The VLDB Journal
© Springer-Verlag 1996

An asymptotically optimal multiversion B-tree

Bruno Becker[2], **Stephan Gschwind**[2], **Thomas Ohler**[2], **Bernhard Seeger**[3], **Peter Widmayer**[1]

[1] Institut für Theoretische Informatik, ETH Zentrum, CH-8092 Zürich, Switzerland
Tel. ++41-1-63-27400, Fax ++41-1-63-21172, email: widmayer@inf.ethz.ch
[2] isys software gmbh, Ensisheimer Str. 2a, D–79110 Freiburg, Germany
[3] Philipps-Universität Marburg, Fachbereich Mathematik, Fachgebiet Informatik, Hans-Meerwein-Strasse, D–35032 Marburg, Germany

Abstract. In a variety of applications, we need to keep track of the development of a data set over time. For maintaining and querying these multiversion data efficiently, external storage structures are an absolute necessity. We propose a multiversion B-tree that supports insertions and deletions of data items at the current version and range queries and exact match queries for any version, current or past. Our multiversion B-tree is asymptotically optimal in the sense that the time and space bounds are asymptotically the same as those of the (single-version) B-tree in the worst case. The technique we present for transforming a (single-version) B-tree into a multiversion B-tree is quite general: it applies to a number of hierarchical external access structures with certain properties directly, and it can be modified for others.

Key words: Information systems – Physical design – Access methods – Versioned data

1 Introduction

The importance of not only maintaining data in their latest version, but also keeping track of their development over time, has been widely recognized (Tansel et al. 1993). Version data in engineering databases (Katz 1990) and time-oriented data (Clifford and Ariav 1986) are two prime examples for situations in which the concepts of versions and time are visible to the user. In multiversion concurrency control (Barghouti and Kaiser 1991; Bernstein et al. 1987), these concepts are transparent to the user, but they are used by the system (e.g. the scheduler) for concurrency control and recovery purposes. In this paper, we are concerned with access structures that support version-based operations on external storage efficiently. We follow the convention of Bernstein et al. (1987) and Driscoll et al. (1989) in that each update to the data creates a new version; note that this differs from the terminology in engineering databases, where an explicit operation exists for creating versions, and versions of design objects are equipped with semantic properties and mechanisms, such as inheritance or change propagation. Our choice of creating a new version after each update turns out not to be restrictive, in the sense that the data-structuring method we propose can be easily adapted to create versions only on request, without loss of efficiency.

We are interested in *asymptotically worst-case* efficient access structures for *external storage* that support at least *insertions*, *deletions*, *exact-match queries* (associative search) – the *dictionary operations* (Sedgewick 1988; Mehlhorn and Tsakalidis 1990; Gonnet and Baeza-Yates 1991) – and *range queries* in addition to application-specific operations such as *purging* of old enough versions in concurrency control. That is, we aim at a theoretical understanding of the fundamentals of multiversion access to data, with little attention to constant factors [studies with this flavor have attracted interest in other areas, too (Kanellakis et al. 1993, Vitter 1991)]. We limit our discussion to the situation in which a change can only be applied to the current version, whereas queries can be performed on any version, current or past. Some authors call this a management problem for *partially persistent* data; we call an access structure that supports the required operations efficiently a *multiversion* structure.

The problem in designing a multiversion access structure lies in the fact that data are on external storage. For *main memory*, there is a recipe for designing a multiversion structure, given a single-version structure. More precisely, any single-version main memory data structure in a very general class, based on pointers from record to record, can be transformed into a multiversion structure, with no change in the *amortized* asymptotic worst-case time and space costs, by applying a general technique (Driscoll et al. 1989). For the special case of balanced binary search trees, this efficiency is achieved even in the worst case per operation – clearly a perfect result.

Given quite a general recipe for transforming single-version main memory data structures into multiversion structures, it is an obvious temptation to apply that recipe accordingly to external access structures. This can be done by simply viewing a block in the external structure as a record in the main memory structure. At first glance, this models block access operations well; unfortunately, it does not model storage space appropriately, in that the size of a block is not taken into consideration. That is, a block is

Correspondence to: P. Widmayer

viewed to store a constant number of data items, and the constant is of no concern. Even worse, the direct application of the recipe consumes one block of storage space for each data item. However, no external data structure can ever be satisfactory unless it stores significantly more than one data item in a block on average; balanced structures, such as the B-tree variants, actually require to store in each block at least some constant fraction of the number of items the block can hold (the latter being called the block capacity b). As a consequence, the space efficiency of this approach is clearly unacceptable, and this also entails an unacceptable time complexity.

It is the contribution of this paper[1] to propose a technique for transforming single-version *external* access structures into multiversion structures, at the cost of a constant factor in time and space requirements, where the block capacity b is *not* considered to be a constant. That is, the asymptotic bounds for the worst case remain the same as for the corresponding single-version structure, but the involved constants change. We call such a multiversion structure *asymptotically optimal*, because the asymptotic worst-case bounds certainly cannot decrease by adding multiversion capabilities to a data structure. Our result holds for a certain class of hierarchical external access structures. It is worth noting that this class contains the B-tree and its variants, not only because the B-tree is an ubiquitous external data structure, but also because an asymptotically optimal multiversion B-tree has not been obtained so far, despite the considerable interest this problem has received in the literature. Since we are interested primarily in the asymptotic efficiency, we will discuss the involved constants only later in the paper. Multiversion structures with excellent asymptotic worst-case bounds for insert and exact-match operations (but *not* for delete) and for related problems have been obtained previously; we will discuss them in some detail later in the paper.

For the sake of concreteness, we base the presentation of our technique in this paper on B-trees; it is implicit how to apply our technique to other hierarchical structures. Each data item stored in the tree consists of a *key* and an *information* part; access to data items is by key only, and the keys are supposed to be taken from some linearly ordered set. Let us restrict our presentation to the following operations:

- *Insert (key,info)*: insert a record with given *key* and *info* component into the *current* version; this operation creates a new version.
- *Delete (key)*: delete the (unique) record with given *key* from the *current* version; this operation creates a new version.
- *Exact-match query (key,version)*: return the (unique) record with given *key* in the given *version*; this operation does not create a new version.
- *Range query (lowkey,highkey,version)*: return all records whose key lies between the given *lowkey* and the given *highkey* in the given *version*; this operation does not create a new version.

Before briefly reviewing the previous approaches of designing a B-tree that supports these operations efficiently, let us state the strongest efficiency requirements that a multiversion B-tree can be expected to satisfy. To this end, consider a sequence of N update operations (insert or delete), applied to the initially empty structure, and let m_i be the number of data items present after the i-th update (we say, in version i), $0 \leq i \leq N$. Then a multiversion B-tree with the following properties holding for each i (all bounds are for the worst case) is the best we can expect:

- For the first i versions, altogether the tree requires $O(i/b)$ blocks of storage space.
- The $(i+1)$-th update (insertion or deletion) accesses and modifies $O(\log_b m_i)$ blocks.
- An exact-match query in version i accesses $O(\log_b m_i)$ blocks.
- A range query in version i that returns r records accesses $O(\log_b m_i + r/b)$ blocks.

The reason why these are lower bounds is the following. For a query to any version i, the required efficiency is the same as if the data present in version i were maintained separately in their own B-tree. For insertions and deletions on the current version, the required efficiency is the same as for a (single-version) B-tree maintaining the data set valid for the current version. In other words, a better multiversion B-tree would immediately yield a better B-tree.

