Concurrency Control and Recovery for Multiversion Database Structures

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

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

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

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

ABSTRACT

In modern database applications access to historical versions of the dataset is becoming increasingly important. Several multiversion structures with corresponding concurrency-control and recovery algorithms exist, but none of these have optimal logarithmic execution times for all actions in all situations. The time-split B^+-tree by Lomet et al. (TSBT) is used in the Immortal DB database prototype, but it does not consolidate pages. The multiversion B^+-tree by Becker et al. (MVBT) is an asymptotically optimal multiversion structure that guarantees logarithmic execution times for all actions, but it lacks concurrency-control algorithms.

It is our plan to design and implement several multiversion index structures with full concurrency-control and ARIES-based recovery algorithms and evaluate their performance. We will experiment with using a multiversion B^+-tree as a historical storage, to which the updates of committed transactions are moved one at a time from a separate B^+-tree. We will also consider using an optimized R-tree to store the multiversion data as two-dimensional line segments. To evaluate these solutions, we will also implement a straightforward B^+-tree based solution that stores the different versions of a data item consecutively; and a solution based on the existing time-split B^+-tree. We expect that the solution that uses a multiversion B^+-tree will be the most efficient.

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*

* T.H. is a Ph.D. student, and the first author of this paper

† E.S.-S. is the official supervisor of T.H.

PIKM'08, October 30, 2008, Napa Valley, California, USA.

General Terms

Algorithms, Design, Performance

Keywords

Physical design, access methods, versioned data, transactions, concurrency, recovery

1. INTRODUCTION

Traditional database systems maintain a single version of the stored data. In many applications, access to historical data is becoming increasingly important. Such applications are, for example, medical-record, legal-record and moving-object databases; scheduling applications and inventory control, engineering design and statistical applications [17, 24, 2, 26]. Storing such *multiversion data* requires a special database structure to maintain good query-time and space constraints. Fortunately, there are a number of database structures designed for storing multiversion data. However, concurrency-control and recovery algorithms for these structures have not been comprehensively addressed.

In database research, the standard structure that is used as the basis of virtually all databases is the B^+-tree [1, 6]. The standard B^+-tree, however, is used to store only a single version of data items. A straightforward extension to multiversion data that uses a B^+-tree for storing the different versions of the data items consecutively may be efficient enough for exact-match queries. Concurrency-control and recovery algorithms for simple structures such as these are also straightforward to design. Standard multiversion concurrency-control algorithms, such as snapshot isolation [4, 7] can be easily applied. However, structures such as these are not adequate for range queries, as consecutive entries of a given version may need to be searched separately.

A better approach to storing multiversion data is to use R-trees [8, 3] as the data storage. After all, multiversion data can be seen as two-dimensional data with keys as the first dimension and version ranges as the second dimension. R-trees do not, in itself, guarantee logarithmic access times in all situations. The standard R-tree in fact requires traversing of multiple paths even when doing an exact-match query, as the key-area regions of sibling pages may overlap. However, when discussing multiversion data, it is worth noticing that

the key-version ranges of data items stored in a multiversion database cannot overlap. This means that the overlap in index pages may be reduced when storing multiversion data.

Another structure, the multiversion B$^+$-tree (MVBT) by Becker et al. [2] is an asymptotically optimal multiversion structure that guarantees logarithmic execution times for all actions and structure-modification operations in any sequences of user actions. Unfortunately, no comprehensive concurrency-control algorithms exist for the MVBT. Another efficient multiversion database, the Immortal DB prototype database by Lomet et al. [15, 16], is based on the time-split B$^+$-tree (TSBT) [17, 18]. Although Immortal DB provides comprehensive concurrency-control and recovery algorithms, the underlying TSBT structure is not optimal in all situations. Furthermore, the TSBT does not consolidate pages. This means that current-version space consumption cannot be reduced once it has reached a certain size.

