Fully Persistent B-Trees

Gerth Stølting Brodal
MADALGO*
Department of Computer Science
Aarhus University, Denmark
gerth@madalgo.au.dk

Spyros Sioutas
Department of Informatics
Ionian University
Corfu, Greece
sioutas@ionio.gr

Konstantinos Tsakalidis
MADALGO*
Department of Computer Science
Aarhus University, Denmark
tsakalid@madalgo.au.dk

Kostas Tsichlas
Department of Informatics
Aristotle University
of Thessaloniki, Greece
tsichlas@csd.auth.gr

Abstract

We present I/O-efficient fully persistent B-Trees that support range searches at any version in $O(\log_B n + t/B)$ I/Os and updates at any version in $O(\log_B n + \log_2 B)$ amortized I/Os, using space $O(m/B)$ disk blocks. By n we denote the number of elements in the accessed version, by m the total number of updates, by t the size of the query's output, and by B the disk block size. The result improves the previous fully persistent B-Trees of Lanka and Mays by a factor of $O(\log_B m)$ for the range query complexity and $O(\log_B n)$ for the update complexity. To achieve the result, we first present a new B-Tree implementation that supports searches and updates in $O(\log_B n)$ I/Os, using $O(n/B)$ blocks of space. Moreover, every update makes in the worst case a constant number of modifications to the data structure. We make these B-Trees fully persistent using an I/O-efficient method for full persistence that is inspired by the *node-splitting* method of Driscoll et al. The method we present is interesting in its own right and can be applied to any external memory pointer based data structure with maximum in-degree d_{in} bounded by a constant and out-degree bounded by $O(B)$, where every node occupies a constant number of blocks on disk. The I/O-overhead per modification to the ephemeral structure is $O(d_{in} \log_2 B)$ amortized I/Os, and the space overhead is $O(d_{in}/B)$ amortized blocks. Access to a field of an ephemeral block is supported in $O(\log_2 d_{in})$ worst case I/Os.

1 Introduction

B-Trees are the most common dynamic dictionary data structures used for external memory [4, 7, 14]. We study the problem of making B-Trees fully persistent in the I/O model [1]. This problem finds applications in the fields of databases [19] and computational geometry [23].

Ordinary dynamic data structures, such as B-Trees, are *ephemeral*, meaning that updates create a new version of the data structure without maintaining previous versions. A *persistent* data structure remembers all versions of the ephemeral data structure as updates are performed to it. Depending on the operations we are allowed to do on previous versions, we get several notions of persistence. If we can only update the version produced last and the other versions are read-only, the data structure is *partially* persistent. In this case the versions form a list (*version list*). A more general case, *full* persistence, allows any version to be updated, yielding a *version tree* instead. In turn, this is a special case of *confluent* persistence, where the additional operation of merging different versions together is allowed. Here, the versions form a directed acyclic graph (*version DAG*). A survey on persistence can be found in [15].

Previous Results The currently most efficient partially persistent B-Trees [2] achieve $O(1)$ I/O-overhead per operation. The currently most efficient fully persistent B-Trees [16] achieve multiplicative $O(\log_B m)$ I/O-overhead per query operation and multiplicative $O(\log_B n)$ I/O-overhead per update operation, where n is the number of elements in the accessed version, m is the total number of updates performed to all versions, and B is the size of the block in the I/O model.

In particular, the most efficient fully persistent B-Trees, which can also be used for confluent persistence, are the fully persistent B$^+$-Trees (FPBT) of Lanka and Mays [16]. They support range queries in $O((\log_B n + t/B)\log_B m)$ I/Os and updates in $O(\log_B^2 n)$ amortized I/Os, using $O(m/B)$ disk blocks of space, where t is the size of the range query's output. Multiple variants of B-Trees have been made partially per-

*Center for Massive Data Algorithmics - a Center of the Danish National Research Foundation

