Transactions on the Multiversion B$^+$-Tree

Tuukka Haapasalo
Helsinki University of Technology
Espoo, Finland
thaapasa@cs.hut.fi

Ibrahim Jaluta
Helsinki University of Technology
Espoo, Finland
ijaluta@cs.hut.fi

Bernhard Seeger
Philipps-University Marburg
Marburg, Germany
seeger@mathematik.uni-marburg.de

Seppo Sippu
University of Helsinki
Helsinki, Finland
sippu@cs.helsinki.fi

Eljas Soisalon-Soininen
Helsinki University of Technology
Espoo, Finland
ess@cs.hut.fi

ABSTRACT

The multiversion B$^+$-tree (MVBT) by Becker et al. assumes a single-data-item update model in which each new version created for a data item is given a timestamp that is unique across the entire MVBT. In this paper, we extend the MVBT model with multi-action transactions such that all (final) data-item versions created by a transaction are given the same timestamp. We show that the MVBT algorithms can be modified to work in a setting in which multiple read-only transactions and a single updating transaction operate concurrently in snapshot isolation on the MVBT, without compromising the asymptotically optimal time complexity of key inserts, key deletes, and key-range scans on any version. The structural consistency and balance of the MVBT is guaranteed by short-duration latching of pages, redo-only logging of structure modifications (version splits, key splits and page merges), and redo-undo logging of key insertions and deletions. The redo pass of our ARIES-based restart-recovery algorithm always produces a structurally consistent and balanced MVBT on which any undo action by a backward-rolling updating transaction can be performed logically if a physical undo is not possible. The standard steal-and-no-force buffering policy is assumed.

1. INTRODUCTION

In many applications, historical data needs to be stored alongside current data. Examples of such applications include medical-record databases, banking software, and moving-object databases, among others [17]. To maintain good query-time and space requirements, a special database structure is required that can store historical versions of data items in a compact form. The version information attached to data items adds a new dimension—the temporal dimension—to the one-dimensional set of non-versioned data items indexed by their primary keys. Thus, it is not surprising that one confronts problems that are similar to those found in general-purpose index structures for multidimensional data, such as R-trees [6], when developing an efficient index structure for versioned data. These structures cannot guarantee a logarithmic time complexity simultaneously for insertions, deletions and searches of data items.

An important goal in indexing versioned data is to guarantee that key-range searches for data items belonging to a given database version are as efficient as for non-versioned data. More specifically, searching for the data items of a given database version v that belong to a given key range should not take significantly much more time than $O(\log m_v + r)$, where m_v is the total number of data items in version v and r is the number of data items in version v whose key belongs to the given key range. The multiversion B$^+$-tree, or MVBT, by Becker et al. [1] possesses this desired property, besides guaranteeing an $O(\log m_v)$ time bound for creating a new version of a data item.

The data-update model adopted by most proposals for indexing versioned data, including the MVBT model [1], assumes that each update action creates a new version of the item (identified by a unique key), so that the version numbers are unique across all versions of all data items. Following Salzberg et al. [17] and Lomet et al. [12], we assume a multi-action-transaction approach in which all data-item versions created by a transaction get the same version number, unless the transaction updates a data item more than once, in which case only the final data-item version gets the version number and is stored in the database. In this approach, each data-item version is still uniquely identified by the pair (k, v), where k is the key of the data item and v is the version number, but the version numbers are only unique across versions of the data item with key k, for each fixed k.

We assume that the creation of a new version of a data item with key k is always based on the most recent committed version of the data item with key k. Thus, as with Becker et al. [1], the versions of a data item with key k always form a linearly ordered version history without any branching (or diverging) versions. This is consistent with multiversion concurrency-control protocols such as snapshot isolation and is also the approach adopted by the transaction-time temporal database engine Immortal DB by Lomet et al. [11].

The data-item versions created by a transaction remain in the database only if the transaction commits; otherwise the transaction is rolled back and the created versions are deleted. Thus, an index structure for versioned data must also support physical deletion of uncommitted versions, even though all committed versions of all data items are to be stored permanently for an indefinite time. This issue has not been addressed in many proposed index structures for temporal data [10, 17, 9]. In other proposed structures [13, 11, 12], rolling back transactions may lead to pages with very few entries (or none at all).

We will use the terms *timestamp* and *version number* interchangeably in this paper, although the terms themselves may be used to imply different semantics. Our structure assumes a *transaction-time model* [19], in which the data items become valid once they have been inserted into the database and invalid once they have been deleted from the database. The mapping between real time and version numbers in the database is beyond the scope of this article, although any increasing integer numbers can be used as version numbers (as long as each data item created by the same transaction gets the same version number; and the version range of each data item deleted by the same transaction is terminated with the same version number).

In this paper, we extend the the single-data-item update model assumed by the MVBT algorithms [1] to the multi-action-transaction update model outlined above. In our model, the MVBT is periodically updated by a multi-action transaction that can roll back some or all of its updates at any time. The updates are made recoverable by an ARIES-based recovery algorithm [14], and a multiversion concurrency-control protocol (such as snapshot isolation) [2, 4] can be used for guaranteeing transactional isolation. Our algorithms work in a concurrent setting in which multiple read-only transactions and a single updating transaction operate on the MVBT simultaneously. Important application areas where this setting is perfectly sufficient include data stream and RFID stream management systems. The setting with multiple concurrent updating transactions is discussed in the conclusions.

Our algorithms have the following properties: (1) all data-item versions created by an updating transaction are given the same version number, consistently with previous transactional versioning schemes; (2) many read-only transactions, each reading consistently from any committed version of the database, can run concurrently with a single updating transaction; (3) all range-query and update actions retain the asymptotic time complexities of the MVBT algorithms; (4) the consistency and balance of the MVBT structure during normal processing are maintained by short-duration latching of pages, assuming the standard steal-and-no-force buffering policy; (5) the consistency and balance of the MVBT structure during transaction aborts and system crashes are maintained by redo-undo logging of the update actions of transactions and by redo-only logging of structure modifications (splits and consolidations); (6) at most five MVBT pages need to be kept write-latched during a structure modification; (7) the redo pass of our ARIES-based restart recovery algorithm will always produce a consistent and balanced MVBT on which the undo actions of the backward-rolling updating transaction (if any) can be performed logically if a physical undo is impossible.

We begin in Sec. 2 by discussing the main features of the MVBT and the problems we encountered when trying to use the original MVBT algorithms by Becker et al. [1] for transactions that consist of multiple updates. Next, in Sec. 3, we define formally our extended MVBT structure, called the transactional multiversion B$^+$-tree (TMVBT), giving detailed invariants that must be retained by any update action and structure-modification operation on the TMVBT. Then, in Sec. 4, we list the actions that a transaction operating on the TMVBT may contain; here we assume that multiple read-only transactions are operating on the TMVBT concurrently with one updating transaction that may contain both read and update (insert and delete) actions. In Sec. 5 we give algorithms to implement these actions, and in Sec. 6 we present the structure-modification operations (key splits, version splits, and merges) that are needed in the algorithms in order to keep the TMVBT in a consistent and balanced state. Finally, in Sec. 7, we compare our algorithms to previous work on indexing temporal data, and in Sec. 8, we present our conclusions and outline our future work on how to use the TMVBT in a fully concurrent setting.

2. MULTIVERSION B$^+$-TREE

The multiversion B$^+$-tree (MVBT) algorithms proposed by Becker et al. [1] modify the standard B$^+$-tree algorithms to store multiple versions. While the pages in a standard B$^+$-tree contain entries of the form (k, w), where k is a key value and w is the data part of a data item (in the case of a leaf page) or the page identifier of a child page (in the case of a non-leaf page), the pages in an MVBT contain entries of the form $((k, [v_1, v_2)), w)$, where $[v_1, v_2)$ is the *version range* of the entry. Initially, the version range of a newly created entry is unbounded, that is, of the form $[v_1, \infty)$. The data part w of a leaf-page entry consists of the values of other attributes of the data item (in the case of a sparse index) or a record identifier of the data item stored elsewhere (in the case of a dense index).