It is our intention to design a multiversion index structure that (1) is optimal in the sense that all actions (on any version) should work in logarithmic time in all situations; and (2) can be used in the fully concurrent scenario with multiple read-only and updating transactions. In contrast to the original MVBT, physical deletion is required for rolling back aborted transactions that need to delete the entries inserted in their forward-rolling phase. On the other hand, the structure must also be able to merge pages to reduce current-version space consumption if enough entries are deleted from the database. Finally, we will also design concurrency-control and recovery algorithms that are based on the *de facto* standard ARIES [21] algorithms.

We have recently written a manuscript on managing multiaction transactions and concurrency control and recovery in the multiversion B$^+$-tree [9]. Our paper provides concurrency-control and recovery algorithms for a scenario where a single updating transaction can run concurrently with multiple read-only transactions. We call the extended structure the transactional multiversion B$^+$-tree (TMVBT). The TMVBT is highly concurrent for the scenario it allows: no locking is required; the single updating transaction write-latches at most five pages at a time (one parent and four sibling pages); and the read-only transactions read-latch only one page at a time (no lock-coupling required). The next step in our research is to design a multiversion structure with concurrency-control and recovery algorithms that can handle multiple concurrent updating and read-only transactions. The planned algorithms will use a separate B$^+$-tree to temporarily store the updates of active transactions. A maintenance transaction will be run frequently to move the committed updates from the separate B$^+$-tree into the TMVBT, thus keeping the B$^+$-tree size small.

To support the analysis of the designed multiversion algorithms, we will develop analytical performance results for the structures and validate them by implementing the designed algorithms for the straightforward B$^+$-tree system, for the combined TMVBT and B$^+$-tree system, for the R-tree system, and for the time-split B$^+$-tree system. The versioned B$^+$-tree and the time-split B$^+$-tree will work as comparison points when evaluating the performance of the other design ideas. We will then simulate a set of transactions on the structures, and measure the amount of required I/O operations, the number of locks and latches acquired and the space consumption for each structure.

The contributions of our planned research are: (1) we will design a fully concurrent multiversion database solution that uses our extended TMVBT as a storage for committed transactions alongside with a separate B$^+$-tree; (2) our aim is to optimize the existing R-tree concurrency-control algorithms for storing multiversion data; (3) we will acquire analytical results for the designed solutions; and (4) we will evaluate the designed solutions against existing database structures.

This paper is organized as follows. In Sec. 2, we begin by defining the multiversion database theory applied in this paper. In Sec. 3, we present the straightforward solution that uses a standard B$^+$-tree for storing multiversion data. This structure is intended to be used as a comparison point when evaluating the performance of the other structures. Next, in Sec. 4, we describe the idea of using an R-tree as a multiversion storage structure. After that, in Sec. 5 we introduce the transactional multiversion B$^+$-tree, and in Sec. 6, we outline our suggestion of using the combination of a TMVBT and a separate B$^+$-tree for building an efficient multiversion database structure. Sec. 7 describes the time-split B$^+$-tree that we are going to use as another comparison point. In Sec. 8, we describe how we are going to evaluate the different structures. Next, in Sec. 9, we describe our future research plans. Finally, in Sec. 10, we give our preliminary conclusions on the presented design ideas.

2. MULTIVERSION DATABASES

While a standard data item in a single-version database is a pair (k, w), where k is the data key and w the stored data, a multiversion data item also includes a *version range* $[v_1, v_2)$ for which it is *alive*. For each key there can be only one data item alive at any version. A data item with an unbounded version range is said to be alive at the current version of the database; or just simply alive. A data item with a bounded version range $[v_1, v_2)$ is said to be *alive at version* v for all versions $v_1 \leq v < v_2$. The version range of a data item begins when the item is inserted into the database and ends when the item is deleted from the database. Updates do not need to be explicitly defined; rather an update at version v will be modeled by ending the version range of the data item to version v and creating a new data item whose version range begins with version v. Because there are multiple ways to represent the version ranges of data items, we will not go into implementation details.

A fully concurrent multiversion database allows any number of read-only and updating transactions to operate concurrently. The read-only transactions must explicitly specify which version they want to read; however, they are only allowed to read versions that were already committed when the read-only transaction began. A global counter v_{cur} is used to track the version number of the latest committed transaction. The updating transactions, on the other hand, must always operate on the most recent version.