This paper presents a multiversion B-tree structure satisfying these efficiency requirements, under the assumption that in a query, access to the root of the requested B-tree has only constant cost [we could even tolerate a cost of $O(\log_b m_i)$, to be asymptotically precise]. We have thus separated the concerns of, first, identifying the requested version, and, second, querying the requested version (that is, the root of the appropriate B-tree). This separation of concerns makes sense because in an application of a multiversion structure, access to the requested version may be supported from the context, such as in concurrency control. For instance, the block address of the requested root block may directly be known (possibly from previous accesses) or only a constant number of versions might be relevant for queries, such that the root block can be accessed in time $O(1)$. This assumption has been made in other papers (Driscoll et al. 1989; Lanka and Mays 1991), allowing the investigation to concentrate on querying within a version. In this paper, we follow this view and try to take advantage of a possibly direct version access for querying a version. We therefore concern ourselves with ways to identify the requested version only later, with little emphasis, since any of a number of search techniques can be applied for this purpose. Note that if we do not separate these issues, but instead assume that the root of the requested B-tree needs to be identified through a search operation, $\Omega(\log_b N)$ instead of $\Omega(\log_b m_i)$ is a lower bound on the run-time of a query, since one item out of as many as N items needs to be found.

In building multiversion structures, there is a general tradeoff between storage space, update time and query time. For instance, building an extra copy of the structure at each update is extremely slow for updates and extremely costly in space, but extremely fast for queries. Near the other extreme, Kolovson and Stonebraker (1989) view versions (time) as an extra dimension and store one-dimensional version intervals in two-dimensional space in an R-tree. As a consequence of

[1] A preliminary version of this paper has been published (Becker et al. 1993).

using an R-tree, they can also maintain one-dimensional key intervals (and not only single keys). This gives good storage space efficiency, but query efficiency need not be as good, because the R-tree gives no guarantee on selectivity. That is, even if access to version i is taken care of in the context, the time to answer a query on version i does not depend on the number of items in that version only, but instead on the total number of all updates. We will discuss other multiversion B-trees suggested in the literature in Sect. 5; none of them achieves asymptotically optimal performance in time and space.

In Sect. 2, we present an optimal multiversion B-tree. Our description suggests a rather general method for transforming hierarchical external data structures into optimal multiversion structures, provided that operations proceed in a certain way along paths between the root and the leaves. But even if the external single version data structure does not precisely follow the operation pattern we request (as in the case of R-trees, for instance), we conjecture that the basic ideas carry over to an extent that makes a corresponding multiversion structure competitive and useful. Section 3 provides an efficiency analysis of our multiversion B-tree, and Sect. 4 adds some thoughts around the main result. Section 5 puts the obtained result into perspective, by comparing it with previous work, and Sect. 6 concludes the paper.

2 An optimal multiversion B-tree

We present our technique to transform single-version external access structures into multiversion structures using the example of the leaf-oriented B-tree.

2.1 The basic idea

To achieve the desired behavior, we associate insertion and deletion versions with items, since items of different lifespans need to be stored in the same block. Let $<$ *key, in_version, del_version, info* $>$ denote a data item, stored in a leaf, with a *key* that is unique for any given version, an associated *info*rmation, and a lifespan from its insertion version *in_version* to its deletion version *del_version*. Similarly, an entry in an inner node of the tree is denoted by $<$ *router, in_version, del_version, reference* $>$; the *router*, together with the *in_version* and *del_version* information on the *referenced* subtree, guides the search for a data item. For example, the B-tree uses a separator key and the R-tree uses a rectangle as a router.

From a bird's eye view, the multiversion B-tree is a directed acyclic graph of B-tree nodes that results from certain incremental changes to an initial B-tree. In particular, the multiversion B-tree embeds a number of B-trees; it has a number of B-tree root nodes that partition the versions from the first to the current one in such a way that each B-tree root stands for an interval of versions. A query for a given version can then be answered by entering the multiversion B-tree at the corresponding root.

Each update (insert or delete operation) creates a new version; the i-th update creates version i. An entry is said to be of version i if its lifespan contains i. A block is said to be *live* if it has not been copied, and *dead* otherwise. In a live block, deletion version $*$ for an entry denotes that the entry has not yet been deleted at present; in a dead block, it indicates that the entry has not been deleted before the block died. For each version i and each block A except the roots of versions, we require that the number of entries of version i in block A is either zero or at least d, where $b = k \cdot d$ for block capacity b and some constant k (assume for simplicity that b, k, d are all integers and b is the same for directory and data blocks); we call this the *weak version condition*.

Operations that do not entail structural changes are performed in the straightforward way that can be inferred from the single-version structure by taking the lifespan of entries into account. That is, an entry inserted by update operation i into a block carries a lifespan of $[i, *)$ at the time of insertion; deletion of an entry by update operation i from a block changes its *del_version* from $*$ to i.

Structural changes are triggered in two ways. First, a *block overflow* occurs as the result of an insertion of an entry into a block that already contains b entries. A block underflow, as in B–trees, for example, cannot occur, since entries are never removed from blocks. However, the weak version condition may be violated in a non-root block as a result of a deletion; such a *weak version underflow* occurs if an entry is deleted in a block with exactly d current entries. Moreover, we say that a weak version underflow occurs in the root of the present version if there is only one live entry (except for the pathological case in which the tree contains only one record in the present version).

The structural modification after a block overflow copies the block and removes all but the current entries from the copy. We call this operation a *version split*; it is comparable to a time split at the current time in Lomet and Salzberg (1989); equivalently, it may be compared to the node-copying operation of Driscoll et al. (1989). In general, a copy produced by this version split may be an almost full block. In that case, a few subsequent insertions would again trigger a version split, resulting in a space cost of $\Theta(1)$ block per insertion. To avoid this and the similar phenomenon of an almost empty block, we request that immediately after a version split, at least $\varepsilon \cdot d + 1$ insert operations or delete operations are necessary to arrive at the next block overflow or version underflow in that block, for some constant ε to be defined more precisely in the next section (assume for simplicity that $\varepsilon \cdot d$ is integer). As a consequence, the number of current entries after a version split must be in the range from $(1 + \varepsilon) \cdot d$ to $(k - \varepsilon) \cdot d$; we call this the *strong version condition*. If a version split leads to less than $(1+\varepsilon) \cdot d$ entries – we say: a *strong version underflow* occurs – a merge is attempted with a copy of a sibling block containing only its current entries. If necessary, this merge must be followed by a version-independent split according to the key values of the items in the block – a *key split*. Similarly, if a version split leads to more than $(k-\varepsilon) \cdot d$ entries in a block – we say: a *strong version overflow* occurs – a key split is performed.

