On Optimal Multiversion Access Structures *

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

[1] Institut für Informatik, Universität Freiburg, Rheinstraße 10–12, D–7800 Freiburg
[2] Institut für Theoretische Informatik, ETH Zentrum, CH–8092 Zürich
[3] Institut für Informatik, Universität München, Leopoldstraße 11b, D–8000 München

Abstract. We propose an asymptotically optimal multiversion B-tree. In our setting, insertions and deletions of data items are allowed only for the present version, whereas range queries and exact match queries are allowed for any version, present or past. The technique we present for transforming a (usual single version) B-tree into a multiversion B-tree is more general: it applies to a number of spatial and non-spatial hierarchical external access structures with certain properties directly, and it can be modified for others. For the B-tree and several other hierarchical external access structures, multiversion capabilities come at no extra cost, neither for storage space nor for runtime, asymptotically in the worst case. The analysis of the behavior of the multiversion B-tree shows that the constant loss of efficiency is low enough to make our suggestion not only a theoretical, but also a practical one.

1 Introduction

The importance of maintaining data not only in their latest version, but also to keep track of their development over time has been widely recognized. Version data in engineering databases [Kat90] and time oriented data [CA86] in geographical information systems [Lan92], e.g. to manage land register data, are two prime examples for situations in which the concepts of versions and time are visible to the user; in multiversion concurrency control [BK91, BHG87], 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 [BHG87, DSST89] 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

* Partially supported by the ESPRIT Basic Research Project "AMUSING".

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 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 spatial access structures for *external storage* that support at least *insertions*, *deletions*, *exact match queries* (associative search) and locality based spatial queries such as e.g. *proximity queries*, in addition to application specific operations like *purging* of old enough versions in concurrency control. We limit our discussion to the situation in which a change can only be applied to the present version, whereas queries can be performed on any version, present 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 spatial 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, asymptotically at no extra amortized time and space costs, by applying a general technique [DSST89]. For the special case of balanced binary search trees, the extra time bounds hold 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: the size of a block is not taken into consideration. That is, a block is 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. In contrast, 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 *block capacity* b. As a consequence, at least the space efficiency of this approach is clearly unacceptable.

It is the contribution of this paper to propose a technique for transforming spatial and non-spatial single version *external* access structures into multiversion structures, with no extra cost in space or time asymptotically, where the block capacity b is not considered to be a constant. This result is achieved for a certain class of hierarchical external access structures. It is important to note that this class contains the B-tree and its variants, not only because the B-tree is a ubiquitous external data structure, but also because B-tree variants play a prominent role in the area of spatial data structures. Here they are used e.g. for maintaining quad trees on external storage by means of their quad code, and, more generally, for maintaining any linearly coded spatial arrangement (for a comprehensive introduction and survey, see [Sam89a, Sam89b]). In addition, it is worth noting that our result solves the long-standing open problem of designing multiversion B-trees with asymptotically optimal extra cost.

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 (such as locational codes of spatial cells). Let us restrict our presentation to the following operations:

- *insert (key,info)*: insert a record with given *key* and *info* component into the *present* version; this operation creates a new version.
- *delete (key)*: delete the (unique) record with given *key* from the *present* 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 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 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 for the bounds being optimum is the fact that for each version i, the efficiency of all operations is the same as if the data present in version i would be maintained separately in its own B-tree. This paper presents a multiversion B-tree structure satisfying these efficiency requirements. For comparison, the next section reviews previous approaches to the problem of designing multiversion B-trees. Section 3 describes and Section 4 analyzes our multiversion B-tree. Section 5 concludes the paper.

2 Previous approaches for B-trees

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, [KS89]

view versions (time) as an extra dimension and store 1-dimensional intervals in 2-dimensional space in an R-tree. This gives good storage space efficiency, but query and update efficiency may be as bad as $\Theta(\log_b N)$. That is, 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. Also with the aim of good storage space efficiency, [EWK90] suggest to organize all keys in the leaves of a B^+-tree, and for each key maintain a further B^+-tree for the set of version-intervals and associated attributes of that key; this results in the same query and update time inefficiency as [KS89].

