong scaling by always loading the index with 10GB of data drawn from a uniform distribution (our experiments evidenced that larger data-sets had no impact on the results). Finally we also scale the number of clients with the number of machines running the key-value store (co-located processes). To help understand the benefits of STI-BT, we created several B$^+$Tree variants:

▷ *Baseline*: a B$^+$Tree built on top of Infinispan without any of the contributions mentioned in this paper.

▷ *Sub-Trees*: this version enhances Baseline by exploiting sub-tree co-location and execution migration. Hence, transactions perform fewer remote accesses. The topmost part, however, is partially replicated. Thus, traversing the first tree nodes results in remote accesses with a high likelihood.

▷ *Dirty*: this version adds Dirty Reads to the Baseline, by exploiting them when accessing inner nodes. In this scheme, tree nodes are placed randomly, which causes a high number of remote accesses. The advantage of this variant is that update transactions (that require commit time validation) involve a smaller number of machines in the commit phase because Dirty Reads need not be validated.

▷ *TopFull*: this version differs from the Baseline by fully replicating the topmost part of the tree (the nodes above $\mathcal{C}$). When traversing these nodes, this variant also uses Dirty Reads, to avoid contacting all machines (due to full replication) during transaction's validation. However, this variant is expected to reduce remote accesses only slightly, as all operations imply traversing partially replicated parts of the tree scattered using random placement.

We start by showing, in Fig. 2, experiments for each of YCSB's workloads using 60 machines, which corresponds to a medium scale deployment given the maximum size of our experimental test-bed. The results highlight the significant speed ups achieved by STI-BT over the Baseline (average throughput improvement of 13.5×) regardless of the workload. By evaluating the relative improvement achieved by each variant, the plots allow us to analyze the relevance of each design contribution. In particular, we obtained the following speedups over the Baseline: 6.6× for Sub-Trees; 2.5× for Dirty; and 1.9× for TopFull. Note that each contribution on its own never reaches more than half the throughput of STI-BT; this fact will be more evident when we look at individual workloads. In Table 2, we show the number of remote data fetches per transaction, and the number of machines contacted per commit. For STI-BT, the remote operations represent only the migration of control flow (on average once per transaction) and rebalance operations that require updating nodes belonging to two sub-trees that are not assigned to the same machine. Sub-Trees requires further remote accesses due to the topmost

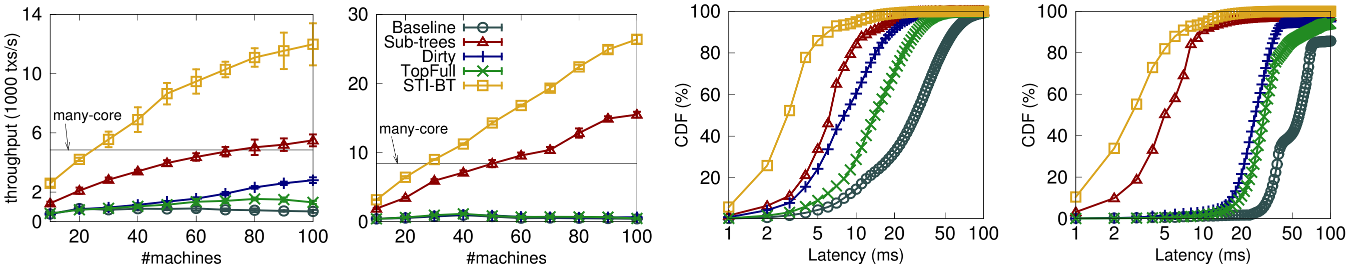
Table 2. Remote gets | machines contacted, relative to Fig.2.

	A	B	C	D	E	F
Baseline	15|19	14|15	10|9	12|14	32|29	14|22
Sub-Trees	3|5	3|5	2|4	2|4	3|5	2|4
Dirty	15|15	14|12	10|9	12|12	30|23	14|15
TopFull	9|18	8|11	7|8	8|12	16|26	11|19
STI-BT	0.2|2	0.3|2	0.1|2	0.1|2	1.4|3	0.1|2

part not being fully replicated. Finally, the Dirty design does not reduce the remote accesses, and the TopFull variant reduces this number only marginally. Note that the Dirty variant reduces the machines contacted by a larger extent than TopFull because of the reduced read-set for validation.

We now look in more detail at Workload A in Fig. 3a, which shows the throughput of each variant as the platform's scale grows, highlighting that STI-BT scales until 100 machines almost linearly even in this challenging, update intensive workload. We also show the peak throughput using a 48-core machine, which does not ensure fault-tolerance (no replication/distribution overhead) for reference purposes of the absolute throughput. In this experiment we note that all other variants besides STI-BT are either not scalable, or achieve inferior performance. In particular, the Sub-Trees variant performs best among them, but its trend stagnates at large scale. This strengthens the relevance of combining the whole set of mechanisms included by STI-BT, as each one alone performs rather poorly. This is also confirmed in the Cumulative Distribution Function of the transactions' execution latencies (see Fig. 3c): while STI-BT processed 90% of the transactions in $6ms$ (or less), this value is, respectively, 2×, 3×, 6× and 10× higher for Sub-Trees, Dirty, TopFull and Baseline.

