Concurrent Updating Transactions on Versioned Data

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

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

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

Modern database applications increasingly often require access to historical versions of the database. Storing such multiversion data in a single-version B^+-tree database index is inefficient, especially for key-range queries. In this article, we present an index structure called the concurrent multiversion B^+-tree (CMVBT) for efficiently storing and querying multiversion data.

The CMVBT structure uses an asymptotically optimal transactional multiversion B^+-tree (TMVBT) index as the main data storage, and a separate B^+-tree index called the versioned B^+-tree (VBT) to hold the updates of active transactions. The updates of committed transactions are moved, one transaction at a time, from the VBT into the TMVBT. This organization of two separate index structures allows us to maintain the asymptotic optimality guarantees of the TMVBT even in the presence of concurrent updating transactions.

We provide concurrent algorithms for updating and reading the CMVBT structure. Our CMVBT algorithms can be used with the standard snapshot isolation concurrency-control and ARIES-based recovery algorithms to allow multiple read-only and updating transactions to operate concurrently on the structure. Transaction rollback is also supported for all updating transactions, either entirely or up to a preset savepoint.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*access methods,recovery and restart*; H.2.4 [**Database Management**]: Systems—*concurrency,transaction processing*

General Terms

Algorithms, Design

IDEAS 2009, September 16-18, Cetraro, Calabria [Italy]
Editor: Bipin C. DESAI

Keywords

Database indexing methods, temporal databases, concurrency control, recovery

1. INTRODUCTION

Modern database applications often require access to historical states of the database, in addition to the current state. Historical data is most efficiently accessed for this purpose with multiversion database indices. For single-version databases, the B-tree [2, 3, 6] is the *de facto* choice for indexing data, with the B^+-tree being the most often used variant. The B^+-tree is an asymptotically optimal index structure for single version data, meaning that the index search operations will be optimal after any sequence of user transactions.

For historical databases, there are a few promising index structures, but none of them is entirely without limitations. The multiversion B^+-tree (MVBT) by Becker et al. [4] is an asymptotically optimal multiversion access structure. The optimality of the MVBT structure guarantees that the performance of the index will not deteriorate under any sequence of user transactions, including key deletions. This is a requirement for large key-range queries to work efficiently. However, the MVBT provides only action-wise versioning, where each new action performed creates a new version of the index. Furthermore, undo actions are not supported, and no optimality-preserving updating algorithms exist for the the MVBT that would allow concurrent updates by multiple updating transactions. Thus, the MVBT index cannot be efficiently used as a transactional multiversion database. The time-split B^+-tree (TSBT) by Lomet and Salzberg [17, 18] is the basis of Microsoft SQL Server's multiversion engine Immortal DB [15, 16]. However, the TSBT has no asymptotically logarithmic access time guarantees for tree traversals in different versions. The performance of the TSBT—especially of the key-range queries—may thus deteriorate with key deletions.

Our goal is to extend the multiversion B^+-tree of Becker et al. [4] to an asymptotically optimal multiversion index structure with concurrent updating algorithms that can be used with standard concurrency-control and recovery algorithms to allow multiple updating and read-only transactions to operate concurrently on the structure. As an initial step, we

have extended the MVBT to support multi-action transactions and designed concurrency-control algorithms that allow a single updating transaction to operate concurrently with multiple read-only transactions [8, 9]. Our extension is called the transactional multiversion B$^+$-tree, or TMVBT.

In this article, we describe how to further extend the TMVBT structure to support multiple concurrent updating transactions. As described in our TMVBT article [8], the TMVBT structure cannot directly support multiple updating transactions without compromising its optimality guarantees. We have thus designed a two-level index structure composed of a permanent TMVBT index used as the main storage of committed versioned data and a temporary B$^+$-tree index called the versioned B$^+$-tree (VBT) used to store the versions created by active or recently committed transactions. A system maintenance transaction is run periodically to move the updates of committed transactions from the temporary VBT index into the TMVBT. The VBT is expected to remain small, and thus it will reside entirely in main memory during normal transaction processing. However, should a long-running transaction require considerable amounts of space, this organization allows us to flush parts of the VBT to disk.

We begin in Sec. 2 by describing the general organization of our new database index structure. In Sec. 3, we explain the temporary versioned B$^+$-tree structure, which is a B$^+$-tree index used for storing multiple versions of keys. After that, in Sec. 4, we briefly review our transactional extension to the multiversion B$^+$-tree. Then, in Sec. 5, we explain the general conventions used in the concurrency-control and recovery algorithms. Sec. 6 describes the system maintenance transaction that is used to move the updates from the VBT into the TMVBT. In Sec. 7, we explain in more detail how user transactions read and update the index structures. Then, in Sec. 8, we describe how our index structure differs from other related index structures. Finally, in Sec. 9, we present our conclusions on the CMVBT index and outline our plans on how to test the design in our future work.

2. CONCURRENT MVBT

The *concurrent multiversion B$^+$-tree* (CMVBT) is composed of two parts: a *versioned B$^+$-tree* (VBT) index that is used as a temporary storage of updates by active transactions, and a *transactional multiversion B$^+$-tree* (TMVBT) index that is used as a final, stable storage of multiversion data. The VBT is described in detail in Sec. 3, and the TMVBT is reviewed in Sec. 4. For user transactions, the TMVBT is a read-only structure. The TMVBT is only updated by a system maintenance transaction, which is periodically run to move the updates of committed transactions from the temporary VBT index into the TMVBT. This setup is illustrated in Figure 1. A similar approach, called *differential indices*, is employed by Pollari-Malmi et al. [23] to group updates falling into the same leaf page and to simplify the recovery algorithms. Mohan and Narang also use temporary indices called *side files* [21] to allow online index construction while the data is being queried and updated. Differential indices were first introduced by Lang et al. [13].