2.2 An example

To illustrate the basic ideas described above, let us discuss the following example of a multiversion B-tree that orga-

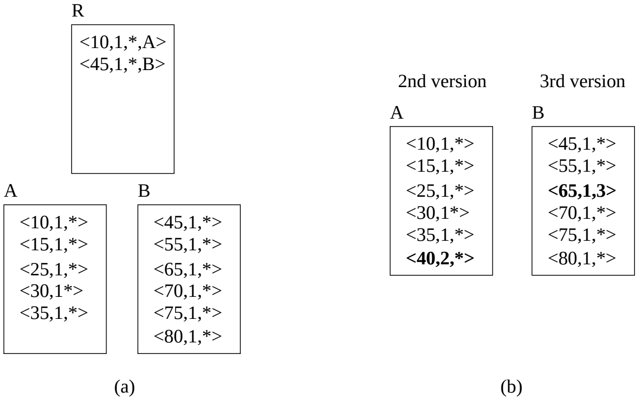

Fig. 1. Development of the multiversion B-tree up to the third version

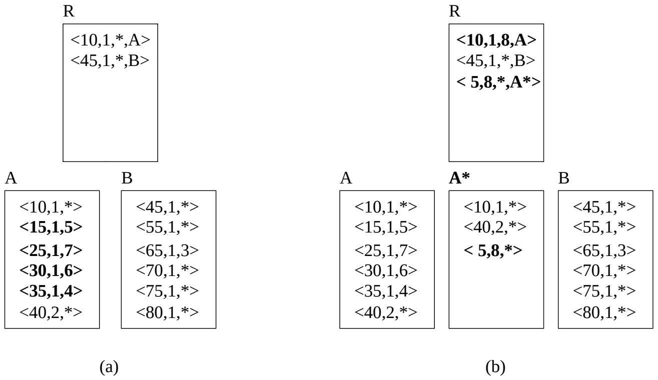

Fig. 2. a The seventh version of the multiversion B-tree; **b** the multiversion B-tree after version split of block A

nizes records with an integer key. The initial situation (i.e. first version) of our multiversion B-tree is given in Fig. 1a.

For the sake of simplicity of our example, we assume that already 11 data records are in the first version. The multiversion B-tree consists of three blocks: a root R and two leaves A and B. The parameters of the multiversion B-tree are set up in the following way: b = 6, d = 2, and ε = 0.5. Hence, after a structural change, a new block contains at least three and at most five current entries.

The second version is created by the operation *insert*(40), adding a new entry to block A. In Figure 1b, for the second and the third version, the result of the corresponding update operation is shown by depicting the block which has been modified. The next operation *delete*(65) creates the third version. As shown in Fig. 1b, for the deletion of a record, the deletion version of the corresponding entry is set to the current version, overwriting the $*$ marker.

To be able to illustrate different underflow and overflow situations, let us assume further updates – *delete*(35), *delete*(15), *delete*(30) and *delete*(25) – resulting in the seventh version of the multiversion B-tree (Fig. 2a).

Now, let us consider two different cases for creating the eighth version of the multiversion B-tree, illustrating the various types of structural changes.

In the first case, we consider the operation *insert*(5) to create the eighth version of the multiversion B-tree. This results in a block overflow of block A that is eliminated by performing a version split on that block. All current entries of block A are now copied into a new live block A*. Because block A* fulfills the strong version condition, no further restructuring is needed. Eventually, the parent block R is updated accordingly (Fig. 2b).

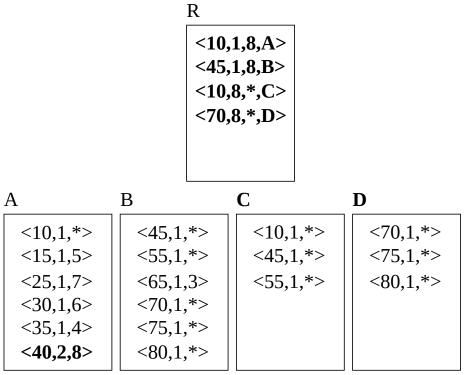

Fig. 3. Structural changes after weak version underflow of block A

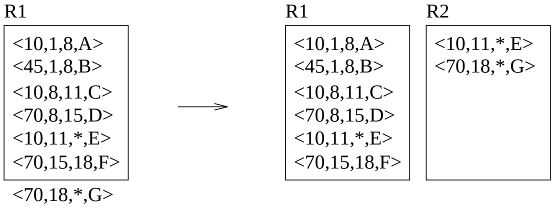

Fig. 4. Creation of two roots $R1$, $R2$ by version split of root block $R1$

In the second case, the eighth version is created by operation *delete*(40), which leads to a weak version underflow, i.e. the number of current entries in block A is less than d (=2). Then, a version split is performed on block A, copying the current entries of block A into a new block A*. Now a strong version underflow occurs in A*, which is treated by merging this block with a block resulting from version split of a sibling block. In our example, B is found to be a sibling. Accordingly, by version split a temporary block B* is created from B and blocks A* and B* are merged. As in our example, a block resulting from a merge can violate the strong version condition. To treat the strong version overflow, a key split is performed, creating two new blocks C and D. Because a key split is always balanced for a B-tree, blocks C and D fulfill the strong version condition. Eventually, the parent block R has to be updated by overwriting the $*$ of the entries which refer to block A and B and inserting two new current entries, referring to blocks C and D (Fig. 3). Now, blocks A and B are dead and blocks C and D are live.

Now let us consider an exact match query in the multiversion B-tree of Fig. 3. A record with key 25 is requested in version 5. First, the root of version 5 is accessed; in our example this is block R. We consider only the entries in the root that belong to version 5. Among these entries we choose the one whose separator key is the greatest key lower than the search key 25 and follow the corresponding reference to the next block. In our example, the search is directed to block A. Eventually, the desired entry $<$25,1,7$>$ is found in block A.

As mentioned before, our multiversion B-tree is not a tree, but a directed acyclic graph. In general, several root

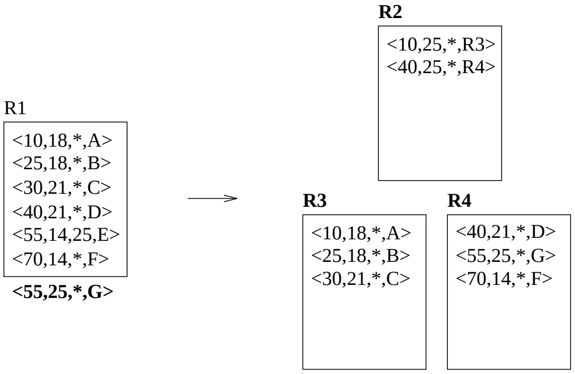

Fig. 5. Key split after strong version overflow of root block $R1$

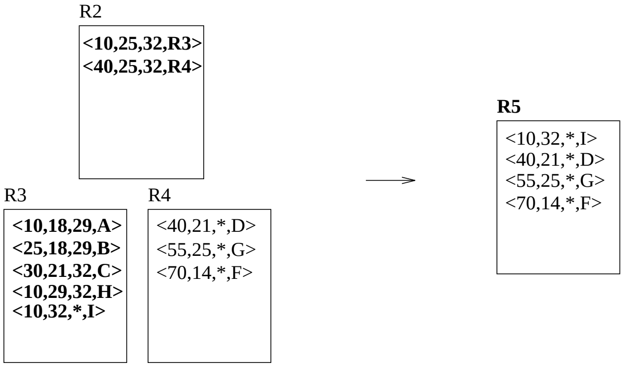

Fig. 6. Weak version underflow of root block $R2$

blocks may exist. This and the effect of structural changes in root blocks is illustrated in Figs. 4–6.

Figure 4 shows an overfull root block $R1$ and the two new roots $R1$, $R2$ resulting from version split of block $R1$. Block $R2$ is the root of the current version, version 18, whereas block $R1$ is the root of versions 1–17. References to roots $R1$ and $R2$ can be stored in an appropriate data structure, supporting access to the root blocks over versions.