Workload E, instead, requests 95% range scans and 5% insertions (see Fig. 3d). The results show that the gains in throughput and latency of STI-BT are even larger than for workload A (except for Sub-Trees). This workload is scan-heavy, for which reason co-locating sub-trees is even more important, as it allows traversals at the leaf nodes to be conducted locally most of the times. Hence, in this workload, the Sub-Trees feature is clearly the most important, whereas the others are of no help. In fact, the latency CDF of the other variants has a longer tail due to traversals that take up to hundreds of milliseconds. Note that these scan requests are wrapped in read-only transactions that are abort-free.



(a) Throughput in A. (b) Throughput in E. (c) Latency CDF in A. (d) Latency CDF in E.

Figure 3. Scalability and Latency CDF (at 60 machines) for YCSB's workloads A (heavy update) and E (short scans).

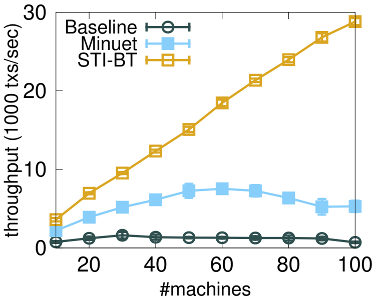


Figure 4. YCSB workload D.

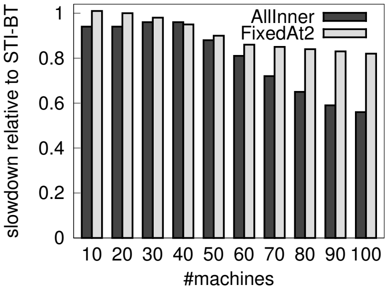


Figure 5. Varying cut-offs ($\mathcal{C}$).

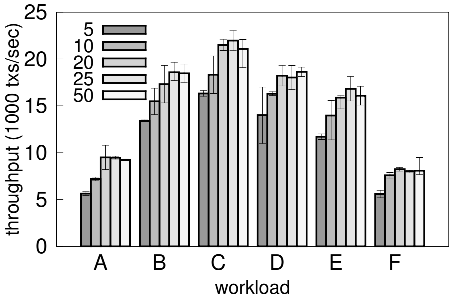


Figure 6. Varying arity (α) in YCSB.

9.2. Fresh Read-Only Transactions

The Minuet B-Tree [26] shares the design of the Dirty variant that we showed above, without any co-location or placement as the Sub-Trees and TopFull variants. Also Minuet relies on multi-versioning, but it adopts a fundamentally different approach to implementing it. STI-BT is layered on top of GMU, which relies on a scalable vector clock-based distributed timestamping mechanism, that avoids to contact any other machines than those maintaining data accessed by the transaction. Conversely, Minuet relies on a shared global clock (maintained by a single machine), which is incremented whenever a read-only transaction requests a fresh view of the data. As we will show, this centralized design hinders performance and compromises scalability.

In Fig. 4 we show detailed results obtained with workload D, which mimics accesses to the latest available data (read-dominated). As the source code of Minuet is not available, we emulated it by using a fully fledged STI-BT where read-only transactions first run an update transaction that increments a partially replicated key-value representing the shared clock. When that increment fails, we do not repeat the update transaction, and instead use the *borrowing technique* introduced in Minuet [26]. Note that the way in which we implemented Minuet is clearly favouring it, as it also benefits from the smart data placement of STI-BT (not used in Minuet). Regardless of that, this Minuet-like solution is clearly not scalable when faced with workloads that require fresh data. Conversely, STI-BT scales up almost to 30 thousand transactions per second with 100 machines while always providing fresh, consistent snapshots.

9.3. Cut-off Adaptation and Tree Arity

To assess the effects of adapting $\mathcal{C}$ as the system scale changes, we consider two alternative, static strategies: AllInner, which fully replicates all inner nodes (similarly to [1]), and FixedAt2, which places the cut-off at depth 2.

In Fig. 5 we show the slowdown of both strategies vs our adaptive mechanism, using Workload A. FixedAt2 performs similarly to STI-BT when the cluster size is small; but as more machines join, the sub-tree assignment becomes unbalanced, as some machines keep more trees than others. AllInner's performance is significantly lower, as rebalance operations are likely to modify fully replicated inner nodes, an inherently non-scalable operation.

In Fig. 6, we show the average throughput of STI-BT across all YCSB workloads while varying the tree arity. We can see that performance increases slightly as the arity increases, and eventually stagnates. This is due to the fact that low arity values lead to deeper trees, which cause a higher number of accesses to the underlying DKV. However, this effect becomes negligible as the arity increases. Overall, this shows that STI-BT ensures stable performance for non-minimal arity values (beyond 20, which motivated the settings of α for the other experiments presented).