Each page p has a *key range*, denoted $\mathrm{kr}(p)$, and a *version range*, denoted $\mathrm{vr}(p)$, such that p contains only entries whose version range overlaps with $\mathrm{vr}(p)$ and whose key belongs to $\mathrm{kr}(p)$. The pages at any given height in the MVBT partition the two-dimensional *key-version space* into disjoint rectangular areas $\mathrm{vr}(p) \times \mathrm{kr}(p)$. All newly created pages have initially an unbounded version range, that is, $\mathrm{vr}(p)$ is of the form $[v, \infty)$. A page with an unbounded version range is called a *live page*. An entry with a version range of the form $[v, \infty)$ residing in a live page is called a *live entry*.

Structure modifications on the MVBT are mostly based on the version-split operation (possibly directly followed by a key split). In a *version split*, a formerly live page p is *killed*, that is, p is split at the current time v_{cur}, and a new live copy p' of p is created. The old page's version range $[v, \infty)$ is cropped to $[v, v_{cur})$, and the new page's version range is set to $[v_{cur}, \infty)$. All the live entries in p are copied to p'. Page p is now considered a *dead page* and is only used for historical queries, and the new page p' is used for current-version queries. Dead pages are never modified again, although they might be deleted later to save space (see the discussion of purging old versions in the MVBT article by Becker et al. [1]).

When a non-leaf page is version-split, the copying of entries creates a new parent, in addition to the old one(s), for the child pages pointed to by those entries. Thus, the

MVBT is not a tree but a directed acyclic graph, which may have several roots. When a root page is split, a new root page is created. However, the old root page is still used as a starting point when searching for historical entries. To accomplish this, a separate structure called *root** is used to store all the different roots of the MVBT. The *root** structure can be implemented, for example, as a B$^+$-tree that contains pointers to the root pages, indexed by their creation versions. It is assumed that the number of different roots is small, and that the *root** structure may even fully reside in main memory.

The main problems with the MVBT algorithms arise if we try to apply them without increasing the version number of the database between the updates. Consider, for example, simply inserting entries with consecutive keys $1, 2, 3, \ldots$ to the MVBT. The scenario is shown in Figure 1, with page capacity of three entries per page. The first three entries can be inserted without problems. The fourth insertion leads to an overflow and therefore a version split is triggered, which in turn causes the version range of the old page and its entries to degenerate into an empty interval $[1, 1)$. This page does not hold any relevant information as it is no longer a part of any version of the database. The pages created earlier on by the same transaction are the real cause of the problem. Our aim is to extend the MVBT algorithms by applying B$^+$-tree-style structure modifications on these pages. In this problem scenario, the page could be key-split directly, without first version-splitting it.

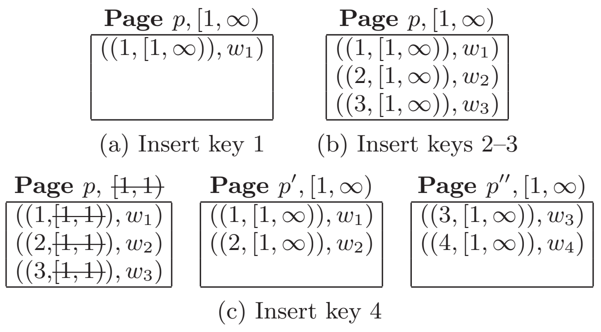


Figure 1: Insertion of key 4 causes an invalid split

As stated before, an index structure for versioned data should support physical deletion of uncommitted entries, to allow for total or partial rollbacks of transactions. Also, when a transaction deletes a key that it has itself inserted, physical deletion should be applied. If we try to delete entries like these by using the MVBT algorithms (with physical deletion added), the number of live entries in a page may fall below the acceptable limit. In a B$^+$-tree, the situation could be remedied with a page merge. In the MVBT, a merge is only possible after a version split. This is problematic, if the page has been created during the execution of the aborting transaction. In this case, the resulting killed page will have an empty version range. Merging without version-splitting is not trivial, because pages may have multiple parents.

3. TRANSACTIONAL MVBT

In this section, we present our modified multiversion B$^+$-tree, called the *transactional multiversion B$^+$-tree*, or the TMVBT, for short. As it turns out, not many changes are needed to the structure of the MVBT of Becker et al. [1] to overcome the problems reported in the previous section.

The page format in the TMVBT is identical to that of the MVBT, with addition of recovery information required for our ARIES-based recovery, such as a Page-LSN field that stores the log sequence number (LSN) of the log record of the latest update on the page. We also assume that each page p explicitly stores the version range, $\mathrm{vr}(p)$, and the key range, $\mathrm{kr}(p)$, of the page and also the height of the page, denoted $\mathrm{height}(p)$, which is 1 for all leaf pages.

v_{cur}	The current, committed version.
v_{act}	The active version.
B	The capacity of pages; for convenience, assumed to be same for all pages.
m_v	The number of live entries of version v in the database.
m_v^p	The number of live entries of version v in page p.
$\mathrm{height}(p)$	The height of page p.
$\mathrm{children}(p)$	The set of children of index page p.
$\mathrm{parent}(p)$	The set of parents of page p; where $p' \in \mathrm{parent}(p) \Leftrightarrow p \in \mathrm{children}(p')$.
$\mathrm{kr}(p)$	The key range of page p.
$\mathrm{vr}(p)$	The version range of page p.
$\mathrm{kvr}(p)$	The key-version range of page p; a rectangular region in key-version space, $\mathrm{kvr}(p) = (\mathrm{kr}(p), \mathrm{vr}(p))$.
min	The minimum number of entries that must be alive at each page that is alive.
s	The minimum amount of user actions (inserts or deletes) required after a structure modification before another one is required; $0 \le s \le \min$
$\min_s$	The minimum number of entries in a new page; $\min_s = \min + s$.
$\max_s$	The maximum number of entries in a new page, $\max_s = B - s$ and $\max_s \ge 2 \times \min_s$.
root*	A separate search structure for locating the root pages for different versions.

Table 1: Terms and variables

In the following discussion we will use the terms and variables listed in Table 1. The variables min, $\min_s$ and $\max_s$ correspond to the MVBT condition variables *weak version*, *strong version underflow*, and *strong version overflow*. These affect the height and storage consumption of the structure and the frequency of structure-modification operations required. The *weak version condition* variable min controls the height of the structure as it controls the minimum fan-out at each level of the TMVBT for every version. The *strong version condition* variables $\min_s$ and $\max_s$ control how many actions are at least required after a structure-modification operation before another structure modification is needed on the page (this is the variable s in the formulas). Note that $\max_s \ge 2 \times \min_s$ must hold so that a page with $\max_s$ entries can always be split into two pages with at least $\min_s$ entries each. An example setting for the variables with page capacity $B = 100$ is $\min = 20$ and $s = 20$. This makes $\min_s = 40$ and $\max_s = 80$. After a page split the entry counts of the new pages must be between 40 and 80 entries. Therefore there will always be space to insert at least 20 new entries or delete 20 old entries before the page is full or the number of live entries falls below 20.

Although the presentations of the variables differ from the previous definition, we can show that the variables are the same as in the MVBT article [1]. The previous definition stated that $\min = d = b/k$, $\min_s = (1 + \epsilon) \times d$ and $\max_s = (k - \epsilon) \times d$, with $b = kd = B$, and k and ϵ being variables that can be selected. If we assign $s = \epsilon d$, we get $\min_s = d + \epsilon d = \min + \epsilon d = \min + s$, and $\max_s = kd - \epsilon d = B - s$, which are the definitions used in this paper.

As discussed in the introduction, we assume that at any time at most one updating transaction on the TMVBT is active. We denote by v_{act} the version number that an active updating transaction assigns to the data-item versions it creates. We denote by v_{cur} the version number assigned to the data-item versions created by the most recent committed transaction that has inserted its updates into the TMVBT. We call an entry with key range $[v_{act}, \infty)$ an *active entry* and all other entries *inactive entries.* The active entries have all been created by the currently active updating transaction. Similarly, *active pages* are pages created by structure-modification operations during an active updating transaction. Active pages may only contain active entries. Thus, the version ranges of all entries in active pages are the same, and therefore the entries cannot have overlapping key ranges. Because of these properties, algorithms such as the standard B$^+$-tree key splits and merges can be applied to active pages. In these operations, the active entries are physically moved from one page to another. Thus, active pages cannot have more than one parent, whereas inactive pages—live or dead—can.

