Distributed B-tree with Weak Consistency

Shah Asaduzzaman and Gregor v. Bochmann

University of Ottawa, Ottawa, ON, K1N 6N5, Canada,
{asad, bochmann}@site.uottawa.ca

Abstract. B-tree is a widely used data-structure indexing data for efficient retrieval. Highly parallel operations are desired by modern-day cloud computing platforms on high-volume and highly dynamic sets of data. This motivates decentralized indexing structures for data-organization that avoid any performance bottleneck. In a decentralized B-tree, parts of the structure are distributed among different processors and some parts are replicated. Maintaining strong consistency among the replicated states of distributed data-structures has been found problematic in large-scale cloud platforms. We show in this paper that data retrieval can be performed correctly with much weaker consistency criteria among the replicated parts of the distributed B-tree. To accommodate the dynamic changes concerning the data insertion/deletion and the distribution of retrieval load, the state of the B-tree is updated by splitting and merging tree-nodes. We provide update algorithms that work under the weak consistency criteria and maintain them. The update operations are initiated independently by individual processors and modify the states of only a few processors in their immediate neighbourhood.

1 Introduction

Massive-scale computing platforms such as computing clouds frequently operate on huge volumes of data. Highly parallel operations are desired by such platforms due to the large number of processing units they have. Consequently, appropriate organization of the data is required such that the high-volume and highly dynamic data set is efficiently accessed and updated without any performance bottleneck.

B-tree is a widely used and well-understood data-structure to index data for efficient retrieval. Highly parallel operations are desired by modern-day cloud computing platforms on high-volume and highly dynamic sets of data. This motivates decentralized indexing structures for data-organization.

In fact, the biggest concern for the cloud computing model, identified in the discussion on the cloud computing research agenda [5] and afterwards, is the enormous overhead and the resulting infeasibility of the strong consistency model assumed in many well-known operations in distributed systems. Thus, it is desired that distributed and replicated-state data structures be designed in a way that they can tolerate some degree of inconsistency and still function appropriately. This motivates us to design a distributed implementation of the

B-tree data structure that works with weak consistency among its replicated components but provides strong consistency in terms of search semantics.

In this paper we identify the consistency conditions that are sufficient for correct and efficient search operation on the distributed B-tree indexing data structure (Section 3). We then define algorithms for updating the data structure keeping these consistency conditions maintained (Section 4). The data structure is generalized for key-spaces of arbitrary dimensions. The system model, assumptions and the particular way of distributing the B-tree structure are introduced in Section 2.

2 System Model and Assumptions

2.1 B-tree structure

We consider the B^+-tree variant of the B-tree, which is possibly the most widely used variant of the data structure. In a B^+-tree, all nodes have the same structure. Each of the leaf nodes maintains data-keys pertaining to a certain range in the key-space. Each internal node effectively maintains a list of entries, each containing a key-range and a pointer to some other node corresponding to this range. B-trees were designed for indexing one-dimensional data-spaces. So, the ranges were effectively expressed by integer data-keys, or points in the linear key-space.

Among the design goals of B-trees were (a) efficient use of disk blocks, and (b) keeping the search tree balanced while growing or shrinking. For efficient disk access, the size of the data stored in each node is maintained to be equal or close to integral multiples of the size of a disk block. A global parameter d defines the maximum number of entries to be held by a node. The root node of the tree describes the whole key-space or *key-universe* and each of the other nodes describes a portion or sub-range of the data-universe. *Describing* a range means dividing the range into sub-ranges and maintaining pointers to the child nodes, each of which describes one sub-range. If n is the number of child pointers or sub-ranges described by a node, $n-1$ data-keys are used to divide the range into n sub-ranges. The relation $\lceil d/2 \rceil \leq n \leq d$ is maintained for all nodes.

We use a generalization of B-tree for key-spaces of arbitrary dimensions, instead of a single dimension. Thus, we avoid any particular way of expressing the division of ranges, such as by points for one dimension as in a B-tree, or by lines or rectangles for two-dimensions as in Rtree [8] or Quad-tree [7]. We assume that each tree-node maintains the definitions of N sub-ranges of the whole range it describes, along with one pointer to another tree-node for each of the sub-ranges. Figure 1(a) shows an example of such B-tree. In the rest of the paper, the term *B-tree* will be used to denote a centralized implementation of such a generalized tree-based indexing structure.

2.2 Distributed Implementations of B-tree

When a huge number of data records are indexed, for the sake of distributing the storage and access load, instead of storing the whole B-tree in one computer, the data-structure is realized using a large number of computers or processing nodes. For disambiguation between tree-nodes and processing nodes, we denote the latter as *processor*, while *node* refers to tree-nodes.