		Update I/Os	Range Query I/Os
Partial	TSB [17]	$\log_B n \log_B m^{\dagger}$	n/B
	MVBT [5]	$\log_B^2 n$	$\log_B m + t/B$
	MVAS [22]	$\log_B^2 n$	$\log_B m + t/B$
	ADT [2]	$\log_B m$	$\log_B m + t/B$
Full	FPBT [16]	$\log_B^2 n$	$(\log_B n + t/B) \log_B m$
	New	$\log_B n + \log_2 B$	$\log_B n + t/B$

Table 1: I/O-Bounds for persistent B-Trees used in an online setting. The number of operations is m, the size of the accessed version is n, the size of the block is B and the size of the range query's output is t. All structures occupy $O(m/B)$ space. † The update time of the TSB is worst case. All other update bounds are amortized.

sistent [4, 7, 18, 13, 14]. Salzberg and Tsotras' [19] survey on persistent access methods and other techniques for time-evolving data provides a comparison among partially persistent B$^+$-Trees used to process databases on disks. They include the Multiversion B-Trees (MVBT) developed by Becker et al. [5], the Multiversion Access Structure (MVAS) of Varman and Verma [22] and the Time-Split B-Trees (TSB) of Lomet and Salzberg [17]. Moreover, the authors in [12] acquire partially persistent *hysterical* B-Trees [18] optimized for offline batched problems. The most efficient implementation of partially persistent B-Trees (ADT) was presented by Arge et al. [2] in order to solve efficiently the static point location problem in the I/O model. They support range queries in $O(\log_B m + t/B)$ I/Os and updates in $O(\log_B m)$ amortized I/Os, using $O(m/B)$ disk blocks. In Table 1 we summarize the partially and fully persistent B-Trees that can be used in an online setting.

All the previous persistent B-Trees follow an approach similar to those of Driscoll et al. [10] who present several generic and efficient techniques to make an ephemeral data structure partially or fully persistent in the pointer machine model. In particular, Driscoll et al. presented two methods in order to achieve full persistence. The *fat node* method that achieves $O(1)$ amortized space whenever an update changes $O(1)$ elements in a node (*update step*), and $O(\log n)$ worst case time overhead whenever $O(1)$ elements in a node are accessed (*access step*) or updated. Lanka and Mays [16] follow a similar method for their FPBT that also yields a logarithmic I/O-overhead per update step. The second method proposed by Driscoll et al. is called *node-splitting* and achieves $O(1)$ amortized space and time overhead per update step and $O(1)$ worst case time overhead per access step. However, it can only be applied to data structures whose underlying graph has its in-degree and out-degree bounded by a constant.

In the pointer machine model, a direct application of the node-splitting method to B-Trees with constant degree is efficient since the in-degree of every node is one. However, applying this method directly to the I/O model will not yield an I/O-efficient fully persistent data structure. The persistent nodes of Driscoll et al. have constant size and thus correspond to at most a constant number of updated elements of the ephemeral structure. However, a persistent node of size $\Theta(B)$ can correspond to $\Theta(B)$ versions of an ephemeral node. In order to find the appropriate version during navigation in the persistent node, as many version-ids must be compared in the version list, using the data structure of [8]. This causes $\Theta(B)$ I/Os in the worst case, since the version list is too large to fit in internal memory. By simple modifications an $O(\log_2 B)$ I/O-overhead per update and access step can be achieved.

