Distributed B-Tree with Weak Consistency

Gregor v. Bochmann and Shah Asaduzzaman[1]

School of Electrical Engineering and Computer Science, University of Ottawa, Canada
{bochmann@eecs.uottawa.ca}
[1] now with Telenav, Inc., USA {shah.asaduzzaman@gmail.com}

Abstract: B-tree is a widely used data-structure indexing data for efficient Retrieval. We consider a decentralized B-tree, were parts of the structure are distributed among different processors and some parts are replicated, thus providing a decentralized indexing structure and parallel operations as desired by modern-day cloud computing platforms. To accommodate the dynamic changes due to data insertion/deletion and changes of the retrieval load, the state of the B-tree is updated by splitting and merging tree-nodes. The traditional update algorithms maintain strong consistency among the replicated states and possibly involve very many tree-nodes. We show in this paper that data retrieval and update can be performed correctly with much weaker consistency criteria. This allows to decompose the necessary updates into smaller update operations that involve only a limited number of tree-nodes, each. We show by analytical models and simulations that with weak consistency the average number of tree-nodes that require updating is reduced compared to the traditional B-tree update algorithms.

Keywords: B-tree, peer-to-peer systems, distributed data retrieval, weak consistency, distributed update operations, distributed databases

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 keeping the tree balanced, a global parameter d is introduced which 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 that describe each of the sub-ranges. If n is the number of child pointers or sub-ranges described by a node, one normally maintains the relation $\lceil d/2 \rceil \leq n \leq d$ in order to balance the amount of information stored in each node. In the case of a one-dimensional key-space (as used in our examples), a sub-range is characterized by two key values, the minimum and maximum key values of the sub-range.

In the following general discussion and the presented algorithms, we assume a generalization of B-trees 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 all our examples in this paper, we use a one-dimensional key-space with

consecutive sub-ranges. 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 the number of data records is huge and/or the access load becomes too large for a single computer, a distributed B-tree must be considered. Simply replicating the whole data structure on several computers is not practical because of the difficulty of the update operations. In this paper we consider what we call "decentralized distribution". For disambiguation between tree-nodes and processing nodes, we denote the latter as processor, while node refers to tree-nodes.

Centralized 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.

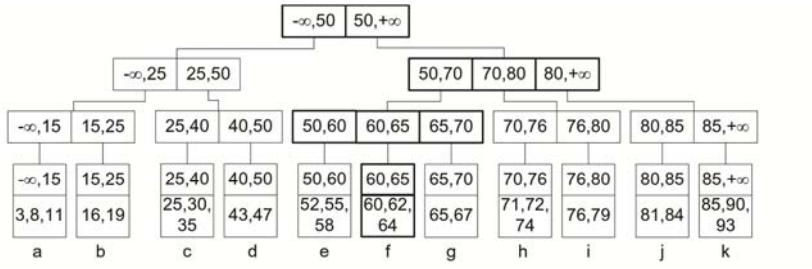
To represent a branch of the tree, each processor *i* maintains a routing table data structure RT_i with multiple levels, each level representing one node of the branch. Level *l* of RT_i , denoted as RT^l_i, corresponds to a level-l node of the B- tree. RT^l_i is a set of entries or tuples c together describing a range LR^l_i in the key-space. Each r is a sub-range of LR^l_i 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^{l-1}_i represents the child node and LR^{l-1}_i = 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 <r, j>. Also, the range r for different entries in an RT^l_i are non-overlapping and the union of these ranges constitutes LR^l_i (this is the same as in a normal B-tree). The lowest level, RT^0_i corresponds to a leaf node of the B-tree, and stores the set of keys in the range LR^0_i delegated to processor *i*, and the pointers to

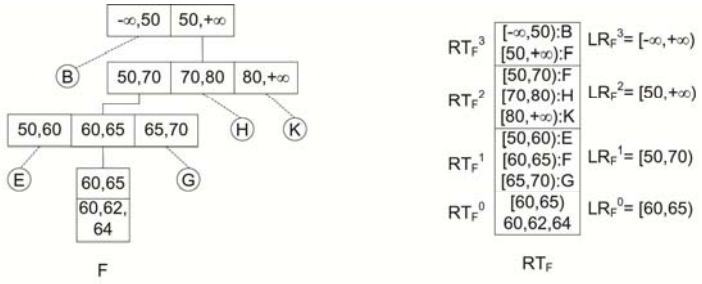
corresponding data items. Note that the size of the tree state maintained at each processor is O(logN), where N is the total number of keys in the whole structure.