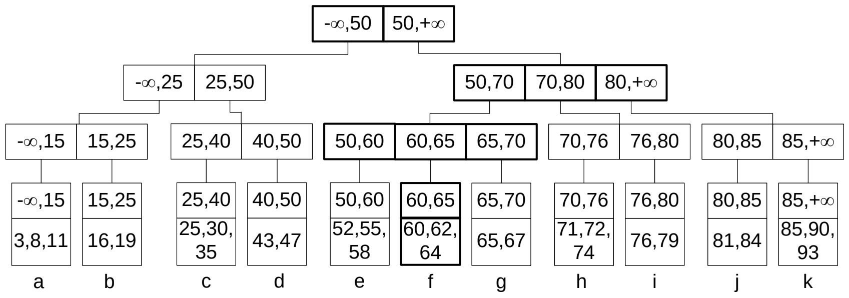
Node-wise distribution The intuitive method for distributing the tree data structure is to place each tree-node on one processor. The scalable distributed B-tree proposed by Aguilera et al. [2] and the tablet hierarchy in the internal representation of Google's Bigtable [6] structure use such representations. Although this allows the update algorithms on the structure for data insertion/deletion to be similar to the centralized version, the processors holding the root or the higher level tree-nodes get overburdened with search traffic. A typical solution to this problem, used in both [2] and [6], is caching or replicating the higher level nodes of the tree in the user or client computers, such that traversing higher level nodes can be avoided. However, this involves additional overhead for maintaining consistency among the replicas, and may not be suitable for highly dynamic data sets.

Decentralized distribution An alternative distribution of the tree structure is possible, following the decentralized design philosophy, assigning equal workload and the same role to each processor node. So, instead of assigning the responsibility of one tree-node to one processor, one branch of the tree, i.e. the path from root to a leaf node, is assigned to one processor. Thus, the higher level tree-nodes are, in a sense, replicated in proportion to their usage, and hence, the workload due to traversal operations is equally distributed among the processors.

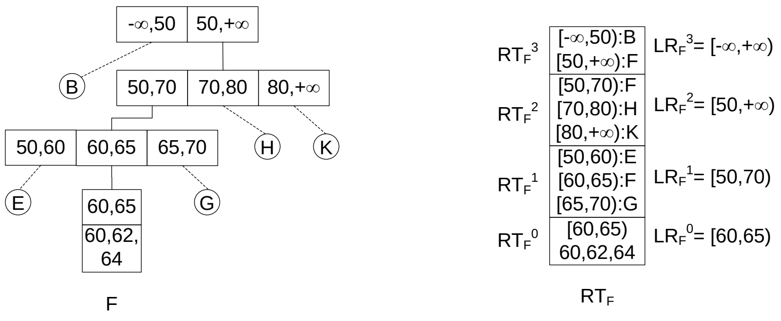
To represent a branch of the tree, each processor i maintains a *routing table* data structure RT_i with multiple levels, each level representing one node of the branch. Level l of RT_i, denoted as RT_i^l, corresponds to a level-l node of the B-tree. RT_i^l is a set of *entries* or tuples $\langle r, j \rangle$, together describing a range LR_i^l in the key-space. Each r is a sub-range of LR_i^l and the corresponding j refers to some processor j (may be i itself) that holds the level $l-1$ node of the B-tree describing r, that is, RT_j^{l-1} represents the child node and $LR_j^{l-1} = r$. Representation of one branch to the leaf-node f for the example B-tree in Figure 1(a) is shown in Figure 1(b).

Because non-leaf nodes are replicated in multiple processors, one for each branch, there are multiple options for j if $l-1$ is a non-leaf level, and any one of them may be chosen for the entry $\langle r, j \rangle$. Also, the range r for different entries in RT_i^l are non-overlapping and the union of these ranges constitutes LR_i^l (this is the same as in a normal B-tree). The lowest level, RT_i^0 corresponds to a leaf node of the B-tree, and stores the set of keys in the range LR_i^0 delegated to processor i, and the pointers to corresponding data items. Note that the size of the tree state maintained at each processor is $O(logN)$, where N is the total number of keys in the whole structure.

A similar distributed implementation of a tree structure has been proposed in [10], called DPTree. Although a DPTree builds the tree-structure on top



(a) An example of B-tree



RT_F^3	[-∞,50):B [50,+∞):F	LR_F^3= [-∞,+∞)
RT_F^2	[50,70):F [70,80):H [80,+∞):K	LR_F^2= [50,+∞)
RT_F^1	[50,60):E [60,65):F [65,70):G	LR_F^1= [50,70)
RT_F^0	[60,65) 60,62,64	LR_F^0= [60,65)
	RT_F	

(b) Consistent decentralized implementation of the B-tree. The view of the tree from processor F(*left*). The routing table maintained by processor F is shown (*right*)

Fig. 1. A B-tree and its consistent decentralized implementation. The leaf-level tree-nodes are referred by small letters $(a, b, c, ...)$ and the processors holding the corresponding leaf-nodes are referred by capital letters $(A, B, C, ...)$

of a distributed hash table used to name and discover the tree nodes, such decentralized structure can be maintained without such overlay, as shown in [3]

2.3 Assumptions

We assume an asynchronous message-passing distributed system [4], where each processor contains its own local memory (or persistent storage), the processors communicate among them through messages, all processors run the same program and there is no master clock to synchronize the events in the processors.

We follow a peer-to-peer model, where the search operation can be initiated from any processor. Thus, the client application may consider any of the processors in the distributed B-tree as a portal to the search service.

For fault-tolerance, a processor in our model may be realized by a small cluster of computers, replicating the state of one processor. Details of implementing a fault-tolerant processor from faulty processing nodes may be found at [11]. We assume that a message sent to another processor is eventually received by that processor in finite amount of time, although messages may be delivered out of order. The message channels may be made reliable through use of an end-to-end transport protocol [1]. We assume a complete network model, where any processor is able to send messages to any other processor as long as the address of that processor is known.