Our Results We obtain fully persistent B-Trees with $O(1)$ I/O-overhead per query operation and additive $O(\log_2 B)$ I/O-overhead per update operation. In particular, we present an implementation of fully persistent B-Trees that supports range queries at any version in $O(\log_B n + t/B)$ I/Os and updates at any version in $O(\log_B n + \log_2 B)$ amortized I/Os, using $O(m/B)$ disk blocks. In Section 2 we present a method for making an external data structure fully persistent, inspired by the node-splitting method of Driscoll et al. [10]. We require that the ephemeral external data structure is pointer-based, and every node of its underlying graph occupies at most a constant number of blocks on disk. This implies that the out-degree of any node is $O(B)$. We moreover require that the maximum in-degree of any node is d_{in}=$O(1)$. Access to a block of the ephemeral data structure (*access step* for the I/O model) causes in the worst case an overhead of $O(\log_2 d_{in})$ I/Os. The update overhead is $O(d_{in} \log_2 B)$ amortized I/Os and the space overhead is $O(d_{in}/B)$ amortized blocks, whenever an update operation makes a constant number of modifications to a node (*update step* for the I/O model). The gist of our method lies on the fact that whenever a node of the structure is accessed by a pointer traversal, the contents of the node for a particular version can be retrieved by at most a predefined number of version-id comparisons. In this way we manage to minimize the I/O-cost of an access step. To manage the persistent nodes of small size, we use a packed memory layout.

In Section 3 we present the *Incremental B-Trees*, an implementation of B-Trees where rebalancing operations due to insertions and deletions are performed *incrementally* over the sequence of succeeding updates. They use $O(n/B)$ blocks of space and support range searches in $O(\log_B n + t/B)$ I/Os. They support in-

sertions and deletions in $O(\log_B n)$ I/Os, and each update operation performs in the worst case $O(1)$ modifications to the tree. In a similar manner, Driscoll et al. applied the *lazy recoloring* technique [21] on *red-black trees* [3] in order to obtain fully persistent red-black trees with $O(\log n)$ amortized time per insertion and deletion, $O(\log n)$ worst case time per access, and $O(m)$ space. Our Incremental B-Trees can be seen as a generalization of this technique to B-Trees. The desired fully persistent B-Trees are achieved by applying to the Incremental B-Trees our method for I/O-efficient full persistence. Since Incremental B-Trees have d_{in}=1 and an update operation makes $O(1)$ modifications, it follows that updating the fully persistent Incremental B-Trees takes $O(\log_B n + \log_2 B)$ amortized I/Os. Fleischer [11] also presents (a, b)-Trees that make $O(1)$ modifications per update operation, however they have d_{in}=$\omega(1)$. Thus, applying our method to them yields less efficient fully persistent B-Trees.

2 Fully Persistent Data Structures in External Memory

In this section we present a generic method that makes a pointer based ephemeral data structure fully persistent in the I/O model, provided that every node of the underlying graph occupies at most a constant number of disk blocks, and the maximum in-degree d_{in} of any node is bounded by a constant. The overhead for accessing an ephemeral node is $O(\log_2 d_{in})$ I/Os in the worst case. The overhead for updating a field in an ephemeral node is $O(d_{in} \log_2 B)$ amortized I/Os and the space overhead is $O(d_{in}/B)$ amortized blocks.

In particular, we require an ephemeral data structure D to be represented by a graph where every *ephemeral node* u contains at most $c_f B$ *fields* for some constant c_f. Each field stores either an *element*, or a *pointer* to another ephemeral node. One ephemeral *entry node* provides access to the graph. An ephemeral node is *empty* if none of its fields contains elements or pointers.

The following interface provides the necessary operations to navigate and to update any version of the fully persistent data structure $\bar{D}$. The interface assumes that the user has only knowledge of the ephemeral structure. The i-th version of $\bar{D}$ is an ephemeral data structure D_i where all nodes, elements and pointers are associated with version i. A `field` is an identifier of a field of a node of D_i. The `value` of the `field` is either the element in the field at version i, or a `pointer` p_i to another node of D_i. Since the `pointer` resides in and points to nodes of the same version, we associate this version with the `pointer`. The `version` i is the unique identifier of D_i.

`pointer` p_i `= Access(version` i`)` returns a `pointer` p_i to the entry node of version i.

`value` x `= Read(pointer` p_i`, field` f`)` returns the `value` x of the `field` f in the node at version i pointed by `pointer` p_i. If x is a `pointer`, it points to a node also at version i.