We now define formally the conditions that must be satisfied by any structurally consistent and balanced TMVBT. A TMVBT is a directed graph that contains pages at different levels, or heights. Pages at the bottom level (at $\text{height}(p) = 1$) are *leaf pages*, which contain (references to) the data items. All pages above the bottom level (at $\text{height}(p) > 1$) are *index pages*, which contain links to child pages. All children are at a height that is one lower than the height of the parent, such that $\forall p \forall p'(p' \in \text{children}(p) \Rightarrow \text{height}(p) = \text{height}(p') + 1)$. As all the links go downward in the graph, the graph is acyclic, and therefore a directed acyclic graph (DAG).

Each page in a TMVBT is associated with a *key-version range* $\text{kvr}(p) = ([k_{\min}, k_{\max}), [v_{\min}, v_{\max}))$. Each level of the TMVBT graph partitions the entire key-version space into disjoint key-version ranges, so that $\forall p \forall p'((p \neq p' \wedge \text{height}(p) = \text{height}(p')) \Rightarrow \text{kvr}(p) \cap \text{kvr}(p') = \emptyset)$. This holds for both index pages and leaf pages.

As stated above, index pages contain child links to pages at a lower level. More precisely, an index page p at $\text{height}(p)$ contains links to all pages at $\text{height}(p) - 1$ whose key-version range intersects with the key-version range of the parent page p. Formally, $\forall p \forall p'((\text{height}(p) = \text{height}(p') + 1 \wedge \text{kvr}(p) \cap \text{kvr}(p') \neq \emptyset) \Rightarrow p' \in \text{children}(p))$. Also, the key range of the child page must be a subset of the parent page's key range, so that $\forall p \forall p'(p' \in \text{children}(p) \Rightarrow \text{kr}(p') \subset \text{kr}(p))$. Thus, a page may have multiple parent pages if the version range of the child extends outside the version range of the parent. For example, there can be a link to a child page that is alive from version v_1 to v_3 in parent pages p and p' if there is a version v_2 that separates these pages, so that $\text{vr}(p) = [v_0, v_2) \wedge \text{vr}(p') = [v_2, v_4) \wedge v_0 \leq v_1 < v_2 \leq v_3 < v_4$.

The key-version ranges in index-page entries are called *routers* to child pages. An entry in an index page that points to a child page p is a pair (r, p), where the router r is the intersection of the key-version range of the parent index page and that of the child page. Initially, the router to page p in the parent page is $\text{kvr}(p) = (\text{kr}(p), \text{vr}(p))$. When a version split is done on the parent page, the router key-version range in the parent is cropped to the active time, even though the child page's version range remains unbounded.

For convenience, we will use the same notation for entries in leaf pages. A leaf-page entry $((k, [v_1, v_2]), w)$, for key k and data-item identifier w, that is alive in versions $[v_1, v_2)$ is, therefore, denoted by $(([k, k+1), [v_1, v_2)), w)$. In this way we can use the same algorithms for manipulating the contents of both leaf and index pages.

Any consistent and balanced TMVBT is required to satisfy the following three invariants. First, by invariant

$$v_{act} = v_{cur} \vee v_{act} = v_{cur} + 1, \tag{1}$$

there can be only one updating transaction active at a time. When $v_{cur} = v_{act}$, there is no updating transaction active, and no insert or delete actions can be executed. By invariant

$$\forall p \forall v \left(\begin{array}{l} m_v^p \geq \min \vee \\ m_v^p = 0 \vee \\ m_v^p = m_v \vee \\ (m_v^p \geq 2 \wedge p \in root^*) \end{array} \right) \tag{2}$$

each page must contain either at least min or zero entries of each version. That is, there can be no pages with only $n \in [1, \min)$ entries that are alive at version v. The exceptions are the situations in which all the live entries of the TMVBT are on a single root page, or the page is a root page with $n \in [2, \min)$ links to leaf pages. This invariant guarantees that all structure-traversal operations within any database version work within time bounds that are logarithmic in the number of live entries at that version. More formally, the worst-case time bound for tree traversal is $O(\log_{\min} m_v)$ for all versions v. Invariant

$$\forall p(\text{vr}(p) = [v_{act}, \infty) \Rightarrow |\,\text{parent}(p)| \leq 1) \tag{3}$$

states that an active page cannot have more than one parent page. This invariant is the foundation of the TMVBT algorithms, and it follows from the fact that all levels of the TMVBT graph partition the key-version space into disjoint regions. The reasoning is that there can be only one parent whose version range intersects with that of a page created at v_{act} (because there can be no parent pages created after the active version); and the child page's key range must be a subset of the parent page's key range (so there can be no other parent with an intersecting key range).

An example of a structurally consistent and balanced TMVBT is given in Figure 2. The white pages represent live pages, and the darker ones dead pages. The page size limits in this example are chosen for illustration purposes only; they do not fill the requirements set out in this paper. The example has been generated by our visualization software with the following action sequence.

- Transaction 1: insert data items with keys 1–9. Note that these entries all have the same timestamp (i.e., the same start version).
- Transaction 2: delete data items with keys 7–9; insert data items with keys 10–15.

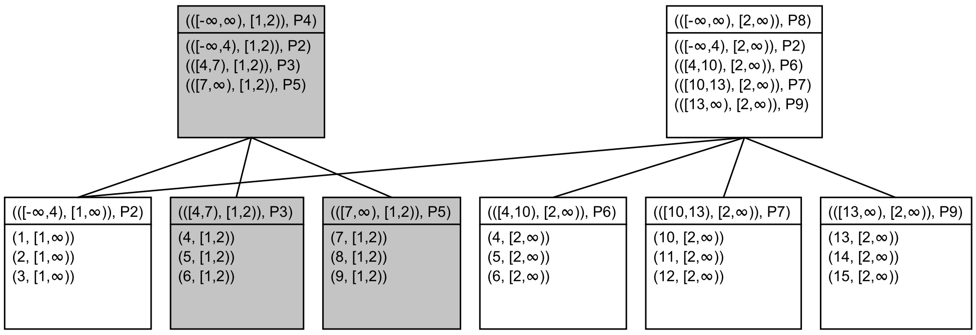


Figure 2: An example of a TMVBT. The page header format and the format of index-page entries is ((key range, version range), page identifier). The format of leaf-page entries is ((key, version range), data-item identifier), but the data-item identifiers have been left out for clarity. White pages are live pages, the darker pages are dead pages. This TMVBT was created by transactions 1 and 2 with transaction 1 inserting keys 1–9, and transaction 2 first deleting keys 7–9 and then inserting keys 10–15.

Figure 2 shows that page P2 containing entries with keys 1–3 is shared by both roots of the TMVBT. Note that page P2 is alive but not active, as it contains entries of previous versions. Thus, it is possible for this page to have more than one parent. The pages P6–P9 are active and contain only entries of the most recent version. Note also that the index page P8 is active even though it contains a router to the inactive page P2, because the router itself is active.

The first insertions by transaction 1 triggered two key-splits (pages P2 → P3 → P5), and a tree height increase (root page P4). The deletions by transaction 2 triggered a consolidation operation (pages P5 ∧ P3 → P6) and the insertions caused leaf-page key-splits (pages P6 → P7 → P9) and a root page version-split (root pages P4 → P8).

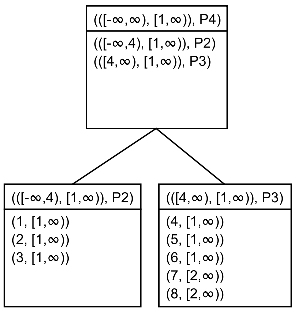


Figure 3: Initial TMVBT. This TMVBT has been created by transaction 1 inserting keys 1–6 and transaction 2 inserting keys 7–8.