In a different setting, [LS89] propose the time-split B-tree, based on the write-once B-tree of [Eas86], for storing multiversion data on both, magnetical and optical WORM disks. Under the basic assumption that *deletions do not occur*, the time-split B-tree (as well as the write-once B-tree) yields good space and time efficiency (for an analysis, see [LS90]) by making clever use of two types of block splits, the usual split according to keys and a split according to time. Even under this restriction (which clearly takes away much of the burden in designing multiversion structures), range queries cannot be performed efficiently: a single range query may require $\Theta(r)$ block accesses, since records in a version that are close in key space are not necessarily clustered in blocks. Therefore, the time-split B-tree is efficient only for insertions and exact match operations. For spatial queries and also most other applications, this is not sufficient. If deletions are allowed, retrieval performance can degenerate. For example, an exact match query in version i may require $\Theta(\log_b N)$ block accesses; that is, it does not depend on the number of data items in version i, but on the total number of update operations in the file.

Several variants of multiversion B^+-trees that also allow updates in the past have been proposed in [LM91], based on the ideas on full persistence in [DSST89]. This is a more general concept than ours; if simplified for our multiversion situation (in the past only queries, no updates), query efficiency depends only on the number of items that belong to the queried version, as desired. Storage space, however, may be as high as $\Theta(N)$ blocks, because each operation may copy an

entire block — a result that is achieved trivially by identifying each node in the *node copying* method of [DSST89] by a block.

As a result, none of the B-trees for multiple versions presented in the literature so far achieve optimum performance in time and space. In the next section, we present an optimum multiversion B-tree; it is suited to spatial applications, because it accomodates multiversion range queries efficiently. 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.

3 Optimal Multiversion Access Structures for External Storage

We present our technique to transform single version external access structures into multiversion structures at the example of the B-tree. 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 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 earliest *in_version* and the latest *del_version* of the *referenced* subtree, guides the search for a data item in the obvious way, for exact match queries as well as for range queries.

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 tells 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 request 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 e.g. in B-trees, cannot occur, since entries are never removed from blocks. However, the weak version condition may be violated as a result of a deletion; such a *weak version underflow* occurs, if an entry is deleted in a block with exactly d present version entries.

The structural modification after a block overflow copies the block and removes all but the present version entries from the copy. We call this operation a *version split*; it is comparable to a time split at the present time in [Eas86]; equivalently, it may be compared to the node copying operation of [DSST89]. Clearly, if no obsolete information is present, no copy needs to be made. Nevertheless, 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 at least $\varepsilon d + 1$ insert operations or delete operations are necessary to arrive at the next block overflow or version underflow in that block, for some constant ε (assume for simplicity that εd is integer) to be defined more precisely in the next section. As a consequence, the number of present version entries after a version split must be in the range from $(1 + \varepsilon)d$ to $(k - \varepsilon)d$; we call this the *strong version*

condition. If a version split leads to less than $(1+\varepsilon)d$ entries – we say: a *strong version underflow* occurs –, a merge is attempted with a copy of a sibling block containing only its present version 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 $(k-\varepsilon)d$ entries in a block – we say: a *strong version overflow* occurs –, a key split is performed. A similar idea is used for the write-once B-tree [Eas86] to eliminate a time-overflow.

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.

To insert a data item, we proceed as follows:

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

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

blockinsert entry e into block A:
enter e into A;
if *block overflow* of A **then**
version split: copy present version entries of A into 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 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 present version entries of D into 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 present version entries of A into 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, present 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 mention that the multiversion structure defined in this way is not a tree, but a directed acyclic graph; the same is true also for other multiversion access structures, like e.g. the time-split B-tree [LS89]. 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. We simply propose to maintain references to root blocks with associated version numbers as data items in a B-tree, where keys are the version numbers alone. It is easy to calculate that the B-tree of version numbers will rarely have depth two and never exceed it, and for all practical purposes can be kept in main memory.

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

4 Efficiency Analysis

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

$(\star)$ $\quad (k-\varepsilon)\cdot d+1 \geq \frac{1}{\alpha}\cdot(1+\varepsilon)\cdot d \quad$ or, equivalently, $\quad k \geq \frac{1}{\alpha}+(1+\frac{1}{\alpha})\cdot\varepsilon-\frac{1}{d}$.

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 $(\star)$ is equivalent to $k \geq 2+3\varepsilon-\frac{1}{d}$.

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 $2d-1$ present version entries in the blocks which have to be merged. Therefore we have:

$(\star\star)$ $\quad 2d-1 \geq (1+\varepsilon)\cdot d \quad$ or, equivalently, $\quad \varepsilon \leq 1-\frac{1}{d}$.