`Write(pointer` p_i`, field` f`, value` x`)` writes the `value` x in the `field` f of the node at version i pointed by `pointer` p_i. If x is a pointer to a node, we require the pointer to be also at version i.

`pointer` p_i `= NewNode(version` i`)` creates a new empty node at version i and returns a `pointer` p_i to it.

`version` j `= NewVersion(version` i`)` creates a new version D_j that is a copy of the current version D_i, and returns a new version identifier for D_j.

By definition of full persistence, the outcome of updating D_i is a new ephemeral structure D_j, where the new version $j \neq i$ becomes a leaf of the version tree. The above interface allows the outcome of updating D_i to be D_i itself. In other words, it provides the extra capability of updating an internal node of the version tree. The user has to explicitly create a new version D_j before every update operation.

2.1 The Structure Our method is inspired by the node-splitting method [10] to which we make non-trivial modifications, such that whenever a node of the structure is accessed by a pointer traversal, the contents of the node for a particular version can be retrieved by at most a predefined number of `version` comparisons.

As defined by full persistence, all the versions of the ephemeral data structure can be represented by a directed rooted *version tree* T. If version j is obtained by modifying version i, version i is the parent of j in T. Similarly to [10] we store the preorder layout of T in a dynamic list that supports *order maintenance* queries [9, 20, 8, 6], called the **global version list (GVL)**. Given two versions i and j, an order maintenance query returns true if i lies before j in the list, and it returns false otherwise. To preserve the preorder layout of T whenever a new version is created, it is inserted in the GVL immediately to the right of its parent version. In this way, the descendants of every version occur consecutively in the GVL. By implementing the GVL as in [8], order maintenance queries are supported in $O(1)$ worst case time and I/Os. The insertion of a version is supported in $O(1)$ worst case time and I/Os, given a pointer to its parent version.

We record all the changes that occur to an ephemeral node u in a linked list of *persistent nodes* $\bar{u}$, called the **family** $\phi(u)$. Each node of $\phi(u)$ stores the

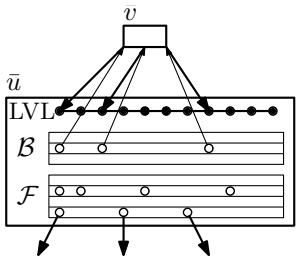


Figure 1: The persistent node $\bar{u}$. The versions stored in the local version list of $\bar{u}$ are represented by black dots. The values stored in $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$ are represented with white dots. The values lie on the column that corresponds to the version they are associated with. When the field in $\mathcal{F}(\bar{u})$ contains pointers, the values contain forward pointers that are represented with thick arrows. The values in $\mathcal{B}(\bar{u})$ contain backward pointers that are represented with thin arrows. They point to the persistent node $\bar{v}$ that contains the corresponding forward pointers. Forward pointers point to their associated version in the local version list of $\bar{u}$.