Figures 3–5 show another example of the TMVBT page operations. In Figure 3, the tree contains historical entries inserted by transaction 1, and some entries inserted by transaction 2.

Figure 4 shows the result of a version split after transaction 2 tried to insert key 9 to the full page P3. The page P3 was version-split into pages P5 and P6. The old data is left stored in the dead page P3, and active copies of the current data have been created to pages P5 and P6. Note that active contents have been physically moved away from page P3.

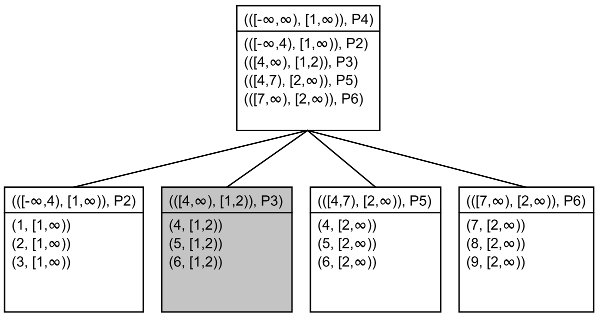


Figure 4: After inserting entry with key 9. Transaction 2 has caused a version-split by inserting key 9.

Figure 5 shows the status of the database after transaction 2 has deleted entries 4–9. Deleting the active entries has caused the pages to underflow and to consolidate by merging them with sibling pages. Note that the page P5 was deallocated when it was merged with a sibling. When the last two active pages were merged, the (current version) tree height was decreased so that all the live entries are in the root page P6. The auxiliary structure *root** contains root pointers to pages P4 (for version 1) and P6 (for version 2).

4. ACTIONS OF TRANSACTIONS

We allow two kinds of transactions to operate concurrently on the TMVBT: one or more read-only transactions and at most one updating transaction at a time. A *read-only transaction* may contain the following actions:

- **begin-read-only**: begins a new read-only transaction; this action records the value $v_{begin} \leftarrow v_{cur}$ for the transaction.
- **query(key k, version v)**: retrieves the data item (k, w) for which the TMVBT contains a leaf-page entry $((k, [v_1, v_2)), w)$ with $v_1 \leq v < \min\{v_2, v_{begin}+1\}$; thus, the transaction is allowed to see only those versions that were committed before the transaction began.
- **range-query(range $[k_1, k_2)$, version v)**: retrieves the set of data items (k, w) for which the TMVBT

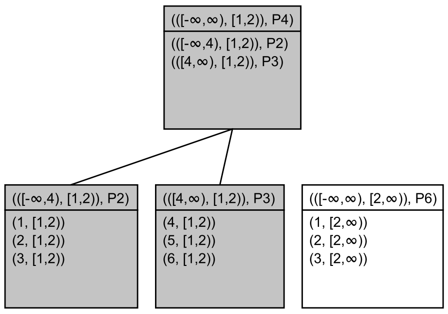


Figure 5: After deleting most of the entries. Transaction 2 deleted keys 4–9, thus shrinking the current version search tree to a single page.

contains leaf-page entries $((k,[v_1,v_2)),w)$ with $v_1 \leq v < \min\{v_2, v_{begin}+1\}$ and $k_1 \leq k < k_2$.

- **commit-read-only**: commits the transaction by removing the transaction from the system.

An *updating transaction* may contain the following actions:

- **begin-update**: begins a new updating transaction; this action increments the active-version counter v_{act}.
- **query(key** k**)**: retrieves the data item (k,w) for which the TMVBT contains a live leaf-page entry $((k,[v_1,\infty)),w)$; thus, the action reads the committed current version of the data item (if the transaction has not updated the data item) or the uncommitted active version (otherwise).
- **range-query(range** $[k_1,k_2)$**)**: retrieves the set of data items (k,w) for which the TMVBT contains live leaf-page entries $((k,[v_1,\infty)),w)$ with $k_1 \leq k < k_2$.
- **insert(key** k**, data** w**)**: a forward-rolling action that is legal when the TMVBT contains no live leaf-page entry of the form $((k,[v,\infty)),w')$; the action inserts the leaf-page entry $((k,[v_{act},\infty)),w)$.
- **delete(key** k**)**: a forward-rolling action that is legal when the TMVBT contains a live leaf-page entry of the form $((k,[v,\infty)),w)$; the action either (1) replaces the leaf-page entry $((k,[v,\infty)),w)$ by $((k,[v,v_{act})),w)$ when $v < v_{act}$; or (2) removes the leaf-page entry $((k,[v,\infty)),w)$ when $v = v_{act}$.
- **commit-update**: commits an updating transaction; this action increments the current-version counter v_{cur}.
- **abort**: labels the updating transaction as aborted and starts the backward-rolling phase.
- **undo-insert(log record** r**)**: a backward-rolling action that undoes the insert action logged with the log record r.
- **undo-delete(log record** r**)**: a backward-rolling action that undoes the delete action logged with the log record r.
- **finish-rollback**: finishes the backward-rolling phase of an aborted updating transaction; this action decrements the active-version counter v_{act}.

An aborted updating transaction is an action sequence consisting of the following actions: (1) a **begin-update** action, (2) a forward-rolling phase consisting of zero or more **query**, **range-query**, **insert**, and **delete** actions, (3) an **abort** action, (4) a backward-rolling phase that consists, in reverse order, of the **undo** actions for the **insert** and **delete** actions of the forward-rolling phase, and (5) a **finish-rollback** action.

To allow for partial rollbacks, a transaction may also contain actions **set-savepoint** p and **rollback-to-savepoint** p in its forward-rolling phase. An action **rollback-to-savepoint** p is followed by the **undo-insert** and **undo-delete** actions for the **insert** and **delete** actions done after setting savepoint p, executed in the reverse order.

All the update actions of an updating transaction are logged onto a physiological write-ahead log as in ARIES [14]. Redo-undo log records are written for an **insert** action, a **delete** action, the incrementation of v_{act} in a **begin-update** action, and the incrementation of v_{cur} in a **commit-update** action, while redo-only log records are written for an **undo-insert** action, an **undo-delete** action, and the decrement of v_{act} in a **finish-rollback** action. The **commit-update** and **finish-rollback** actions also write separate commit log records and force the contents of the log buffer onto disk. A read-only transaction does not create any log records; it only stores transient control information in the active-transactions table when it begins, and removes that information when it commits.

5. IMPLEMENTATION OF THE ACTIONS

We assume that the physical consistency of the database during normal processing is maintained by short-duration latching [14] of pages, so that the server process or thread that executes a transaction keeps a page p read-latched for the time a read action is performed on p, and write-latched for the time an update action is performed on p. We also assume that the buffer manager applies the standard steal-and-no-force buffering policy. These assumptions are in accordance with the ARIES recovery algorithm [14].

In a fully dynamic index structure in which any inserted data can be physically deleted at any time, *latch-coupling* (also called *crabbing* by Gray and Reuter [5]) is the standard way to guarantee the validity of traversed search paths in all circumstances. In a general situation, the validity of the traversed path can be ascertained by releasing the latch on the parent page only after the latch on a child page has been acquired. Latch-coupling is deadlock-free if all latches are acquired in a certain order, such as first top-down, then left-to-right. However, in the case of a TMVBT the fact that inactive data is never deleted or moved, together with our assumption that a read-only transaction only reads inactive data, implies that **query** and **range-query** actions of read-only transactions need not do latch-coupling, so that a parent page may be unlatched before acquiring a latch on a child page.

Accordingly, an action **query**(k,v) in a read-only transaction can be implemented as follows. First, the root page for version v is located from *root** and read-latched. Then the TMVBT is traversed using read latches until the leaf page p is found that covers k and v, that is, $k \in \mathrm{kr}(p)$ and $v \in \mathrm{vr}(p)$. At each index page p on the traversed path, the next page on the path is the child page q of p with $k \in \mathrm{kr}(q)$ and $v \in \mathrm{vr}(q)$. Once the page identifier q of the child page has been determined, the read latch on the parent page p is released and the child page q is read-latched.