An example of a CMVBT database setup is given in Figure 2. The example shows a database with five committed transactions (versions 1–5), and two active, uncommitted transactions (with temporary identifiers 102 and 104). The variable v_{commit} denotes the commit-time version number of

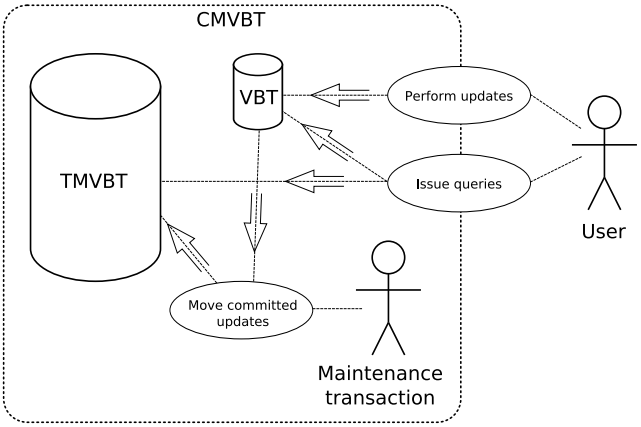


Figure 1: The CMVBT structure organization. The user issues queries both to the VBT and to the TMVBT, but updates are performed only on the VBT. A system maintenance transaction is run periodically to apply the updates from the VBT into the TMVBT.

the latest committed transaction. The temporary identifiers 103 and 101 correspond to committed versions 4 and 5, respectively. The concepts present in the image are explained in the following paragraphs.

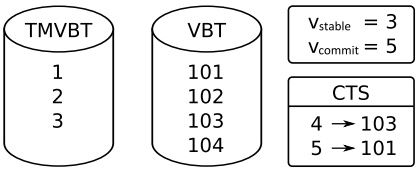


Figure 2: A logical example of a CMVBT. The database contains five committed versions, versions 1–5. The TMVBT index contains all the updates of committed versions 1–3, and the updates of committed versions 4–5 are still located in the VBT index (with transient transaction identifiers 103 and 101).

Following Lomet et al. [16], we assign a transient transaction identifier *start-time*(T) for each new transaction T. The updates of the transaction T will be stored to the VBT index with this transient identifier used as the VBT entry version number (identifiers 101–104 in the example situation of Figure 2). The version numbers of committed transactions, however, must be based on their committing order, not their starting order. Therefore, when an active transaction T commits, we assign it a permanent commit-time-based version number, denoted *commit-time*(T) (versions 1–5 in the example image). This version number defines the ordering of committed transactions. When the updates of a committed transaction T are moved to the TMVBT, the commit-time version number is known, and will be used in the TMVBT index. Thus, the TMVBT stores commit-time version numbers exclusively, and the VBT index stores only transient transaction identifiers. The transient transaction identifiers are internal to the database system, and not visible to the users. Thus, the users only use the commit-time version numbers when issuing history queries to the database. Both of these version numbers can be based either on the system

time, or on an increasing counter value, as long as they are unique and ordered consecutively. In the following discussion, we assume that the version numbers are based on an increasing counter value.

Since the transient transaction identifiers and the actual, commit-time version numbers of the transactions may differ, a mapping from commit-time version numbers to the transient identifiers is maintained in an in-memory hash-table called the *commit-to-start* table, or CTS table. In the example image of Figure 2, the updates of committed transactions with commit-time version numbers 4 and 5 are still located in the VBT with transient transaction identifiers 103 and 101. A mapping $t_c \rightarrow t_s$, for a committed transaction T with *commit-time*$(T) = t_c$ and *start-time*$(T) = t_s$, is deleted from the CTS table once the maintenance transaction has moved all updates of transaction T from the VBT index into the TMVBT and committed. The CTS table does not need to be backed to disk. If the system fails, the CTS table can be reconstructed based on the log file contents during the analysis pass of the ARIES restart recovery [20].

The CMVBT structure maintains a variable v_{stable} that tells which commit-time versions are already reflected in the TMVBT index. In other words, for each version $v \leq v_{\text{stable}}$, all the updates of a transaction T with *commit-time*$(T) = v$ have been moved to the TMVBT. These versions are called *stable* versions. This is stated by the following invariant:

$$\begin{array}{l} \forall v : v \leq v_{\text{stable}} \Rightarrow \textit{updates-of}(v) \in \text{TMVBT} \\ \forall v : v > v_{\text{stable}} \Rightarrow \textit{updates-of}(v) \in \text{VBT} \end{array} \quad (1)$$

The updates of all committed transactions with commit-time version numbers larger than v_{stable} are still located in the VBT. These versions are called *transient* versions. A committed transaction can thus be either transient or stable. In the example image of Figure 1, the commit-time versions 1–3 are stable, and the commit-time versions 4–5 are still transient.

When the maintenance transaction is running, some or all of the updates of a single version, named the *move version* v_{move}, can be located in both of the structures. The second invariant, namely

$$v_{\text{move}} = v_{\text{stable}} \vee v_{\text{move}} = v_{\text{stable}} + 1, \quad (2)$$

states that the move version is either the committed version (in which case the maintenance transaction is not running); or it is one higher than the committed version. When the maintenance transaction is running, the user transactions still use the committed version counter v_{stable} to direct the search for the correct version of the data items. In the example image of Figure 2, the maintenance transaction could be moving updates of transaction T with *commit-time*$(T) = 4$ from the VBT to the TMVBT.

When multiple active transactions update a single key, all the different values of the same key are stored in the VBT, ordered by the start timestamps of the transactions that created them. More formally, when a transaction T with *start-time*$(T) = v$ updates an entry with key k by inserting value w (or by deleting the key, in which case $w = \perp$, a special marker used to mark deletion), it will either (1) insert a new entry (k, v, w) into the VBT if no entry (k, v, w') exists; or (2) replace the existing entry (k, v, w') by (k, v, w). When updating transactions commit, the updated values are left in the VBT until the maintenance transaction moves the updates to the TMVBT.

Because the ordering of the committed transactions may differ from the ordering of the transient identifiers of the transactions, the entry values in the VBT might not be in the correct order. Therefore, reading transactions that wish to read a version $v > v_{\text{stable}}$ need to find the transient identifiers of the transactions with commit-time version numbers $v_i : v_{\text{stable}} < v_i \leq v$, and search the VBT for all these versions, so that the most recent update is found. In the example situation of Figure 2, a user querying for keys that are present in a commit-time version 5 needs to find all keys in the VBT stored with transient transaction identifiers 103 and 101 (corresponding to commit-time versions 4 and 5). This is relevant for situations when the transaction with commit-time version 4 has inserted a key that has not been modified by the transaction with commit-time version 5 (and is thus still present in the commit-time version 5).

