Efficient Bulk Updates on Multiversion B-trees

Daniar Achakeev and Bernhard Seeger
Philipps-Universität Marburg
Marburg, Germany
{achakeye,seeger}@mathematik.uni-marburg.de

ABSTRACT

Partial persistent index structures support efficient access to current and past versions of objects, while updates are allowed on the current version. The Multiversion B-Tree (MVBT) represents a partially persistent index-structure with both, asymptotic worst-case performance and excellent performance in real life applications. Updates are performed tuple-by-tuple with the same asymptotic performance as for standard B^+trees. To the best of our knowledge, there is no efficient algorithm for bulk loading and bulk update of MVBT and other partially persistent index structures. In this paper, we propose the first loading algorithm for MVBT that meets the lower-bound of external sorting. In addition, our approach is also applicable to bulk updates. This is achieved by combining two basic technologies, weight balancing and buffer tree. Our extensive set of experiments confirm the theoretical findings: Our loading algorithm runs considerably faster than performing updates tuple-by-tuple.

1. INTRODUCTION

Undoubtedly, the importance of temporal data management has grown in recent years. An efficient processing of historical data enables a wide range of new real world applications, e.g. fraud detection and complex business analytics. Organization of historical data is also of great importance in science [26] and Web applications. The need for temporal database support has led to the development of so-called transaction-time database systems offered by many commercial vendors like Oracle [23], IBM [16], Teradata [2] and Microsoft [21].

Databases with transaction time support are characterized as follows: Firstly, they are continuously growing in size, since updates of data items always preserve the old version, while deletions are only logically performed. Because linear scanning historical versions becomes extremely expensive (at query time), there is an urgent demand for efficient temporal indexes. In addition to supporting (range) queries on arbitrary old versions and ranges of versions, these indexes must have excellent update performance, in order to cope with high volumes of continuously arriving updates. The required update performance cannot be achieved by simply applying updates one by one. Instead, updates have to be applied in bulk. Thus, an index on transaction-time databases have to support bulk updates efficiently. For an initially empty index, this problem is also termed bulk loading.

The problems of bulk updates and bulk loading have been adequately addressed for B-trees. Unfortunately, traditional B-trees are not appropriate for efficiently indexing transaction-time databases. This has led to the development of many temporal extensions for B-trees [25]. The multiversion B-tree (MVBT) [8] is the first partially persistent index-structure with optimal worst-case guarantees for insertions, updates, deletions and temporal key-range queries. However, efficient algorithms for bulk loading and bulk updates are available neither for the MVBT nor for other partially persistent B-trees like e.g. the time-split B-tree [20, 21]. The design of efficient bulk algorithms for partially persistent B-trees is still challenging [20, 8].

In this paper, we address bulk update and bulk loading problem of partially persistent B-trees. Without loss of generality, we discuss these algorithms in the context of the MVBT. In contrast to previous results [9], we show that the original buffer tree technique alone is not appropriate for loading MVBTs. Updates have to be applied in a certain order, however, the buffer tree introduces too much asynchrony within the loading process, leading to a serious violation of the MVBT invariants. Here, we present the first asymptotically optimal bulk loading algorithm that meets the lower bound of external sorting. Our novel approach is based on a sophisticated combination of the buffer tree [6] and weight balancing technique [7]. Weight balancing is necessary for controlling the degree of asynchrony. Our loading algorithm is directly applicable to bulk updates on MVBTs with only very little changes. The excellent performance of our algorithms comes at the little price that the generated index levels are different to the ones of the original counterpart. However, all asymptotic performance guarantees of the MVBT are still met. Because of the structural differences, we decided to use the acronym $MVBT^+$ for the resulting partial persistent B-tree generated by our bulk loading or bulk update algorithms. We summarize the contributions of this paper as follows:

- This work presents the first asymptotically optimal bulk loading algorithm for partially persistent index-structures, e.g. MVBT. Bulk loading only requires the same asymptotic I/O cost as external sorting, while all

Proceedings of the VLDB Endowment, Vol. 6, No. 14

other performance guarantees of the original MVBT are still maintained.

- We show that the loading algorithm can also be used for efficient bulk updates on the MVBT.
- Our experiments show that our bulk algorithms are not only of theoretical interest, but also of practical relevance. Then our bulk algorithms provide substantial performance improvements in comparison to applying updates one by one (which is the standard method today).

The remaining part of the paper is organized as follows: In Section 2, we introduce a few preliminaries. We review related work in Section 3 and discuss the MVBT in detail. In Section 4, we outline the basic idea of our new bulk loading algorithm. In addition, we show that it is impossible to use the original buffer tree for loading the MVBT. In Section 5, we present the details of our asymptotically optimal bulk algorithms and introduce MVBT$^+$. We outline the theoretical results on runtime in Section 5.3. In Section 6 we briefly present our algorithm for bulk update. We discuss our experimental result in Section 7. Section 8 concludes the paper.

2. PRELIMINARIES

In this paper we tackle the problem of managing records in a partially persistent file consisting of multiple versions. A versioned record in the file is given by $< k, t_s, t_e, inf >$ where k is a key. $[t_s, t_e)$ represents a version interval in which the key is valid, and inf is the payload. A versioned record is alive in the most recent version if its t_e field carries the special character "*". Otherwise the versioned record is dead. Versioned records can be depicted as intervals in a two-dimensional space, consisting of a time dimension (X-axis) and a key dimension (Y-axis). The i-th version of the partially persistent file consists of all versioned records $< k, t_s, t_e, inf >$ with $i \in [t_s, t_e)$. Update operations are allowed only on the most recent version, but queries are supported on any version. Whenever an update operation (insert, delete) is posed, a new version now is created and a new record with version interval $[now, *)$ is inserted (in case of insert) or a live record is deleted (in case of delete). Note that a deletion corresponds to closing the interval of a live entry by assigning now to the t_e field. An update on a versioned record is simply a concatenation of insert and delete (without incrementing the version number before delete).

Due to the excellent worst case performance, a partial persistent B-tree, e.g. MVBT [8], is used as underlying structure for supporting queries on arbitrary version and version ranges [10]. The leaves of the MVBT consist of versioned records. In addition, the version concept is also carried over to the index entries, i.e., an index entry also comprises a time interval $[ts_i, te_i)$.

In the following, we consider the bulk loading and bulk update problem for the MVBT. Let us consider a sequence of input records $e_i =< k, inf, ops >$, $1 \leq i \leq N$, where $ops \in \{insert, delete, update\}$. For $i = 1, \ldots, N$, the operation of e_i is performed on the most recent version of the MVBT using tuple $< k, inf >$ as input. For bulk loading we assume the initial tree to be empty, while the tree already consists of N' live records for the bulk update problem.

Symbol	Description
N'	number of live records
N	problem size (in number of operations)
N_i	number of entries live at version i
M	memory capacity
B	block capacity
d	minimal live entries per block

Table 1: Important notations

These problems differ from the equivalent ones on ephemeral indexes, e.g. B-tree and R-trees, in the sense that the partial persistent semantics requires a strict ordering of the update operations. In fact, this renders a direct application of traditional loading techniques impossible.

We use the classical I/O model proposed in [1], where storage is partitioned into pages of fixed size. B is the maximum number of records per page and M denotes the available memory (in terms of records). Our problem in this paper are discussed for a centralized setting. An extension to a distributed setting is possible for the loading problem by splitting the input in the time dimension into partitions using optimal splitters [18] and building up a MVBT for each partition in parallel.

For the MVBT we use the following notation: N is the number of updates, N_i denotes the number of records live at version i. Parameter $d \in \Theta(B)$ denotes the minimum number of live records in a page. We count tree levels $l = 0, 1, \ldots,$ bottom up starting from the leaf level. Our notations are summarized in Table 1.

3. RELATED WORK

In this section, we review the problems related to partial-persistence, corresponding index structures, as well as their methods for bulk-loading und bulk-updates. In one subsection, we discuss in detail MVBT that serves as the target index for our bulk loading and bulk update techniques.