versions of u for a subinterval of the global version list GVL. To implement the linked list, the persistent node $\bar{u}$ contains a pointer $c(\bar{u})$ to the next persistent node in $\phi(u)$. For every field f of the corresponding ephemeral node u, the persistent node $\bar{u}$ stores a set $\mathcal{F}_f(\bar{u})$. If field f stores elements, then $\mathcal{F}_f(\bar{u})$ contains pairs (version i, value x) where x is the element stored in f at version i. Else, if field f stores pointers, then $\mathcal{F}_f(\bar{u})$ contains pairs (version i, pointer $\overrightarrow{p}$) where the *forward pointer* $\overrightarrow{p}$ corresponds to the ephemeral pointer p that is stored in f at version i. If p points to the node v at version i, then $\overrightarrow{p}$ points to the persistent node $\bar{v}$ in $\phi(v)$ that corresponds to node v at version i. For every persistent node $\bar{v}$ that contains forward pointers pointing to $\bar{u}$, the persistent node $\bar{u}$ stores a set $\mathcal{B}_{\bar{v}}(\bar{u})$ of pairs (version i, pointer $\overleftarrow{p}$) where the *backward pointer* $\overleftarrow{p}$ points to $\bar{v}$. Backward pointers do not correspond to ephemeral pointers and they are only used by the persistent mechanism to accommodate updates. The pairs in the sets $\mathcal{F}_f(\bar{u})$ and $\mathcal{B}_{\bar{v}}(\bar{u})$ are sorted with respect to the order of their first component (version i) in the GVL. We denote $\mathcal{F}(\bar{u})=\cup_{f\in\bar{u}}\mathcal{F}_f(\bar{u})$ and $\mathcal{B}(\bar{u})=\cup_{\bar{v}\to\bar{u}}\mathcal{B}_{\bar{v}}(\bar{u})$, where $\mathcal{F}_f(\bar{v})$ contains a forward pointer to $\bar{u}$. Finally, the persistent node $\bar{u}$ contains a *local version list* LVL($\bar{u}$) that stores all the versions i in the pairs of $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$, sorted with respect to their order in the GVL. The first version in the LVL($\bar{u}$) is the *version $i_{\bar{u}}$ of the persistent node $\bar{u}$*. Figure 1 illustrates a persistent node $\bar{u}$.

To provide access to the structure we maintain an **access array** whose i-th position store the pair (version i, pointer $\overrightarrow{p}$), where $\overrightarrow{p}$ is a forward pointer to the persistent node that corresponds to the entry node at version i, and i is a pointer to version i in the GVL. We define the *size* of a persistent node $\bar{u}$ to be the number of pairs in $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$. This dominates the number of versions in the LVL($\bar{u}$), since there exists at least one pair per version. We call a persistent node *small* if its size is at most $\frac{c_f}{2}B$. To utilize space efficiently, we pack all families of small size in an **auxiliary linked list**.

Our structure satisfies the following invariants:

Invariant 1. *Every set in $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$ contains a pair with the version $i_{\bar{u}}$ of the persistent node $\bar{u}$.*

Invariant 2. *The size of a persistent node $\bar{u}$ that is not stored in the auxiliary linked list is $\frac{c_f}{2}B \le |\bar{u}| \le c_{max}B$ for $c_{max} = \Omega(c_f(d_{in} + \frac{d_{in}^2}{B}))$.*

Invariant 3. *For every forward pointer $\overrightarrow{p}$ that points to the persistent node $\bar{v}$ and resides in a pair $(i, \overrightarrow{p})$ of $\mathcal{F}_f(\bar{u})$ or in a pair of the access array, there exists a pair $(i, \overleftarrow{p})$ in $\mathcal{B}_{\bar{u}}(\bar{v})$ where the backward pointer $\overleftarrow{p}$ points to the persistent node $\bar{u}$ or to the i-th position of the access array, respectively.*

Invariant 1 ensures that an ephemeral node u can be retrieved by accessing exactly one persistent node $\bar{u}$ in the family $\phi(u)$. Invariant 2 ensures that a persistent node $\bar{u}$ occupies at most a constant number of blocks. Invariant 3 associates a forward pointer $\overrightarrow{p}$ with a corresponding backward pointer $\overleftarrow{p}$. It moreover ensures that the version i of the forward pointer $\overrightarrow{p}$ belongs to the LVL of the pointed persistent node $\bar{v}$. A forward pointer $\overrightarrow{p}$ at version i points to the version i in the LVL($\bar{v}$). It suffices for backward pointer $\overleftarrow{p}$ to only point to the persistent node $\bar{u}$.