A similar distributed implementation of a tree structure has been proposed in [10], called DPTree. Although a DPTree builds the tree-structure on top 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] (see also [14]). In this paper we do not consider distributed hash tables because we want to support range queries.

The above references consider that a single (global) B-tree is distributed and partially replicated over all the processors. We call this situation global consistency. It implies that updates of nodes in the upper part of the B-tree, which are replicated in many processors, require complex update operations. We consider in this paper a situation with weak consistency where an update of any node in the B-tree involves only a few processors.



(a) An example of a B-Tree



(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, ...)

2.3 Assumptions

We assume in this paper a decentralized distributed implementation of a B-tree as described above. We assume that the processors can use an asynchronous message-passing 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 $d_t \in LR^0_i$ (or $\text{union}_{i \in P}$ includes 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.

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 $<r,j> \in RT^l_i$, such that $d_t \in r$
8: Forward the query to j as *Search(j,* d_t *,* l *-1)*
9: **end if**

For range search, instead of finding one $<r,j> \in RT^l_i$, all $\{<r,j> \mid r \cap r_t \neq \phi\}$ 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 Globally 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 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 F2. 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 , F2 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 , F2 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 fan-out of only 2 - 4 involves atomic update of the states at 10 - 12 processors.

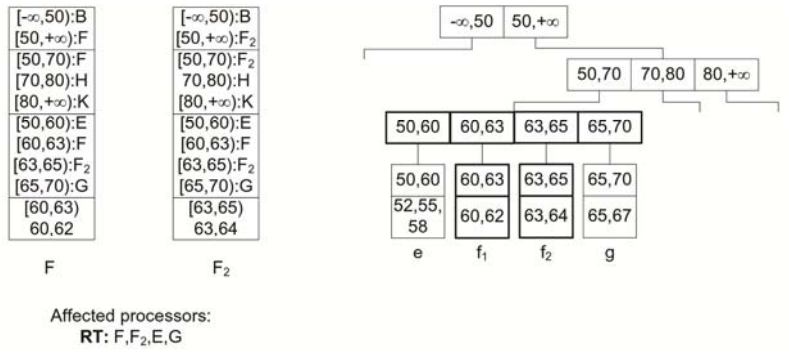
The huge overhead of large-scale atomic updates in the consistent decentralized B-tree structure motivates us to look for weaker consistency conditions that are easy to maintain through much smaller-scale updates, and yet sufficient for correct search operations.

3.3 How Much Consistency is Needed?

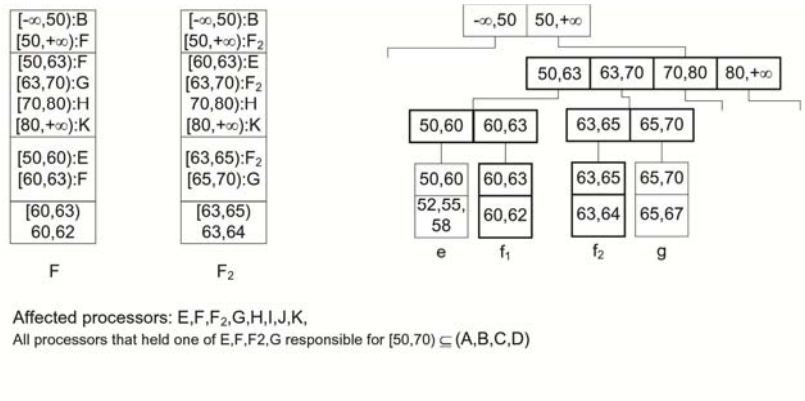
Here we define consistency conditions among the components of the decentralized B-tree structure maintained by different processors that are sufficient for ensuring the correctness of the search operation through Algorithm 1, but weaker than the constraint that all component-states are consistent with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