Partial persistence is a well-known concept in algorithm design [11] and computational geometry [13, 5]. In the field of databases, the problem of partial persistence is better known as versioning, transaction-time and system-time [17]. Many commercial vendors like Oracle [23], IBM [16], Teradata [2] and Microsoft [21] have developed systems for transaction-time support. Versioning is also an integral part of key-value stores [3]. Recently, versioning has also played a key role in developing robust transactional file systems [24]. Event processing system can take benefit from storing temporal data in event stores [12, 15].

In order to support (range) queries for old versions with the same query performance as an ordinary B$^+$tree on these versions, special partially persistent B-trees have been proposed [20, 8, 5, 28, 21, 30]. An excellent survey is given by Salzberg and Tsotras [25]. The MVBT has been the first partial persistent B-tree with asymptotically optimal performance. MVBT and other partial persistent B-trees have significantly influenced index design for moving object databases [27, 14]. However, the problem of bulk loading and bulk updates have not been addressed in these papers.

To the best of our knowledge, the first approach to bulk loading a partial persistent B-tree has been proposed by Goodrich et al. [13]. They built a persistent B-tree with

branching degree $\sqrt{\frac{M}{B}}$ in I/O complexity of external sorting to solve a geometric off-line problem. Our problem differs in that we address the loading of an online persistent B-tree with branching degree B, a problem that is so far unsolved [13]. Bulk loading of the MVBT was already addressed by Bercken et al. [9]. The authors used a generic buffer tree framework [4, 9, 6]. Although this approach is applicable to loading R-trees, loading of MVBT is only possible for insertions only. For mixed workloads consisting of insertions, deletions and updates, this approach cannot be used. We will show major problems of this approach in Section 4. Our new loading algorithm is applicable to arbitrary workloads while all asymptotic performance guarantees of the MVBT are fully maintained.

An interesting loading algorithm for R-trees is presented in [6], using the buffer-tree framework. This algorithm loads the leaf level and index levels simultaneously. The advantage is that this approach is not limited to bulk loading only, but also suitable for bulk update. Unfortunately, the loading approach cannot be used for partially persistent B-trees. However, our new bulk loading algorithm also loads all levels simultaneously. Therefore, our loading algorithm can also be used for bulk updates as well, with very little changes.

Recently, Zhang et al. [30] presented a memory optimized tuple-by-tuple on-line loading algorithm for the HV-tree, an advanced version of the Time-Split B-tree (TSBT) [21]. The primary goal was to provide fast access to recent data in memory and to move old data efficiently to secondary storage. In contrast to our problem, the HV-tree assumes that all live nodes can be kept in memory. This assumption is not always valid, as the size of databases can still be larger than the available main memory. As a consequence, no worst-case performance guarantees are given. In addition, the loading algorithm still relies on executing one update at a time, while our approach achieves substantial higher improvements of the bulk update time from processing updates in batches.

Due to the continuously growing size of a versioned database, distributing this data among multiple nodes is becoming more and more important. In [18], a new method is presented for determining splitters for a set of versioned records (represented as intervals in a two-dimensional space). This method could be easily combined with our loading algorithms to obtain a distributed loading technique. In this paper, however, we focus on the centralized case and leave a detailed discussion on distributed techniques for future work.

3.1 MVBT

MVBT (multiversion B-tree) is an asymptotically optimal partial persistent B$^+$tree. It has $O(N)$ space complexity and supports (key range) queries at version i with the same asymptotic complexity as an ordinary B$^+$tree that only stores the i-th version. I/O time for the i-th update is $O(\log_B N_i)$. Loading of MVBT is performed update by update and requires $O(N \log_B N)$ I/Os in the worst case. MVBT is actually a direct acyclic graph (DAG), providing a condensed physical representation of N B$^+$trees (one for each version) [8]. As proposed by Discroll et al. [11], MVBT stores pointers to historical roots in a separate (B$^+$tree) termed *root**. The DAG and root* of a MVBT are illustrated in Figure 1.

The asymptotic bounds on query and update time are achieved by preserving the so-called *weak-version condition*:

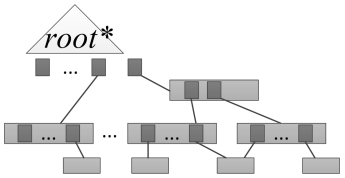

Figure 1: MVBT structure.

A linear fraction of the capacity $d = \frac{B}{4}$ in a live node is reserved for live data. The remaining portion $\frac{3B}{4}$ can be used for historical (dead) entries. For the sake of simplicity, we use these specific settings throughout the paper without loss of generality. We refer to [8] for a detailed discussion on parameter settings.

Reorganization of a live node is triggered if there are not enough live entries in the node (i.e., the weak version condition is violated) or the physical capacity B is exceeded. In order to use only linear space, the so-called *strong-version* condition has to be satisfied: the number of live entries is to be between $\frac{3B}{8}$ and $\frac{7B}{8}$ for nodes that just have been involved in a reorganization. Therefore, such a node accepts at least $\Theta(B)$ updates (insertions, deletions) until its next reorganization will be triggered.

We discuss the specific reorganization operations of MVBT using the four two-dimensional partitionings of the time-key space shown in Figure 2. Each (leaf) node of MVBT corresponds to a rectilinear rectangle. Intervals represent the versioned records; black and red ones refer to dead and life entries, respectively. We assume that an update at version t_i triggers a reorganization. Reorganization of a node always starts with a time split where live entries at version t_i are copied from node v to a new live node v_l (see left upper plot in Figure 2). If the strong version condition is violated for v_l, additional reorganization steps are triggered. If v_l has more than $\frac{7B}{8}$ live entries, a *key-split* is first performed, see right upper plot in Figure 2. Similar to a split in a B$^+$tree, entries are evenly distributed among two nodes using a split value from the key dimension. If v_l contains less than $\frac{3B}{8}$ live entries, a *merge* with a live sibling node v_n is triggered. After a time split on v_n, live entries from v_n are inserted into v_l, as seen in the left lower plot of Figure 2. If the number of live entries in v_l is greater than $\frac{7B}{8}$, an additional key-split has to be performed on v_l. This is illustrated in the right lower plot of Figure 2. Thus, the two new live nodes satisfy the strong-version condition. Hereafter we use the term *node reorganization* to refer to time split, merge or key-split. Note that at most two new nodes can be created during one reorganization.

Algorithm 1 describes the insert procedure of MVBT given a record $e =< k, inf >$ at time ts. The path to a live leaf node is computed in lines 1-4. In each level *chooseSubTree* searches for the matching live index entry using key k. Afterwards, the versioned record is inserted in the leaf node. If either the weak version condition or the capacity constraint is violated, a reorganization of the leaf node will be triggered (lines 5-7).

Index entries of new live nodes are inserted in the parent node (lines 8-18). If the live root is reorganized, a new live root is created (lines 10 - 15) and a corresponding index entry is inserted in *root**. In case of an additional key-split, the height of the live MVBT tree increases (lines 12-13).

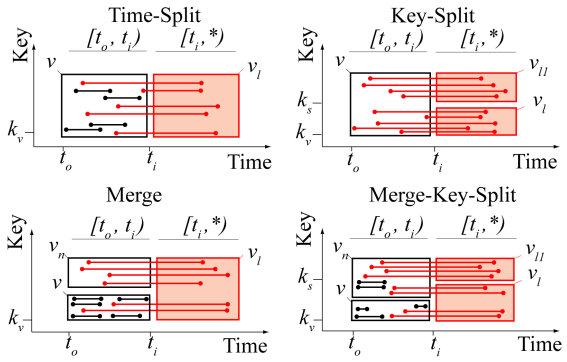

Figure 2: MVBT node reorganization operations

Otherwise, the height remains the same and the live root is replaced by its temporal successor (line 14-15). In order to support efficient key-range queries over a given time interval, a leaf is linked with its predecessor(s), see line 7 in Algorithm 1. Every leaf manages up to two backward pointers to its temporal predecessor(s). This allows to start the processing at the right border of the search rectangle and to traverse backwards through the leaf level. The details for query processing are given in [10].