4.1 Runtime analysis

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,.... Assuming that the valid *root* reference can be found for each version i with at most one block access – an extremely conservative count, as in [LM91] –, the number of block accesses for searching a data item x in version i is at most $2+\lceil\log_d m_i\rceil$, 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. Therefore, we conclude:

Theorem 1 *The number of block accesses for searching a data item in version i is $2+\lceil\log_d m_i\rceil$ 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 $O(\lceil\log_d m_i\rceil + r/b)$ in the worst case.*

For comparison, the data structure in [LM91], when updates in the past are disallowed, yields a worst case complexity for exact match queries of $2 + 2 \cdot \lceil \log_{d'} m_i \rceil$ with $b = 2 \cdot d' - 1$ (note that in that case, only entries with constant length need to be handled in the directory of [LM91]). The asymptotic efficiency of exact match queries is, hence, the same, and for realistic choices of k and d (e.g. $d \geq 10, k \leq 5$), our access paths are shorter by at least 25 %, a certain fringe benefit.

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 four directory blocks have to be modified or created. Therefore we have:

Theorem 3 *The number of block accesses and modifications for the $(i+1)$-th update operation is $2 + 5 \cdot \lceil \log_d m_i \rceil$ in the worst case.*

In contrast to the write-once B-tree [Eas86] and the time-split B-tree [LS89], the results of our structure (presented in Theorems 1, 2 and 3) depend on m_i, the number of data entries in the i-th version. Whenever deletions would be allowed, both, the time-split B-tree and the write-once B-tree, would require $1 + \log_{b/2} N$ block accesses in the worst case for an exact match query. Thus, retrieval performance would actually depend not on m_i, but on N instead.

4.2 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 over- or underfull, 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. So 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

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

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 present 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* at 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 becomes also *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. Because A^l is the only *live* parent for the *live* blocks referenced from A^l, and A^l has initially fulfilled the strong version condition, 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. Therefore at least $(\varepsilon \cdot d+1) \cdot k \cdot d$ space was used since then. As an induction hypothesis, assume that the amortized space cost per update on level $l-1$ is at most C^{l-1}, with $C^0 = \frac{2 \cdot k}{\varepsilon}$. 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 the *block overflow*. The space cost for the *version split* and the subsequent *key split* of A^l is $2 \cdot k \cdot d$. Therefore, the amortized space cost for the operations in the subtree of A^l is at most

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

for $l = 1, .., L$.

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

$$\left(\frac{2}{d \cdot \varepsilon}\right) \cdot \frac{2 \cdot k}{\varepsilon} + \left(\frac{2}{d \cdot \varepsilon}\right)^2 \cdot \frac{2 \cdot k}{\varepsilon} + ...+ \left(\frac{2}{d \cdot \varepsilon}\right)^L \cdot \frac{2 \cdot k}{\varepsilon}.$$

For $d \geq \frac{4}{\varepsilon}$, we have $(\frac{2}{d \cdot \varepsilon})^l \leq (\frac{1}{2})^l$ for $l \geq 1$.

Hence,

$$S_{dir} \leq \frac{2 \cdot k}{\varepsilon} \cdot \sum_{l=1}^{L} \left(\frac{1}{2}\right)^l \leq \frac{2 \cdot k}{\varepsilon} = O(1).$$

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

This is in contrast to the mechanism of [LM91], were the amortized space cost per update operation can be as high as $\Theta(b)$. Note that the condition that $d \geq \frac{4}{\varepsilon}$ can be easily satisfied in all practically relevant circumstances. 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 a 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 $(\star)$. Choosing $\varepsilon = 1 - \frac{1}{d}$ gives bounds for the strong version condition of $2d - 1$ and $(k - 1) \cdot d + 1$.

5 Conclusion

In this paper, we have presented a technique to transform spatial and non-spatial single version hierarchical external storage access structures into multiversion structures. We have shown that our technique delivers multiversion capabilities asymptotically free of charge for B-trees. 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. A prominent example of such an access structure is the B^+-tree; other examples that are important in spatial applications include the cell-tree [GB91], the BANG-File [Fre87], and the R-tree family [Gut84, Gre89, BKSS90], 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, but may even be multidimensional geometric objects of non-zero size.