A *read-only transaction* may thus contain the following actions:

- **begin-read-only**: begins a new read-only transaction. This action records the latest committed version of the database $v_{begin} \leftarrow v_{cur}$.
- **query(key k, version v)**: retrieves from the database a data item (k, w) that is alive at version $v \leq v_{begin}$, if such an item exists.

- **range-query(range $[k_1, k_2)$, version v)**: retrieves the set of data items (k, w) that are alive at version $v \leq v_{begin}$ with $k_1 \leq k < k_2$.

- **commit-read-only**: commits the read-only transaction.

An updating transaction does not specify any version number in its actions. A multiversion concurrency-control algorithm must be applied to maintain a serializable view of the most recent version of the database for each updating transaction. Depending on the timestamping methodology used, it may be possible that the commit-time timestamp of updating transactions is not known during their execution. In such a scenario, *temporary* or *dummy* timestamps can be used during the execution of the updating transactions. These temporary timestamps can be based, for example, on the start-time timestamps of the transactions. The temporary timestamps need to be changed to the commit-time timestamp of the transaction at some point after the transaction has committed. This technique is called *lazy timestamping* [28, 16].

An *updating transaction* may thus contain the following actions:

- **begin-update**: begins a new updating transaction.

- **query(key k)**: retrieves the live data item (k, w), if such an item exists.

- **range-query(range $[k_1, k_2)$)**: retrieves the set of live data items (k, w) with $k_1 \leq k < k_2$.

- **insert(key k, data w)**: a forward-rolling action that is legal when the database does not contain a live data item (k, w'); this action inserts a data item (k, w) into the database with version range $[v, \infty)$. The version v is the commit-time timestamp of the updating transaction, or a temporary timestamp.

- **delete(key k)**: a forward-rolling action that is legal when the database contains a live data item (k, w). This action deletes the data item from the database by setting the end version of the item to version v. The version v is the commit-time timestamp of the updating transaction, or a temporary timestamp.

- **commit-update**: commits the updating transaction.

- **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.

3. VERSIONED B$^+$-TREE

A straightforward solution for storing multiversion data is to store the data in a B$^+$-tree which has been extended to store the version information. An advantage of this solution is that tried and tested algorithms for concurrency control, recovery and tree-structure maintenance exist. We call this solution a *versioned B$^+$-tree*. This straightforward solution stores tuples of the form $(k, [v_1, v_2), w)$. Because each key k is unique for any given version v, the version ranges of all the data items with key k cannot overlap. When inserting a data item, the end version v_2 is initially $v_2 = \infty$. Deleting a data item at version v is performed logically by replacing the tuple $(k, [v_1, \infty), w)$ with $(k, [v_1, v), w)$, $v > v_1$. In this structure, the B$^+$-tree algorithms are modified to use the combination of k and v_1 to order the data items in such a way that $(k, v_1) < (k', v_1')$ if $(k < k') \ \vee \ (k = k' \wedge v_1 < v_1')$. The situation $k = k' \wedge v_1 = v_1'$ is not possible, because version-ranges of data items with the same key cannot overlap.

Search validity for the versioned B$^+$-tree is guaranteed by using the latch-coupling protocol [20, 19] when traversing the tree. The logical state of the database as seen by different transactions is maintained with snapshot isolation [4, 7]. All operations are logged using an ARIES-based write-ahead logging protocol. Structure-modification operations are executed as separate atomic actions that transform a structurally consistent tree into another structurally consistent tree. The structure-modification operations are executed top-down to maintain the tree consistency between each operation.

The problem with this solution is that the search times may grow far too large, especially for range queries. For (single item) queries, inserts and deletes, the search time is logarithmic in the amount of entries in the B$^+$-tree. The theoretical maximum amount of entries in a versioned B$^+$-tree at version v is $O(nv)$, where n is the largest amount of entries a transaction has updated. Thus the worst case search time for the single-data-item queries is $O(\log nv)$. However, in contrast to the standard B$^+$-tree, range queries cannot benefit from the located leaf-page of the previous keys, as the data items of the next key may be far away from the previous key. Thus, in the worst case, the performance of the range query may even deteriorate to $O(k \log nv)$, where $k = k_2 - k_1$ is the size of the searched key range (separate tree traversals for each key).