Figure 5 illustrates the case that after the version split a strong version overflow occurs and a key split becomes necessary. In this case, a new root block ($R2$) is allocated, which stores entries referring to the two blocks $R3$ and $R4$ resulting from key split of the copy of root $R1$. By that, the height of the subtree valid for the current version, version 25, has grown.

Figure 6 shows the shrinking of a subtree. By several data block merges, the number of current entries in $R3$ has shrunk, a weak version underflow occurred. To handle this underflow, block copies of $R3$ and $R4$ are created and merged into a block $R5$. Since this causes a weak version underflow of block $R2$, $R5$ becomes the new root block valid for the current version.

2.3 The multiversion operations in detail

To make these restructuring operations more precise, let us now present the main points in a semi-formal algorithmic notation. In order to present the main idea without obstructing irrelevant details, we assume that an exact-match query in the single-version structure returns a block in which the searched item is stored if it is present in the structure. For the same reason, we ignore the treatment of the end of the recursion in our operations, when a change propagates up to the root of the tree.

To insert a data item, we proceed as follows:

```
insert key $k$, current version $i$, information info :
  {assume $k$ is not yet present}
  exact-match query for $k$ in version $i$ leads to block $A$;
  blockinsert $< k, i, *, info >$ into $A$.
```

Here, **blockinsert** is defined as follows:

```
blockinsert entry $e$ into block $A$:
  enter $e$ into $A$;
  {this may momentarily lead to a block overflow in $A$,
  conceptually; such an overflow is eliminated immediately}
  if  block overflow of $A$  then
    version split: copy current entries of $A$ into a new
    block $B$;
    blockinsert entry referencing $B$ into father of $A$;
    if  strong version underflow of $B$  then
      merge $B$
    elsif  strong version overflow of $B$  then
      treat strong version overflow of $B$.
```

Note that after a version split, the deletion version stored in the father entry referring to the dead block must be adjusted to represent the version of the version split, in order to guide subsequent searches correctly.

Merging a block makes use of the fact that a suitable sibling can always be found in the access structure:

```
merge block $B$:
  identify a sibling $D$ of $B$ to be merged;
  version split: copy current entries of $D$ into a new
  block $E$; unite $B$ and $E$ into $B$ and discard $E$;
  if  strong version overflow of $B$  then
    treat strong version overflow of $B$
    {no weak version underflow possible in father of $B$}
  else
    adapt router to $B$ in father of $B$;
    check weak version underflow of father of $B$.
```

Essentially, a strong version overflow is treated by a key split of the entries according to their key or router values:

```
treat strong version overflow of block $A$:
  key split: distribute entries of $A$ evenly among $A$ and
  $B$; adapt router to $A$ in father of $A$;
  blockinsert entry referencing $B$ into father of $A$.
```

A weak version underflow leads to a version split and a merge:

```
check weak version underflow of block $A$:
  if  weak version underflow of $A$  then
    version split: copy current entries of $A$ into a new
    block $B$;
    blockinsert entry referencing $B$ into father of $A$;
    merge $B$.
```

This completes the description of the insertion of an item into a block. To delete an item, we proceed as follows:

```
delete key $k$, current version $i$ {assume $k$ is present}:
  exact match query for $k$ in version $i$ leads to block $A$;
  blockdelete $k, i$ from $A$.

blockdelete key $k$, version $i$ from block $A$:
  change entry $< k, i^{'}, *, info >$ into $< k, i^{'}, i, info >$
  in $A$;
  check weak version underflow of $A$.
```

This completes the more detailed presentation of update operations. Let us repeat that the multiversion structure defined in this way is not a tree, but a directed acyclic graph. In general, more than one *root* block may exist. Since the number of root blocks to be expected is very small, maintaining these blocks is not a major data organization problem; see Sect. 4 for a suggestion.

In the next section, we show in an analysis that the basic operations actually do lead to the desired behavior.

3 Efficiency analysis

Recall that a block is *live* if it was not copied up to the current version, *dead* otherwise. N is the number of update operations performed on the data structure from the beginning up to the current version, m_i is the number of data items present in version i.

What are the restrictions for the choice of k and ε? First, after a *key split*, the resulting blocks must fulfill the strong version condition. Before a key split on a block A is performed, A contains at least $(k-\varepsilon)\cdot d+1$ entries. After the key split operation that distributes the entries of A among two blocks, both blocks must contain at least $(1+\varepsilon)\cdot d$ entries. Therefore, the following inequality must hold:

$$\begin{array}{l}(k-\varepsilon)\cdot d+1 \geq \frac{1}{\alpha}\cdot(1+\varepsilon)\cdot d \\ \text{or, equivalently, } k \geq \frac{1}{\alpha}+(1+\frac{1}{\alpha})\cdot\varepsilon-\frac{1}{d}\end{array} \quad (1)$$

Here, α depends on the underlying access structure. It denotes the constant fraction of data entries that are guaranteed to be in a new node. For example, $\alpha = 0.5$ is fulfilled for B-trees, i.e. inequality 1 is equivalent to $k \geq 2+3\cdot\varepsilon-\frac{1}{d}$.

Second, no strong version underflow is allowed for a block A resulting from a *merge* operation. Before a merge operation is performed, together there are at least $2\cdot d-1$ current entries in the blocks which have to be merged. Therefore we have:

$$\begin{array}{l}2\cdot d-1 \geq (1+\varepsilon)\cdot d \\ \text{or, equivalently, } \varepsilon \leq 1-\frac{1}{d}\end{array} \quad (2)$$

3.1 Run-time analysis

As introduced before, for our multiversion B-tree we have separated the concerns of identifying the root block of the requested version and querying the requested version. For the following analysis we assume that, supported from the application context, the appropriate root block is given.

Recall that our multiversion structures are based on leaf-oriented balanced-access structures. The data blocks are on level 0, the directory blocks are on level 1,2,.... Then the number of block accesses for searching a data item x in version i is at most $\lceil\log_d m_i\rceil$, because each directory block on the path from the root of version i to the leaf where x is stored has at least d references of i. Given direct access to the root of the version in question, we conclude:

Theorem 1 *The number of block accesses for searching a data item in version i is $\lceil\log_d m_i\rceil$ in the worst case.*

The arguments above can be extended to range queries that are answered by traversing the corresponding umbrella-like part of a subtree of the tree for the queried version:

Theorem 2 *The number of block accesses for answering a range query in version i that returns r data items is $O(\lceil\log_d m_i\rceil + r/d)$ in the worst case.*

The $(i+1)$-th update operation first performs an exact match query in version i and then modifies at least one data block A. If A violates the weak version condition, up to three other data blocks have to be created or modified. In this case, the parent of A – say A' – has to be modified. Again, this can lead to a violation of the weak version condition of A'. In the worst case, this situation occurs on each directory level up to the root of version i. On each directory level, at most five directory blocks have to be accessed, modified or created. Therefore we have:

Theorem 3 *The number of block accesses and modifications for the $(i+1)$-th update operation is $5\cdot\lceil\log_d m_i\rceil$ in the worst case.*

3.2 Space analysis

We analyze the worst-case space utilization over the sequence of the N update operations. The crucial factor in the analysis is the fact that a *version split*, if necessary followed by a *merge* or a *key split*, leads to new blocks which fulfill the strong version condition. Therefore we need a certain number of update operations on these blocks before the next underflow or overflow situation on these blocks can occur. To be more precise, we consider the utilization of data blocks and of directory blocks separately.