We define the *valid interval* of a pair (i, x) in $\mathcal{F}_f(\bar{u})$ to be the set of versions in the GVL for which field f has the particular value x. In particular, it is the interval of versions in the GVL from version i up to but not including version j. Version j is the version in the next pair of $\mathcal{F}_f(\bar{u})$, if this pair exists. Otherwise, j is the version in the first pair of $\mathcal{F}_f(c(\bar{u}))$, if $c(\bar{u})$ exists. Otherwise, the valid interval is up to the last version in the GVL. By Invariant 3 it follows that the valid interval of a pair $(i, \overrightarrow{p})$ in $\mathcal{F}_f(\bar{u})$, where $\overrightarrow{p}$ is a forward pointer to the persistent node $\bar{v}$, is identical to the valid interval of the pair $(i, \overleftarrow{p})$ in $\mathcal{B}_{\bar{u}}(\bar{v})$, where the backward pointer $\overleftarrow{p}$ corresponds to $\overrightarrow{p}$. A pair $(i, \overrightarrow{p})$ where the forward pointer $\overrightarrow{p}$ points to the persistent node $\bar{v}$, implements the `pointers` p_j that point to v at every version j that belongs to the valid interval of the pair. All versions occur in the access array, since `NewVersion` is called for every version created. Thus, the valid interval of

a pair $(i, \overrightarrow{p})$ in the access array is only version i. We define the *valid interval of a persistent node* $\bar{u}$ to be the union of the valid intervals of the pairs in $\mathcal{F}(\bar{u})$ and $\mathcal{B}(\bar{u})$. In particular, it is the interval of versions in the GVL from version $i_{\bar{u}}$ up to but not including version $i_{c(\bar{u})}$, if $c(\bar{u})$ exists. Otherwise, it is up to the last version in the GVL.

We define the *span of a forward pointer* $\overrightarrow{p}$ that points to the persistent node $\bar{v}$ to be the versions in the intersection of the valid interval of the pair that contains $\overrightarrow{p}$ with the LVL($\bar{v}$). The backward pointer $\overleftarrow{p}$ that corresponds to $\overrightarrow{p}$ has the same span as $\overrightarrow{p}$. Invariant 4 below ensures that whenever a persistent node is accessed by traversing a forward pointer, the content of the persistent node for a particular version can be retrieved by comparing against at most d versions (or $O(\log d)$ by binary searching).

INVARIANT 4. *Let $\pi \geq d_{in}+9$ be a constant. The size of the span of every forward pointer is $d \in \mathbb{N}$ where $1 \leq d \leq 2\pi$.*

2.2 Algorithms Here we present the implementation of the user-interface. Operations `Write`, `NewNode`, and `NewVersion` immediately restore Invariants 1 and 3. This may cause at most d_{in} forward pointers to violate Invariant 4 and some persistent nodes to violate Invariant 2. The auxiliary subroutine `Repair()` restores those invariants utilizing an auxiliary *violation queue*.

We say that a version j *precedes* version i in the local version list LVL, if j is the rightmost version in the LVL that is not to the right of version i in the global version list GVL. Note that version i precedes itself when it belongs to the set. We denote by i^+ the version immediately to the right of version i in the GVL.

pointer p_i = Access(version i). We return the forward pointer in the i-th position of the access array, since it points to the entry node at version i.

value x = Read(pointer p_i, field f). Let `pointer` p_i point to the ephemeral node u at version i. Let $\bar{u}$ be the persistent node in $\phi(u)$ whose valid interval contains version i. To return the value x that field f has in the ephemeral node u at version i, we locate the pair in $\mathcal{F}_f(\bar{u})$ whose valid interval contains version i. Figure 2 illustrates the setting for the operation `Read`.

The pairs in $\mathcal{F}(\bar{u})$ whose valid intervals contain version i, also contain the version j that precedes version i in the LVL($\bar{u}$). We determine j by searching in the LVL($\bar{u}$) as following. Let the pair $(i', \overrightarrow{p})$ contain the forward pointer that implements `pointer` p_i. By Invariant 3 version i' belongs to the LVL($\bar{u}$). Since version i belongs to the valid interval of this pair, version i' lies to the left of version i in the GVL. If $i' \neq j$, then version j lies to the right of version i' in the LVL($\bar{u}$). Version j belongs to the span of $\overrightarrow{p}$.