3 Search and Updates in Decentralized B-tree

3.1 Search Algorithm

To search a target key d_t (or a range r_t) in the decentralized B-tree, the primary goal is to find the processor i (or a set of processors P) such that $d_t \in LR_i^0$ (or $\bigcup_{i \in P} LR_i^0 \supseteq r_t$). The search can be initiated from any processor. Navigation of the request from the initiator to the target processor is performed by Algorithm 1. The initiator processor calls the Algorithm 1 with level l parameter equal to the topmost level of its own routing table.

For range search, instead of finding one $\langle r, j \rangle \in RT_i^l$, all $\langle r, j \rangle | r \cap r_t \neq \phi$ are looked up and the navigation proceeds next level to all the j's in parallel. The time complexity of both point and range search algorithms are clearly $O(logN)$, although the message complexity is higher for the range search ($O(N)$ in the worst case, if all the processors are included in r_t).

3.2 Updates in a Consistent Decentralize B-tree

The data structure needs to be updated as keys are inserted or deleted. The B-tree data structure grows with key-insertion by splitting a node when the number of entries overflows, and shrinks with key-deletion by merging two sibling nodes. In the decentralized B-tree, leaf level split and merger are simple. However, because non-leaf nodes are replicated in many processors, split/merge operations

Algorithm 1 Search(i, d_t, l)

```
1: Initiator: processor $i$
2: Condition: a query received to resolve $d_t$ at level $l$
3: Action:
4: if $l = 0$ then
5:     Result is processor $i$
6: else
7:     Find $\langle r, j\rangle \in RT_i^l$ s.t. $d_t \in r$
8:     Forward the query to $j$ as Search($j$, $d_t$, $l-1$)
9: end if
```

in non-leaf levels require a large number of nodes to be updated atomically, which may require the updates to be coordinated by a single master processor. In the worst case, when the state of the root node is changed, the update needs to be atomically propagated to all the processors.

Figure 2(a) illustrates the split of the tree node f after insertion of data element 63 by maintaining a single consistent view of the tree at each processors. Splitting at the leaf level is relatively simple. Part of the data-keys at processor F is now moved to a new processor F_2. Because the level-1 tree node is modified, level-1 at processors E and G need to be updated. Also, whichever processor held F responsible for its level-0 range $[60, 65)$ need to be updated about the change.

When the level-1 tree-node, containing the range $[50, 70)$ needs to be split (Figure 2(b)), it involves splitting the level-1 of processors E, F, F_2 and G. This causes the level-2 tree-node to have one new entry, which requires all the processors E through K to add an entry at their level-2. Finally, whichever processors held any of E, F, F_2 or G responsible for its level-1 range now need to update their pointers. Thus even a level-1 split for consistent B-tree with fanout of only $2-4$ involves atomic update of the states at $10-12$ processors.

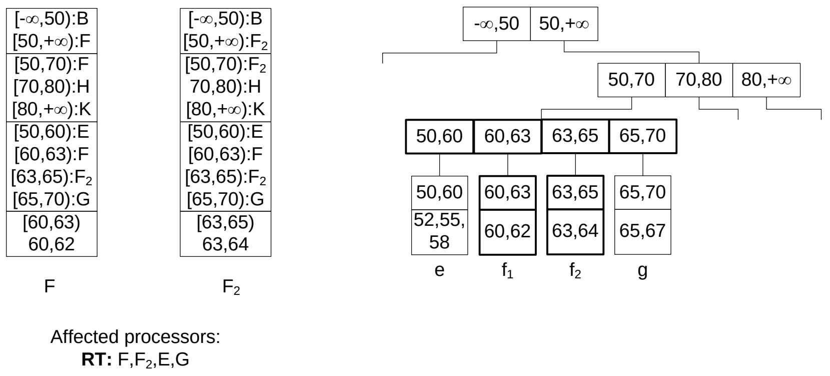
The huge overhead of large-scale atomic updates in the consistent decentralized B-tree structure motivates us to look for weaker consistency conditions that are easy to maintain through much smaller-scale updates, and yet sufficient for correct search operations.

3.3 How Much Consistency is Needed?

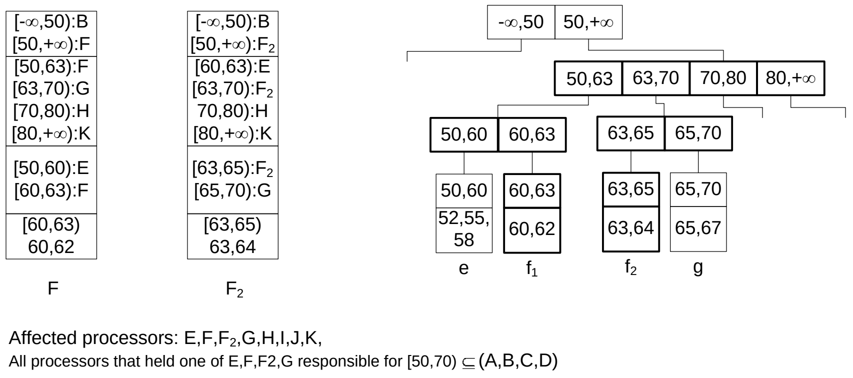
Here we define consistency conditions among the components of the decentralized B-tree structure maintained by different processors that are sufficient for ensuring the correctness of the search operation through Algorithm 1, but weaker than the constraint that all component-states are consistent with a single global B-tree.