4. TWO-DIMENSIONAL R-TREE

Another design idea is to use a standard R-tree as a storage for multiversion data, with data keys and version ranges used as the dimensions. A data entry with key k and version range $[v_1, v_2)$ therefore occupies a one-dimensional line segment $([k, k], [v_1, v_2))$ of the key-version space. An important property of the keys stored in the database is that the key-version ranges of different data items cannot overlap. This may allow us to optimize the R-tree algorithms by reducing the overlapping of index pages.

Using R-trees to store multiversion data has been discussed, for example, by Kolovson and Stonebraker [11]. In their paper, they present two design ideas that periodically move historical entries from the current-version R-tree into a historical R-tree that is possibly stored in a write-once medium. However, their paper does not discuss concurrency control and recovery. Concurrency control and recovery for

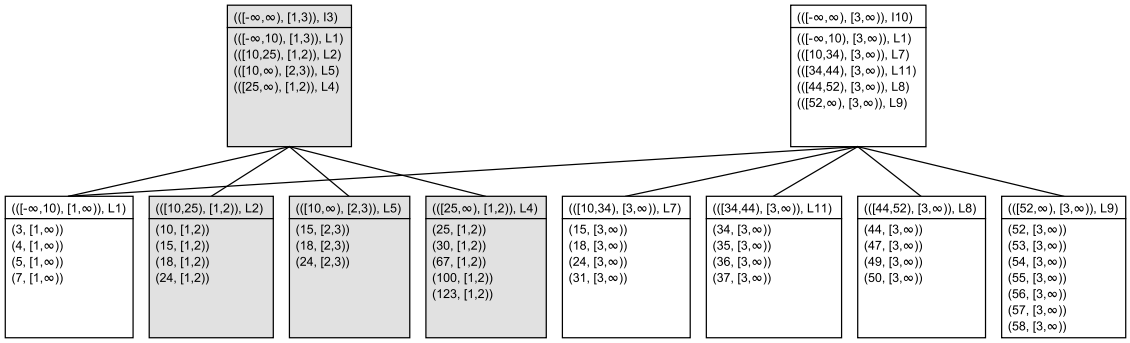

Figure 1: An example of a transactional MVBT.

R-trees has been discussed in [23, 5, 12, 13, 27]. The approaches to R-tree concurrency control in these articles can be divided into two main techniques: lock-coupling-based algorithms [23, 5] and link-based algorithms [12, 13, 27].

The R-tree *lock-coupling-based* concurrency-control algorithms are based on a concept that is similar to the standard latch-coupling method used in B^+-trees. Because the children of an R-tree index page may overlap, the search may need to go through multiple subtrees to find the proper leaf page. Thus all the parent pages in the search path need to be kept locked until all their children have been examined. To obtain a better level of concurrency, the search is performed level-by-level using breadth-first search (instead of depth-first search) to keep the parents locked for as short a time as possible. Recovery in R-trees has been described by Chen and Huang [5]; they report that they use a recovery strategy that is based on the write-ahead-logging (WAL) protocol. The problem with the lock-coupling-based concurrency control is that entire subtrees are held locked while the search is still going on.

The *link-based* concurrency-control schemes, on the other hand, are based on an idea similar to concurrency-control schemes in B-link-trees [14]. The original B-link-tree concurrency-control schemes allow for high concurrency because searchers do not need to keep multiple pages locked simultaneously. Rather, searchers notice split pages from changed key ranges and traverse *right links* to find the keys that were moved away from the split page. However, in R-trees, the (n-dimensional) keys cannot be similarly ordered and thus the link-based R-tree algorithms use a logical sequence number [12], or node sequence number [13, 27] based method to determine whether a child page has been changed after the parent page was released. The problem with this approach (as with the B-link-tree approach) is that empty-page deletion can only be performed when there are no searchers that could have stored links to these pages. Empty-page deletion is dealt with either by using signaling locks [13] or by marking the pages with a *generation counter* [12], but the extra effort required makes the link-based algorithms more complicated than the lock-coupling algorithms.