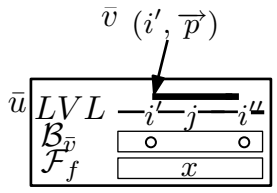


Figure 2: Operation `Read`(p_i,f). The thick arrow represents the forward pointer $\overrightarrow{p}$ that implements `pointer` p_i of node v at `version` i. The white dot represents the backward pointers that point to $\bar{v}$. The thick horizontal line represents the span of $\overrightarrow{p}$. We assume that version i does not belong to the LVL($\bar{u}$).

We perform a binary search over the versions of the span in $\overrightarrow{p}$ in the LVL($\bar{u}$). Every comparison is implemented by an order maintenance query between the accessed version in the span and version i. In this way, we locate the rightmost version j in the span for which the order maintenance query returns true. At least one order maintenance query returns true, since version i' lies to the left of version i in the GVL. We find the pair of $\mathcal{F}_f(\bar{u})$ with the version that precedes version j in the LVL($\bar{u}$), and return the value it contains.

Write(pointer p_i, field f, value x). Let `pointer` p_i point to the ephemeral node u. Let $\bar{u}$ be the persistent node in $\phi(u)$ whose valid interval contains version i. As in `Read`, we find the version j that precedes version i in the LVL($\bar{u}$), and the pair (j', y) in $\mathcal{F}_f(\bar{u})$ whose valid interval contains version i. Figure 3 illustrates the setting before and after the operation `Write`.

If $j'=i$, we merely replace y with x, and add pair (i^+, y) to $\mathcal{F}_f(\bar{u})$. In this case, version i is the currently updated version and it belongs to the LVL($\bar{u}$). Otherwise, we add both the pairs (i, x) and (i^+, y) to $\mathcal{F}_f(\bar{u})$. By this way version i belongs only to the valid interval of the pair (i, x). Moreover, the versions that belonged to the valid interval of the pair (j', y) and succeed version i in the GVL, continue having the previous value y. If there is already a pair in $\mathcal{F}_f(\bar{u})$ with version i^+, it suffices to only add the pair (i, x). If $\mathcal{F}_f(\bar{u})$ is empty, we add the pairs $(i_{\bar{u}}, null)$, (i, x) and $(i^+, null)$ instead, where $i_{\bar{u}}$ is the version of the persistent node $\bar{u}$.

Version i is inserted in LVL($\bar{u}$) immediately to the right of version j. Unless version i^+ already exists in the LVL($\bar{u}$), i^+ is inserted immediately to the right of version i. These insertions may cause at most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ that point to $\bar{u}$ to violate Invariant 4. The persistent nodes that contain them have to be inserted to the violation queue. To find the forward pointers $\overrightarrow{P_{\bar{u}}}$, we determine the corresponding

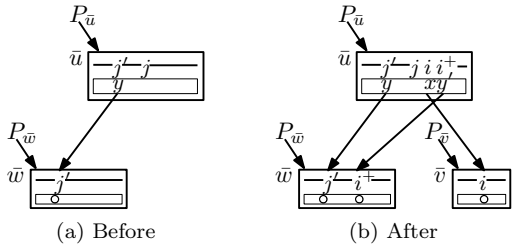
backward pointers in $\bar{u}$. In particular, we find all the pairs $(k, \overleftarrow{p})$ in $\mathcal{B}(\bar{u})$ whose valid intervals contain the inserted versions, and check if there are more than 2π versions in LVL($\bar{u}$) between version k and the version of the next pair in $\mathcal{B}_{\bar{z}}(\bar{u})$. If so, we access the persistent node $\bar{z}$ pointed by $\overleftarrow{p}$ and mark the pair in $\mathcal{F}_f(\bar{z})$ with the corresponding forward pointer. We insert $\bar{z}$ to the violation queue, unless it is already there.