An action **range-query**$([k_1, k_2), v)$ is implemented similarly, except that for each index page p in the search path we need to traverse all subtrees rooted at child pages q with $[k_1, k_2) \cap \mathrm{kr}(q) \neq \emptyset$ and $v \in \mathrm{vr}(q)$. If there are more than one such child page q, then all but the first are pushed into a stack, and the traversal proceeds to the subtree rooted at the first child. When a subtree has been searched, a page (if any) is popped from the stack and read-latched, and the search is continued at the subtree rooted at that page.

An action **query**(k) in an updating transaction is implemented as the action **query**(k, v_{act}), and an action **range-query**$([k_1, k_2))$ as the action **range-query**$([k_1, k_2), v_{act})$. These actions may read active data, but latch-coupling is still not needed, because the data can move from a page to another only in a structure modification performed by the process (or process thread) that is generating the updating transaction itself.

For more efficiency, the TMVBT may contain direct links to the root page of version v_{act} so that the queries in updating transactions do not need to use the *root** structure to find it. Similarly, if it is expected that applications read the committed version most often, the root page of version v_{cur} can also be tracked separately.

We assume that all TMVBT traversals maintain a *saved path*, that is, an array *path* local to the server process or thread in question and indexed by the height of pages, so that *path*$[i]$.page gives the page identifier, *path*$[i]$.LSN gives the Page-LSN, *path*$[i]$.kr gives the key range, and *path*$[i]$.vr gives the version range, of the page at height i on the latest traversed path.

The saved-path concept can be used to accelerate **query** and **range-query** actions by starting the traversal at the lowest-level page in the saved path that, according to the saved information, covers the entire search space. This page is known to be the correct page to start the tree traversal, because (1) for read-only transactions, the inactive data is never moved away from the pages; and (2) for updating transactions, there can be no other updating transaction that would invalidate the data in the saved path of the current updating transaction.

The global variables v_{cur} and v_{act} are maintained in the permanent database and their reading and writing is protected by locking. A **begin-read-only** action acquires a short-duration read lock on v_{cur} for reading its value, and a **commit-update** action acquires a commit-duration write lock on it for incrementing its value. A **begin-update** action acquires a commit-duration write lock on v_{act}, thus guaranteeing that at most one updating transaction is active at a time. The decrement of v_{act} in a **finish-rollback** action is performed under the protection of that lock.

For **insert**(k, w) and **delete**(k), the TMVBT is traversed with read latches as for **query**(k, v_{act}), except that the target leaf page p is write-latched. As in **query**(k) and **range-query**$([k_1, k_2))$, no latch-coupling is needed. If the target leaf page p can accommodate the update, so that it is possible to perform the update on the page without violating the required balance conditions, then the update is done on p, a redo-undo log record is generated, its LSN is stamped in the Page-LSN field of p, and the write latch on p is released.

In the case of **insert**(k, w), the target leaf page p can accommodate the update if it has room for inserting the entry $((k, [v_{act}, \infty)), w)$. In the case of **delete**(k), page p can accommodate the update if replacing the entry $((k, [v, \infty)), w), v \neq v_{act}$ by $((k, [v, v_{act})), w)$ or removing the entry $((k, [v_{act}, \infty)), w)$ does not decrease the number of live entries in the page below the required minimum min.

When the target leaf page p cannot accommodate the update, structure modifications are needed. These operations are explained in Sec. 6. For inserts, the operation **split** is called before the insert action can proceed. For deletes, the page p needs to be consolidated with operation **consolidate** before the entry can be deleted from the page. For the structure-modification operations, the page p will be left write-latched on the saved path. After the operations, the saved path will contain the proper write-latched leaf page p' whose key range contains k. As with the earlier situation, the update is now done on page p', a redo-undo log record is generated, the LSN is stamped on p' and page p' is unlatched.

As will be explained in the next section, the structure modifications (page splits or merges) are done in a top-down, level-by-level manner, logging the structure modification done at each level using a single redo-only log record. Each of these structure modifications involves at maximum five pages on two adjacent levels. The structure modifications result in a target leaf page that can accommodate the **insert** or **delete** action in question.

An undo action, **undo-insert**(r) or **undo-delete**(r), is performed as a physical undo if possible and as a logical undo otherwise [14]. For a physical undo, the page mentioned in the log record r is write-latched and the Page-LSN field is examined. If the Page-LSN field still contains the LSN of r, or if the page contents show that the page is the correct target for the undo action and the page can accommodate the undo action, the undo action is performed on the page, a redo-only log record is generated, its LSN is stamped in the Page-LSN field of the page, and the page is unlatched.

The target leaf page p can accommodate the undo of an action **insert**(k, w) if the inserted entry $((k, [v_{act}, \infty)), w)$ can be removed from p without decreasing the number of live entries in p below the required minimum min. The target leaf page p can always accommodate the undo of an action **delete**(k) if this action replaced the entry $((k, [v, \infty)), w)$ by the entry $((k, [v, v_{act})), w)$ in p. If the action removed the leaf-page entry $((k, [v_{act}, \infty)), w)$ from p, then p can accommodate the undo action if it has room for the entry.

If the page mentioned in the log record r cannot be seen to be the correct target for the undo action or if the page cannot accommodate the undo action, a *logical undo* is performed, starting with a search for the key mentioned in r and including any structure modifications that may be necessary to make the target page accommodate the undo action.

The following theorem states that the asymptotic bounds of the MVBT are maintained for queries:

Theorem 1 Let us denote by m_v the number of live data items in database version v. Assuming that all structure-modification operations maintain the structural consistency and balance of the TMVBT and that locating the root page for any version v takes only constant time, (1) the action **query**(k, v) is performed in time $O(\log_{\min} m_v)$; (2) the action **query**(k) is performed in time $O(\log_{\min} m_v)$, where v is the active database version v_{act}; (3) the action **range-query**$([k_1, k_2), v)$ is performed in time $O(\log_{\min} m_v + r)$, where r is the number of data items in version v whose key belongs to the key range $[k_1, k_2)$; (4) the action **range-query**$([k_1, k_2))$ is performed in time $O(\log_{\min} m_v + r)$, where $v = v_{act}$ and r is the number of data items in version v_{act} whose key belongs to $[k_1, k_2)$. $\square$

The following theorem follows directly from the definitions of the **query** and **range-query** actions and from the fact that only one updating transaction can be active at a time:

Theorem 2 Our algorithms produce a snapshot-isolated [2] serializable schedule for the transactions. $\square$

6. STRUCTURE MODIFICATIONS

With the user-transaction actions defined, we will now concentrate on the structure-modification operations **split** and **consolidate**. The general convention in these operations is that each structure-modification operations transforms a structurally consistent and balanced TMVBT into another structurally consistent and balanced TMVBT. Each operation is logged with a single redo-only log record, so that structure modifications are never undone when transaction aborts or system fails [7, 8]. The structure-modification operations need to be performed top-down, starting from the highest page on the search path that requires splitting or consolidation.

The actual implementation of the operations traverses the search path bottom-up in order to determine which kind of a structure modification is needed at each level, yet without performing any modification. When a parent page which does not need any modification is encountered, the search path is traversed top-down, and the structure-modification operations are performed level-by-level, logging each operation with a single redo-only log record. The search path state is guaranteed to remain valid throughout the operations because only one updating transaction can be active at a time. Thus, the information about the traversed search path stored in the saved path can be trusted. For clarity, the algorithms presented in this section only describe the structure-modification operations applied at a single level.

Before explaining the two structure-modification operations, split and consolidate, we will first define the *page-killing* operation. Page killing is not a separate structure-modification operation, but it is used by both split and consolidate. This operation takes an inactive page p, marks it as killed, and creates a new active page p' that holds the live contents of p. An overview of this operation is shown in Algorithm 1. The page p and its parent q (located from the saved path) both need to be write-latched.