First, any processor should be able to initiate the search, so every processor should maintain a description of the key-space universe (U) at the topmost level of its routing table. We call this condition *invariant of universal coverage* –

– A_U: $\forall i : LR_i^m = U$, where m is the highest level in RT_i



(a) Splitting level-0 range $[60, 65)$



(b) Splitting level-1 range $[50, 70)$

Fig. 2. Updates in a consistent decentralized B-tree. The original tree is shown in Figure 1

For correct navigation, if an entry $\langle r, j\rangle$ is in RT_i^l, then its target j must describe at least the range r at level $l-1$. Formally, this defines the *invariant of navigability* –

– A_N: $\forall i \forall l : \langle r, j\rangle \in RT_i^l \Rightarrow r \subseteq LR_j^{l-1}$

Another condition is necessary depending on the semantics of the search operation. If we allow different processors to have overlapping local ranges at the leaf level, then for a search query d_t, $d_t \in LR_i^0 \cap LR_j^0, i \neq j$, Algorithm 1 ensures delivery of the query to at least one of i and j. This result is correct, if all keys in $LR_i^0 \cap LR_j^0$ are available in both RT_i^0 and RT_j^0. If overlapping coverage of ranges by different processor does not imply such exact replication of all keys in the common range, then the usual semantics of search requires the query to reach all such processors. To keep things simple, we impose the following *invariant of disjoint local range* –

– A_{LR}: $\forall i \forall j : i \neq j \Rightarrow LR_i^0 \cap LR_j^0 = \phi$

Theorem 31 *A_U, A_N and A_LR are sufficient conditions for correctness of exact and range search operations in the decentralized B-tree using Algorithm 1.*

Proof. Line 8 of Algorithm 1 ensures that the algorithm proceeds at least one level towards level 0 at each hop. Thus the algorithm terminates in a number of steps not larger than the maximum number of levels in the routing table of any processor.

At each hop in the navigation, an entry $\langle r, j\rangle \in RT_i^l, d_t \in r$ must always be found at Line 7. A_U ensures that such an entry is always found at the topmost level of the initiating processor. A_N ensures that, if such an $\langle r_1, p\rangle$ is found at level l of current processor i, an entry $\langle r_2, q\rangle, d_t \in r_2$ can be found at level $l-1$ of the next hop processor p. So, by induction, we observe that the query is finally forwarded to a processor p s.t. $LR_p^0 \ni d_t$. A_{LR} ensures that only one such processor exists. The proof can be easily extended to show the correctness of the range-search algorithm.

The decentralized B-tree structure that maintains the conditions A_U, A_N and A_R, in general, is a weakly-consistent structure, because several conditions valid in the consistent decentralized B-tree structure have been relaxed. For example, the equality relation in the condition $\forall i \forall l : \langle r, j\rangle \in RT_i^l \Rightarrow r = LR_j^{l-1}$ valid for the consistent structure is relaxed to the inclusion ($\supseteq$) relation. Also, in the consistent structure, each level of the routing table contains a self-pointer, i.e. the condition $\forall i \forall l > 0 : \exists \langle r, i\rangle \in RT_i^l$ is valid, but this is not maintained in the weakly-consistent structure. In addition, the number of levels of the routing table may be different for different processors. The condition that each node in the tree must maintain a number of entries n such that $\lceil d/2 \rceil \leq n \leq d$, is also relaxed. The lower and upper limits are now rather soft-limits. As a result, the cascaded split or merge operations are treated as separate update operations.

Figure 3 shows how a weakly-consistent B-tree structure may grow through insertion of data-keys. Initially, through the first three steps, the view of the tree