More formally, we define $t_{\text{commits}}(v)$ to be the set of transient committed versions: $t_{\text{commits}}(v) = \{v_c : v_{\text{stable}} < v_c \leq v\}$. In the example of Figure 2, $t_{\text{commits}}(5) = \{4, 5\}$. Furthermore, we define the set $t_{\text{starts}}(v)$ to contain the transient identifiers associated with the commit versions: $t_{\text{starts}}(v) = \{v_s : v_c \in t_{\text{commits}}(v) \wedge v_s = CTS[v_c]\}$. In the example, $t_{\text{starts}}(5) = \{103, 101\}$. Now, when looking for the most recent update of key k, the VBT must be searched looking for the updates of any version $v_s \in t_{\text{starts}}(v)$. After all these updates for any given key have been found, they must be ordered according to the ordering of the corresponding commit-time version numbers in $t_{\text{commits}}(v)$. After this, the most recent update is known to be the last update in the ordered list. If no updates on the key k are found in the VBT, the most recent update can be found from the TMVBT.

This is actually not an MVBT-specific issue, but is also present in other multiversion database index structures that allow multiple concurrent updating transactions to commit in an order that is not the same as their corresponding starting order. Lomet et al. [16] have solved this problem by using a technique called *lazy timestamping*, in which they change the version numbers of entries of a committed transaction when the entries are first accessed after the transaction has committed. This technique also requires a lookup table (called *persistent timestamp table*) for converting temporary version numbers to commit-time version numbers. In our technique the version numbers of entries are corrected when the updates are moved to the TMVBT for permanent storage.

3. VERSIONED B$^+$-TREE

In this section, we describe a B$^+$-tree index for storing multiple versions of a data item. We define a *versioned B$^+$-tree*, or *VBT* for short, to be a B$^+$-tree that stores entries of the form (k, v, w), where k is the entry key, v is the version of the data item, and w is the value associated with the entry. The entries are ordered first by the key k, and then by the version v so that $(k_1, v_1, w_1) < (k_2, v_2, w_2)$ if and only if $k_1 < k_2$ or $(k_1 = k_2) \wedge (v_1 < v_2)$. This is an unambiguous ordering since the entries are uniquely identified by the pair (k, v). Entry deletions are recorded by inserting an entry deletion marker $(k, v, \perp)$ into the VBT. The version number v in a VBT entry (k, v, w) is the transient identifier *start-time*(T) of the transaction T that created the versioned entry. The final version number of the entry with which it will be stored in the TMVBT will be the commit timestamp *commit-time*(T) of transaction T, if T commits.

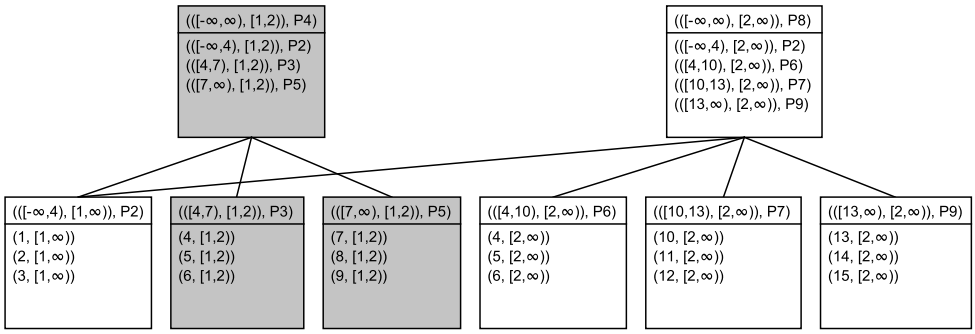


Figure 3: An example of a TMVBT. The page header format and the format of index-page entries is ((key range, version range), page identifier). The format of leaf-page entries is ((key, version range), data-item identifier), but the data-item identifiers have been left out for clarity. White pages are 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.

If the transaction does not commit, but aborts, then the versioned entry will be physically deleted from the VBT, and hence will never appear in the TMVBT.

While this extension to the standard B^+-tree by itself is functionally sufficient for storing multiversion data, it cannot be used as the basis of any larger database index, as it cannot support range queries efficiently. This is because the consecutive keys of any given version are not clustered together, and need to be searched individually. In our approach, we use the VBT as a small, temporary index where only the updates of active transactions are stored. Once a transaction T has committed, the updates performed by T will be applied to the TMVBT index, which is the main index structure. The updates of transactions are stored in the VBT using the aforementioned entries: an entry (k, v, w) is used to record an insertion (or update) of a key k, while an entry $(k, v, \perp)$ is used to record a key deletion.

We further define the VBT to have sibling pointers at each level pointing to the next page at the same level, like in a B^{link}-tree [14]. These pointers will be used to accelerate key-range scans in the tree. Structure-modification operations, on the other hand, are performed as atomic operations that transform a structurally valid VBT into another structurally valid VBT, as described by Jaluta et al. [10, 11].

All updates by user transactions are performed by inserting keys to the VBT. Therefore, all undo actions on the VBT are physical key deletions. However, if a single transaction updates the same key multiple times, the previous values stored with the key are overwritten, and need to be restored when rolling back the transaction to a preset savepoint. Thus we write standard physiological redo-undo log records to log forward-rolling update actions, and redo-only log records to log undo actions, of user transactions. The structure-modification operations are logged with redo-only log records, so that they are never undone. These log records are used to bring the VBT up-to-date after a single transaction has crashed. If the entire database system crashes, it is possible to bring the VBT up-to-date either by using ARIES-based recovery [20, 19], or by reconstructing the index logically based on the log contents, including only the entries of committed transactions.

4. TRANSACTIONAL MVBT

In this section, we briefly review our transactional extension (TMVBT) [8] to the multiversion B^+-tree (MVBT) by Becker et al. [4]. The MVBT is an asymptotically optimal multiversion index structure. In practice, the MVBT guarantees that all actions on version v have the same asymptotic page access bounds as a single version B^+-tree index that contains only the entries that belong to the version v. This result holds for all versions v of the database. Entries that belong to the most recent version of the MVBT are called *live* entries. Entries that have been deleted from the MVBT are not physically deleted but marked as *killed*. Similarly, entries belonging to a version v are said to be *live at version* v. The pages of the MVBT index form a directed acyclic graph (DAG), instead of a tree. However, a subset of these pages forms a *search tree* for version v, for all versions v. The subset is formed of all pages that are live at version v. The search trees of different versions may be overlapping, that is, they may reuse same pages. The asymptotically optimal bounds are maintained by guaranteeing that each page p that belongs to the search tree of version v contains at least min entries that are live at version v, where min is a configurable variable. Thus, the complexity of all tree traversal operations in the search tree of version v is logarithmic in the number of entries live at version v. Other index structures do not have this kind of asymptotic guarantee for each version. This makes the MVBT a good choice for applications where historical data needs to be queried often, especially in presence of key-range queries.