Algorithm 1: Insert

```
   Input: Entry e, Time Stamp ts
1  cR ← root;
2  while cR does not point to leaf do
3      node ← GetNode (cR) and push node in Path;
4      cR ← ChooseSubTree (node, e, ts)//search live
       index entry;
5  InsertLeaf ( leaf ← GetNode (cR) , e, ts);
6  idx[] ← SplitNode ( leaf, ts)// perform reorganization
   if needed, if merge or merge-key-split find neighbor;
7  link created successor nodes with leaf;
8  while idx[] is not null do
9      get parent node, logically delete cR, parent ← pop
       Path;
10     if parent is null then
11         store cR as historical root ;
12         if was key-split then
13             create new root node and insert new live
               successors idx[] in it;
14         else
15             replace root with new created live successor;
16     else
17         insert new live successors idx[] in parent;
18         idx[] ← SplitNode ( ParentNode, ts);
```

4. BASIC IDEAS OF BULK LOADING

In this section, we outline our approach to bulk loading a partial persistent B-tree, which is closely related to MVBT. Our goal is to provide a loading solution that requires $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os. In fact, this is the lower bound for loading because external sorting and loading of ordinary B-trees cannot be faster.

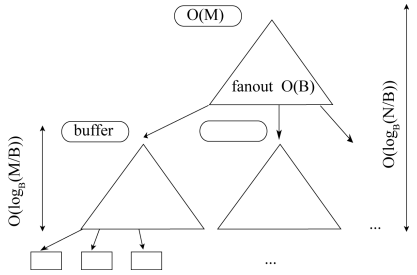

Figure 3: Buffer Tree by Arge et al.

In order to design an efficient loading algorithm, we use two techniques in combination with the MVBT. The one is the *buffer tree* technique [6] and the other is *weight balancing* [7]. We call this MVBT extension MVBT$^+$ because it maintains the worst-case performance properties of MVBT and additionally supports efficient loading in asymptotically optimal number of I/Os. Each of these two techniques contributes to the efficiency of MVBT$^+$:

1. The buffer tree yields the same I/O time as external sorting. The key idea of this technique is to transport data in batches between levels. Figure 3 shows the general buffer tree architecture with buffers attached to subtree roots.

2. Weight balancing controls the synchrony of buffer emptying processes. It guarantees that the MVBT$^+$ still maintains the MVBT invariants without giving up its worst-case performance.

The buffer tree attaches buffers to the nodes on each $i \cdot \left\lfloor \log_{B/4}\frac{M}{16B} \right\rfloor$ with $i = 1, 2, \ldots \Theta(\frac{\log_B N/B}{\log_{B/4} M/16B})$ level, see Figure 3. The buffer size is limited to $\frac{M}{2B}$ pages. This allows to keep all live nodes of a sub-tree of height $\log_{B/4}\frac{M}{16B}$ in memory using at most $\frac{M}{2B}$ I/O. In Section 5.2 we explain the choice of these parameters in detail. We use the following terminology hereafter: a leaf (node) is the node on level 0, an index node is on level $l > 0$, a buffer node is the node on levels $i \cdot \left\lfloor \log_{B/4}\frac{M}{16B} \right\rfloor$ with $i = 1, \ldots$. A buffer is associated to a buffer node.

4.1 The Problems of Buffer Trees

In the following, we show that is not sufficient to use the buffer tree only for loading a MVBT. In fact, synchronization of buffer emptying is required; As reorganizations within the MVBT are temporally ordered according to the time stamps of the updates. If the buffers are emptied only after they are filled completely, some subtrees will evolve uncontrolled in time. Moreover, the node reorganizations of the child nodes can force the parent node also to evolve in time. This becomes a serious problem when other child nodes still contain historical data in their buffers. When these buffers are emptied later in time, the parent node as well as siblings could already be dead. As a consequence, an insertion of their index entries in the parent node is not allowed, and a required merge with a sibling is impossible. In both cases, the MVBT invariants are violated.

The first problem termed *parent-child problem* is illustrated in Figure 4. Initially, there are one parent node p and two child nodes u and v. A time-split of u also leads to

a split of parent p. The new parent node p' is created with time interval $[t_8, *)$, while its child v is already vaild at time t_2. Later the buffer emptying of node v creates two nodes v' and v''. The time interval of v' is $[t_4, t_6)$ which does not fit to the time interval of parent node p'. Therefore, it would be required to insert an index entry (referring to v') into the dead node p. However, an insertion into a dead node is not allowed for MVBT.

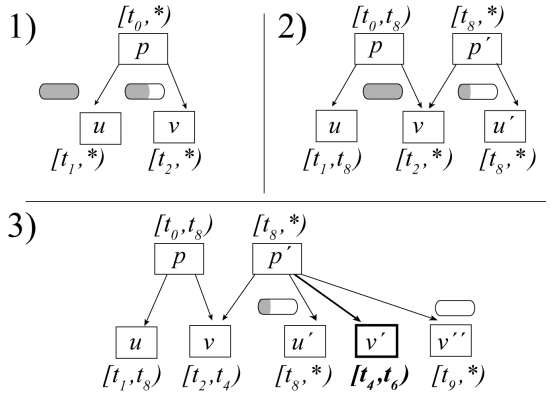

Figure 4: Parent-Child problem

The second problem termed *sibling problem* is illustrated in Figure 5. Initially, there are one parent p and two child nodes u and v. Further, node u is a key sibling of v. Node u evolves much faster than its sibling v. This causes a time-split at t_{10} and a new node u' is created with a time interval $[t_{10}, *)$. Later in time, the buffer of node v is emptied. Due to the historical data in the buffer, a time split is performed at time t_3. Therefore, the interval of v is closed and a new node v' with a time interval $[t_3, *)$ is created. Because v' contains less than $\frac{3B}{8}$ records, a merge is triggered with a node that is alive at t_3. This would be the key sibling node u, but u is already dead. However, a merge with a dead node is not allowed for the MVBT.

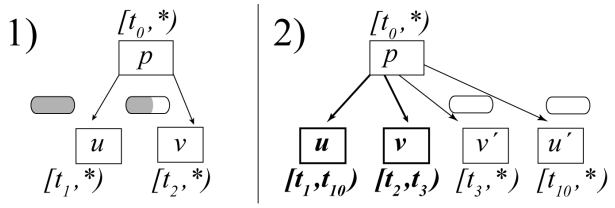

Figure 5: Sibling problem

4.2 A Case for Weight Balancing

In order to avoid the parent-child problem and the sibling problem, we applied the weight balancing technique [7]. The idea is to prevent these problems even without knowing the precise closing time of an entry time interval. Instead, we introduce a so-called *safe interval* where the closing time is estimated by the lower-bound of the number of operations required until the next reorganization will happen. Only if there is no overlap among two safe intervals (belonging either to siblings or to a parent and child), an additional reorganization step will be triggered. The larger the safe intervals, fewer of these forced reorganizations are necessary. Similar to B-tree, MVBT requires only $\Theta(B)$ operations to trigger the next reorganization after the last was performed. Unfortunately, this causes very short safe intervals and many forced reorganization steps. In contrast, weight balancing allows much more operations until the next reorganization step has to be triggered. More precisely, the number of operations is asymptotically equal to the number of records in the associated subtree. This leads to very long safe intervals and a very low probability that there is no overlap among safe intervals. Moreover, when a buffer of a subtree has to be emptied and the time interval of the associated entry moves to the future, the safe interval of the sibling is forced to overlap. If necessary, the buffer of the sibling has to be emptied first.

Similar to [7], weight information has to be maintained for each node. However, we use two different weight counters for a node v. The live weight $w(v)$ tracks the live records in the associated subtree (including its root buffer), whereas the operation weight $t(v)$ tracks the number of update and insert operations. The live weight is used to preserve the MVBT invariants, e.g. weak version condition. The operation weight $t(v)$ allows to estimate the closing time of the safe interval.

Both weights capture the temporal progress of a node. We constrain the ratio of node weights on the same level by a constant. With each level the weights $w(v)$ and $t(v)$ increase exponentially. Further, at least $O(w(v))$ operations are needed to reorganize node v on level l again.