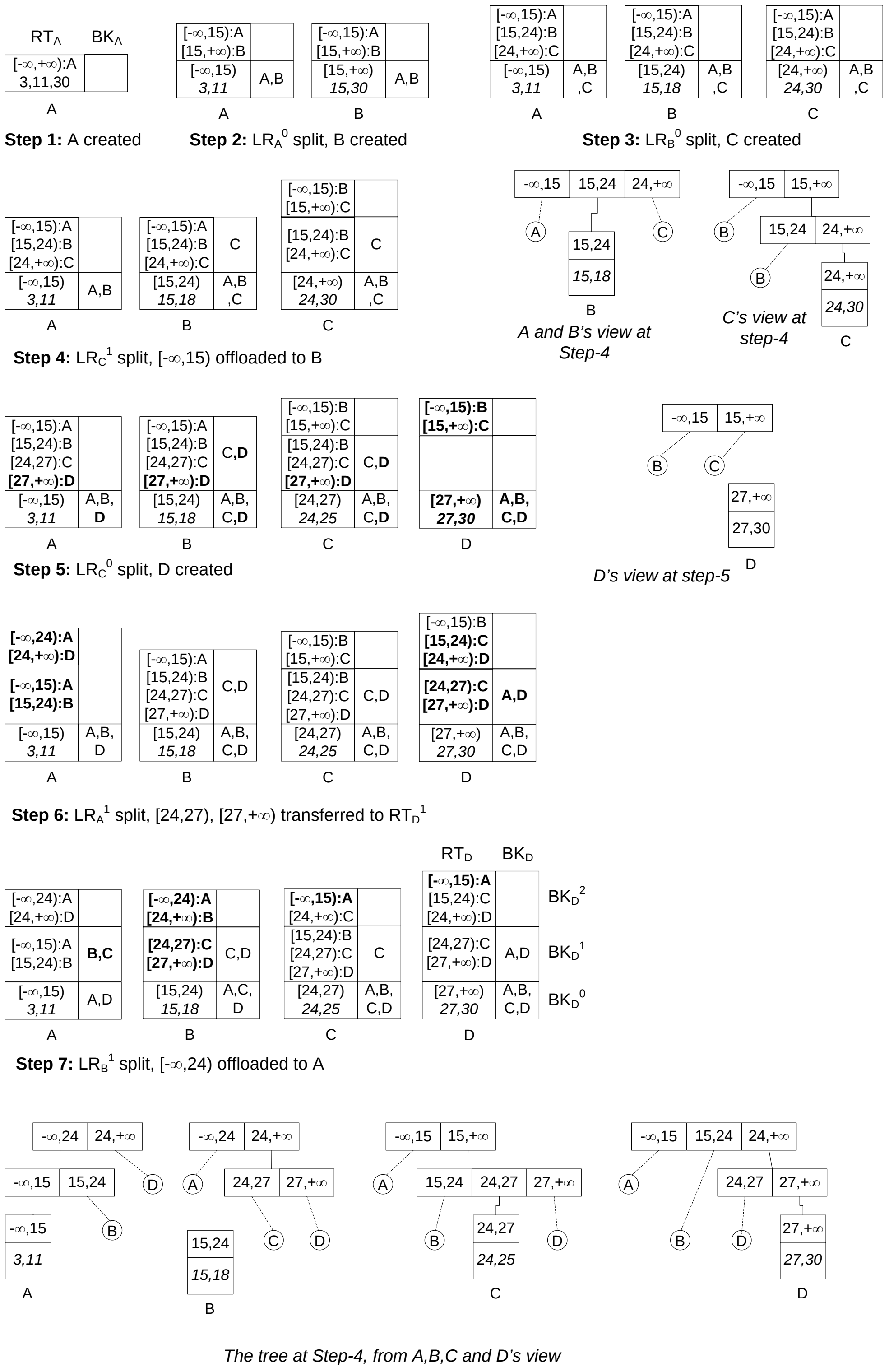

RT_A BK_A
[-∞,+∞):A
3,11,30
Step 1: A created
Step 2: LR_A^0 split, B created
Step 3: LR_B^0 split, C created
A and B's view at Step-4
C's view at step-4
Step 4: LR_C^1 split, [-∞,15) offloaded to B
Step 5: LR_C^0 split, D created
D's view at step-5
Step 6: LR_A^1 split, [24,27), [27,+∞) transferred to RT_D^1
RT_D BK_D
BK_D^2
BK_D^1
BK_D^0
Step 7: LR_B^1 split, [-∞,24) offloaded to A
A
B
C
D
The tree at Step-4, from A,B,C and D's view

Fig. 3. Evolution of a weak-consistent B-tree with asynchronous updates. To facilitate asynchronous update, each processor i maintains a table BK_i in addition to RT_i. BK_i^l holds the names of all processors that hold RT_i^l responsible for the whole or some part of LR_i^l

remains consistent for all three processors A, B and C. From step-4 onwards, different processors may have different views of the tree. It may be noted that with such weak-consistent updates, the view of the data structure at some processors may no longer remain a single connected tree. Rather, the view may be of several disconnected segments of the tree. Nevertheless, each processor maintains sufficient information to route any search query originated at any processor.

The update operations are initiated independently and asynchronously by individual processors. Compared to the updates in a consistent B-tree shown in Figure 2, which, even for a level-1 split, require updating the states of a large number of processors atomically, updates here are much less invasive. For example, starting from the same states as in Figure 2, weak-consistent updates at level-1 could be initiated independently by the processors E,F,F_2 and G, and each of them would involve the states of $3 - 4$ processors known to them by routing table entries. The algorithms presented in the next section will explain these asynchronous updates.

Although the weak consistency leads different processors to maintain different views of the tree, the search operation remains correct and can be initiated from any processor. The search always progresses one level at each hop, and thus, terminates after a number of hops equal to the maximum height of the tree (i.e. the maximum level in the routing table) in any of the views.

4 Updates with Weak Consistency

In this section we describe the atomic update operations needed to adapt the decentralized B-tree structure when data keys are inserted or deleted, or when some processor is overloaded. The basic insertion and deletion operations works similarly as in traditional B-tree, i.e. first the target key (or its position) is searched, and then the deletion or insertion is performed. Insertion of keys may cause overflow in a leaf level node, which then splits, and the split may be cascaded to higher level nodes. Similarly deletion of keys cause underflow and triggers merger of nodes. Because lower and upper limits in the number of entries are now soft-limits, the cascading splits (or mergers) are treated as separate atomic operations. Here we define the atomic split and merge update operations for leaf level and non-leaf level separately. The update algorithms assume that the three consistency conditions A_U, A_N and A_{LR} are satisfied when the update is started, and assure that the conditions will be satisfied again when the update completes.