For data blocks, one update operation can lead to at most one overflow or underflow situation. We distinguish four types of situations:

- *Version split only:* One block A becomes dead and one new live block B is created. A was the first data block in the data structure or has fulfilled initially – after its creation – the strong version condition. If it becomes overfull, at least $\varepsilon\cdot d+1$ operations must have taken place on A since its creation. So the amortized space cost for each of these operations is at most $\frac{k\cdot d}{\varepsilon\cdot d+1}$.
- *Version split and key split:* One block A becomes dead and two new live blocks $B1$ and $B2$ are created. Again, at least $\varepsilon\cdot d+1$ operations must have taken place on A and therefore the amortized space cost for each of these operations is at most $\frac{2\cdot k\cdot d}{\varepsilon\cdot d+1}$.

- *Version split and merge without key split:* Two blocks $A1$ and $A2$ become dead and one new live block B is created. On $A1$ or $A2$ at least $\varepsilon \cdot d + 1$ operations must have taken place. Thus, the amortized space cost for each of these operations is at most $\frac{k \cdot d}{\varepsilon \cdot d+1}$.
- *Version split and merge with key split:* Two blocks $A1$ and $A2$ become dead and two new live blocks $B1$ and $B2$ are created. Again, on $A1$ or $A2$ at least $\varepsilon \cdot d + 1$ operations must have taken place. The amortized space cost for each of these operations is at most $\frac{2 \cdot k \cdot d}{\varepsilon \cdot d+1}$.

In all cases the amortized data block space cost per update operation S_{dat} is at most

$$\frac{2 \cdot k \cdot d}{\varepsilon \cdot d + 1} < \frac{2 \cdot k}{\varepsilon} = O(1) \tag{3}$$

For directory blocks, one update operation can lead to at most one *block overflow* or *version underflow* situation on each directory level up to the directory level of the *root* in the current version. Let L denote the maximum level that occurs during the N operations. To look precisely at the different underflow and overflow situations, we distinguish between directory blocks that are roots during their lifetime and inner blocks.

Let A^l denote an *inner directory block* of level l. We call a reference in A^l *dead*, if it is a reference to a dead block, *live* otherwise. The following situations can cause a weak version underflow or a block overflow of A^l:

- One reference in A^l becomes dead and one new reference has to be inserted into A^l. This can cause a *block overflow* with the creation of two new directory blocks.
- One reference in A^l becomes dead and two new references have to be inserted into A^l. This can cause a *block overflow* with the creation of two new directory blocks.
- Two references in A^l become dead and one new reference has to be inserted into A^l. This can cause a *weak version underflow* or a *block overflow*. In the case of a weak version underflow, a sibling of A^l also becomes dead, and up to two new directory blocks are created.
- Two references in A^l become dead and two new references have to be inserted into A^l. This can cause a *block overflow* with the creation of two new directory blocks.

Note that if a directory block is the root of the data structure in version i, a weak version underflow does not lead to a new copy of the block. A block overflow of a root block is treated in the same manner as a block overflow of an inner block.

We explain the amortized space cost per operation for the first case. The extension to the other cases and the root blocks is straightforward and yields the same result. A^l is the only live parent for the live blocks referenced from A^l and has initially fulfilled the strong version condition. Therefore, in the subtree of A^l on level $l-1$ at least $\varepsilon \cdot d+1$ new blocks have been created between the creation of A^l and the block overflow of A^l. Hence, at least $(\varepsilon \cdot d + 1) \cdot k \cdot d$ space was used. Let us assume that the amortized space cost per update on level $l - 1$ is at most C^{l-1}. Then it follows that at least $\frac{(\varepsilon \cdot d+1) \cdot k \cdot d}{C^{l-1}}$ operations have taken place in the subtree of A^l between the creation of A^l and its block overflow. The space cost for the version split of A^l and the subsequent key split is $2 \cdot k \cdot d$. Therefore, the amortized space cost per update on level l is at most

$$C^l < 2 \cdot k \cdot d \cdot \frac{C^{l-1}}{(\varepsilon \cdot d + 1) \cdot k \cdot d} < \frac{2}{\varepsilon \cdot d} \cdot C^{l-1} \tag{4}$$

for $1 \leq l \leq L$. With $C^0 := S_{data}$, i.e. $C^0 = \frac{2 \cdot k}{\varepsilon}$ (from inequality 3), we can rewrite inequality 4:

$$C^l < \left(\frac{2}{\varepsilon \cdot d}\right)^l \cdot C^0 = \left(\frac{2}{\varepsilon \cdot d}\right)^l \cdot \frac{2 \cdot k}{\varepsilon} \tag{5}$$

for $1 \leq l \leq L$.

Therefore, the total amortized directory block space cost per operation S_{dir} is at most:

$$S_{dir} < \sum_{l=1}^{L} C^l = \frac{2 \cdot k}{\varepsilon} \cdot \sum_{l=1}^{L} \left(\frac{2}{\varepsilon \cdot d}\right)^l \tag{6}$$

For $d > \frac{2}{\varepsilon}$, which can easily be satisfied in all practically relevant circumstances, we get:

$$S_{dir} < \frac{2 \cdot k}{\varepsilon} \cdot \sum_{l=1}^{\infty} \left(\frac{2}{\varepsilon \cdot d}\right)^l = O(1) \tag{7}$$

In summary, from inequalities 3 and 7 we can conclude:

Theorem 4 *The worst-case amortized space cost per update operation $S = S_{dat} + S_{dir}$ is $O(1)$ if $d \geq \frac{2}{\varepsilon}$.*

In total, we get:

Theorem 5 *The multiversion B-tree constructed in the described way from the single-version B-tree is asymptotically optimal in the worst case in time and space for all considered operations.*

The analysis shows that for a given block capacity b it is useful for the time complexity to choose d large and k small. To guarantee good space utilization it is useful to choose ε maximum, that is equal to $1 - \frac{1}{d}$, and k as small as possible without violating inequality 1. Choosing $\varepsilon = 1 - \frac{1}{d}$ gives bounds for the strong version condition of $2 \cdot d - 1$ and $(k-1) \cdot d + 1$. For instance, for block capacity $b = 25$ we get $k = 5$, $d = 5$, and $\varepsilon = 0.8$. In the worst case, this implies that we have 11.5 ($\frac{2 \cdot k}{\varepsilon} - 1$) redundant records for each key on average. Because this is quite a high number, we implemented the multiversion B-tree and ran a number of experiments with the above parameters and $N = 100\,000$ update operations. It turned out that in all experiments, we had between 1.31 and 1.70 redundant records for each key on average. Hence, our worst-case bounds are extremely pessimistic and do not imply high constant costs on average.

4 Thoughts around the main result

In the following, we present some of the thoughts around the main result that may be interesting or important in practice. First, we discuss the organization of the access to the requested B-tree root; this also solves the problem of time-oriented access, where query points in time differ from version creation times, and of maintaining user-defined versions. Second, we show how to efficiently remove the oldest versions, in order to save storage space. Our thoughts are intended to demonstrate the high potential of adapting the multiversion B-tree to different settings and different requirements. The given list of modifications and extensions is not meant to be exhaustive; additions to this list should be performed as needed.