Weight balancing requires a weight information attributed to each node and maintained during the loading process. This results in the following modifications of the original buffer tree emptying process:

- Weight information is updated during buffer emptying process.
- Buffer emptying is triggered either if the buffer is full or the weight conditions of the node are violated.
- Buffer emptying is forced if the node safe intervals do not overlap. However, as we apply weight balancing, I/O costs will not asymptotically increase.
- Node reorganization is performed in a top-down manner only. By this, the new entries can be stored in the parent node without causing any overflow again. This facilitates the implementation of the buffer tree (in comparison to its original counterpart).

Due to this top-down node reorganizations and emptying of buffers, we avoid *parent-child* problem and the *sibling problem*. As a consequence, the loading time of our approach is asymptotically optimal. The details are given in the next section.

5. BULK LOADING DETAILS

In this section we explain our MVBT$^+$ bulk loading methods. We first discuss the loading process using three procedures *Bulkload*, *ClearBuffer* and *PushDownEntry* (see Algorithms 2, 3 and 4). We also discuss how to apply the weight-balancing technique to loading. The main theorem proof is outlined at the end of the section.

Algorithm 2: Bulkload

```
   Input: InputData
1  initialize S, rootBuffer, root points to first leaf node;
2  foreach Entry e in InputData do
3      if rootBuffer size ≥ M/4 then
4          for i = 1 to M/4 do
5              PushDownEntry (dequeue entry, root, null, S,
               ts(entry));
6          foreach idx in S do ClearBuffer (idx) ;
7      append e to rootBuffer;
8  ClearAllBuffers ();
```

Algorithm 3: PushDownEntry

```
   Input: Record e, Index Entry idx, Parent Node v_p,
          Stack S, Time Stamp ts
1  cR ← idx ;
2  if cR violates weight condition then
3      splitType ← ComputeSplitType () ;
4      if cR is not root and has buffer then
5          remove cR from S and ClearBuffer (cR) ;
6          if splitType is merge or merge-key-split then
7              nR ← find neighbor of cR;
8              remove nR from S and ClearBuffer (nR);
9      idx[] ← SplitNode ( v ←GetNode (cR) , v_p, ts,
       splitType) ;
10     if splitType is key-split then
11         reassign cR depending on key split ;
12     if v_p is null then
13         new root node handling ... // see Algorithm 1
           lines 10-15 ;
14     else
15         ExpandParent (idx[]);
16 UpdateWeight (cR);
17 if cR has buffer and buffer size ≥ M/4 and not yet in S
   then
18     push cR in S ;
19 if cR is not root and has buffer then
20     Enqueue (e, buffer of cR) ;
21     return;
22 if cR points to leaf node then
23     load leaf node and insert entry e into leaf node;
24     return;
25 v ←GetNode (cR) and cR ← ChooseSubTree (v, e) ;
26 PushDownEntry (e, cR, v, S) //recursive call ;
```

5.1 Buffer Tree Loading

Loading starts at the root of a tree (see Algorithm 2). Data is pushed towards the leaf nodes in batches if either the root buffer is full or a node weight condition is violated. MVBT$^+$ points to a buffer of the current live root (see line 1). The batch size is $\frac{M}{4}$ (see line 3). Buffer nodes pointers with buffer sizes greater than $\frac{M}{4}$ are stored in stack S. Buffers of the buffer nodes stored in S are emptied right after the root buffer. Finally, function *ClearAllBuffers* (line 10) is called to empty the buffers of all buffer nodes in breadth-first manner (level-by-level).

For the $\frac{M}{4}$ entries of the root buffer *PushDownEntry* is called, see Algorithm 3. This function has the following parameters: record to be inserted, root entry of the subtree, its parent node and the pointer to stack S. It routes the entries to the next buffer nodes or leaf nodes using the original MVBT routing algorithm (see line 25). Routing through $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ levels of the subtree is done in memory. Note that we consider only live index nodes for routing. In Lemma 2, we show that the live nodes of a subtree of height $\log_{\frac{B}{4}} \frac{M}{16B}$ always fit in memory.

We also update weight information of the nodes in line 16 of Algorithm 3. The weight information is recorded in the corresponding index entries. Moreover, buffer nodes are pushed into S if the size of their buffer is greater than $\frac{M}{4}$ (see lines 17-18).

Before we route a record one level down, the weight conditions of the node are checked (see line 2, Algorithm 3). If the conditions are violated, node reorganization is triggered (merge, time-split, key-split or merge-key-split). For buffer nodes, we empty their buffers first, since we need to enforce an overlap of the safe intervals. This ensures that there is no entry in the buffer with a time stamp smaller than that of the entry to be inserted. Otherwise, some records could lie outside the lower interval boundaries.

Thereafter, the node is reorganized in a top-down fashion (see call *SplitNode*, line 9, Algorithm 3). For the routing, only the live part of a tree is considered. Logically deleted nodes are released from memory. Index entries of the newly created nodes are posted to the parent node (see function *ExpandParent*). It is crucial for our algorithm that *this function does not trigger recursive parent splits towards the root node* due to weight balancing. If we perform a key-split or a merge-key-split, we adjust the node pointer according to the key of the entry (line 10, Algorithm 3). The root node split is handled in line 12 of Algorithm 3 similar to Algorithm 1 lines 10-15.

The procedure *ClearBuffer* (see Algorithm 4) is executed if either one of the following cases occur. 1. the buffer has more than $M/4$ records. 2. the buffer node has to be reorganized (e.g. due to weight violation) or 3. the procedure *ClearAllBuffers* is executed (see line 8, Algorithm 2, lines 4-8 Algorithm 3). In case of 2 or 3, buffers are emptied completely. We push records in two batches of maximal size $M/4$. After processing each batch, we clear all child buffers with more than $M/4$ (see line 8 and 13, Algorithm 4). In addition, buffers belonging to the lowest buffer nodes are also completely emptied (even if they contain more than $M/4$ records). In all other cases, only the first $M/4$ records of a buffer are pushed down to the next buffer node. By this, we ensure that no buffer will have more than $M/2$ records and we avoid cascading buffer emptying processes [6].

5.2 Weight Balancing

Recall that we use two weight counters to track the temporal progress of a node v. Live weight $w(v)$ is used for live records. $t(v)$ tracks the number of update and insert records pushed to a node v (or to its buffer) on level l since its creation. We need this weight counter to properly estimate the safe interval as execution of the *update* operation (logical delete followed by insert) on a leaf node does not change the number of live entries.

Algorithm 4: ClearBuffer

```
Input: Index Entry idx
1  buffer ← load buffer of idx ;
2  v ← load the node of idx ;
3  initialize S;
4  for i = 1 to min(M/4, buffer size) do
5      e ← Dequeue (buffer) ;
6      cRoot ← ChooseSubTree (v, e ) ;
7      PushDownEntry (e, cRoot, v, S, ts(e)) ;
8  foreach i in S do ClearBuffer (i) ;
9  if idx has weight violation or level == ⌊log_{B/4} M/16B⌋
   or ClearAllBuffers then
10     foreach e in buffer do
11         cRoot ← ChooseSubTree (v, e) ;
12         PushDownEntry (e, cRoot, v, S, ts(e)) ;
13     foreach i in S do ClearBuffer (i) ;
```

The original index entries of MVBT are extended with two weight counters, w and t. MVBT$^+$ maintains w and t for each live node in its associated live index entry. Both counters are updated in a top-down fashion (see Algorithm 3 line 21). Every time a record is passed one level down, we update the node weight information as follows:

$$w(v) = \begin{cases} w(v)+1 & \text{if insert} \\ w(v)-1 & \text{if delete} \\ w(v) & \text{if update} \end{cases}$$

$$t(v) = \begin{cases} t(v)+1 & \text{if insert} \\ t(v) & \text{if delete} \\ t(v)+1 & \text{if update} \end{cases}$$

Recall that the original MVBT triggers time splits and further reorganizations like merges or key-splits only with respect to the number of physical entries. This occurs if a node has not enough live entries or the physical bound is achieved (see Section 3.1). We carry over the same idea to weight balancing. Thus, we limit live and operation weights $w(v)$ and $t(v)$ (length of safe interval) for each node. The allowed ranges exponentially increase with each level. The ratio of the minimal and the maximal value of $w(v)$ and $t(v)$ for level l is a constant.