The update algorithms are triggered independently by any processor. One principle followed in the design of the update algorithms is to modify the states of a minimal number of processors. Specifically, the modification is limited to the *neighbour* processors only, i.e. the processors known to the local routing table of the initiating processor. To facilitate the updates, an additional table called backward pointer table is maintained by each processor (the table of processor i is denoted as BK_i). For any processor i, BK_i has the same number of levels as RT_i. $j \in BK_i^l$ if and only if $\langle r, i \rangle \in RT_j^{l+1}$.

Each update operation may need to modify the states (routing tables) of a few neighbouring processors. To ensure correctness in the presence of concurrent updates, some concurrency control mechanism must ensure atomicity of each update. To allow a higher degree of parallelism, a version-number-based optimistic transaction protocols may be used [9]. In this method, a counter or version number is maintained for the state of each processor. The version number is incremented whenever the state is successfully modified. The initiating processor that executes the update algorithm reads the necessary states along with their version numbers. After computing the modified state locally, it then attempts to commit the new states to appropriate processors. The update transaction is aborted if any of the states in the write-set has a different version number than the one that was read initially. Aborted transactions are retried at a later time. While describing the update algorithms, we clearly mention which processor initiates it (*initiator*), and which state in which processors are read (*Readset*) and updated (*Writeset*). Version control may be applied at different granularities on the states. Each row of RT and BK tables at each processor may be separately versioned for maximum parallelism.

4.1 Split Algorithms

Algorithm 2 describes the procedure to split the local range LR_i^0 of processor i into 2 disjoint ranges $LR1$ and LR_2, and offloading LR_2 to a newly recruited processor j. Because i looses part of LR_i^0, $\forall p \in BK_i^0$, RT_p^1 need to add entries pointing to the new processor j instead of i for the lost part of the range. BK_i^0 may include i if RT_i^1 has a self-entry (Lines 10-17). In addition to the leaf level, the topmost level of the new processor j's routing table, RT_j^m is initialized by a replica of RT_i^m (Line 18). Mid-levels of RT_j remains empty. For the nodes newly pointed to by j at level m, their backward pointers are updated (Line 19).

Algorithm 3 is executed when processor i wants to offload some entries from its routing table RT_i^l at level $l > 0$. Unlike the case of leaf-level split, no new processor is recruited here. So, the major challenge here is to find an existing processor j, whose routing table at the same level, RT_j^l, either already contains some entries covering some common range with LR_i^l, or, have some space to take few entries from RT_i^l. In a consistent distributed B-tree, $\forall j | \langle r, j \rangle \in RT_i^l$, $RT_j^l = RT_i^l$. Thus, neighbours in RT_i^l are natural target for offloading part of RT_i^l. In the weak-consistent structure, it is not certain that such a j will be found in RT_j^l, so, other neighbours are searched including all backward pointers. Also, in the leaf-level split, the mid-levels of the new processor's routing table are kept empty. Non-leaf level splits are initiated for the lowest overloaded level. So, there is high possibility of finding a j in RT_i^l with empty space in RT_j^l.

Once a suitable j is found, the update procedure is straightforward. The entries are transferred from RT_i^l to RT_j^l and BK_p^{l-1} are updated for the processors corresponding to the transferred entries (Lines 8-13). Then for the processors in BK_i^l, i.e. those who held i responsible for some part of LR_i^l, now need to update for the range shifted to j, by adding a new entry in the level $l+1$ of their routing

Algorithm 2 SplitLeafNode(i)

```
1: Initiator: processor $i$
2: Condition: $RT_i^0$ is overloaded, in terms of storage or access load
3: Readset = $\{LR_i^0, RT_i, BK_i^0, \forall_{p\in BK_i^0} RT_p^1\}$
4: Writeset = $\{RT_i^0, LR_i^0, BK_i^0, RT_j^0, LR_j, BK_j^0, RT_j^m, \forall_{p\in BK_i^0} RT_p^1, \forall_{p\in RT_i^m} BK_p^{m-1}\}$
5: Action:
6: Partition $RT_i^0$ into 2 disjoint sets of keys $D_1$, $D_2$ and $LR_i^0$ into 2 disjoint ranges $LR_1$, $LR_2$, accordingly
7: Find 1 new processor $j$
8: $RT_i^0 \leftarrow D_1$, $LR_i^0 \leftarrow LR_1$
9: $RT_j^0 \leftarrow D_2$, $LR_j^0 \leftarrow LR_2$
10: for $\forall p \in BK_i^0$ do
11:     there must exist $\langle x, i\rangle \in RT_p^1$
12:     if $x \backslash LR_1 \neq \phi$ then
13:         $RT_p^1 \leftarrow RT_p^1 \backslash \langle x, i\rangle \cup \{\langle x\cap LR_1, i\rangle, \langle x\cap LR_2, j\rangle\}$
14:         $BK_j^0 \leftarrow BK_j^0 \cup \{p\}$
15:     end if
16:     if $x\cap LR_1 = \phi$ then $BK_i^0 \leftarrow BK_i^0 \backslash \{p\}$ endif
17: end for
18: $RT_j^m \leftarrow RT_i^m$, where $m$ is the topmost level of $RT_i$
19: $\forall p | \langle r, p\rangle \in RT_i^m : BK_p^{m-1} \leftarrow BK_p^{m-1} \cup \{j\}$
```

tables. Backward pointers of i and j are also updated accordingly (Lines 12-12). Finally, if the topmost level of RT_i^l is split, one additional level is added to hold the pointer to the transferred range, such that the whole universe is described.