4.1 Access to the requested version

Our presentation of the multiversion B-tree so far assumes that access to the root of a version is taken care of in the context of the application. If this is not the case, a search structure may be used to guide the access. As an example, a B-tree maintaining the version intervals of the multiversion B-tree root nodes in its leaves serves this purpose. Even in its most direct application, this access structure to the roots of the multiversion B-tree (we call it *root**) allows access to a root as well as insertion of a new root into root* in time $O(\log_b p)$, where p is the number of roots being maintained. The space efficiency of such a B-tree is obviously $O(p)$. Since p is less than $\lceil N/d \rceil$, the storage cost of root* is $O(N/b)$ and the search for a key in a multiversion query can be realized in time $O(\log_b N)$ in total, including the search for the appropriate version.

In most cases, we expect that the number of roots is much less than $\lceil N/d \rceil$. Consider, for example, the situation when the current version data set has been created by a sequence of insertions only, beginning at an empty structure. Then, the left path of the current B-tree contains all the roots of the multiversion B-tree. Therefore, the number of roots is only $O(\log_b N)$ which is considerably less than the worst-case results of the general case.

Furthermore, root* can be used to support time-oriented queries. If our setup changes from versions to time, such that each key has an insertion time stamp and a deletion time stamp, root* supports queries for any point in time (not necessarily coinciding with some insertion or deletion time) in the standard B-tree fashion.

Moreover, root* can be tuned to achieve even higher performance by observing that a new multiversion root can only be added at the high end of the current version or time spectrum. Therefore, a split of a node of root* can be made totally unbalanced: the node of the lower key range is full, whereas the node of the higher key range contains just one key, namely the new one. As a consequence, all nodes in root* are full, except those on the rightmost path. This straightforward approach is somewhat reminiscent of the append-only tree (Segev and Gunadhi 1989), where an entry pointer to the rightmost node for each level of the tree is maintained in addition, in order to favor queries to the recent past. Then, access to the records of the current (and recent past) version can be organized more efficiently, leading to a path length of $O(\log_b m_N)$. Therefore, the worst-case time bound for range queries to the current version for the MVBT tree is $O(\log_b m_N + r/b)$. An update costs time $O(\log_b N)$ in the worst case, because a change may propagate up to the root of the root* B-tree. Amortized over a sequence of updates, however, the worst-case cost of a single update is only $O(\log_b m_N)$, for the following reasons: First, in root*, the entry pointing to the current root is found in $O(1)$. Second, the record to perform the update in the current B-tree is found in $O(\log_b m_N)$. Third, the remaining effort to perform the update has only constant amortized cost (Huddleston and Mehlhorn 1982). Overall, this proves our statement.

Other access structures may be plugged in to serve as root*. For instance, if a high locality of reference to nearby versions is required, a finger search tree may be the method of choice (Huddleston and Mehlhorn 1982). To summarize, root* has the potential to be tuned to the particular application.

4.2 Purging old versions

The operation of removing the oldest versions from disk, the so-called *purge* operation, is very important in multiversion access structures, because maintaining all versions forever may be too costly in terms of storage space. Under the assumption that old versions are accessed significantly less frequently than newer ones, the amount of secondary storage can be reduced substantially by moving old versions to tertiary storage (e.g. optical disks) or, whenever the application permits, by simply deleting them (e.g. in multiversion concurrency control).

The deletion of versions older than a specified version i can be supported easily in the multiversion B-tree. A straightforward approach would be to search for all blocks which have been split by a version split in a version less than or equal to i. This search starts at the root blocks valid for version i and older. Performing a depth-first search, all blocks fulfilling the above condition can immediately be deallocated. The disadvantage of this approach is that it may access many blocks for a few that can be purged. A more efficient approach accesses only blocks that must be purged: An additional data structure is used to keep track for each node of the most recent (i.e. newest) version for which this node is relevant in a query. Since this version is just the version before the version in which the node dies, this defines a linear order on the nodes; a simple first-in-first-out queue will therefore suffice to perform all operations efficiently. Whenever a node dies, a corresponding entry is added to the tail of the queue. Whenever the oldest versions before some version i are to be deleted, triggered by the user or by some other mechanism such as concurrency control, the corresponding head entries of the queue are removed, as well as the corresponding multiversion B-tree nodes.

Note that the removal of a node from a multiversion B-tree may leave the tree in an inconsistent state: there may be pointers in the tree that point to the node that is no longer present. Nevertheless, this inconsistency is not harmful, as long as no search for a deleted version (older than version i) initiates: A search may encounter a dangling pointer but will never follow it.

5 Related work

A number of investigations on how to maintain multiversion data (historical data, time-dependent data) on external storage have been presented in the literature. Often, the goal of these investigations has been somewhat different from our goal in designing the multiversion B-tree. Nevertheless, some previous proposals pursue almost the same objective as we do, and others have been influential in setting the stage. To put our work into its proper perspective, we present a synopsis of relevant previous work in this section.

Kolovson and Stonebraker (1989) discussed the problem of maintaining multiversion data using two external storage media, magnetic disk for recent data and WORM optical disk for historical versions. They proposed two approaches, both using the R-tree index (Guttman 1984), to organize data records: according to their key values in one, according to their lifespans in the other dimension. The approaches differ by the techniques of moving data and index blocks from magnetic disk to WORM disk, also called vacuuming. In the first approach, vacuuming is triggered in the following way. If the size of the index on magnetic disk reaches a given threshold, a vacuuming process moves a given fraction of the oldest (i.e. dead) data blocks to WORM disk and – recursively up the tree – those directory blocks that refer only to blocks already stored on WORM disk. The second approach maintains two R-trees, one completely on magnetic disk, the other with the upper levels on magnetic and all levels below on WORM disk. Again, if the size of the R-tree completely stored on magnetic disk reaches a threshold, all its blocks except the root are moved to WORM disk. Then, references to the blocks below the root level, now stored on WORM disk, are inserted into the corresponding level of the R-tree on magnetic disk. Updates are only performed on the R-tree that completely resides on magnetic disk, while queries may affect both R-trees. Both approaches presented by Kolovson and Stonebraker (1989) support the same operations as the multiversion B-tree (MVBT). Additionally, queries over version intervals can be answered.

In both approaches, the height of the R-trees is $\Theta(\log_b N)$; remember that N is the total number of updates to the tree, and b is the maximum number of entries in a tree node. Therefore, each insertion needs time $\Theta(\log_b N)$; this compares with amortized time $\Theta(\log_b m_N)$ in the MVBT, since access to the newest version is always immediate. Deletion must be implemented as modification of the corresponding R-tree entry. For that, the affected entry has to be searched in the tree before modification. Because of overlapping regions in the R-tree, the search for a record may necessitate a traversal of the whole index tree in the worst case. Therefore, deletion can be extremely expensive in the worst case; this compares with worst-case time $O(\log_b m_N)$ in the MVBT. The same arguments show that exact-match queries and range queries on a given version may access $\Theta(N/b)$ blocks in the worst case. This compares with a worst-case time for exact-match queries and range queries of $\Theta(\log_b N)$ and $\Theta(\log_b N + r/b)$, respectively. Note, however, that the goals of these approaches have been somewhat different from our goal of building a multiversion B-tree.

Because no data are replicated, the space efficiency of Kolovson and Stonebraker's approaches is perfect. However, especially for sets of records with lifespans of non-uniformly distributed lengths, Kolovson and Stonebraker observed a decreasing efficiency for the R-tree.