AU : for all i : $LR^m_i = U$, where m is the highest level in RT_i



(a) Splitting leve-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 <r, j> is in RT^{l}_{i} , then its target j must describe at least the range r at level *l - 1*. Formally, this defines the **invariant of navigability:**

AN : for all *i* and *l* : <r, j> ϵ RT^{l}_{i} implies r is included in LR^{l-1}_{j}

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 for d_t , where d_t ϵ LR^{0}_{i} and d_t ϵ LR^{0}_{j} with $i \neq j$, Algorithm 1 ensures delivery of the query to at least one of i and j. This result is correct, if all keys in the intersection of LR^{0}_{i} and LR^{0}_{j} are available in both TR^{0}_{i} and TR^{0}_{j} . 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 ranges:**

ALR : for all *i* and *l* : i ≠ j implies LR^0_i and LR^0_j are disjoint.

Theorem 3.1: *AU, AN and ALR are sufficient conditions for the 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 $<r, j> \epsilon RT^l_i$, $d_t \epsilon r$ must always be found at Line 7. AU ensures that, if such an $<r_1, p>$ is found at level *l* of the current processor *i*, an entry $<r_2, q>$, $d_t \epsilon r_2$ 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 such that $d_t \epsilon LR^0_p$. ALR 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 AU , AN and AR, in general, is a weakly-consistent structure, because several conditions valid in the consistent decentralized B-tree structure have been relaxed. For example, in the normal B-Tree structure, we have a stronger invariant of navigablity

AN(strong) : for all *i* and *l* : $<r, j> \epsilon RT^l_i$ implies $r = LR^{l-1}_i$

where r is equal to LR^{l-1}_i instead of included. Also, in the consistent structure, each level of the routing table contains a self-pointer, i.e. the condition

Self-pointer: for all *i* and *l* : there exist $<r, j> \epsilon RT^l_i$

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 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, F2 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.