We adapt weight balancing as follows: the branching factor a is set to $a = \frac{B}{4}$. The valid live weight of node v on level l is between:

$$\textit{live-condition: } a^l \cdot \frac{B}{4} \leq w(v) \leq a^l \cdot B \tag{1}$$

The valid operation weight $t(v)$ of the node v on level l is between:

$$\textit{operation-condition: } w(v) \leq t(v) \leq a^l \cdot B \tag{2}$$

Immediately after node reorganization, the live weight and operation weight of the newly created node should fulfill the following *Strong-weight-condition:* (see Table 2):

$$a^l \cdot \frac{3B}{8} \leq w(v) \leq t(v) \leq a^l \cdot \frac{7B}{8} \tag{3}$$

Setting	Description
$a = \frac{B}{4}$	branching factor
$a^l \cdot \frac{B}{4} \leq w(v) \leq a^l \cdot B$	valid live weight
$w(t) \leq t(v) \leq a^l \cdot B$	valid operation weight
$a^l \cdot \frac{3B}{8} \leq w(v) \leq t(v) \leq a^l \cdot \frac{7B}{8}$	strong weight condition
$i \cdot \left\lfloor \log_{\frac{B}{4}} \frac{M}{16B} \right\rfloor$, i= 1, ...	buffer levels
$\frac{M}{2}$	maximal buffer capacity

Table 2: MVBT$^+$ settings

Node reorganization is triggered in two cases: if either

$$w(v) \leq a^l \cdot \frac{B}{4}$$

or

$$t(v) \geq a^l \cdot B$$

These two conditions are checked in line 2 of Algorithm 3. In line 9 of the algorithm function *SplitNode* is called. It runs the node reorganization and logically deletes the current index entry in the parent node. Each reorganization of node v on level l starts with a time split. It creates a new node v_t and copies live entries into v_t.

The new live weight $w(v_t)$ and operation weight $t(v_t)$ are set to value of $w(v)$. If $w(v_t)$ violates *strong-weight-condition* (see Condition 3) one of the following operations are executed: merge, key-split and merge-key-split (see Table 3). In case of a merge, the live weight w_n of siblings have to be considered to choose the type of reorganization (see Table 3). We release the dead node from memory and use the free space for a new live node. In case of a merge operation we update $w(v_t)$ and $t(v_t)$ as follows $w(v_t) \leftarrow w(v_t) + w_n$ and $t(v_t) \leftarrow t(v_t) + w_n$.

The original key-split and merge-key-split of MVBT are also adapted to weight balancing. Key-splits on leaf nodes are identical to MVBT key-splits. This is different for index nodes. For these nodes, we iterate over the sequence of index entries and sum up their weights until the sum is at least equal to $(\frac{1}{2} \cdot a^l \cdot \frac{7B}{8})$ ($\frac{1}{2}$ of the upper bound of *strong-weight-condition*). Then, we split at the child entry where the iteration stopped. Afterwards, we adjust the weight counters of the created nodes. Since the maximal weight of this child is limited by $a^{l-1}B$, we find an almost balanced split. The branching factor a should be greater than 16. This follows from the following inequality [7, 8]:

$$\frac{1}{2} \cdot (a^l \frac{7B}{8}) - a^{l-1}B \geq a^l \frac{3B}{8} \tag{4}$$

Hence, this inequality ensures that a balanced split can be always found.

The lower bound of the live weight is $a^l \frac{B}{4}$. This ensures that after merging, the live weight of a new node is always greater than the lower bound of *strong-weight-condition* (since $2 \cdot a^l \frac{B}{4} \geq a^l \frac{3B}{8}$). The weight constraints and our reorganization operations guarantee that a minimum of $\frac{a^l B}{8}$ operations (inserts, deletes, updates) have to occur before node v on level l is reorganized again.

In line 15 of Algorithm 3, the newly created index entries are posted to the parent node (function call *ExpandParent*). Recall again that the insertions of these new entries do not cause a split of the parent node. Let us sum important

w	t	w_n neighbor live weight	Operation
$a^l(\frac{3B}{8}) < w < a^l(\frac{7B}{8})$	$t \geq a^l B$		*time-split*
$w \geq a^l(\frac{7B}{8})$	$t \geq a^l B$		*key-split*
$w \leq a^l(\frac{3B}{8})$	$t \geq a^l B$	$w_n + w < a^l(\frac{7B}{8})$	*merge*
$w \leq a^l(\frac{3B}{8})$	$t \geq a^l B$	$w_n + w \geq a^l(\frac{7B}{8})$	*merge-key-split*
$w \leq a^l(\frac{B}{4})$	$t < a^l B$	$w_n + w < a^l(\frac{7B}{8})$	*merge*
$w \leq a^l(\frac{B}{4})$	$t < a^l B$	$w_n + w \geq a^l(\frac{7B}{8})$	*merge-key-split*

Table 3: MVBT$^+$ node reorganization conditions

properties in the following lemata; their proofs can be found in the appendix.

Lemma 1. For $a = \frac{B}{4}$, the number of entries in a MVBT$^+$ node is at most $6B = \Theta(B)$.

According to Lemma 1, index nodes can occupy up to 6 pages in worst case. These pages are simply organized as a linked list. The function *ExpandParent* lazily appends new pages to the index node in at most constant time. At first glance, this seems to affect the practical performance of the loading algorithm. However, we did not observe this worst case of 6 pages in our experiments. In only one of our experiments we observed a list of two pages. In all other cases, the index node corresponds to one physical page.

Lemma 2. The number of live entries in the node on level $l > 0$ of MVBT$^+$ is between $\frac{B}{16}$ and B. The number of live nodes in a subtree of height $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ is bounded by $\frac{M}{2B}$.

Lemma 2 ensures that the live part of a subtree fits always in memory, since for the routing only live nodes are considered. This explains also the choice of height $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ for the buffer-tree configuration.

Before a node reorganization is executed, we check if the node has a buffer (see lines 4-8 in Algorithm 3). The buffer of the buffer nodes is emptied if the weight condition of the node is violated. The maximum number of entries in the buffer is limited to $\frac{M}{2}$. We call the *ClearBuffer* function and if the node should be merged we call also this function for a neighbor node. Buffers are always emptied before node reorganization to enforce an overlap of safe intervals (we synchronize nodes according to the time dimension). This operation must not happen frequently, since stopping the process and buffer emptying have worst case $O(M/B)$ I/O costs per buffer level. At least after each $\Theta(M)$ operation, a weight violation on the buffer nodes occurs.

Lemma 3. MVBT$^+$ nodes on level $i \cdot \left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ with $i = 1, \ldots$ are reorganized again after at least $\Theta(M)$ operations (insertions, deletions, updates).

After the buffer is emptied completely, we continue with node reorganization. Finally, we assign a new buffer to a live node and drop buffers of temporal predecessors.

MVBT$^+$ has the same asymptotic I/O bounds on update and space as a MVBT loaded by tuple-by-tuple method. The following lemma holds for MVBT$^+$.

Lemma 4. The MVBT$^+$ height loaded with N records is $O(\log_B N)$ and its space is $O(N)$.

5.3 Outline Runtime

In this section, we outline the proof of the following theorem:

Theorem 1. The cost for loading N records in an initially empty MVBT$^+$ is $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os.

First, we consider the costs of emptying buffer of node v on level $l = i \cdot \left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ and the node reorganization. Thereafter, we discuss the emptying of all buffers after insertions of N records (see function *ClearAllBuffers* in Algorithm 2). Records are pushed down in batches of size $\frac{M}{4}$ towards leafs starting from the root node. We consider two cases of buffer emptying, caused either by buffer overflow or by the node weight violation.

Buffer-Overflow: I/O costs of loading a subtree of height $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ in memory is $O(M/2B)$ I/Os (see Lemma 2). The overall split costs are limited by $O(M/B)$ I/O between two buffer levels, since the weight of the subtree root is not violated (see Lemma 2). Thus, $\frac{M}{4}$ entries routed one level down causing $O(\frac{M}{B})$ I/Os. The I/O cost per entry is $O(\frac{1}{B})$. Entries pass $O(\log_B N/B)$ nodes before they are inserted in the leaf nodes. Yet, we pay I/Os only on each $l = i \left\lfloor \cdot \log_{B/4} \frac{M}{16B} \right\rfloor$ level. Thus, the overall I/O cost per entry is $O(\frac{1}{B} \cdot \frac{\log_{B/4} N_i/B}{log_{\frac{B}{4}} \frac{M}{16B}}) = O(\frac{1}{B} \cdot \log_{\frac{M}{B}} N/B)$.