The original MVBT by Becker et al. [4], however, assumes a single-update transaction model, in which each update causes the database version number to increase. Also, concurrency-control and recovery algorithms were not discussed. Our transactional extension to the MVBT, called the *transactional multiversion B^+-tree*, or *TMVBT* [8, 9], extends the transaction model to allow multiple updates to be applied within a single transaction. We described an efficient concurrency-control and recovery scheme that allows a single updating transaction to operate concurrently with multiple read-only transactions. We have shown that our extended algorithms maintain all the asymptotically optimal access time guarantees of the MVBT.

Our extensions are based on the notion of active and inactive pages and entries. An entry (or a page) is *active* when it has been created by the active updating transaction. Similarly, an entry (or a page) becomes *inactive* immediately when the active updating transaction commits. Active pages may only contain active entries. Active entries can be physically moved around and deleted, while inactive entries are always left in place, with active copies created of them if necessary. This organization allows us to treat the active data physically, in the same way as entries are treated in a B^+-tree, and thus we can maintain the optimality guarantees within the active version even with multiple updates. An example of a TMVBT index is given in Figure 3. More detailed examples can be found in our previous article on the structure [8].

5. CONCURRENCY AND RECOVERY

The CMVBT allows us to use various approaches for concurrency control and recovery. Our approach for recovery follows the ARIES algorithms [20, 19] with physiological logging and standard steal-and-no-force page buffering policy. Each structure-modification operation on both the VBT and the TMVBT is logged using a single physiological redo-only log record, so that interrupted tree-structure modifications are never rolled back (undone) when a transaction aborts or system fails. This approach is described by Jaluta et al. [10, 11]. The actions of user transactions on the VBT are logged with standard redo-undo and redo-only log records, as described in more detail in Sec. 3. These log records are required to support total and partial rollbacks of active transactions. All update actions on the TMVBT are performed by the maintenance transaction, which is never aborted or undone. Therefore, redo-only log records are sufficient for logging the actions performed on the TMVBT. For efficiency, we employ separate page buffers for the TMVBT and the VBT. This way the entire VBT index is kept in main memory during normal transaction processing.

Concurrency control on the key level is provided by using the snapshot isolation (SI) algorithms [5]. In snapshot isolation, a consistent state of the database is maintained for each transaction. The state maintained for a transaction T is called a *snapshot*, and it reflects the committed state of the database as of the time T started (denoted *snapshot-time*(T), see Sec. 7). Read-only transactions always read data from their own snapshot, and thus do not require locks to protect the keys from concurrent modifications. Updates to the database are performed by adding a new version of the updated key to the snapshot of the updating transaction, allowing updating transactions to read their own modifications directly from the snapshot. Consistency for updating transactions is guaranteed by checking that overlapping transactions do not make updates to the same keys. Snapshot isolation is an obvious choice for multiversion structures, because the entire history of the database is preserved, and thus the snapshot state is available for each transaction. The definition of snapshot isolation by Berenson et al. [5] does not include details on how the consistency checks of SI should be implemented. The commercial Oracle database uses a combination of locking and versioning to achieve snapshot isolation [22]. In Oracle's implementation, when transaction T needs to modify key k, it acquires a commit-duration exclusive lock on k, such that:

- If another transaction T' already holds an exclusive lock on k, T is blocked until T' either commits or aborts. If T' finally commits, T is aborted; if T' aborts, T can continue.
- If a transaction T' with overlapping execution time has committed after T began (i.e., *commit-time*(T') > *snapshot-time*(T)), and T' has updated key k, then T is aborted immediately.

Note that the exclusive locks taken by updating transactions in this approach only block other updating transactions, because read-only transactions take no locks. Following Oracle, we use this approach to achieve snapshot isolation.

Structural consistency of the VBT index is maintained by standard page-latching operations, with latch-coupling (also called crabbing by Gray and Reuter [7]) applied to ensure child-link consistency during tree traversals. We define the latching order to be top-down, left-to-right for all transactions, and we disallow upgrading read latches to write latches. These are necessary (and sufficient) restrictions to avoid deadlocks caused by page latching [7]. For the TMVBT, pages need to be latched, but latch-coupling is not always required. This is because the only transaction that is allowed to update the TMVBT is the system maintenance transaction, and there can be only one such transaction running at a time. Thus, the maintenance transaction does not need to do latch-coupling. Furthermore, as explained in our TMVBT paper [8, 9], readers that read inactive data do not need latch-coupling either. Therefore read-only transactions that are reading stable versions do not need latch-coupling. This is explained in more detail in Sec. 7. Further details of concurrency control and recovery are embedded in the explanations of the algorithms themselves.

6. MAINTENANCE TRANSACTION

The maintenance transaction is a system-generated transaction that is run periodically to move the updates of committed transactions, one at a time, from the VBT index into the TMVBT. To ascertain correct operation with concurrent user transactions, the updates must be moved in such a way that the system transaction does not cause user transactions to miss any keys. This is easiest to accomplish by first applying the updates to the TMVBT, then increasing the stable version counter v_{stable}, and only then removing the updates from the VBT. A consequence of this approach is that the user transactions must be prepared to possibly encounter the same update twice when scanning the index structures.

The maintenance transaction performs the following steps:

1. Update the move version counter $v_{\text{move}} \leftarrow v_{\text{stable}} + 1$. Find out the corresponding transient transaction identifier $v_s \leftarrow \text{CTS}[v_{\text{move}}]$.
2. Scan through the VBT, and find all updates of version v_s. Apply the updates to the TMVBT, changing the version number from v_s to v_{move}.
3. Update the stable version number to $v_{\text{stable}} \leftarrow v_{\text{move}}$.
4. Scan through the VBT a second time, and physically delete all entries with version v_s.
5. Remove the mapping $v_{\text{move}} \rightarrow v_s$ from the CTS table.