For B-trees, the number of block accesses for insertion, deletion, exact match query and range query increases only by a small constant in the worst case; the same holds for the number of blocks needed to store the data. That is, our suggestion is not merely of theoretical, but also of practical interest. Previously, asymptotically optimal multiversion behavior had not even been achieved for B-trees.

While in our model, each update creates a new version and hence version numbers are consecutive integers, our technique still works whenever version numbers are monotonically increasing. For instance, associating a (real-valued) point in time with each version – e.g. its creation time – makes it possible to pose queries referring to the data present at any given point in time.

It should be clear that the multiversion structures designed according to our technique are not limited to the basic operations we considered explicitly. Furthermore, the multiversion structures are flexible enough to sustain even structural additions. As an example case, consider the possibility that a multiversion database may run out of space as the number of versions keeps growing. In some applications, old versions can be moved to an archival storage medium, such as an optical WORM disk; in others, such as multiversion concurrency control, versions older than the time stamp of the oldest active operation can be deleted from the database. Our structures can be made to support these *purge* operations by maintaining a *purge queue* that contains all the dead blocks in the order of increasing time stamps. After a version split, the block that became dead is enqueued; a purge operation with a given time stamp dequeues all dead blocks of lower time stamp and simply frees their storage space, without performing any other changes in the multiversion structure. As a consequence, the multiversion structure may be left with references to blocks that have been purged. This is not a problem in the described setting, since the validity of a reference can be determined by comparing it with the time stamp of the oldest version that has not been purged yet. Additionally, when a block overflow occurs, entries with references to purged blocks can be deleted, such that a version split may be unnecessary.

We conjecture that our technique may be useful also for spatial and non-spatial access structures that do not satisfy all of our requirements, such as hierarchical grid files. In that case, the derived performance guarantees do not carry over without change. This is clearly due to the fact that the 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.

References

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. *Addison Wesley Publ. Co., Reading, Massachusetts*, 1987.

[BK91] N.S. Barghouti and G.E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. *ACM SIGMOD International Conf. on Management of Data*, 19:322–331, 1990.

[CA86] J. Clifford and G. Ariav. Temporal data management: models and systems. *New directions for database systems, Eds. Ariav, G. and Clifford, J. Ablex, Publishing Co., Norwood, N.J.*, pages 168–186, 1986.

[DSST89] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Comp. and System Sci.*, 38:86–124, 1989.

[Eas86] M. Easton. Key-sequence data sets on indelible storage. *IBM J. Res. Development 30, 3*, pages 230–241, 1986.

[EWK90] R. Elmasri, G. Wuu, and Y.-J. Kim. The time index: An access structure for temporal data. *ACM SIGMOD International Conf. on Management of Data*, 19:1–12, 1990.

[Fre87] M.W. Freeston. The BANG-file: a new kind of grid file. *ACM SIGMOD International Conf. on Management of Data*, 16:260–269, 1987.

[Gre89] D. Greene. An implementation and performance analysis of spatial access methods. *Fifth IEEE International Conference on Data Engineering, Los Angeles*, 5:606–615, 1989.

[GB91] O. Günther and J. Bilmes. Tree-based access methods for spatial databases: implementation and performance evaluation. *IEEE Trans. on Knowledge and Data Eng.*, pages 342–356, 1991.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD International Conf. on Management of Data*, 12:47–57, 1984.

[Kat90] R.H. Katz. Towards a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, 1990.

[KS89] C. Kolovson and M. Stonebraker. Indexing techniques for historical databases. *IEEE Trans. on Knowledge and Data Eng.*, pages 127–137, 1989.

[Lan92] G. Langran. Time in Geographical Information Systems. *Taylor and Francis*, 1992.

[LM91] S. Lanka and E. Mays. Fully persistent B^+–trees. *ACM SIGMOD International Conf. on Management of Data*, 20:426–435, 1991.

[LS89] D. Lomet and B. Salzberg. Access methods for multiversion data. *ACM SIGMOD International Conf. on Management of Data*, 18:315–324, 1989.

[LS90] D. Lomet and B. Salzberg. The performance of a multiversion access method. *ACM SIGMOD International Conf. on Management of Data*, 19:353–363, 1990.

[Sam89a] H. Samet. The design and analysis of spatial data structures. *Addison-Wesley*, 1989.

[Sam89b] H. Samet. Applications of spatial data structures: computer graphics, image processing, and GIS. *Addison-Wesley*, 1989.