Weight-Violation: Reorganization of a buffer node stops the buffer emptying process and triggers up to two buffer emptying processes. In case of merge or merge-key-split we also empty the buffer of the sibling node. In the worst case, we write dirty subtree nodes to the disk using at most $O(\frac{M}{2B})$ I/Os (see Lemma 2). The buffers of both nodes contains up to $\frac{M}{2}$ entries each, since we push data in portions of $\frac{M}{4}$ records. For emptying both buffers we pay up to $O(\frac{M}{B})$ I/Os. Total I/O costs are bounded by $O(\frac{M}{B})$. In worst case, we pay $O(M/B)$ I/O for buffer node reorganization after each $\Theta(M)$ operations (see Lemma 3). The total worst case costs of lower buffer nodes splits are $O(N/B)$. The costs for all remain buffer levels are: $N \cdot \sum_{i=1}^{O(\log_{M/B} N/B)} \frac{O(\frac{M}{B})}{(\frac{B}{4})^{i \cdot \log_{B/4} M/16B} \cdot \frac{B}{8}}$ $\leq O(\frac{N}{B} \log_{M/B} \frac{N}{B})$.

Thus, after insertion of N entries in an empty MVBT$^+$ we pay $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ for emptying full buffers and buffer node reorganizations.

Finally, we show that the emptying of all buffers after insertion of N operation entries is bounded by $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os. As shown above, costs for emptying full buffers and for emptying due to weight violation is bounded by $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os. The costs of emptying the remaining non-full buffers is bounded by $O(N/B)$. Since the lowest buffer node level is

$l = \left\lfloor \log_{\frac{B}{4}} M/16B \right\rfloor$, the number of buffer nodes after insertion of N operation entries is limited by $O(\frac{N}{B}/(M/B))$. The worst-case-cost of the buffer emptying process is $O(M/B)$. Therefore, on each $O(\frac{N}{B}/(M/B))$ we pay $O(M/B)$ I/O resulting the $O(N/B)$ bound. Combining this result with buffer emptying caused by buffer overflow and weight violations before *ClearAllBuffers* yields the desired asymptotic bounds.

6. BULK UPDATE

In this section, we briefly describe how to insert a sequence of records efficiently into a non-empty MVBT$^+$ whose buffers are entirely empty. To implement a bulk update, we follow the ideas presented in [6] with a minor modification of algorithm 2: we use the current live root and its buffer instead of a pointer to an empty leaf node. Bulk update appends records to a current root buffer and if applicable pushes entries towards leaf nodes. We call function *ClearAllBuffers* by the end of procedure. Since we load records into existing MVBT$^+$, the records are routed only through the live nodes. By this, we obtain the following I/O cost for a bulk update:

Theorem 2. The cost for a bulk update of N records on an existing MVBT$^+$ with N' live records and empty buffers is $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N+N'}{B} + \frac{N'}{B})$ I/Os.

Analogous to the proof of Theorem 1, we obtain the I/O bound.

7. EXPERIMENTS

In this section, we report the main results of our algorithms for bulk loading and bulk update. We discuss the MVBT$^+$ and MVBT query performance and compare the results to those of a bulk-loaded R-trees.

7.1 Workload Generation

Workloads used in our experiments are designed similarly to other experimental studies with versioned databases [21, 22, 30]. We consider two different types of workloads: one for index loading and the other for queries. We investigate I/O loading performance and space utilization as a function of number of update and delete operations.

Our loading workload consists of six files: $d50$, $u0$, $u25$, $u50$, $u75$, $u100$. Each of them contains $10'000'000$ operations. Larger workloads were not considered, because the experiments for tuple by tuple loading took already almost two days.

For all data sets, the first $1'000'000$ operations are insertions (10% of the data set). The remaining 90% of the file consists of a mix of insertions, deletions and updates. Data sets are named after these specific operations. For example the file $d50$ consists of $1'000'000$ insertions followed by a mix of insertions ($4'500'000$) and deletions ($4'500'000$). File $u75$ consists of $1'000'000$ initial insertions followed by a mix of insertions ($2'250'000$) and updates ($6'750'000$). Inserted record keys are obtained from a permutation of $1, ..., k$, where k denotes the total number of insertions in the particular workload. Deletions and updates randomly select one from all live records. In the following, we use the term update to refer to all of these operations.

For each file from the loading workload we consider three query files qr_1, qr_2, qr_3. Each query file contains two-dimensional range queries (key range and time range) of the same absolute selectivity. qr_1 consist of $10'000$ queries with 100 answers. qr_2 consists of $1'000$ queries with $1'000$ answers and qr_3 contains $1'000$ queries with $10'000$ answers. Queries are uniformly distributed in the two-dimensional space.

	MVBT$^+$		MVBT-LRU	
	B leaf	B index	B leaf	B index
4KB	97	97	97	121
8KB	197	197	197	245
16KB	397	397	397	493

Table 4: Node capacity

7.2 Experimental Setup

All algorithms are implemented in Java using the XXL library [29]. MVBT$^+$ is implemented on top of the existing MVBT by associating two additional weight counters w and t to an index entry.

We conducted our experiments on a system running Windows 7 equipped with an Intel I7 CPU, 8GB of main memory, a magnetic disk (WD Caviar Black 1002FAEX, 1TB) and a SSD (Corsair Force 3 SSD, 120 GB). To avoid an impact of the operating system on our experiments, we used only the raw device interface.

We ran our experiments with pages of different sizes: 4KB, 8KB and 16KB. Table 4 reports the page capacities in the number of tuples. Each page contains header information that occupies 102 bytes (like the level, number of entries, and pointers to temporal predecessors). The size of a versioned record in a leaf occupies 41 bytes (17 bytes for the time interval, 8 bytes for the key and 16 bytes for the payload). The size of an MVBT index entry is 33 bytes (8 bytes for the node pointer, 8 bytes for the key, 17 bytes for a time interval), whereas the size of a MVBT$^+$ index entry is 41 bytes (because of the additional weight counters w and t).

The available main memory varied from 0.8 MB up to 16 MB. These numbers sound very small, however the size of the memory has only a marginal impact (base in the log-factor) on the loading performance. We assigned buffers to each $i \cdot \max\left\{\left\lfloor \log_{B/4}(M/16B) \right\rfloor, 1\right\}$ MVBT$^+$ level. Thus, one buffer is assigned to each internal live node. Buffer size varied from 200 KB up to 4MB because it equals to $\frac{1}{4}$ of the available memory.

Wall clock time and number of I/Os were employed to measure loading performance. The query performance was measured by the number of I/Os and the number of leaf accesses.

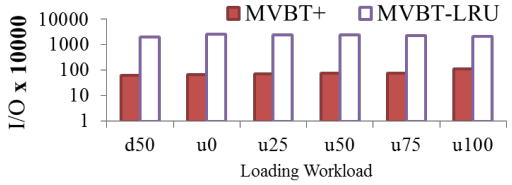

Figure 6: Loading performance (logarithmic scale) of MVBT-LRU and MVBT$^+$ (page size = 8 KB, memory size = 1.6 MB)

7.3 Bulk Loading Results

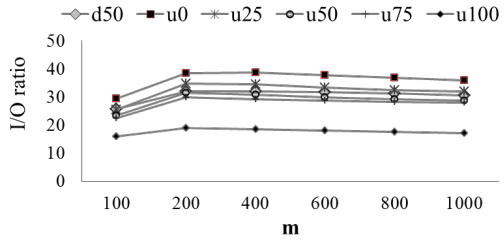

Figure 7: I/O Ratio of MVBT-LRU and MVBT$^+$ as a function of the memory size(page size = 8KB)

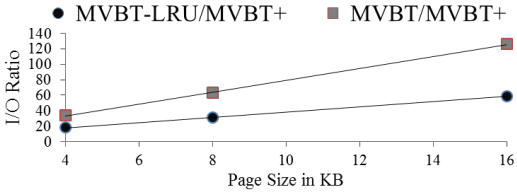

Figure 8: I/O ratio as a function of the page size ($u50$ data set, memory size $m = M/B = 400$)

In this section, we compare the performance of our new bulk loading method to iterative MVBT loading (update by update). MVBT-LRU and MVBT refer to iterative loading with and without a LRU-Buffer, respectively. MVBT-LRU and MVBT$^+$ always received an equal amount of memory.