In order to achieve better query and update performance for such data distributions, Kolovson and Stonebraker have proposed *segment R-trees* (SR-trees; Kolovson and Stonebraker 1991), a hybrid of segment trees (Bentley 1977) and R-trees (Guttman 1984). Skeleton SR-trees operate with a preset data space partitioning, based on an assumption about the data distribution. In the performance evaluation presented by Kolovson and Stonebraker (1991), the SR-trees never outperformed R-trees in the non-skeleton variant. However, skeleton SR-trees have better performance than skeleton R-trees for non-uniformly distributed interval lengths and query regions of very high or very low aspect ratio. The approach of (skeleton)SR-trees suffers from the same major inefficiencies as using R-trees to store multiversion data. There is no good worst-case guarantee for deletions, exact-match queries, and range queries.

Elmasri et al. (1990, 1991) proposed the *time index* for maintaining historical data. The time index supports all the operations of our setting, plus range queries over versions. In the time index, data records are organized in a B^+-tree according to versions (time). For each version, a bucket is maintained for storing all data records (or references to it) valid for that version. Elmasri et al. proposed several modifications of the basic approach to reduce the high redundancy resulting from this data organization. However, assuming that each update creates a new version, the space efficiency of all those variants may be as bad as $\Theta(N^2/b)$ in the worst case. An insertion of a record in the time index may create a new bucket containing all records for the new version. In the worst case, this operation requires $\Theta(N/b)$ time. Moreover, the time index does not support range queries efficiently: range query efficiency may be as bad as $\Theta(\log_b N + N/b)$ in the worst case.

The Write-once B-tree (WOBT), proposed by Easton (1986), is a variation of the B^+-tree; it is completely stored on a WORM medium, e.g. an optical disk. Because of the write-once characteristic, all versions of data are kept forever. If version numbers are assigned to the index and data records, multiversion queries can be answered in a straightforward way. To treat an overflow of a data or an index block in the WOBT, first a version split must be performed, because the overflow block itself cannot be rewritten. Afterwards, if the current entries occupy more than a given fraction of the new block (e.g. two-thirds), a key split is performed on the block before writing it to external memory. So far, the WOBT split policy is comparable to the one of the MVBT. One major difference is the treatment of a root split: if a root is split in the WOBT, a new root block is allocated that initially contains three references. One reference is pointing to the old root block, whereas the other references are pointing to the blocks obtained from splitting the old root. Thus, a WOBT has one root, and all the paths from the root to a data block have the same length $\Theta(\log_b N)$. In contrast, if a root is split in the MVBT, the reference to the new root is inserted into root*, the data structure organizing the root blocks.

Under the pessimistic assumption that the computation of the root of an arbitrary non-current version requires $\Theta(\log_b N)$ time, the MVBT is still more time-efficient than the WOBT for updates and queries on the current version. Recall that the root of the MVBT valid for the current version – and for some recent non-current versions – can be accessed in time $O(1)$ by maintaining a direct reference to this block. Therefore, queries to these versions and updates to the current version are more efficient than $O(\log_b N + r/b)$ and $O(\log_b N)$, the respective bounds in the WOBT. Moreover, the WOBT is restricted to insertions and modifications of the non-key part of records, while the MVBT supports both insertions and deletions.

In order to reduce storage costs and to improve performance of queries on the current version, Lomet and Salzberg (1989) proposed a variant of the WOBT, the *time-split B-tree* (TSBT). The TSBT spans over magnetic and WORM disk. All live blocks are stored on magnetic disk, while a dead block migrates to WORM disk during a version split. Lomet and Salzberg distinguish split policies for index blocks from those for data blocks.

For splitting data blocks, the following two basic types of splits can be performed in the TSBT. First, in contrast to the WOBT, the version (time) used for a version split (time split) of a data block is not restricted to the current version, but can be chosen arbitrarily. Second, a key split can be performed on a data block instead of a version (time) split. Lomet and Salzberg (1989, 1990) discussed the effects of different data block split policies, with emphasis on space cost. The space cost is given as the sum of storage cost on magnetic and WORM disk. For *data block split*, the following three split policies were proposed:

- The *WOBT policy* is the split policy as used in the WOBT.
- The *time-of-last-update policy* performs a version split with the version of the last update. This reduces the number of entries to be kept in the dead block after a version split, and therefore the storage space needed on WORM disk. The number of entries in the live block remains unchanged in comparison to a version split of the WOBT. As for the WOBT, a key split will be performed immediately after a version split, if the current entries occupy at least a given fraction of the new block (e.g. two-thirds).
- The *isolated-key-split policy* performs a key split if at least a given fraction of the entries (e.g. two-thirds) of the overfull node belongs to the current version. Otherwise, a version split with the current version is performed. In comparison with the two split policies described above, this split policy reduces redundancy and therefore storage space: a version split is not performed if it would be immediately followed by a key split. The disadvantage of this policy is that by a key split the dead entries of the block are spread over two blocks. This decreases the performance for range queries to non-current versions. Consequently, in contrast to the MVBT it is not guaranteed that a block contains for each version either no entries or $\Theta(b)$ entries. Then, a range query in the worst case requires $\Theta(N/b)$ blocks, independent of the size of the response set and independent of the number of records in the corresponding version. In comparison with the TSBT, the MVBT requires more storage space [but it is still $O(N)$] to cluster versions appropriately such that range queries can be answered with $O(\log_b N + r/b)$ disk accesses.

For index blocks, split policies cannot be the same as for data blocks. The problem of using data block split policies on index nodes is the following: a dead index block may still contain references to live blocks on the next lower level of the tree. If such a live block becomes dead (i.e. it migrates to optical disk), the corresponding references have to be updated in the parent nodes. However, this would require that the dead blocks are stored on a write-many storage medium.

In their first paper on the TSBT, Lomet and Salzberg (1989) discuss the effects of using version and key splits for index block splitting. An index block split policy based on key splits avoids redundancy, but leads to an index which gives no selectivity according to versions. Moreover, an index block may contain entries which cannot be separated by a key split. Therefore, for the simulations presented in Lomet and Salzberg (1990), the authors applied another policy for index block splitting. A version split is performed using the insertion version of the oldest index entry that is still valid for the current version. Then, a dead block contains only non-current index entries and therefore it can be written onto WORM disk. In addition to the redundancy that this entails in index blocks, the main problem of this split policy is that such a split version may not exist. In this case, a key split is possible. This separates not only current entries (as desired), but also dead ones. As a consequence, the TSBT does not have a lower bound on the number of entries for a version in an index block.

Lanka and Mays (1991) presented three approaches for fully persistent B^+-trees. Full persistence means that changes can be applied to any version, current or past, creating a new version. Because this concept of multiple versions of data is more general than ours, all three approaches also can be used to maintain our type of multiversion data (partially persistent data). Like the MVBT, and in contrast to the WOBT and the TSBT, all the proposed techniques support insertions and deletions.

The first approach, the *fat node method*, is based on the idea that each node, index node or leaf with data items is fat enough to store all versions of all its entries. Lanka and Mays proposed implementing such a fat node as a set of blocks, one block per version, and a version block, containing references to each of the blocks. Although query and update efficiency for any given version i is $O(\log_b m_i)$ [based on the assumption that the root block for version i can be accessed in time $O(1)$], this obviously leads to storage cost of $\Theta(1)$ blocks per update. Moreover, it is doubtful whether one physical block is sufficient to implement a version block, as assumed in the paper.