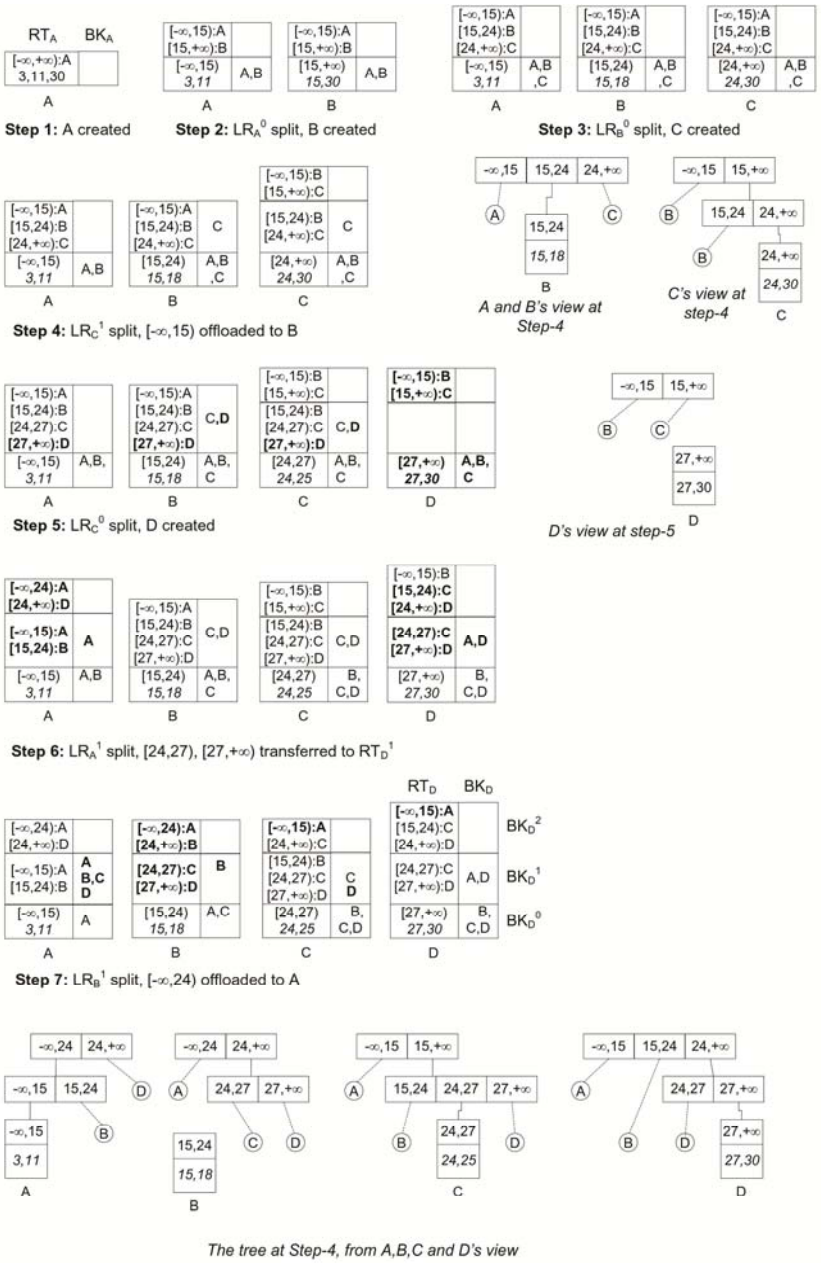


Fig. 3. Evolution of a weak-consistent B-tree with asynchronous updates. To facilitate asynchronous update, each processor i maintains a table BK_i in addition to RT_i . BK^l_i holds the names of all processors that hold RT^l_i responsible for the whole or some part of LR^l_i

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, F2 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 a 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 AU , AN and ALR 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 . $j \in BK_i$ if and only if $\langle r, i \rangle \in RT^{l+1}_j$.

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^0_i of processor *i* into 2 disjoint ranges LR_1 and LR_2 , and offloading LR_2 to a newly recruited processor *j*. Because *i* loses part of LR^0_i , for all p ϵ BK^0_i , RT^1_p need to add entries pointing to the new processor *j* instead of *i* for the lost part of the range. BK^0_i may include *i* if RT^1_i has a self-entry (Lines 6-13). In addition to the leaf level, the topmost level of the new processor j's routing table, RT^m_j is initialized by a replica of RT^m_i (Line 13). Mid-levels of RT_j remains empty. For the nodes newly pointed to by *j* at level m, their backward pointers are updated (Line 14).

Algorithm 3 is executed when processor *i* wants to offload some entries from its routing table RT^l_i 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^l_j , either already contains some entries covering some common range with LR^l_i, or, have some space to take few entries from RT^l_i. In a consistent distributed B-tree, we have for all j : if <r, j> ϵ RT^l_j then $RT^l_j = RT^l_i$. Thus, neighbours in RT^l_i are natural target for offloading part of RT^l_i. In the weak-consistent structure, it is not certain that such a *j* will be found in RT^l_j , 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^l_i with empty space in RT^l_j .

Once a suitable *j* is found, the update procedure is straightforward. The entries are transferred from RT^l_i to RT^l_j and BK^{l-1}_p are updated for the processors corresponding to the transferred entries (Lines 6-8). Then for the processors in BK^l_i, i.e. those who held *i* responsible for some part of LR^l_i , now need to update for the range shifted to *j*, by adding a new entry in the level l + 1 of their routing tables. Backward pointers of *i* and *j* are also updated accordingly (Lines 9-13). Finally, if the topmost level of RT^l_i is split, one additional level is added to hold the pointer to the transferred range, such that the whole universe is described.

Algorithm 2: SplitLeafNode(*i*)
1: **Initiator:** processor i
2: **Condition:** RT^0_i is overloaded, in terms of storage or access load
3: **Action:**

4: Partition RT^0_i into two disjoint sets of keys D_1, D_2
and LR^0_i into two corresponding ranges LR_1, LR_2
5: Find a new processor *j*
6: $RT^0_i := D_1$; $RT^0_j := D_2$;
7: $LR^0_i := LR_1$; $LR^0_j := LR_2$;
8: **for all** p ϵ BK^0_i **do**
9: there must exist <x,i> ϵ RT^1_p with x = LR^0_i
10: $RT^1_p := RT^1_p \setminus$ <x,i> union {< LR_1, i> , < LR_2, j> }
11: add {p} to BK^0_j
12: **end-for**
13: $RT^m_j := RT^m_i$ where m is the topmost level of RT_i
14: **for all** p **such that** <r, p> ϵ RT^m_i **do** add {j} to BK^{m-1}_p