For concurrency control, Kornacker et al. [13] propose a hybrid locking mechanism which uses two-phase locking for existing entries and applies predicate locking at page level to avoid phantom insertions. The searchers attach the search predicate to all the pages they traverse. An insert operation thus needs to only check the predicates at the leaf page. However, predicate locking is very expensive compared to page locking, because all the predicates must be checked. Recovery of the link-based algorithms is based on write-ahead logging and separating the update operations into content-changing operations (insert, delete) and structure-modification operations (page split, page delete) [12, 13, 27]. The structure-modification operations are logged as separate atomic actions that transform a structurally consistent tree into another structurally consistent tree.

The current R-tree concurrency-control and recovery algorithms have many drawbacks. The lock-coupling-based algorithms are straightforward enough, but they may keep a large portion of the R-tree locked, thus reducing concurrency. The link-based algorithms seem to offer more concurrency, but physical record deletion, empty-page deletion and phantom avoidance are still very complicated. Our plan is to design ARIES-based concurrency-control and recovery algorithms for a standard R-tree structure that is used as a multiversion data storage. We hope to design straightforward and elegant algorithms that take into account the fact that the stored data is line segments instead of rectangles.

5. TRANSACTIONAL MVBT

The transactional multiversion B^+-tree (TMVBT) [9] is our extension to the asymptotically optimal multiversion B^+-tree (MVBT) by Becker et al. [2]. The multiversion B^+-tree stores tuples of the form $(k, [v_1, v_2), w)$, where w is either the data item (or pointer; in leaf pages), or a child page pointer (in index pages). The pages of a given level of the MVBT cover disjoint rectangular regions of the key-version space. The basic operation in the MVBT is the *version-split operation*, which splits a page at the latest version. Page data entries (data items or child page pointers) whose version range extends over the split version are duplicated in the two pages. Thus, if an index page splits, the child page pointers are duplicated and the child pages will have multiple parents. Key splits and page merges in the original MVBT are possible only directly after a version split.

While the original MVBT by Becker et al. [2] assumes a single-update transaction model, in which the database version number must change between each update, our transactional MVBT allows transactions to contain multiple updates. Key splits and page merges are also allowed without version splits for pages that have been created earlier by the same transaction. We have also designed a simple concurrency-control scheme that allows a single updat-

ing transaction to run concurrently with multiple read-only transactions. Our concurrency-control and recovery algorithms apply write-ahead logging and assume the standard steal-and-no-force buffering policy. The extended algorithms also allow the active transaction to physically roll back, leaving no trace of the intermediate updates performed on the structure. Recovery for the structure is maintained by ARIES-based logging of the actions and structure-modification operations. Each structure-modification operation is executed as a separate atomic action that is logged with a single redo-only log record [10] that transforms a structurally consistent TMVBT into another structurally consistent TMVBT. The user actions are logged with physiological redo-undo log records.

Our algorithms retain all the asymptotically optimal time-complexity guarantees of the original MVBT algorithms. The original structure (and thus also the TMVBT) guarantees access times that are logarithmic in the number of entries of any given version, for all versions. More specifically, the running times for all actions (query, range query, insert and delete) for any version v have an execution time of $O(\log m_v)$, where m_v is the amount of entries of version v in the database. In a standard B$^+$-tree, the logarithmic base is usually $B/2$, where B is the page capacity. In the MVBT, the logarithmic base is around $B/5$.

An example of a structurally consistent transactional multiversion B$^+$-tree is given in Fig. 1. The white pages in the image are live pages, and the dark ones are dead. The example has been generated by our visualization software with the following action sequence:

- Transaction 1: insert data items with keys 10, 5, 7, 15, 123, 3, 4, 18, 24, 25, 30, 67, and 100.
- Transaction 2: delete data items with keys 100, 25, 30, 67, 123, and 10.
- Transaction 3: insert data items with keys 31, 44, 47, 49, 50, 52–58, 34, 36, 35, and 37.

We believe that it is not possible to further extend the transactional multiversion B$^+$-tree to support full concurrency without losing some of the structure's logarithmic guarantees. If multiple concurrent active transactions are allowed to insert different versions of entries to pages, this may even lead to a situation where it is impossible to split the page.