Figure 6 reports the total number of I/Os required for loading MVBT$^+$ and MVBT for each workload file. We used a page size of 8 KB and a fixed memory size of 1.6 MB. Results (number of I/Os) are given on a logarithmic scale. MVBT$^+$ clearly outperforms MVBT by a factor between 18 and 40.

Figure 7 depicts the ratio of I/Os required to load MVBT-LRU and MVBT$^+$ as a function of memory size. The best results for MVBT$^+$ are achieved for 200 pages. For larger memory sizes, I/O performance of MVBT-LRU improves slightly faster than the one of MVBT$^+$. The (relative) number of I/Os of MVBT$^+$ also increases with a growing number of updates. Due to a smaller number of live versions, the buffer of MVBT-LRU becomes more effective. For updates only (file $u100$), the LRU buffer contains all internal MVBT nodes, while a lot of reorganization steps are triggered for MVBT$^+$. Thus, the MVBT$^+$ performance improvements are only a factor of 15.

In Figure 8, the I/O ratio is depicted as a function of the page size for loading the data set $u50$. The memory capacity is set to 400 pages in total. The lower curve displays the ratio between I/Os required for loading MVBT-LRU and the ones required for loading MVBT$^+$. As expected from our theoretical results, we observe a linear improvement of the I/O performance with an increasing page size. For 16 KB pages, MVBT$^+$ runs faster than MVBT by a factor of 58. The upper curve illustrates the worst-case for MVBT when no LRU buffer is used. The curve shows the relative performance gains of MVBT in comparison to MVBT$^+$.

Figure 9 a and 9 b depict loading time in minutes, using either magnetic disk or SSD for all workload files. For the magnetic disk, loading MVBT$^+$ takes between 30 and 60 minutes, while MVBT-LRU requires between 30 and 40

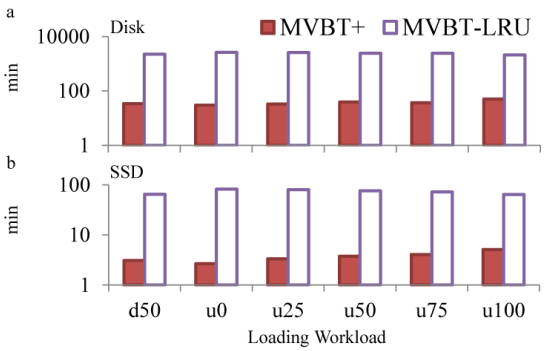

Figure 9: Loading times (logarithmic scale) of MVBT-LRU and MVBT$^+$ (page size = 8KB, memory size = 1.6 MB) a) magnetic disk, b) SSD

File	$d50$	$u0$	$u25$	$u50$	$u75$	$u100$
MVBT$^+$	629	1157	1241	1253	1261	1239
MVBT	605	1153	1215	1223	1235	1196

Table 5: Storage utilization of MVBT$^+$ and MVBT

hours. Greater performance improvements (in comparison to the pure I/O numbers) are due to MVBT$^+$ uses a larger number of sequential I/Os than MVBT-LRU. For SSD, loading times of MVBT$^+$ are between 3 and 5 minutes, while MVBT-LRU requires between 64 and 82 minutes.

Figure 10 depicts the average space utilization of index and leaf nodes. The leaf node utilization of MVBT$^+$ does not differ from the original MVBT. According to Lemma 1 the utilization of an index node is limited by $O(B)$. More precisely, an index node of MVBT$^+$ contains at most $6 \cdot B$ and at least $\frac{B}{16}$ entries. In our experiments, we observed that on average there are $B/2$ entries in one index node. This is less than for the original MVBT. We did not observe more than B per index node except very rarely for data set $u100$.

Table 5 shows space consumption of the resulting trees. The total space required for MVBT$^+$ increases only slightly in comparison to MVBT. This is due to the larger index entries in which the weight counters w and t have to be kept. Moreover, weight balancing results in a lower storage utilization in the index nodes as shown in Figure 10 b.

7.4 Bulk Update Results

In addition to loading, we also conducted a series of experiments to measure the I/O efficiency of bulk updates on a given MVBT$^+$ and MVBT-LRU, respectively. For each data set we first executed 5′000′000 updates (50% of the total updates). Thereafter, we processed the remaining updates with a sequence of bulk updates (with a given batch size). Bulk updates on MVBT-LRU are again implemented by calling the update function one by one. Figure 11 depicts the I/O ratio of MVBT-LRU and MVBT$^+$ as a function of batch size. The memory size was set to 200 pages. MVBT-LRU required slightly less I/Os than MVBT$^+$ for batch sizes with less than 10′000 updates. The reason is that after the updates of the entire batch are performed many buffers contains only one or very few update operation. However, these buffers are forced to be emptied because this has to be performed after the batch. Note that these results are still in agreement with the asymptotically optimal worst-case bounds of Theorem 2.

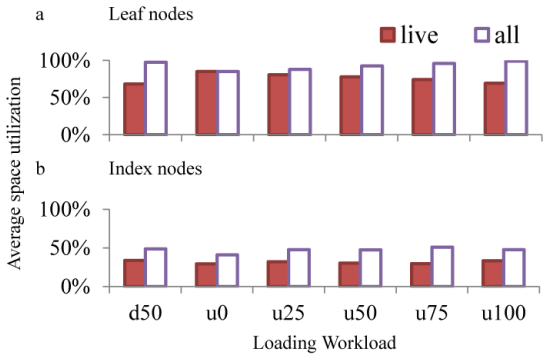

Figure 10: Average storage utilization of leaf and index nodes (MVBT$^+$, 8KB pages B=197=100%) a) leaf storage utilization, b) index node storage utilization

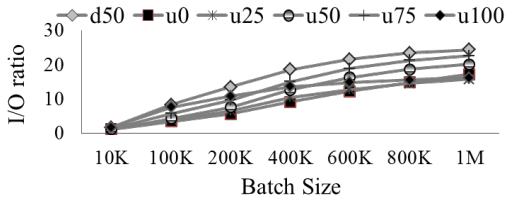

Figure 11: Bulk update, I/O Ratio MVBT-LRU / MVBT$^+$ 8 KB pages

For batch sizes with more than 10′000 updates, the situation is different and MVBT$^+$ is still superior. For a batch size of 400K , for example, the MVBT$^+$ improvements over the MVBT-LRU are again between 9 (for file $u0$) and 18 (for file $d50$).

7.5 Query Workload Results

We conducted a series of experiments running the query workloads on each data file. As expected, we observed almost the same number of leaf accesses for MVBT$^+$ and MVBT. There are marginal differences, as merging of leaves might differ for MVBT and MVBT$^+$. Due to different split strategies for index nodes, the original sibling of a leaf might belong to a different parent node in case of MVBT$^+$. The number of accesses to index nodes is higher for MVBT$^+$ in comparison to MVBT (Table 6). The average number of I/Os per query are reported for the three query files on data file $u50$. However, only for small queries (qr_1) the increase in the number of I/Os for MVBT$^+$ over MVBT is close to 10%.

Workload file $u50$		qr_1	qr_2	qr_3
MVBT$^+$	I/O	4.75	11.98	85.23
	leafs	1.78	8.88	80.62
MVBT-LRU	I/O	4.18	11.33	83.74
	leafs	1.79	8.89	80.8
R-TREE	I/O	116.3	124.92	197.34
	leafs	104.2	112.8	184.8

Table 6: I/Os and leaf accesses for query workload qr_1, qr_2, qr_3 and for MVBT$^+$, MVBT and R-tree

Additionally, we also report the query performance of an R-tree in Table 6. We built the R-tree using STR bulk loading algorithm [19], a popular loading method that is also utilized in commercial systems. Our results clearly show that the R-tree query performance is inferior to the MVBT performance. The reason is simply the high overlap among the nodes of the R-tree. This is particularly noticeable for small queries.