The *fat field method* is an improvement on the fat node method, storing entries of different versions in the same block. To describe which versions a B^+-tree entry belongs to, each entry is extended by a field representing its insertion version and the set of its deletion versions. Applying the fat field method to our multiversion data, the structure of an entry is equal to that of a MVBT entry, because only one deletion version can occur. Also comparable to the MVBT, the fat field method guarantees for each block and each version in the block that a number of entries proportional to the block capacity (namely 50%) is stored in that block. If for any version less than half of the entries belong to that version, a version split and a merge is performed. The split policy is a version split, followed by a key split if the block is still overfull. In contrast to the MVBT, for the fat field method a block may be full after split or merge. That means that after a constant number of updates, the next split or merge may be triggered, leading to a worst-case storage cost of $\Theta(1)$ blocks per update. As for the fat node method, the query performance analysis for the fat field method is based on the assumption that each version block fits into one physical block. This assumption is not realistic for organizing a high number of versions in the structure.

The third approach proposed by Lanka and Mays (1991) is the *pure version block method*. In this technique, a B^+-tree index is built over the key values of the data items. This technique does not give any selectivity according to versions.

As a result, we conclude that the approaches for multiversion B-trees proposed in the literature have their merits in exposing many interesting ideas and achieving good performance in one or the other aspect. Nevertheless, none of them achieves asymptotic worst-case optimality both in the time for all operations and in space. Therefore, we feel the MVBT to be a worthwhile addition to the list of multiversion external B-trees.

6 Conclusion

In this paper, we have presented a technique to transform certain single-version hierarchical external storage access structures into multiversion structures. We have shown that our technique delivers multiversion capabilities with no change in asymptotic worst-case performance for B-trees, if we assume that the root block for a requested version is given from the application context. Otherwise, a search structure for the appropriate root block can be tuned to the particular requirements. The properties of B-trees that we have used include the following characteristics of access structures:

1. The access structure is a rooted tree of external storage blocks.
2. Data items are stored in the leaves (data blocks) of the tree; the inner nodes (directory blocks) store routing information.
3. The tree is balanced; typically, all leaves are on the same level.
4. The tree can be restructured by splitting blocks or by merging blocks with siblings along a path between the root and a leaf.
5. A block split can be balanced; that is, each of the two resulting blocks is guaranteed to contain at least a constant fraction α, $0 < \alpha \leq 0.5$, of the entries.

Single-version access structures satisfying these requirements are therefore the prime candidates for carrying over and applying our technique. Examples of such access structures other than the B-tree include the cell-tree (Günther and Bilmes 1991), the BANG file (Freeston 1987), and the R-tree family (Guttman 1984, Greene 1989, Beckmann et al. 1990), whenever reinsertion of data items can be replaced by block merge without loss of geometric clustering. Note that the data items are not limited to one-dimensional points.

We conjecture that our technique may be useful also for access structures that do not satisfy all of our requirements, such as hierarchical grid files. In that case, the performance guarantees derived for the MVBT do not carry over without change. This is clearly due to the fact that these performance guarantees do not hold for the single-version structure in the first place. However, we do not know in sufficient generality how the performance of an arbitrary external access structure changes if it is transformed into a multiversion structure along the lines of our technique.

Acknowledgements. We want to thank an anonymous referee for an extraordinary effort and thorough discussion that led to a great improvement in the presentation of the paper. This work was partially supported by grants ESPRIT 6881 of the European Community and Wi810/2–5 of the Deutsche Forschungsgemeinschaft DFG.

References

Barghouti NS, Kaiser GE (1991) Concurrency control in advanced database applications. ACM Comput Surv 23:269–317

Becker B, Gschwind S, Ohler T, Seeger B, Widmayer P (1993) On optimal multiversion access structures. In: 3rd International Symposium on Large Spatial Databases. (Lecture Notes in Computer Science, vol 692) Springer, Berlin Heidelberg New York, pp 123–141

Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. ACM SIGMOD International Conference on Management of Data 19:322–331

Bentley JL (1977) Algorithms for Klee's rectangle problems. Computer Science Department, Carnegie-Mellon University, Pittsburg, Pa

Bernstein PA, Hadzilacos V, Goodman N (1987) Concurrency control and recovery in database systems. Addison Wesley, Reading, Mass

Clifford J, Ariav G (1986) Temporal data management: models and systems. In: Ariav G, Clifford J (eds) New directions for database systems. Ablex, Norwood, NJ, pp 168–186

Driscoll JR, Sarnak N, Sleator DD, Tarjan RE (1989) Making data structures persistent. J Comput Syst Sci 38:86–124

Easton M (1986) Key-sequence data sets on indelible storage. IBM J Res Dev 30:230–241

Elmasri R, Wuu G, Kim Y-J (1990) The time index: an access structure for temporal data. 16th International Conference on Very Large Data Bases, pp 1–12

Elmasri R, Wuu G, Kim Y-J (1991) Efficient implementation techniques for the time index. Seventh IEEE International Conference on Data Engineering, pp 102–111

Freeston MW (1987) The BANG-file: a new kind of grid file. ACM SIGMOD International Conference on Management of Data 16:260–269

Gonnet GH, Baeza-Yates R (1991) Handbook of algorithms and data structures: in PASCAL and C. Addison-Wesley, Reading, Mass

Greene, D (1989) An implementation and performance analysis of spatial access methods. Fifth IEEE International Conference on Data Engineering, pp 606–615

Günther O, Bilmes J (1991) Tree-based access methods for spatial databases: implementation and performance evaluation. IEEE Trans Knowl Data Eng 3:342–356

Guttman A (1984) R-trees: a dynamic index structure for spatial searching. ACM SIGMOD International Conference on Management of Data 12:47–57

Huddleston S, Mehlhorn K (1982) A new data structure for representing sorted lists. Acta Inform 17:157–184

Kanellakis PC, Ramaswamy S, Vengroff DE, Vitter JS (1993) Indexing for data models with constraints and classes. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems 12:233–243

Katz RH (1990) Towards a unified framework for version modeling in engineering databases. ACM Comput Surv 22:375–408

Kolovson C, Stonebraker M (1989) Indexing techniques for historical databases. Fifth IEEE International Conference on Data Engineering, pp 127–137

Kolovson C, Stonebraker M (1991) Segment indexes: dynamic indexing techniques for multi-dimensional interval data. ACM SIGMOD International Conference on Management of Data 20:138–147

Lanka S, Mays E (1991) Fully persistent B^+–trees. ACM SIGMOD International Conference on Management of Data 20:426–435

Lomet D, Salzberg B (1989) Access methods for multiversion data. ACM SIGMOD International Conference on Management of Data 18:315–324

Lomet D, Salzberg B (1990) The performance of a multiversion access method. ACM SIGMOD International Conference on Management of Data 19:353–363

Mehlhorn K, Tsakalidis A (1990) Data structures. In: Leeuwen J van (ed) Handbook of theoretical computer science, vol A: Algorithms and complexity. Elsevier, Amsterdam, pp 301–341

Sedgewick R (1988) Algorithms. Addison-Wesley, Reading, Mass

Segev A, Gunadhi H (1989) Event-join optimization in temporal relational databases. 15th International Conference on Very Large Data Bases, pp 205–215

Tansel, AU, Clifford J, Gadia S, Jajodia S, Segev A, Snodgrass R (1993) Temporal databases – theory, design, implementation. Benjamin/Cummings, Redwood City, Calif

Vitter JS, (1991) Efficient memory access in large-scale computation. In: 8th Annual Symposium on Theoretical Aspects of Computer Science. (Lecture Notes in Computer Science, vol 480) Springer, Berlin Heidelberg New York, pp 26–41