6. TMVBT WITH VERSIONED B$^+$-TREE

As the transactional MVBT only allows a single updating transaction to run at a time, we propose a setting in which the TMVBT is used as a storage for committed historical transactions. In this setting, a small separate versioned B$^+$-tree is used to store the updates of active transactions. The updates of committed transactions can then be moved from the versioned B$^+$-tree into the transactional MVBT by a periodically-run maintenance transaction in commit order. This does compromise on the logarithmic execution-time-guarantees of tree traversal for the active transactions. However, as it is assumed that the number of active transactions is not too large, the versioned B$^+$-tree will stay small, and the asymptotically worse access times of the B$^+$-tree will not affect the overall performance of the structure significantly. For bulk updates, it may be more efficient to apply them directly to the TMVBT, without going through the versioned B$^+$-tree. This can be done off-line, or by blocking access for updating transactions for the duration of the bulk update.

The basic idea of this setting is that there is a historical version counter v_{his} that determines which transaction updates have been moved to the transactional MVBT. The updates of all transactions with version number $v \leq v_{his}$ are fully stored in the TMVBT. Read-only transactions that wish to read a version v from the database operate by the following rules: (1) if $v \leq v_{his}$, the transaction can directly fetch the data item from the TMVBT; and (2) if $v > v_{his}$, the transaction may need to check both the versioned B$^+$-tree and the TMVBT to retrieve the correct version of the data item. In the latter case, the read-only transaction must first search the versioned B$^+$-tree to see if an update has been made to the data item by a committed but not-yet-moved transaction with timestamp $v_t \leq v$. If no update entry was found, the transaction must read the most recent historical version from the TMVBT. Updating transactions must similarly go through both the versioned B$^+$-tree and the TMVBT to search for the latest version of the data item. Range queries always need to search through both structures at the same time. When progressing to the next key, both of the structures need to be searched to determine the smallest next key. The standard *saved path* concept can be used to reduce the amount of I/O operations required. However, because there are two separate structures, the transaction processes or threads need to maintain two saved path variables; one for the TMVBT and one for the versioned B$^+$-tree.

The logical state of the latest versions is maintained by using a standard multiversion concurrency-control protocol, such as snapshot isolation [4, 7]. A major benefit in this setting is that it is very easy to apply lazy timestamping [28, 16]. The updating transactions can use temporary timestamps, which are stored on the versioned B$^+$-tree. A mapping from temporary timestamps to commit-time timestamps is maintained in a simple structure called the transaction-timestamp table, or TTT. There is no need to revisit the entries in the versioned B$^+$-tree when the transaction first commits. Instead, when the maintenance transaction moves the updates of a committed transaction from the versioned B$^+$-tree into the TMVBT, it can change the temporary timestamps of the updates into the commit-time timestamps. When the updates of the transaction have been moved, the mapping for this transaction is removed from the TTT.

When the maintenance transaction is run, another counter v_{mov} is set to $v_{mov} \leftarrow v_{his} + 1$. This indicates that the maintenance transaction is in progress and is copying updates of the committed transaction with version $v = v_{mov}$ from the B$^+$-tree into the TMVBT. During this phase, the active transactions may read versions $v < v_{mov}$ from the TMVBT, and transactions with version $v \geq v_{mov}$ from the versioned B$^+$-tree. Note that this definition does not conflict with the earlier definition of the usage of the historical counter v_{his}. After the entries have been copied, the history version counter v_{his} is incremented to $v_{his} \leftarrow v_{mov}$. At this point, transactions can read the committed version v_{mov} from the TMVBT. The next step in the maintenance transaction is to remove the copied updates of the transaction with version $v = v_{mov}$ from the versioned B$^+$-tree. After the deletion is complete, the maintenance transaction commits.

Recovery in this database organization is a combination of the recovery algorithms of the versioned B$^+$-tree (see Sec. 3)

and transactional MVBT (see Sec. 5). The actions of maintenance transactions (i.e., all the actions performed on the TMVBT) can be logged with redo-only log records, as all the data required for completing the maintenance transaction are always available. This means, however, that the standard ARIES-based recovery algorithms would need to be modified. Another approach is to simply log the actions with standard redo-undo log records and roll back the maintenance transaction if a system crash occurs.