The actions performed by the maintenance transaction are logged using redo-only log records. If the system crashes during the execution of the maintenance transaction T_m, the redo phase of the restart recovery will redo all actions performed by T_m to bring the database pages into a consistent state and restart the maintenance transaction. All the steps of the maintenance transaction are idempotent, meaning that performing them multiple times has the same effect as performing them once. That is, $f(f(x)) = f(x)$ for all actions f of the maintenance transaction T_m. Thus, when the maintenance transaction is restarted after a system crash, it will automatically skip those actions that already have been performed.

At the beginning, in step 1, the move version counter is increased to show that the maintenance transaction is running. The system must guarantee that there are never two maintenance transactions running at the same time. Thus, at the very beginning, the maintenance transaction takes a commit-duration exclusive lock on a global *maintenance-transaction* marker. This action is logged by a redo-only log record that identifies the action (start the maintenance transaction) and contains the new v_{move} counter value and the corresponding transient transaction identifier v_s.

At the next step, step 2, the updates are copied from the VBT into the TMVBT. The system transaction scans through both structures at the same time, using a saved path for the TMVBT and using latch-coupling for the VBT to maintain structural consistency. However, as the system transaction is the only transaction updating the TMVBT, no latch-coupling on the TMVBT is required, and thus only one page needs to be latched at a time (except during structure-modification operations). Because the TMVBT does not have sibling links, the saved path needs to be backtracked from time to time to locate the next leaf page. Again, because no other transaction can update the TMVBT concurrently, it is safe to relatch pages on the saved path when backtracking. In the VBT, we can use the sibling links defined in Sec. 3 to traverse through the entire tree efficiently. Because we are only reading the updates from the VBT at this point, it is sufficient to only scan through all the leaf pages of the VBT without ever backtracking. Thus no saved path is needed for the VBT for this step. Each update that is performed to the TMVBT is logged using a redo-only log record.

Step 3 updates the stable version counter v_{stable}. The counter is protected by simply latching the page where the counter is stored. At this point we know that all the updates of transaction T with *commit-time*$(T) = v_{\text{move}}$ have been applied to the TMVBT. New read-only transactions can therefore read the version v_{move} directly from the TMVBT. Thus, first a write latch is acquired on the page containing the counter, the counter is then updated to $v_{\text{stable}} \leftarrow v_{\text{move}}$, and the write latch is released. To speed up recovery, this action is also logged with a single redo-only log record, and the log is forced onto the disk at this point. If this log record is found after a system crash, steps 1–3 of the maintenance transaction can be skipped entirely, and the maintenance transaction can continue at step 4 to finish the transaction.

Next, at step 4, the updates are cleared from the VBT. It is safe to delete these updates, because although deleting the entries may cause concurrent user transactions to miss an update they expected to find in the VBT, the user transactions will find the missed update later on from the TMVBT, where it has already been applied to at this point. To correctly maintain the structural consistency of the VBT, the tree must be traversed using a depth-first search so that structure-modification operations can be performed if pages contain too few entries after entry deletions. During the depth-first search, latch-coupling will be applied, first top-down, then left-to-right, as explained earlier. The delete actions are logged with redo-only log records.

Finally, in step 5, the temporary transaction identifier mapping is removed from the commit-to-start (CTS) hash table. The maintenance transaction commits by writing a commit log record. The log must be forced onto disk at this point.

The following theorem states the correctness and complexity of the maintenance transaction:

Theorem 1. Let us denote by n the number of updates applied by the earliest transient committed transaction with version v_{move}, by n_V the number of entries in the VBT and by n_T the number of live entries in the TMVBT at version $v_{\text{move}} - 1$. The maintenance transaction correctly transforms the transient version v_{move} into a stable version by applying the updates of the version v_{move} into the TMVBT, and by removing these updates from the VBT. The maintenance transaction requires at most $O(n_V/B + \min\{n \log n_T, (n_T + n)/B\})$ page accesses, where B is the page capacity (assumed, for clarity, to be the same for both structures).

Proof. The complexity of the maintenance transaction is composed of the VBT and TMVBT index scans of steps 2 and 4. The complexity of both scans of the VBT is the same, $O(n_V/B)$, because a full scan is required for both of the steps. Maintaining the entire saved path from root to leaf for the VBT in the deletion phase does not add to the asymptotic complexity of the scan. Applying the n updates into the TMVBT requires $O(n \log n_T)$ page accesses in the worst case, because a single update operation in the TMVBT requires at most $O(\log n_T)$ page accesses (see Theorem 4 in our TMVBT article [8]). However, the leaf pages of the version tree of the latest version of the TMVBT are accessed at most once, so the operation is also bound by $O((n_T + n)/B)$; that is, a full leaf-level page scan of the latest version of the TMVBT plus possible new pages created by insertions. □

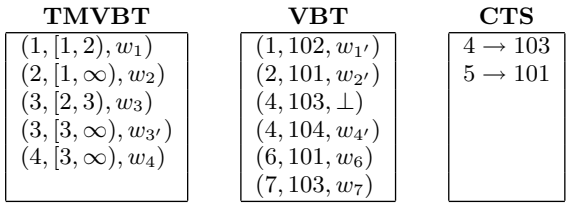
7. USER TRANSACTIONS

We allow two kinds of user transactions to operate concurrently on the CMVBT: read-only transactions and updating transactions. There are no restrictions on how many transactions of either type can be running at the same time. A read-only transaction T_r may issue single-key queries, range queries and next-key queries for any version v that was committed at the beginning of T_r. The version v must be specified at the beginning of the read-only transaction T_r, and T_r will see a consistent view of the database at this selected version. This is in accordance with the snapshot isolation protocol.

An updating transaction T_u may consist of any sequence of updates (key insertions and deletions), and queries of version *snapshot-time*(T_u) (single-key, range, or next-key queries), where *snapshot-time*(T_u) is defined to be the latest version that was committed when T_u began, that is, v_{commit}. The updating transaction T_u may also set savepoints and rollback to any preset savepoint. Rolling back to a preset save-

point is accomplished by going through the redo-undo log records, as explained in the ARIES algorithms [20].

An example of the possible contents of the CMVBT index is given in Figure 4. This example represents the same situation as the previous example of Figure 2, but now the actual entries stored in the different indices are shown.