8. CONCLUSIONS

In this paper we presented MVBT$^+$, which is the first partially persistent B$^+$tree that supports bulk loading in an asymptotically optimal number of I/Os and maintains all worst-case performance guarantees of the multiversion B-tree (MVBT). The results of our experimental studies showed that excellent loading times can also be achieved for various storage devices (magnetic disks, SSD, main memory). In comparison to previous loading approaches, i.e., loading by iterative updates, MVBT$^+$ loading is substantial faster by a factor linear to the page capacity. As MVBT$^+$ uses a weight balancing technique, fill degree of non-leaf nodes is slightly lower than for the original MVBT, but this leads to only a slight deterioration of the MVBT$^+$ query performance.

In our future work, we plan to carry over our loading approach to those many partial persistent index-structures that have been derived from the MVBT.

9. ACKNOWLEDGMENTS

The authors would like to thank Anne Sophie Knöller, Marc Seidemann, and the anonymous referees for their insightful and helpful feedback that led to a great improvement of the paper.

10. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. P. Pakala. Temporal query processing in teradata. In *EDBT*, pages 573–578, 2013.

[3] Apache:. Apache hbase. http://hbase.apache.org/.

[4] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[5] L. Arge, A. Danner, and S.-M. Teh. I/o-efficient point location using persistent b-trees. In *ALENEX*, pages 82–92, 2003.

[6] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002.

[7] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *FOCS*, pages 560–, Washington, DC, USA, 1996. IEEE Computer Society.

[8] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.

[9] J. V. d. Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB '97*, pages 406–415, 1997.

[10] J. V. den Bercken and B. Seeger. Query processing techniques for multiversion access methods. In *VLDB*, pages 168–179, 1996.
[11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.
[12] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In *SIGMOD*, pages 847–854, 2009.
[13] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry (preliminary version). In *FOCS*, pages 714–723, 1993.
[14] M. Hadjieleftheriou, G. Kollios, J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *The VLDB Journal*, 15(2):143–164, June 2006.
[15] B. Hoßbach and B. Seeger. Anomaly management using complex event processing: extending data base technology paper. In *EDBT*, pages 149–154, 2013.
[16] IBM:. *A Matter of Time: Temporal Data Management in DB2 for z/OS*. http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/.
[17] K. Kulkarni and J.-E. Michels. Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43, Oct. 2012.
[18] W. Le, F. Li, Y. Tao, and R. Christensen. Optimal splitters for temporal and multi-version databases. In *SIGMOD*, 2013.
[19] S. Leutenegger, M. A. Lopez, and J. Edgington. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506, 1997.
[20] D. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD*, pages 315–324, 1989.
[21] D. B. Lomet, M. Hong, R. V. Nehme, and R. Zhang. Transaction time indexing with version compression. *PVLDB*, 1(1):870–881, 2008.
[22] D. B. Lomet and F. Li. Improving transaction-time dbms performance and functionality. In *ICDE*, pages 581–591, 2009.
[23] Oracle:. *Total Recall*. http://www.oracle.com/technetwork/database/application-development/total-recall-1667156.html.
[24] O. Rodeh. B-trees, shadowing, and clones. *TOS*, 3(4), 2008.
[25] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, June 1999.
[26] A. Seering, P. Cudré-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *ICDE*, pages 1013–1024, 2012.
[27] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.
[28] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Trans. Knowl. Data Eng.*, 9(3):391–409, 1997.
[29] XXL:. Xxl-java-libary. xxl.googlecode.com.
[30] R. Zhang and M. Stradling. The hv-tree: a memory hierarchy aware version index. *PVLDB*, 3(1):397–408, 2010.

APPENDIX

Here we outline the proofs of lemmas.

PROOF. of Lemma 1: For $l = 0$ we have the same settings as MVBT leafs. Let v be node on level $l > 0$. To proof the lemma we compute the maximal number of entries that can be pushed before next reorganization. Since the minimal live weight of the node is $a^l \cdot \frac{3B}{8}$ (see equations 3) we can perform up to $a^l \cdot \frac{5B}{8}$ inserts (*Operation-condition*) and $a^l \cdot \frac{6B}{8}$ deletes (*Live-condition*) on this node. In total we can push up to $a^l \cdot \frac{11B}{8}$ operations until $t(v)$ overflows or the minimal bound of $w(v)$ is achieved. According to the weight constraints, a child node of the node v is reorganized after at least $a^{l-1} \cdot \frac{B}{8}$ operations. After each child split we produce 2 new entries in worst. Thus, after $a^l \cdot \frac{11B}{8}$ operations we create: $\frac{2 \cdot a^l \cdot \frac{11B}{8}}{a^{l-1} \cdot \frac{B}{8}} = 22a$ new entries. The maximal number of entries stored in the node with weight $a^l \cdot \frac{3B}{8}$ before inserting $a^l \cdot \frac{11B}{8}$ is $\frac{a^l \cdot \frac{3B}{8}}{a^{l-1}\frac{B}{4}} = \frac{3a}{2}$. Thus, total number of entries stored in the node is $22a + \frac{3a}{2} \leq 24a = 6B$ for $a = \frac{B}{4}$. □

PROOF. of Lemma 2: (the proof is similar to weight property [7]) We consider node v on level l. The maximal live weight is $a^l B$. The minimal weight of the live child is $a^{l-1}\frac{B}{4}$. Thus, the maximal number of live entries is $a^l B / a^{l-1}\frac{B}{4} = 4a$. Since $a = \frac{B}{4}$ we get the bound. The minimal live weight of the node is $a^l \frac{B}{4}$ (also operation weight $t(v)$, since $w(t) \leq t(w)$). The maximal live weight of the child is $a^{l-1}B$. Thus, the minimal number of entries is $a^l \frac{B}{4} / a^{l-1}B = a/4 = B/16$. We consider a sub-tree T of height $i \cdot \left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$ on level i. Without loss of generality, we assume that $\log_{B/4} \frac{M}{16B}$ is an integer value. The maximal live weight of T is $a^{i \cdot \log_{B/4} \frac{M}{16B}} \cdot B$. Since the minimal weight of the tree on level $(i-1) \cdot \log_{B/4} \frac{M}{16B}$ is $a^{(i-1)\cdot \log_{B/4} \frac{M}{16B}} \cdot B/4$. Using the same argument as above, we obtain maximal number of buffer nodes $\frac{M}{4B}$, referenced by T. Thus, the overall number of live nodes is bounded by $2 \cdot \frac{M}{4B}$. □

PROOF. of Lemma 3: We consider the lowest buffer node v. The lowest level is $\left\lfloor \log_{B/4} \frac{M}{16B} \right\rfloor$. According to weight conditions the minimal number of entries needed for next reorganization is $a^l \frac{B}{8}$. Thus, the node v is reorganized after $a^{\log_{B/4} \frac{M}{16B}} \frac{B}{8}$ operations. Since $a = \frac{B}{4}$ we get $\frac{M}{16B} \cdot \frac{B}{8} = \frac{M}{128} = \Theta(M)$. □

PROOF. of Lemma 4:

A sub tree with a root node on level l references $O(a^l \cdot B)$ live elements in the live leaf nodes. After insertion of N operation entries at most N entries are alive. The minimal live weight of the live root node is at least $N \geq 2 \cdot a^{l-1} \cdot \frac{B}{4}$. Since $a = \frac{B}{4}$ level of the root node is $O(\log_B N)$.

A leaf node (level = 0) is reorganized at least after performing $\frac{B}{8}$ operations. In worst we create two new leaf nodes. Node v node on level l is reorganized after at least $a^l \frac{B}{8}$ operations. In worst we create also two new nodes. Thus, after N operations we create up to: $2 \cdot N \sum_{l=0}^{\log_a N} \frac{1}{\frac{B}{8} \cdot a^l} \leq 16 \cdot N/B \sum_{l=0}^{\infty} \frac{1}{a^l}$ nodes. Since $a \geq 16$ and the node capacity is $O(B)$ according to lemma 1 we obtain the result of $O(N)$ space. □