4.2 Merge Algorithms

When there are too few data items in a processor i, it decides to release itself by merging its items and routing table with those of another processor. Algorithm 4 describes the update procedure for such a merger. A suitable partner j for the merger is found in RT_p^1, where p points to i for some range $x \subseteq LR_i^0$. If RT_p^1 is pointing to j for some other range y, then after the merger, the two entries $\langle x, i\rangle$ and $\langle y, j\rangle$ can be merged into $\langle x \cup y, j\rangle$. Because processor i is being released, all levels of its routing table are merged with the corresponding level of j's routing table (Line 8). Accordingly, $\forall_l \forall_{p\in BK_i^l} : RT_p^{l+1}$ are updated (Line 9).

Similar to level-0 merger, if any other level l of the routing table of a processor i is found underloaded, the entries of that level can be merged with the same-level entries in another processor. The merging partner, j is found in similar way as before, from $\forall_{p\in BK_i^l} RT_p^{l+1}$, so that after the merger one entry is eliminated there (Line 10). If RT_p^{l+1} is the topmost level, and contains only one entry after the merger, that level may potentially be eliminated (Line 13).

Theorem 41 *The update algorithms, Algorithms 2, 3, 4 and 5 maintain the invariants A_U, A_N and A_{LR}.*

Algorithm 3 SplitNonLeafNode(i, l)

```
1: **Initiator:** processor $i$
2: **Condition:** $|RT_i^l|$, has too many entries or causing too much routing load
3: **Readset** $= \{RT_i^l,\ BK_i^l,\ \forall_{p|\langle r,p\rangle \in RT_i \vee p \in BK_i} : RT_p^l$ (to find $j$), $\forall_{p \in BK_i^l} RT_p^{l+1}\}$
4: **Writeset** $= \{RT_i^l,\ BK_i^l,\ RT_j^l,\ BK_j^l,\ \forall_{p \in BK_i^l} RT_p^{l+1}\}$
5: **Action:**
6: Find $j|\langle r,j\rangle \in RT_i \vee j \in BK_i$ s.t. $RT_j^l$ have some space for at least 2 entries or $LR_j^l$ has some overlap with $LR_i^l$. Multiple such $j$ (say, $j_k$) may be chosen.
7: Partition $LR_i^l$ into two disjoint subsets $E_s$ and $E_x$, and partition $LR_i^l$ into disjoint ranges $R_s$ and $R_x$ accordingly. $R_x$ may be partitioned into multiple sub-ranges $R_{x_k}$ according to available $j_k$.
8: **for** $\forall k$ **do**
9:     $RT_{j_k}^l \leftarrow RT_{j_k}^l \cup E_{x_k}$
10:    $RT_i^l \leftarrow RT_i^l \backslash E_{x_k}$
11:    $\forall p|\langle r,p\rangle \in E_{x_k} : BK_p^{l-1} \leftarrow BK_p^{l-1} \backslash \{i\} \cup \{j_k\}$
12: **end for**
13: **for** $\forall p \in BK_i^l$ **do**
14:    there must exist $\langle x,i\rangle \in RT_p^{l+1}$
15:    **if** $x \backslash R_s \neq \phi$ **then**
16:        $RT_p^{l+1} \leftarrow$
               $RT_p^{l+1} \backslash \langle x,i\rangle \cup \{\langle x \cap R_s, i\rangle\} \bigcup_{\forall k} \{\langle R_{x_k}, j_k\rangle\}$
17:        $BK_j^l \leftarrow BK_j^l \cup \{p\}$
18:    **end if**
19:    **if** $x \cap R_s = \phi$ **then** $BK_i^l \leftarrow BK_i^l \backslash \{p\}$ **endif**
20: **end for**
21: **if** $RT_i^l$ is the topmost level in $RT_i$ **then**
22:    $RT_i^{l+1} \leftarrow \{\langle R_s, i\rangle\} \bigcup_{\forall k} \{\langle R_{x_k}, j_k\rangle\}$, $BK_i^l \leftarrow BK_i^l \cup \{i\}$
23: **end if**
```

Algorithm 4 MergeLeafNode(i)

```
1: **Initiator:** processor $i$
2: **Condition:** $RT_i^0$ has too few keys
3: **Readset** $= \{RT_i,\ \forall_{p \in BK_i^0} RT_p^1\}$
4: **Writeset** $= \{RT_j,\ \forall_l \forall_{p \in BK_i^l} RT_p^{l+1}\}$
5: **Action:**
6: Select $j|j \neq i \wedge \langle r,j\rangle \in \bigcup_{\forall p \in BK_i^0} RT_p^1$
7: $RT_j^0 \leftarrow RT_j^0 \cup RT_i^0$, $LR_j^0 \leftarrow LR_j^0 \cup LR_i^0$
8: $\forall l RT_j^l \leftarrow RT_j^l \cup RT_i^l$
9: $\forall l \forall p \in BK_i^l$: replace $i$ by $j$ in $RT_p^{l+1}$ and merge ranges as necessary
10: Release processor $i$
```