History of transactions

T_1, *commit-time*$(T_1) = 1$: Insert 1, 2
T_2, *commit-time*$(T_2) = 2$: Insert 3, delete 1
T_3, *commit-time*$(T_3) = 3$: Insert 3, 4
T_4, *commit-time*$(T_4) = 4$: Insert 7, delete 4
T_5, *commit-time*$(T_5) = 5$: Insert 2, 6
T_6, *start-time*$(T_6) = 102$: Insert 1
T_7, *start-time*$(T_7) = 104$: Insert 4

Figure 4: Example of data entries stored in a CMVBT index. This example represents the same situation as Figure 2. The format of entries in the TMVBT is (key, version range, data), and the format of entries in the VBT is (key, version, data). In addition to the committed transactions, the VBT also contains entries of two active transactions with transient identifiers 102 and 104.

For the query and update actions, we can distinguish between the following different algorithms that must be defined to use the CMVBT. These are: (1) performing an update action (insert or delete); (2) querying for a key of a stable version; and (3) querying for a key of a transient version. Also, to support range queries and next-key queries, we need to define algorithms for (4) querying for the next key of a stable version; and (5) querying for the next key of a transient version. In the following discussion, we present algorithms for performing these actions, first in a general level, and then in more detail.

For the first point, performing an update action on the CMVBT is reasonably straightforward: all we need to do is record the action to the VBT, using proper top-down, left-to-right latch-coupling for tree traversal. An insertion (or update) of a key k by transaction T with *start-time*$(T) = v$ is recorded by adding the entry (k, v, w), where w is the entry value; and deletion of a key k by the same transaction is recorded by adding the entry $(k, v, \perp)$. If the VBT index already contains an entry (k, v, w') with the same key and version, the old entry will be replaced by the new entry. Otherwise, all update actions always add a new key into the VBT index. For concurrency control, T acquires a commit-duration exclusive lock on the key k, as explained in Sec. 5. All the update actions of a forward-rolling updating transaction are logged with a single physiological redo-undo log record. For more details on logging of update actions on the VBT, refer to Sec. 3.

Querying for stable versions is also straightforward. Both read-only and updating transactions can use the same algorithms for querying the index structures, and we will thus use the term *reader* to refer to a read-only transaction or an updating transaction that is performing a query action. When a version is stable, and the reader knows it to be stable (the version was stable at the beginning of the read action), the reader can directly query for the version from the TMVBT, as explained in our TMVBT article [8, 9]. In this situation, the TMVBT index can be traversed without latch-coupling, and saved paths can be used for key-range and next-key queries without having to check for page validity when relatching them. This is possible because all pages traversed and entries read are inactive, and thus guaranteed to remain where they are located.

When querying for a transient version, more care needs to be taken. First of all, the reader cannot know beforehand which index structure contains the most recent update that precedes the queried version. The relevant version might be the last update in the TMVBT, or there may be an intermediate update in the VBT. In the example of Figure 4, when querying for a key of commit-time version 5, the relevant version for key 3 is the last update in the TMVBT (by transaction T_3); and the relevant version for key 2 is the update in the VBT (by transaction T_5 with transient transaction identifier 101). Thus, whenever querying for a transient version, both structures may need to be checked. In more detail, for single-key queries (with the key given), it is sufficient to search the VBT if an update on the key is found from there; only if no such update is found from the VBT do we need to consult the TMVBT to find the latest update on the key. However, when performing next-key queries, we always need to check both structures, because it is impossible to determine which structure contains the nearest key otherwise. Also, because the maintenance transaction may be moving the updates of the move version v_{move} to the TMVBT, the reading algorithms must be prepared to miss some updates in the VBT, and possibly re-encounter some updates in the TMVBT. However, the situation is amended by organizing the read actions in such a way that the VBT is always consulted first, and the TMVBT only afterwards. This guarantees that no update will be entirely missed.

The general algorithm for querying for a single key is given in Algorithm 1. As the stable version counter is protected by latches, a read latch must be taken on the page that contains the counter to read the stable version counter value. The latch can be released immediately after the counter value has been read. It is also possible to read the counter value once, at the beginning of the transaction, and then cache the result. This may lead to a read-only transaction trying to find updates from the VBT that have already been moved away from there, but this does not cause problems, because the updates will be found from the TMVBT later on.

Querying for a single key from the TMVBT (key-query-TMVBT) works exactly as explained in our TMVBT article [8, 9], except that the third parameter is used to indicate whether the reader might need to read active data and must therefore use latch-coupling. In other words, if the version is known to be stable, the traversed paths in the TMVBT are known to be inactive and thus remain stable. If the reader is looking for the latest update of a transient version, the updates might be active in the TMVBT because a concurrent maintenance transaction is writing them, and thus the reader must use latch-coupling to ascertain path validity and check that previously released pages are still valid when relatching them. When reading the pages of the TMVBT, the

Algorithm 1 key-query(k, v, T)

```
1. if v ≤ v_stable then // Is v stable?
2.    w ← key-query-TMVBT(k, v, false)
3. else
4.    w ← key-query-VBT(k, v, T)
5.    if w = ∅ then // If no updates found from VBT
6.       w ← key-query-TMVBT(k, v, true)
7.    else if w = ⊥ then // Latest update is a deletion
8.       return ∅
9.    end if
10. end if
11. return w
```

reader can determine whether a page has been invalidated by a concurrent structure-modification operation (for example, when relatching pages on a previously released saved path) by internally caching the page-LSN before releasing latches and comparing the cached value to the value in the page after it has been relatched. If the new page-LSN is larger than the cached one, the page has been modified by the maintenance transaction, and it is necessary to either re-traverse the entire path from the root of the TMVBT or to backtrack up the path until an unchanged page is found.

Querying for a single update from the VBT is described in Algorithm 2. The algorithm generates a reverse start-to-commit (STC) mapping for all relevant commit-time versions, mapping the possible updates created by the transaction itself to infinity so that they always take precedence over other updates. A transient identifier list V is also generated. This list contains the keys of the STC map. The query-all-VBT(k, V) function finds all entries (k, v) from the VBT with key k and version $v \in V$. This function is implemented with a standard tree-traversal operation using latch-coupling. After finding all the possibly relevant updates, the key-query-VBT function orders the updates in transaction commit order, and returns the latest update. If there are no updates on the key k in the VBT, the function returns the null value $\emptyset$. If the last update is a deletion, the function returns the deletion marker $\perp$. These values must be separate so that the single-key query algorithm knows whether to continue searching from the TMVBT or not.