Algorithm 2: SplitNonLeafNode(*i, l*)
1: **Initiator:** processor i
2: **Condition:** RT^l_i has too many entries or is causing too much routing load
3: **Action:**
4: find some existing processor j such that RT^l_j is empty
or has some overlap with LR^l_i
5: partition LR^l_i into two subranges R_s and R_x
where R_x is equal to the overlap (if there is one)
and partition RT^l_i into two corresponding sets E_s and E_x
6: $RT^l_i := RT^l_i \setminus E_x$
7: if there is no overlap do $RT^l_j := E_x$
8: **for all** p **such that** <r, p> ϵ E_x **do** delete {i} from BK^{l-1}_p and add {j}
9: **for all** p ϵ BK^l_i **do**
10: there must exist <x,i> ϵ RT^{l+1}_p where x is included in LR^l_i
11: remove <x, i> from RT^{l+1}_p and add < R_s, i>
12: add < R_x, j> to RT^{l+1}_p
13: add {p} to BK^l_j ; **if** x ∩ R_s = ϕ **do** remove {p} from BK^l_i
14: **if** *l* is the topmost level of the routing table RT_i **do**
15: RT^{l+1}_i := {< R_s, i> , < R_x, j> } ; add {i} to BK^l_i

For finding an existing processor j that can be used for off-loading some of the entries from RT^l_i (see line 4 of the SplitNonLeafNode algorithm), we have explored two algorithms by simulation [12] : (1) The Ping algorithm checks the descendants of RT^l_i in the distributed B-Tree to check whether their processor contains at level *l* en empty routing table or a table that has an overlap with RT^l_i . If no suitable processor j is among them, then the algorithm checks the next-lower descendents of RT^l_i in the tree, possibly until level 0 is reached. (2) The Ping-Pong algorithm goes down one level (like the Ping algorithm) but then follows the back-pointers that point to routing tables at level *l*. Because of the two steps, a larger number of processors is reached. Again, if no suitable processor j is found, the Ping-Pong process is repeated by going down two levels and going up two levels, and so on.

Our simulation studies confirmed that both of these algorithms always find a suitable processor j in reasonable time on average [12].

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, described in detail in an earlier version of this paper [13], describes the update procedure for such a merger. First a suitable partner *j* for the merger is selected among the processors backpointed at level 0, such that RT^1_j points to *i* for some range x included in LR^0_i. If RT^1_i is pointing to *j* for some other range y, then after the merger, the two entries <x, *i*> and <y, *j*> can be merged into <x union y, *j*>. Because processor *i* is being released, all levels of its routing table are merged with the corresponding level of *j*'s routing table. Accordingly, for all *l* and $p \in BK^l_j$, RT^{l+1}_p are updated.

Similar to the level-0 merger, if any other level *l* of the routing table of a processor *i* is found under-loaded, the entries of that level can be merged with the same-level entries in another processor. The merging partner, *j*, is found in a similar way as before, among the $p \in RT^{l+1}_p$, so that after the merger one entry is eliminated there. If RT^{l+1}_p is the topmost level, and contains only one entry after the merger, then that level may potentially be eliminated. This procedure, called Algorithm 5, is also described in [13].

4.3 Proving the invariants

Theorem 4.1: *The update algorithms, Algorithms 2, 3, 4 and 5 maintain the invariants AU , AN and ALR .*

AU : Algorithm 2 maintains AU in the newly joined processor *j* by copying the top level of the routing table of *i* (Line 14). In Algorithm 3, the range LR^{l+1}_p in all $p \in BK^l_i$ remains unchanged by the modification following Line 9. If RT^l_i is the topmost level in RT_i then the additional update in Line 14 ensures AU for processor *i*. Concerning Algorithms 4 and 5, see [13].