An unsolved issue in the moved-transaction-version processing still remains. When the maintenance transaction has copied the updates of a transaction T with version $v = v_{mov}$, the active transactions need to be somehow notified that the correct place to search for version v_{mov} is now the TMVBT. It would be infeasible for the active transactions to check the version counter v_{his} (which has now been set to v_{mov}) between each single operation. One possible solution could be to track the count of active transactions that have read v_{his} and are using the information to read the structures. When the system maintenance transaction has copied the entries, it would then wait until the count of transactions that are using the old version number reaches zero.

The idea of using a separate B$^+$-tree for storing the updates of active transactions is closely related to the idea of using *differential indices* [25] or a *side file* [22]. The differential indices are used to group together the updates of several transactions in order to insert them more efficiently into the main index. The side file is used to store the updates of ongoing transactions for the duration of an index-building operation. A crucial difference is that due to the properties of the TMVBT, we have to move the updates of transactions one transaction at a time in order to maintain the logarithmic query-time bounds of the TMVBT tree-traversal for all versions. However, we can still benefit from the fact that records with consecutive keys updated by the same transaction will be moved to the TMVBT in ascending order. The maintenance transaction can therefore speed up the copying process by keeping track of the traversed path via the saved path variable. In the optimal situation, many consecutive updates will thus be applied directly to the same page.

7. TIME-SPLIT B$^+$-TREE

The time-split B$^+$-tree (TSBT) by Lomet et al. [17, 18, 15, 16] is a multiversion structure that stores the multiversion data in a way similar to the multiversion B$^+$-tree. The structure does not impose strict restrictions on the number of live data items for each version. Thus, both time and key splits are always possible for any page. Another benefit of the time-split B$^+$-tree is that the historical data is moved when a time-split is triggered. That is, when page p is split, the old data is moved to a new page p', and p continues to be the storage for current data. In contrast, in the multiversion B$^+$-tree, the old data stays in the split page p, and transactions use the new page p' for further operations. Moving the old page is possible as only historical pages may have multiple parents, and thus the single parent can be updated. This allows historical data to be migrated to a slower, possibly write-once storage medium.

There are, however, some drawbacks in the TSBT. For example, the page splitting algorithms do not guarantee that the pages contain any specific amount of entries alive at any given version. Furthermore, the pages of a TSBT are never merged or deleted. Thus, the tree-traversal time for the current version of the database cannot decrease, even if all the entries are deleted from the database. For these reasons, we believe that the multiversion B$^+$-tree can be more efficient for a range of applications.

However, because the structure does not impose entry-count restrictions for different versions, the concurrency-control and recovery algorithms are simpler. It is sufficient to use a standard multiversion concurrency-control protocol (such as snapshot isolation), and to apply the updates directly to the TSBT. If a transaction rolls back, an inserted entry can be physically removed from the TSBT, and a deleted entry can be reinserted to the page. This is not possible in the MVBT in all situations, as the physical deletion may reduce the number of live entries in a certain page below acceptable limits.

8. EVALUATION OF RESULTS

We will evaluate our concurrent and recoverable algorithms by comparing them to existing alternative algorithms. We will first develop analytical results for the suggested design ideas, and then validate the obtained results by simulation. The original multiversion B$^+$-tree, for example, guarantees that at least a configurable number of user actions must be performed before a structure-modification operations is required. Our extended algorithms maintain these guarantees. We can thus calculate the maximum amount of operations required for a given set of user actions in a multiversion B$^+$-tree of a given height. From this it is also possible to determine the maximum growth of the MVBT for a given set of actions. We can also calculate these values for the other structures. For tree traversals, the multiversion B$^+$-tree guarantees access times that are logarithmic in the number of live entries for the queried version. None of the other structures can guarantee this for all versions.

We will also design and implement a simulator to simulate the different multiversion structures and concurrency-control and recovery algorithms to validate the obtained analytical results. We will generate sets of random data that will be inserted, deleted and read from the database structures concurrently. To determine the effects of different key distributions, we will generate both data with uniformly distributed random keys and keys distributed according to the Gaussian distribution. We will allow several transactions to run concurrently. Also, we will perform several tests that have large bulk insertions in them to stress-test the combined TMVBT and B$^+$-tree system.