Algorithm 1 Kill page p, parent page q

```
$p' \leftarrow$ **create** and **write-latch** a new page
copy all live entries of $p$ to $p'$
end the version range of the router to $p$ in $q$
insert router to $p'$ in $q$
```

The operation begins by allocating the new page p', write-latching it and formatting it as a TMVBT page. All the live entries of page p are now either copied or moved to page p' in such a way that all active entries are physically moved to p' to maintain invariant (3), and all inactive live entries are copied to p'. The version ranges of the copied entries are split at version v_{act}, so that a version range $[v, \infty)$ is changed to $[v, v_{act})$ in p and to $[v_{act}, \infty)$ in p'. Thus, the logical state of the database at all versions prior to v_{act} is maintained in page p, but the page p is no longer part of the active version. The router to page p in the parent page q must be updated by settings its end version to v_{act}. Also, the router for the new page is inserted into the parent. After this, the old page p is replaced by the active page p' in the saved path. All pages modified by this operation are kept write-latched, as the acquired page latches can be released only after the relevant log record has been written.

The page-killing operation maintains invariant (2), as the newly created active page p' contains only entries of version v_{act} (at least min entries), and the dead page p no longer contains any entries of the version v_{act}. The entry counts of other versions in page p are not affected by the operation. Invariant (3) is also maintained, as the active entries are physically moved from page p to the new page p'.

The **split** operation is a structure-modification operation that splits a page that has become full. The operation is similar to the MVBT version-split operation, with the exception that active pages are directly key-split. This operation is triggered by an **insert** action when a data page has become full, and also to split full index pages along the search path. At the beginning, the page p to be split is retrieved from the saved path along with its parent page q, and both pages are write-latched for modification. As explained in the beginning of this section, we expect that parent pages in the saved path have already been split so that the parent page of p can accommodate the routers to the new pages created by the split operation. An overview of the actual operation is simple: if page p is active, it will be key-split, and if it is inactive, it will be version-split. Key-splitting and version-splitting will be defined in more detail below. The split operation is described generally in Algorithm 2. Note that the bottom-up checking phase needs to do the same checks that are described in Algorithm 2 to determine which kind of a split needs to be done. When performing the actual structure modification, the same checks must be performed again (as described here), or results saved during the checking phase can be used.

Key-splitting an active page is similar to a page-split operation in a standard B$^+$-tree. It is accomplished by creating a new page p', and redistributing the entries of page p between p and p'. Note that all the entries of page p must be alive and active because p is active. The new page p' is thus allocated, write-latched and formatted. After that, the upper half of the keys in page p are moved to page p', and the router to page p in the parent page q is adjusted. Also, a router to the new page p' is now inserted to the parent.

An exception to the key-split algorithm is the situation where page p has no parent page (saved path contains only the page p). This happens when page p is a root page. In this situation a new parent page r is allocated, write-latched and formatted; and routers to pages p and p' are inserted to it. The new root is attached to the TMVBT by inserting the page identifier of r to *root** with version v_{act}. This may replace an existing root identifier, if the TMVBT already contains a root created by the same updating transaction. The rest of the key-split operation is otherwise similar to the normal situation.

The *version-split* operation begins by killing page p with the page-killing operation defined earlier. The new active page is denoted by p'. At this point, page p' may contain too few or too many entries to satisfy the strong version conditions. If the entry count of p' is less than $\min_s$, the

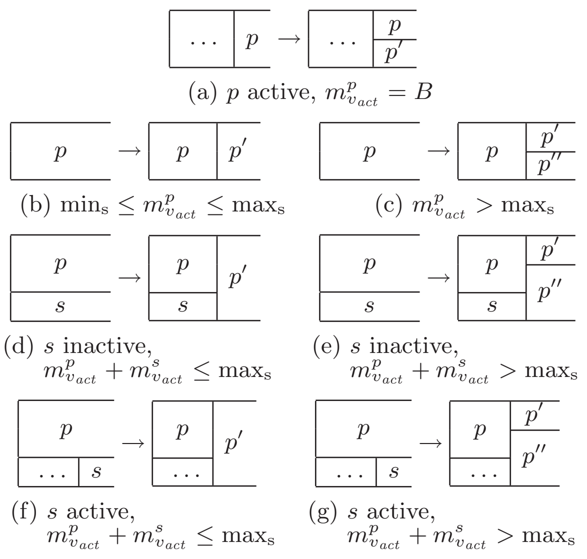
Algorithm 2 Split page p, parent page q

```
if $p$ is active then // Figure 6(a), key split
  $p' \leftarrow$ create and write-latch a new page
  redistribute entries of $p$ between $p$ and $p'$
  insert router to $p'$ in $q$
  adjust router to $p$ in $q$
else // Figures 6(b)–6(g), version split
  $p' \leftarrow$ kill page $p$
  insert router to $p'$ in $q$
  if $m^p_{v_{act}} > \max_s$ then // Figure 6(c)
    $p'' \leftarrow$ create and write-latch a new page
    redistribute entries of $p'$ between $p'$ and $p''$
    insert router to $p''$ in $q$
    adjust router to $p'$ in $q$
  else if $m^p_{v_{act}} < \min_s$ then // Figures 6(d)–6(g)
    $s \leftarrow$ find a live sibling page of $p'$ from $q$
    write-latch $s$
    if $s$ is active then // Figures 6(f),6(g)
      $p'' \leftarrow s$
    else // Figures 6(d),6(e)
      $p'' \leftarrow$ kill page $s$
    end if
    if $m^p_{v_{act}} + m^s_{v_{act}} > \max_s$ then // Figures 6(e),6(g)
      redistribute entries of $p'$ and $p''$
      adjust router to $p'$ in $q$
      adjust router to $p''$ in $q$
    else // Figures 6(d),6(f)
      move all entries of $p''$ to $p'$
      adjust router to $p'$ in $q$
      remove router to $p''$ from $q$
      deallocate $p''$
    end if
  else // Figure 6(b)
    // No further action required
  end if
end if
log the operation using a redo-only log record
release page latches
```

page will be merged with a sibling page by consolidating it in the same way as the **consolidate** operation does. If the entry count of p' is more than $\max_s$, the page will be key-split in the same way as active pages are split.

The entire split operation consisting of either a key split or a version split (possibly followed by a key split or consolidation) is logged with a single redo-only log record containing the page identifiers of all pages involved, that is, pages p, p', p'', s and q. The log record must also contain information of all the entries moved or copied between pages. As usual, all the pages are kept latched until the log record has been generated and its LSN stamped in the Page-LSN fields of the pages. The split operation is finished by replacing the old page p in the saved path with the proper active page, and by unlatching all other pages related to the split operation.

All possible split scenarios for inactive pages are shown in Figure 6. In the figure, the horizontal axis represents version ranges, and the vertical axis shows key ranges. In the presented scenarios, page p is split. Page s is the sibling page that is located from the parent of p found in the saved path. Pages p' and p'' are new pages created by the operation. As can be seen from the figures, all the scenarios preserve the initial key-version extents of p and s. That is, the new pages cover exactly the same region in key-version space as the old pages did. Therefore the version-split operation cannot cause overlap or create gaps in the key-version space at any level of the TMVBT structure.



(a) p active, $m^p_{v_{act}} = B$

(b) $\min_s \le m^p_{v_{act}} \le \max_s$

(c) $m^p_{v_{act}} > \max_s$

(d) s inactive, $m^p_{v_{act}} + m^s_{v_{act}} \le \max_s$

(e) s inactive, $m^p_{v_{act}} + m^s_{v_{act}} > \max_s$

(f) s active, $m^p_{v_{act}} + m^s_{v_{act}} \le \max_s$

(g) s active, $m^p_{v_{act}} + m^s_{v_{act}} > \max_s$

Figure 6: Page-split scenarios for page p. The horizontal axis represents version ranges, and the vertical axis key ranges. Case (a) represents an ordinary key split, (b) a version split, (c) a version split followed by a key split, (d) a version split followed by a merge with an inactive sibling, (e) a version split followed by a redistribution of live entries with an inactive sibling, (f) a version split followed by a merge with an active sibling, and (g) a version split followed by a redistribution of live entries with an active sibling.