AN : AN can be violated only when LR^l_i for some processor *i* and some level *l* is reduced. In Algorithm 2, LR^1_i is reduced, and so, RT^1_p is updated for all $p \in BK^0_i$ to maintain AN (Line 10). A similar update is performed in Line 13 of Algorithm 3, for the reduction in LR^l_i.

ALR : Violation of ALR is possible only when RT^0_i 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 7). In Algorithm 4, LR^0_i and LR^0_j are merged into LR^0_j , and then processor *i* is removed. So no overlap is created. Algorithms 3 does 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 AU , AN and ALR are valid. So, by induction over successive updates, it can be proved using Theorem 4.1 that all three invariants are always maintained for the structure. Also, all four update algorithms work assuming the three invariants only. Validity of the back-

ward pointers is also maintained in these algorithms whenever a forward pointer is updated.

5 Discussion

As mentioned in Section 2.2, we use a decentralized distributed implementation of the B-Tree similar to the BPTree [10] or as in [3]. The main difference is that we relax in this paper the requirement for global consistency of the B-Tree data structure. Through the use of the weak consistency the tree update operations (split and merge) can be performed much more effectively. In the decentralized B-tree architecture, the average number of processors involved in a routing table split operation at level *l* (the SplitNonLeafNode algorithm described above) is (2 + b) which is independent of the level. Here the number 2 accounts for the processor *i* of the node being split and the node *j* to which some of the entries are transferred. b represents the average number of back-pointers of the split node. This value is equal to the average fan-out of a node, which is ¾ d (if we assume that the relation $\lceil d/2 \rceil \leq n \leq d$ is maintained).

In the case of a B-Tree with global consistency, all copies of the routing table at level *l* must be updated concurrently in a single transaction. Since the number of copies of a routing table at level *l* is b^l, this becomes a very big number when the level is close to the root, and if the root table must be updated, this involves all processors.

This shows the main advantage of weak consistency. The main disadvantage of weak consistency is the fact that it is more difficult to find a suitable processor for node splitting or merging due to the irregular structure of the B-tree after repeated data insertions and deletions. We note, however, that the average number of tree nodes to be updated due to a single data object insertion is the same for strong and weak consistency. If we assume that the number of data objects per processor is also limited by d, then the probability that a data insertion leads to the splitting of a leaf node is equal to 2/d (assuming that the number m of data objects is in the range $\lceil d/2 \rceil \leq m \leq d$). And such a split will lead to the update of b routing tables at level 1, where again, for each of these updates, there is the probability of 2/d that the routing table at level 1 will be split (i.e. b * 2/d = 1.5 splits at level 1), and so on. Therefore, the average number of routing tables to be split after one data object insertion is given by $2/d\ (1 + 1.5 + 1.5^2 + \ldots + 1.5^N)$. In the case of the B-Tree with weak consistency, these splits can be performed as separate transactions, each involving only (2+b) processors (as mentioned above). In the case of strong consistency, the probability of having a split at level *l* after a data object insertion is equal to $(2/d)^{l+1}$, however, the number of processors involved for an split at level *l* would be $(\frac{3}{4}\ d)^{l}$, which becomes prohibitive for the root node.

6 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. Through our simulation studies, we have shown that the depth of the tree can be maintained over a long period of tree update operations at the optimal level of L = log(N), where N is the number of processors in the system.

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, pp.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. Unpublished document, http://arxiv.org/abs/0903.3759, 2009.
4. A. Bar-Noy and D. Dolev. A partial equivalence between shared-memory and message-passing in an asyn. fail-stop distr. env., 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.
12. K.B. Hafaiedh, Studying the properties of a distributed decentralized B+ tree with weak consistency, Master Thesis, University of Ottawa, Oct. 2012.
13. S. Asaduzzaman, G.v. Bochmann, Distributed B-tree with weak consistency, unpublished report, see http://www.site.uottawa.ca/~bochmann/Curriculum/Pub/2010 - Distributed B-tree with weak consistency.pdf
14. S. Asaduzzaman, G.v. Bochmann, A locality preserving routing overlay using geographic coordinates, IEEE Intern. Conf. on Internet Multimedia Systems Architecture and Application, Bangalore, India, Dec. 2009.