The values we will measure are (1) the number of I/O operations required for a single operation; (2) the number of I/O required for a transaction; (3) the size of the database (in pages); and (4) the number of exclusive locks and (5) latches held at the same time during a transaction. Running time of transactions can also be measured, but we expect that the number of I/O operations required is the most important factor in the running time.

Based on the results of the experimental analysis, we can determine how far the straightforward versioned B$^+$-tree can be used. We expect that at some point the performance of range queries will deteriorate, and the more complex structures will be far superior. The optimality of the MVBT structure suggests that the transactional MVBT with versioned B$^+$-tree should perform reasonably well. However, the simpler organization of an R-tree structure may well compete with the combination of two separate structures.

We expect that the time-split B^+-tree will have results similar to the MVBT structure. However, because the MVBT is optimal, we expect it to perform better with a wider range of transactions.

9. FUTURE WORK

Our research interest is currently focused on designing concurrency-control and ARIES-based recovery algorithms for various database structures. In this paper, we have outlined the ongoing research for the dissertation of the first author. Our plan for the future is to research multidimensional structures in general, to be used for either storing multidimensional data or multiversion data. Our main focus is in providing asymptotically optimal structures and algorithms that can be used as general tools for multiple purposes.

10. CONCLUSIONS

Our research concentrates on designing, implementing and evaluating a near-optimal and fully concurrent multiversion database structure. The current database structures and algorithms all seem to have some drawbacks which cause the performance of the structure to deteriorate in some situations. We have initiated a research that is aimed at finding a near-optimal database organization for storing multiversion data. Our initial findings suggest that such a structure can be constructed, at least if we relax the logarithmic-time-bound requirements for the active transactions. Querying for historical versions is guaranteed to be logarithmic for the concurrent transactional MVBT structure. This structure is therefore optimal for applications that most often read historical data. For applications that mostly access current data, the time-split B^+-tree and the multiversion B^+-tree should have performance ratings that are close to each other for most situations.

11. ACKNOWLEDGMENTS

This work has been funded by the Academy of Finland.

12. COPYRIGHT

13. REFERENCES

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

[2] 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.

[3] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331. ACM New York, NY, USA, 1990.

[4] 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.

[5] J. Chen, Y. Huang, and Y. Chin. A study of concurrent operations on R-trees. *Information Sciences*, 98(1):263–300, 1997.

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

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

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

[9] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen. Transactions on the Multiversion B^+-tree. To be submitted for publication.

[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] C. Kolovson and M. Stonebraker. Indexing techniques for historical databases. In *Data Engineering, 1989. Proceedings. Fifth International Conference on*, pages 127–137, 1989.

[12] M. Kornacker and D. Banks. High-concurrency locking in R-trees. In *Proceedings of VLDB*, pages 134–145, 1995.

[13] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and recovery in generalized search trees. *ACM SIGMOD Record*, 26(2):62–72, 1997.

[14] P. Lehman and B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)*, 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, ICDE 2006*, pages 35–46, 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] D. Lomet and B. Salzberg. Access method concurrency with recovery. *ACM SIGMOD Record*, 21(2):351–360, 1992.

[20] 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*, pages 371–380. ACM Press New York, NY, USA, 1992.

[21] 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.

[22] 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.

[23] V. Ng and T. Kameda. Concurrent accesses to R-trees. In *Proceedings of the Third International Symposium on Advances in Spatial Databases*, pages 142–161. Springer-Verlag London, UK, 1993.

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

[25] 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.

[26] 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.

[27] S. Song, Y. Kim, and J. Yoo. An enhanced concurrency control scheme for multidimensional index structures. *IEEE Transactions on Knowledge and Data Engineering*, pages 97–111, 2004.

[28] K. Torp, C. Jensen, and R. Snodgrass. Effective timestamping in databases. *The VLDB Journal—The International Journal on Very Large Data Bases*, 8(3-4):267–288, 2000.