We will now consider invariant (2) when splitting inactive pages. After p has been killed, the resulting page p' contains $e = m^p_{v_{act}}$ entries, where $e \in [\min, B]$. This is because split is only called when the page has become full, and by invariant (2) it must contain at least min live entries. If $e < \min_s$, the page will be consolidated with another page to avoid thrashing. If $e > \max_s$, the page will be key-split into two pages (of which both will have more than $\min_s$ entries). When splitting active pages, the full active page can always be split into two active pages with $B/2 > \min_s$ entries. Invariant (3) is also maintained, because page killing maintains the invariant, as does moving active entries between two active pages.

The **consolidate** operation is a structure-modification operation that merges a page with a sibling page before the page live-entry count falls below acceptable limits. This operation is triggered by the **delete** action, if too many entries have been deleted from a page (weak version condition). This operation is also used to consolidate index pages in the search path. An overview of the operation is shown in Algorithm 3. The operation begins by retrieving the page to be consolidated from the saved path and by write-latching it. The parent page needs to be modified, so it is also retrieved from the path, and write-latched for modification. As with the **split** operation, it is assumed that the parent can accommodate the insertions or deletions possibly triggered by this operations (insertion of up to two new routers; or deletion of a single router). Again, the bottom-up checking phase needs to do the same checks that are shown in Algorithm 3, and these results can be saved.

Algorithm 3 Consolidate page p, parent page q

```
$s \leftarrow$ find live adjacent sibling of $p$
write-latch s
if $p$ is active then // Figures 7(a)–7(d)
    $p' \leftarrow p$
else // Figures 7(e)–7(h)
    $p' \leftarrow$ kill page $p$
end if
if $s$ is active then // Figures 7(a),7(b),7(e),7(f)
    $p'' \leftarrow s$
else // Figures 7(c),7(d),7(g),7(h)
    $p'' \leftarrow$ kill page $s$
end if
if $m^p_{v_{act}} + m^s_{v_{act}} \leq \max_s$ then // 7(a),7(c),7(e),7(g)
    move all entries of $p''$ to $p'$
    remove router to $p''$ from $q$
    adjust router to $p'$ in $q$
    deallocate $p''$
else // Figures 7(b),7(d),7(f),7(h)
    redistribute entries between $p'$ and $p''$
    adjust router to $p'$ in $q$
    adjust router to $p''$ in $q$
end if
log the operation using a redo-only log record
release page latches
```

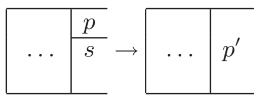
The consolidation begins by finding a live adjacent sibling page s from the parent page q, and write-latching it. Such a page is guaranteed to be found, because by invariant (2), the parent page must contain at least two live entries, and these must be adjacent. If either page p or the sibling page s is inactive, it will be killed with the page-killing operation defined earlier. We denote the active pages with p' (from page p) and p'' (from page s). The consolidation operation can now merge the pages p' and p''.

The merging is similar to the standard B$^+$-tree merge operation. If the combined entry count of pages p' and p'' (equivalently, live-entry count of pages p and s) is larger than $\max_s$, the entries will be redistributed between the two pages. If the entry count is below or equal to $\max_s$, the entries will be moved to page p', and page p'' will be deallocated by removing the router to it from the parent and deallocating the page from the corresponding space-map page. The router to page p' (and to page p'' in the former case) in the parent q need to be updated to match the new key ranges. This cannot result in invalid links to pages p' and p'' from other parents, because the pages are active, and therefore have only one parent by invariant (3).

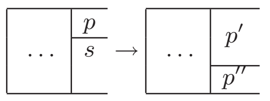
An exception to the normal operation of **consolidate** is when the current index root live-entry count falls to one. In this case, the root of the active version can be replaced in the TMVBT with the single remaining child by inserting the child page identifier to *root** with version v_{act}. This may replace an already existing root, if *root** contains a root created by the same transaction earlier on. If the old root page is an active page, it may be deallocated at this point.

The entire consolidation operation, including the possible root page update operation, is logged with a single redo-only log record containing the page identifiers of all related pages—this means pages p, s, p', p'' and q. The log record must also contain information of all the moved entries. The saved path must now be returned to a proper state, so that the proper active page p' or p'' is placed in the saved path to replace the consolidated page. The operation finishes by releasing the write latches on the pages.

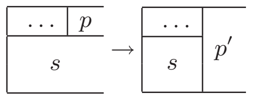
The possible consolidation scenarios for page p are shown in Figure 7. In the figure, page p is consolidated with a live sibling page s. For consistency with Figure 6, the resulting active pages are denoted p' and p'', even though $p' = p$ when the page p is active.



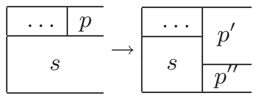
(a) p active, s active, $m^p_{v_{act}} + m^s_{v_{act}} \leq \max_s$



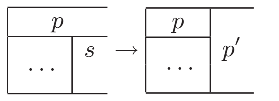
(b) p active, s active, $m^p_{v_{act}} + m^s_{v_{act}} > \max_s$



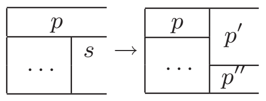
(c) p active, s inactive, $m^p_{v_{act}} + m^s_{v_{act}} \leq \max_s$



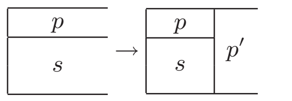
(d) p active, s inactive, $m^p_{v_{act}} + m^s_{v_{act}} > \max_s$



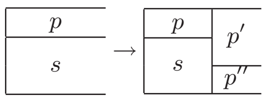
(e) p inactive, s active, $m^p_{v_{act}} + m^s_{v_{act}} \leq \max_s$



(f) p inactive, s active, $m^p_{v_{act}} + m^s_{v_{act}} > \max_s$



(g) p inactive, s inactive, $m^p_{v_{act}} + m^s_{v_{act}} \leq \max_s$



(h) p inactive, s inactive, $m^p_{v_{act}} + m^s_{v_{act}} > \max_s$

Figure 7: Page consolidations scenarios for page p.

Let us now prove that invariant (2) is maintained throughout this operation, and that the new pages contain between $\min_s$ and $\max_s$ entries to avoid page thrashing. The possible page-killing operations maintain the invariant regarding the historical pages, as explained earlier. Page p must have exactly min live entries, while s can have between min and B live entries. This range of entries $\{e \mid 2 \times \min \leq e \leq \min + B\}$ is in the range that merging can handle. In the minimum case, $2 \times \min \geq \min_s$ entries are moved to page p', and page p'' is deleted. In the maximum case, $\min + B$ entries are distributed between p' and p'', resulting in more than $\min_s$ and less than $\max_s$ entries per page. The former holds trivially (as we are distributing more than $\max_s$ entries between two pages), and the latter holds because $\min + B = (\min_s - s) + (\max_s + s) = \min_s + \max_s$.

Invariant (3) is also maintained, as both killing an inactive page and merging two active pages physically move the active entries. Thus, the parent count of active pages is not affected.

Lemma 3 Any structure modification needed in the implementation of **insert**, **delete**, **undo-insert** and **undo-delete** actions keeps at most five TMVBT pages latched simultaneously and transforms a structurally consistent and balanced TMVBT into a structurally consistent and balanced one. For any of the actions, at most $h+1$ structure modifications are needed, where h is the height of the tree in the active database version v_{act}. □

The following theorem states that the update actions also maintain the asymptotic bounds of the MVBT:

Theorem 4 Assuming that locating the root page for any database version v takes only constant time, each of the actions **insert**, **delete**, **undo-insert**, and **undo-delete** is performed in time $O(\log_{\min} m_v)$, where m_v is the number of data items in the active database version $v = v_{act}$. □

In restart recovery from a system crash, an ARIES-based recovery algorithm [14, 7, 8] is used.