Algorithm 5 MergeNonLeafNode(i, l)

```
1: Initiator: processor $i$
2: Condition: $RT_i^l$ has too few data items
3: Readset = $\{RT_i^l, \forall_{p\in BK_i^l} RT_p^{l+1}\}$
4: Writeset = $\{RT_i^l, RT_j^l, \forall_{p\in BK_i^l} RT_p^{l+1}\}$
5: Action:
6: Select $j | j \neq i \wedge \langle r, j\rangle \in \bigcup_{\forall p\in BK_i^l} RT_p^{l+1}$
7: $RT_j^l \leftarrow RT_j^l \cup RT_i^l$, merge ranges in $RT_j^l$ as necessary
8: if $l$ is not the topmost level of $RT_i$ then
9:     $RT_i^l \leftarrow \phi$
10:    $\forall p \in BK_i^l$ : replace $i$ by $j$ in $RT_p^{l+1}$, and merge entries as necessary
11: end if
12: if for any $p \in BK_i^l$, $RT_p^{l+1}$ is the top-most level of $RT_p$ and contains only one entry pointing to $p$ then
13:     Delete $RT_p^{l+1}$ and remove $p$ from $BK_p^l$
14: end if
15: $BK_i^l \leftarrow \phi$
```

Proof. A_U: Algorithm 2 maintains A_U in the newly joined processor j by copying the top level of the routing table of i (Line 18). In Algorithm 3, the range LR_p^{l+1} in $\forall p \in BK_i^l$ remains unchanged after the modification in Line 13. If RT_i^l is the topmost level in RT_i then the additional update in Line 22 ensures A_U for processor i. In Algorithm 4, processor i is released and processor j takes all the entries of RT_i for all levels, so A_U is maintained for j. In Algorithm 5, RT_i^l is not emptied if l is the topmost level (Line 9). The removal of topmost level of RT_p with only one entry (Line 13) does not violate A_U either.

A_N: A_N can be violated only when LR_i^l for some processor i and some level l is reduced. In Algorithm 2, LR_i^0 is reduced, and so, RT_p^1 is updated for $\forall p \in BK_i^0$ to maintain A_N (Line 13). A similar update is performed in Line 16 of Algorithm 3, for reduction in LR_i^l, in Line 9 of Algorithm 4 for removal of $\forall l RT_i^l$, and in Line 10 of Algorithm 5 for removal of RT_i^l.

A_{LR}: Violation of A_{LR} is possible only when RT_i^0 is created or extended for any i. In Algorithm 2, LR^0 is modified for processors i and j only, and no overlap is formed (Line 9). In Algorithm 4, LR_i^0 and LR_j^0 are merged into LR_j^0, and then processor i is removed. So no overlap is created. Algorithms 3 and 5 do not modify LR^0 of any processor.

In an elementary state of the decentralized B-tree structure, when there is only one processor having only one level in its routing table, all the invariants A_U, A_N and A_{LR} are valid. So, by induction over successive updates, it can be proved using Theorem 41 that all three invariants are always maintained for the structure. Also, all four update algorithms work assuming the three invariants only. Validity of the backward pointers are also maintained in these algorithms whenever a forward pointer is updated.

5 Conclusion

We have demonstrated that it is possible to distribute a B-tree for data retrieval over a large number of processors with partial replication of the interior nodes of the tree over the different processors without full consistency. Enforcing only weak consistency conditions necessary for the correct operation of the retrieval function, it is possible to define tree update operations that can be initiated by one of the processors and would involve only the local state of the tree and the state in a few neighbour nodes, without requiring simultaneous updates in all processors that have a replica of the state being updated.

We have proved that the new update algorithms maintain our weak consistency conditions and that these conditions guarantee correct operation of the data retrieval algorithm that requires L steps where L is the depth of the B-tree.

We plan to study in the future whether the depth of the tree obtained over a long period of tree update operations can be maintained at the optimal level of L = log(N) where N is the number of processors in the system. We also plan to determine the average number of processors involved in a single split or merge update operation, and perform a more thorough comparison between the two decentralized B-tree organizations with full and weak consistency, to determine the performance of the update operations in the two settings.

References

1. Y. Afek and E. Gafni. End-to-end communication in unreliable networks. In *PODC'88: Proc. 7th ACM Symposium on Principles of Distributed Computing*, pages 131–148, 1988.
2. M. K. Aguilera, W. Golab, and M. S. Shah. A practical scalable distributed B-tree. *Proc. VLDB Endow.*, 1(1):598–609, 2008.
3. S. Asaduzzaman and G. v Bochmann. GeoP2P: An adaptive peer-to-peer overlay for efficient search and update of spatial information. *CoRR*, abs/0903.3759, 2009.
4. A. Bar-Noy and D. Dolev. A partial equivalence between shared-memory and message-passing in an asyncrhonous fail-stop distributed environment. *Mathematical Systems Theory*, 26:21–39, 1993.
5. K. Birman, G. Chockler, and R. v Renesse. Toward a cloud computing research agenda. *ACM SIGACT News*, 40(2):68–80, 2009.
6. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. *ACM Transactions on Computing Systems*, 26(2):1–26, 2008.
7. R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.
8. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84*, pages 47–57, Jun. 1984.
9. H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.
10. M. Li, W. Lee, and A. Sivasubramaniam. DPTree: A Balanced Tree Based Indexing Framework for Peer-to-Peer Systems. In *14th IEEE ICNP*, pages 12–21, Nov. 2006.

11. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.