In the example situation of Figure 4, the reverse STC mapping when querying for commit-time version 5 with a read-only transaction T_r is $\{101 \to 5,\ 103 \to 4\}$. If the querying transaction is an updating transaction T_u with transient transaction identifier *start-time*(T_u) = 104, the STC mapping is $\{101 \to 5,\ 103 \to 4,\ 104 \to \infty\}$. For example, if T_u queries for key 4 (at transient commit-time version 5), the STC is generated as explained above, and the start version list $V = \{101, 103, 104\}$. The VBT query finds the entries $(4, 103, \perp)$ and $(4, 104, w_{4'})$, which are then converted to commit-time entries and placed in the ordered result map $K_c = \{4 \to \perp, \infty \to w_{4'}\}$. The query returns the last value from this map, $w_{4'}$, as the result.

When a user transaction T is querying for a single key k of a transient version $v > v_{\text{stable}}$, and the maintenance transaction T_m is ongoing, one of the following situations may occur, regarding an update of version v_{move} when that update is the latest update on the key k:

1. T finds the update from the VBT. The update is not yet in the TMVBT. This is the normal situation, and no special processing is required.

Algorithm 2 key-query-VBT(k, v, T)

```
1. STC ← ∅
2. V ← ∅
3. for each v_c ∈ {v_stable + 1, ..., v} do
4.    v_s ← CTS[v_c] // transient transaction identifier
5.    STC[v_s] ← v_c
6.    V ← V ∪ v_s
7. end for
8. if T is an updating transaction then
9.    STC[start-time(T)] ← ∞
10.   V ← V ∪ start-time(T)
11. end if
12. K_s ← query-all-VBT(k, V)
13. K_c ← ∅
14. for each (k, v, w) ∈ K_s do
15.   v_c ← STC[v]
16.   K_c[v_c] ← w
17. end for
18. return entry K_c[v_c] with highest version v_c
```

2. T finds the update from the VBT. The update has been applied to the TMVBT by the maintenance transaction T_m. Because the update was still found from the VBT, the TMVBT is not searched, and thus this situation is similar to the first one.

3. T does not find the update of version v_{move} from the VBT, because the maintenance transaction T_m has already deleted it. Because no update was found from the VBT, the TMVBT is scanned for the latest update. At this point, the maintenance transaction T_m has already applied the update to the TMVBT, so the update is found from there.

If there is a newer update with commit-time version v_c such that $v_{\text{move}} < v_c \leq$ *snapshot-time*(T), then this update will be found from the VBT in all of the above situations, and it will be returned directly without ever consulting the TMVBT.

For next-key queries, the general algorithm is described in Algorithm 3. When querying for stable versions, it is again sufficient to only scan through the TMVBT to find the next key. However, with transient versions, both structures must always be searched to find the next key.

When searching for a transient version, the next keys from both structures need to be retrieved alternatingly. In Algorithm 3, starting from line 4, the current key is initialized to the previously found key. After this, at the beginning of the infinite loop, both of the index structures are scanned to the next key, and we check which key is nearest to the previously fetched key. If the next nearest key is found from the TMVBT (line 8), we can return the key and the corresponding value directly. If, on the other hand, the nearest key is found from the VBT (line 10), we must check whether the latest update in the VBT is an insertion (and not a deletion) before returning the key. Similarly, if the keys fetched from both structures are the same (line 14), we need to check that the latest update on the key in the VBT is not a deletion. If the latest update is a deletion, we need to skip this key and scan forward to find the next keys. In this situation, we must always scan both structures forward, because an ongoing maintenance transaction might have changed the nearest key in the TMVBT.

As with the function key-query-VBT, the function next-

Algorithm 3 next-key-query(k, v, T)

```
1. if v ≤ v_stable then // Is v stable?
2.   return next-key-query-TMVBT(k, v, false)
3. else
4.   k_c ← k
5.   loop
6.     (k_1, w_1) ← next-key-query-VBT(k_c, v, T)
7.     (k_2, w_2) ← next-key-query-TMVBT(k_c, v, true)
8.     if k_1 > k_2 then // Nearest key is in TMVBT
9.       return (k_2, w_2)
10.    else if k_1 < k_2 then // Nearest key is in VBT
11.      if w_1 ≠ ⊥ then // Is w_1 an insertion?
12.        return (k_1, w_1)
13.      end if
14.    else // Same key returned from both structures
15.      if k_1 = ∅ then // No more keys in either index
16.        return ∅
17.      else if w_1 ≠ ⊥ then // Is w_1 an insertion?
18.        return (k_1, w_1)
19.      end if
20.    end if
21.    k_c ← k_1 // Scan forward in both indices
22.  end loop
23. end if
```

key-query-VBT must find the transient transaction identifiers of all the commit-time versions between v_{stable} and v, find updates matching these start-time versions, and order the updates based on their corresponding commit-time versions. The actual implementation of the next-key query should use saved paths to accelerate the next-key queries from the VBT and the TMVBT. With saved paths, most of the consecutive next-key calls to the VBT and CMVBT will fall to the same leaf page, and will thus reuse the existing path without requiring any additional I/O operations. To further enhance the operation, the latches and page fixes required for the VBT and TMVBT index structures need not be released at all between the next-key-query sub-operations in the main loop of Algorithm 3.

As an example of a next-key query, suppose a read-only transaction T_r is querying the next key from key 3, at version 4, in the example situation of Figure 4. At the beginning of the next-key-query function, the next entries are fetched from both structures: $(k_1, w_1) = (4, \perp)$ (from the VBT) and $(k_2, w_2) = (4, w_4)$ (from the TMVBT). Because the keys of both entries are the same, the one retrieved from the VBT takes precedence. However, because it is a deletion marker, we need to continue the search to find the next keys. Thus, the algorithm continues by finding the next entries from key 4 from both of the structures: $(k_1, w_1) = (7, w_7)$ (from the VBT) and $(k_2, w_2) = \emptyset$ (from the TMVBT). Because the TMVBT has no more entries, the VBT entry is the next key, and the key-value pair $(7, w_7)$ is thus returned to the user.

The following theorems state the correctness and complexity of the user actions, expressed in how many pages need to be visited:

Theorem 2. The update action correctly records a key update (key insertion or deletion) in the VBT. A single update action requires $O(\log n_V)$ page accesses, where n_V is the number of entries in the VBT.