Theorem 5 In the event of a system crash, the redo pass of restart recovery produces a structurally consistent and balanced TMVBT on which the undo actions by a backward-rolling updating transaction (if any) can be performed logically if a physical undo is impossible. □

7. RELATED WORK

The idea of storing historical versions in the database is not new. Easton's write-once balanced tree (WOBT, [3]) stores multiple versions of data on indelible storage; however, the structure only keeps track of the current version, and the mechanism has been designed based on the fact that written contents cannot be deleted from the write-once media. Moreover, when a page is split, the current entries are moved to a new page and the old entries remain in the old page. In WOBT algorithms, it is not described how to access the historical versions of the data.

Lomet's and Salzberg's time-split B^+-tree (TSBT, [13]) stores multiple versions of data. The TSBT allows access to historical versions, but it does not do page consolidation (merging). The current pages can only be split into pages with smaller key ranges, they are never merged with adjacent pages as in the MVBT [1]. This could be a problem for applications that actively delete old data from the database and use increasing identifiers as keys. In this situation pages near the high end of the key range need to be continuously split to contain new entries with high keys, while pages at the low end of the key range will contain practically no entries of the current version. As the amount of pages belonging to the current version increases, so does the structure height, which leads to longer search paths. Similarly, range queries will need to traverse more pages as leaf pages contain few entries of the current version (or none at all).

Another method for accessing temporal data is hashing. Kollios and Tsotras describe a hashing method for accessing temporal data with non-branching history [10]. Hashing is very efficient for an exact-match query, with an access time of $O(1)$ in ideal situations. However, hashing cannot support efficient key-range queries. In the MVBT, the time complexities of all the actions are logarithmic.

For a comprehensive presentation and comparison of different multiversion access methods, the reader is referred to Salzberg and Tsotras [18].

Lomet et al. have chosen the time-split B^+-tree (TSBT) as the basis when implementing multiversion support to Immortal DB, which is built into the commercial Microsoft SQL Server [11]. When performing a version-based split (time-split) in the TSBT, a new page is created for the historical contents of the old page. Thus, pages that contain old data can be moved to a slower tertiary storage after the pages become historical. This is indeed a problem in the MVBT algorithms; the old page must be left in place, because there can be an unknown number of references to it from historical parent pages (unless all the parents of a given child page are somehow tracked). A simple solution would be to maintain a separate mapping from page identifiers to actual locations in the storage media. In this way, the old pages could be moved to a tertiary storage when they are killed.

Jouini and Jomier [9] recently published a paper comparing three different approaches for indexing multiversion data with branched evolution. However, none of the structures presented can guarantee the logarithmic query-time and space bounds of the MVBT.

Salzberg et al. have also published an interesting framework for accessing multiversion data [17]. Their algorithms, however, work with branching histories, while the MVBT only works with a linear history. That is, in their framework, new versions can be created from any existing version, thus creating version branches. The version numbers are increasing integer numbers, and the branching information is stored in a separate version tree. This adds complexity to the algorithms, and also makes it non-trivial to determine whether two version numbers are on the same branch. Also, because of the simpler version scheme, the MVBT does not need to store null records (which are used to mark item deletions) and therefore does not have ghost pages (which are pages that only contain null records). Our extended MVBT algorithms allow key merges without version-splitting for active pages. For these reasons we believe that the extended MVBT algorithms are more efficient for database systems with linear histories.

As explained in Sec. 6, each structure modification on the TMVBT only involves pages on two adjacent levels of the structure; each such modification turns a structurally consistent and balanced TMVBT into a structurally consistent and balanced one and is logged using a single redo-only log record. This technique has been designed for B^+-tree-like structures by Jaluta et al. [7, 8].

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have extended the original multiversion B^+-tree (MVBT) algorithms by Becker et al. [1] to support multiple updates within a single updating transaction. Our extended algorithms maintain the same asymptotic bounds as the original MVBT algorithms (Theorems 1 and 4). None of the user actions or structure-modification operations cause pages to consolidate or split earlier than when using the previous algorithms. Also, all new pages have the same minimum and maximum entry counts as the previous algorithms for each operation that creates new pages—thus, the same number of operations is required before new pages need to be split or consolidated. Furthermore, the new operations modify the same number of pages than the original MVBT algorithms. At most four (plus one) pages need to be modified per level in a single operation; this is the case for example when two inactive sibling pages are version-split and their live entries are distributed between two new pages. A maximum number of five pages must therefore be held write-latched at the same time, as the parent page must also be modified.

We have also designed a simple concurrency control and recovery scheme that supports a single updating transaction and multiple concurrent read-only transactions. Because the inactive data read by read-only transactions does not move between pages, the applied latching scheme allows for high concurrency for the read-only transactions. The recovery algorithms are in line with the *de facto* standard ARIES algorithms and produce a structurally consistent and balanced TMVBT structure after a system crash (Theorem 5).

We believe that it is not possible to directly extend our TMVBT algorithms to the fully concurrent setting in which multiple updating transactions operate concurrently on the TMVBT. The first problem is that transactions need to have a version number assigned when they first perform updates on the TMVBT. At that point, however, the commit order of the transactions may not be known. If the modifications are not performed in commit order, the database will become inconsistent. On the other hand, forcing transactions to commit in their starting order or in the order in which they perform their first update action could delay indefinitely the commit of transactions. It is also unclear what would be the definition of active pages and entries with multiple updating transactions. Also, backward-rolling aborted transactions that delete entries they have inserted in their forward-rolling phase can cause the structure to have too few live entries at some pages. Thus, it is only possible to insert entries of one version at a time to the TMVBT.

The problems encountered suggest a database organization in which the TMVBT is used as a storage for committed data, while a (smaller) B$^+$-tree is used to store the updates of active updating transactions. A system-generated maintenance transaction can then be run periodically to move the updates of the committed transactions from the B$^+$-tree to the TMVBT one version at a time in the commit order of the transactions. This organization would also allow the version numbers used in the TMVBT to follow the commit order of transactions even when the commit order is different from the starting order (cf. Immortal DB [12]). Similar techniques involving a smaller, temporary index have been published before, called *differential indices* by Pollari-Malmi et al. [16] and *side files* by Mohan and Narang [15]. Designing this database organization is the topic of our future work with the multiversion B$^+$-tree.

9. ACKNOWLEDGMENTS

This work has been funded by the Academy of Finland.

10. COPYRIGHT

11. REFERENCES

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal—The International Journal on Very Large Data Bases*, 5(4):264–275, 1996.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM New York, NY, USA, 1995.

[3] M. Easton. Key-sequence data sets on indelible storage. *IBM Journal of Research and Development*, 30(3):230–241, 1986.

[4] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.

[5] J. Gray and A. Reuter. *Transaction processing: concepts and techniques.* Morgan Kaufmann, 1993.

[6] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57. ACM Press New York, NY, USA, 1984.

[7] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency control and recovery for balanced B-link trees. *The VLDB Journal—The International Journal on Very Large Data Bases*, 14(2):257–277, 2005.

[8] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. B-tree concurrency control and recovery in page-server database systems. *ACM Transactions on Database Systems*, 31(1):82–132, Mar 2006.

[9] K. Jouini and G. Jomier. Indexing multiversion databases. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, pages 915–918. ACM New York, NY, USA, 2007.

[10] G. Kollios and V. Tsotras. Hashing methods for temporal data. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):902–919, 2002.

[11] D. Lomet, R. Barga, M. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for SQL server. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 939–941, 2005.

[12] D. Lomet, R. Barga, M. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Transaction time support inside a database engine. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 35–46, 2006.

[13] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 315–324, 1989.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[15] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 361–370. ACM Press New York, NY, USA, 1992.

[16] K. Pollari-Malmi, J. Ruuth, and E. Soisalon-Soininen. Concurrency control for B-trees with differential indices. In *Proceedings of the International Database Engineering and Applications Symposium*, pages 287–296, 2000.

[17] B. Salzberg, L. Jiang, D. Lomet, M. Barrena, J. Shan, and E. Kanoulas. A framework for access methods for

versioned data. In *Proceedings of the 9th International Conference on Extending Database Technology*, pages 730–747, 2004.

[18] B. Salzberg and V. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[19] G. Özsoyoğlu and R. Snodgrass. Temporal and real-time databases: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.