10. Conclusions

In this paper we have presented STI-BT, a scalable solution to index data transactionally on a DKV store. STI-BT allows overcoming one of the inherent, and most severe limitations of this emerging type of platforms: the lack of (efficient) transactional indexes for non-primary attributes. STI-BT ensures that the index is transactionally consistent with the data, sparing programmers from the complexity of asynchronous/weakly consistent indexing solutions. This is achieved without compromising the key strength points that have determined the success of DKV stores, namely scalability and elasticity.

STI-BT combines a number of innovative mechanisms aimed to i) maximize data locality, ii) achieve optimal efficiency at any scale, and iii) minimize data contention. We integrated STI-BT in a mainstream transactional DKV store, and conducted an extensive experimental study. Our results demonstrate its scalability and efficiency, by achieving linear scalability in a cluster of 100 commodity machines, and up to 5.4× speed-ups over state of the art solutions.

References

[1] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A Practical Scalable Distributed B-tree. *Journal Proc. VLDB Endowment*, 1(1):598–609, August 2008.

[2] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[3] Saeed Alaei, Mohammad Ghodsi, and Mohammad Toossi. Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries. *Computer Communications*, 33(1):73–82, January 2010.

[4] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Proc. International Symposium on Reliable and Distributed Systems (SRDS)*, pages 163–172, 2013.

[5] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

[6] James Corbett et al. Spanner: Google's globally-distributed database. In *Proc. Conference on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2012.

[7] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1):1–45, April 2013.

[8] Giuseppe DeCandia et. al. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[9] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional Auto Scaler: Elastic Scaling of In-memory Transactional Data Grids. In *Proc. 9th International Conference on Autonomic Computing (ICAC)*, pages 125–134, 2012.

[10] Nuno Diegues and Paolo Romano. Bumper: Sheltering Transactions from Conflicts. In *Proc. International Symposium on Reliable and Distributed Systems (SRDS)*, pages 185–194, 2013.

[11] Nuno Diegues and Paolo Romano. STI-BT: A scalable transactional index. Technical Report 24, INESC-ID, September 2013.

[12] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *Proc. Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 25–36, 2012.

[13] Goetz Graefe. A survey of B-tree locking techniques. *ACM Transactions on Database Systems*, 35(3):1–26, July 2010.

[14] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. 22nd Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.

[15] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. BATON: a balanced tree structure for peer-to-peer networks. In *Proc. 31st International Conference on Very Large Data Bases (VLDB)*, pages 661–672, 2005.

[16] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. Symposium on Theory of Computing (STOC)*, pages 654–663, 1997.

[17] Junwhan Kim, Roberto Palmieri, and Binoy Ravindran. Enhancing Concurrency in Distributed Transactional Memory through Commutativity. In *Proc. Euro-Par*, pages 150–161, 2013.

[18] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[19] Dionysios Logothetis and Kenneth Yocum. Ad-hoc data processing in the cloud. *Journal Proc. VLDB Endowment*, 1(2):1472–1475, August 2008.

[20] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proc. Conference on Symposium on Opearting Systems Design and Implementation (OSDI)*, pages 8–24, 2004.

[21] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. AutoPlacer: scalable self-tuning data placement in distributed key-value stores. In *Proc. International Conference on Autonomic Computing (ICAC)*, pages 119–131, 2013.

[22] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *Proc. International Conference on Distributed Computing Systems (ICDCS)*, pages 455–465, June 2012.

[23] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. SCORe: A Scalable One-Copy Serializable Partial Replication Protocol. In *Proc. Middleware*, pages 456–475, 2012.

[24] Cristina Schmidt and Manish Parashar. Squid: Enabling search in DHT-based systems. *Journal of Parallel and Distributed Computing*, 68(7):962–975, 2008.

[25] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 385–400, 2011.

[26] Benjamin Sowell, Wojciech Golab, and Mehul A. Shah. Minuet: A Scalable Distributed Multiversion B-tree. *Journal Proc. VLDB Endowment*, 5(9):884–895, May 2012.

[27] Alexandru Turcu, Roberto Palmieri, and Binoy Ravindran. Automated data partitioning for highly scalable and strongly consistent transactions. In *Proc. International Systems and Storage Conference (SYSTOR)*, 2014.

[28] Alexandru Turcu, Binoy Ravindran, and Roberto Palmieri. Hyflow2: a high performance distributed transactional memory framework in scala. In *Proc. on Principles and Practices of Programming on the Java Platform (PPPJ)*, pages 79–88, 2013.

[29] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multi-dimensional data in a cloud system. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 591–602, 2010.

[30] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient B-tree Based Indexing for Cloud Data Processing. *Journal Proc. VLDB Endowment*, 3(1-2):1207–1218, September 2010.

[31] Chong Zhang, Weidong Xiao, Daquan Tang, and Jiuyang Tang. P2p-based multidimensional indexing methods: A survey. *Journal of Systems and Software*, 84(12):2348–2362, 2011.