Proof. The update action simply needs to traverse the VBT once to insert the update marker, so the complexity $O(\log n_V)$ comes from the tree traversal. As with standard B$^+$-trees, any possibly required structure-modification operation does not affect the asymptotic complexity of the action. □

Theorem 3. The key-query action for key k and version v correctly returns the most recent committed version v' of the queried key, relative to version v so that $v' \leq v$. A single key-query action for a stable version v requires $O(\log n_T)$ page accesses, where n_T is the number of entries in the TMVBT that are live at version v. A single key-query action for a transient version v requires at most $O(\log n_V + (n_a + n_t)/B + \log n_T)$ page accesses, where n_V is the number of entries in the VBT, n_T is the number of entries in the TMVBT that are live at version v, n_a is the number of active transactions, n_t is the number of transient versions, and B is the page capacity.

Proof. The key-query action for a stable version similarly only does a single query to the TMVBT, and thus has a complexity of $O(\log n_T)$ [8]. For a transient version, the VBT search requires $O(\log n_V)$ page accesses for the initial tree traversal, and at most an additional $O((n_a+n_t)/B)$ leaf page accesses to locate all versions that might be relevant to the key query. Finally, the query for a transient version may need to further query the value from the TMVBT, thus adding the $O(\log n_T)$ term to the complexity. □

8. RELATED WORK

While there are many other index structures for multiversion data, the majority of them are not optimal under all histories of user transactions, at least for all actions (single key queries, key-range queries, insertions and deletions). The only other optimal multiversion database—to our knowledge—is the multiversion access structure (MVAS) of Varman and Verma [24]. The MVAS is similar in structure to the MVBT, with notable differences only in the splitting policies and the access lists of the MVAS that are used for version-range queries. Thus it also has the same limitations as the original MVBT of Becker et al. [4]. The general database organization described in this article should be applicable to the basic structure of the MVAS also. However, it is unclear to us whether the access lists of the MVBT would directly support the new concurrent database organization.

Another structure similar to the MVBT is the time-split B$^+$-tree (TSBT) of Lomet et al. [17, 18], which is used as the basis of the multiversion database engine Immortal DB in Microsoft SQL Server [15, 16]. 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 not possible in the MVBT (or its extensions). However, in the TSBT, certain types of user transactions may cause the index structure performance to degrade. This is because leaf pages are never merged with sibling pages. Thus, key deletions can leave some leaf pages with only a few live entries (or none at all), thereby degrading key-range query performance.

Another approach for storing multiversion data is key hashing. The hashing structure of Kollios and Tsotras [12], for example, is very efficient for exact-match queries, with an access time of $O(1)$ in ideal situations. However, key-range queries cannot be performed efficiently on hashing structures, because consecutive entries of any given version are

not clustered together. Likewise, it is impossible to find out the next key of any given key, as the keys are not ordered in the index structure.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have designed a concurrent multiversion database index structure called the concurrent multiversion B^+-tree (CMVBT) which is based on the asymptotically optimal multiversion B^+-tree (MVBT) of Becker et al. [4]. The CMVBT uses a separate versioned B^+-tree (VBT) index to store the updates of active transactions. Once the active transactions have committed, the updates are moved, one transaction at a time, from the VBT into the main transactional multiversion B^+-tree (TMVBT) index. The TMVBT is our transactional extension to the MVBT that extends the MVBT to support multi-action transactions [8, 9]. The CMVBT index retains the optimality of the TMVBT for stable committed versions, and is thus guaranteed to perform optimally under any histories of user transactions.

We have designed algorithms for updating and reading the CMVBT structure concurrently. Standard concurrency-control and recovery algorithms can be used with these algorithms to allow multiple transactions to operate concurrently on the CMVBT structure. The snapshot isolation algorithms [5] are especially well suited for use with our multidimensional index structure, and they guarantee snapshot isolation for all transactions. Our organization allows the data item version numbers to be efficiently changed from the transient transaction identifiers into the commit-time version numbers. This is a non-trivial issue that often requires special book-keeping arrangements in other multiversion index structures that support commit-time version numbering. The TMVBT index retains all the optimal access time guarantees of the MVBT, and therefore queries for stable versions are optimal. The VBT index is kept small by constantly moving the updates of committed transactions into the TMVBT index. Thus, the VBT can be kept entirely in main memory, and we expect that it does not affect the overall query times of the CMVBT much.

We are currently working on implementing the CMVBT algorithms in a simulated database environment. Our plan is to evaluate the performance of the suggested design by running different transactions on the database simulator. We will also implement some of the other multiversion index structures, such as the time-split B^+-tree of Lomet et al. [17, 18], and run the same tests on them. Thus we will be able to compare our algorithms with other indexing approaches. Our expectation is that for transaction histories with little or no key deletes, the CMVBT will perform on the same level as the TSBT. However, for transaction histories with frequent key deletions, we expect the CMVBT to outperform the TSBT because of the the optimality of the underlying TMVBT index structure. This should be especially apparent with large key-range queries.

10. ACKNOWLEDGMENTS

This work has been funded by the Academy of Finland.

11. COPYRIGHT

12. REFERENCES

[1] *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM Press New York, NY, USA, 1992.

[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[3] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[4] 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.

[5] 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 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM New York, NY, USA, 1995.

[6] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

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

[8] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen. Transactions on the multiversion B^+-tree. In *Proceedings of the 12th International Conference on Extending Database Technology*, pages 1064–1075, 2009.

[9] T. Haapasalo, I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency control and recovery for multiversion database structures. In *Proceedings of the 2nd PhD Workshop on Information and Knowledge Management*, pages 73–80, 2008.

[10] 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.

[11] 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.

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

[13] S. Lang, J. Driscoll, and J. Jou. Batch insertion for tree structured file organizations—improving differential database representation. *Information Systems*, 11(2):167–175, 1986.

[14] P. Lehman and B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, Dec 1981.

[15] 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.

[16] 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*, 2006.

[17] 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.

[18] D. Lomet and B. Salzberg. The performance of a multiversion access method. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 353–363. ACM New York, NY, USA, 1990.

[19] C. Mohan. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* [1], pages 371–380.

[20] 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.

[21] 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* [1], pages 361–370.

[22] Oracle. Oracle Database Concepts 11*g* Release 1 (11.1). `http://download.oracle.com/docs/cd/B28359_01/server.111/b28318/toc.htm`, Apr 2009.

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

[24] P. Varman and R. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.