If x is a `pointer` to an ephemeral node v at version i, the argument `pointer` x_i is implemented by a forward pointer $\overrightarrow{x}$ to the persistent node $\bar{v}$ in $\phi(v)$ whose valid interval contains version i. Version i belongs to the span of $\overrightarrow{x}$. We add to $\mathcal{F}_f(\bar{u})$ the pairs $(i, \overrightarrow{x})$ and $(i^+, \overrightarrow{y}')$ instead, where $\overrightarrow{y}'$ is a forward pointer to the persistent node $\bar{w}$ pointed by the forward pointer $\overrightarrow{y}$ of the pair $(j', \overrightarrow{y})$ in $\mathcal{F}_f(\bar{u})$. We restore Invariant 3 for the added pair $(i, \overrightarrow{x})$ by inserting the corresponding backward pointer $\overleftarrow{x}$ to $\mathcal{B}(\bar{v})$. In particular, we add the pair $(i, \overleftarrow{x})$ to $\mathcal{B}_{\bar{u}}(\bar{v})$, where the backward pointer $\overleftarrow{x}$ points to the persistent node $\bar{u}$ and corresponds to $\overrightarrow{x}$. If $\mathcal{B}_{\bar{u}}(\bar{v})$ contains only the pair $(i, \overleftarrow{x})$, then we also add the pair $(i^+, null)$ in order to determine the span of $\overrightarrow{x}$ without traversing $\overleftarrow{x}$. We perform a binary search in the span of $\overrightarrow{x}$ in order to find the version in the LVL($\bar{v}$) that precedes version i. Unless it is i itself, we insert version i immediately to the right of it. The at most d_{in} forward pointers $\overrightarrow{P_{\bar{v}}}$ that violate Invariant 4 are processed as described above. We set $\overrightarrow{x}$ to point to version i in the LVL($\bar{v}$). The added pair implements `pointer` x_i. If $(i^+, \overrightarrow{y}')$ was also added, we restore Invariant 3 for $\overrightarrow{y}'$ as described above. Version i^+ belongs to the span of $\overrightarrow{y}$. The at most d_{in} forward pointers $\overrightarrow{P_{\bar{w}}}$ may violate Invariant 4. Notice that $\bar{u}$ may contain a pointer from $\overrightarrow{P_{\bar{v}}}$ or $\overrightarrow{P_{\bar{w}}}$.

The insertion of pairs in the persistent nodes $\bar{u}$, $\bar{v}$ and $\bar{w}$ increases their size. If the nodes are not small anymore due to the insertion, we remove them from the auxiliary linked list and move them to an empty block. If they violate Invariant 2, we insert them to the violation queue, unless they are already there. Finally, `Repair` is called.

pointer p_i = NewNode(version i). We create a new family $\phi(u)$ which consists of one empty persistent node $\bar{u}$. We insert version i to the LVL($\bar{u}$), so that $\bar{u}$ satisfies Invariant 1. All fields of $\bar{u}$ are emprty. Node $\bar{u}$ is added to the auxiliary linked list since it is small. We return a forward pointer to version i in the LVL($\bar{u}$).

version j = NewVersion(version i). We traverse the pointer stored at the i-th position of the access array to find the position of version i in the GVL. We insert version j immediately to the right of version i in the GVL. We insert in the j-th position of the access array a pointer to version j in the GVL. Let $\bar{u}$ be the persistent node pointed by the forward pointer in the i-th position of the access array. We insert in the j-th position of the access array a forward pointer $\overleftarrow{p}$ to $\bar{u}$, and add a pair $(j, \overleftarrow{p})$ to $\mathcal{B}(\bar{u})$ where the backward pointer $\overleftarrow{p}$ points to the j-th position of the access array and corresponds to $\overrightarrow{p}$. At most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ that point to $\bar{u}$ may violate Invariant 4. If $\bar{u}$ violates Invariant 2 we insert it to the violation queue, unless it is already there. Finally, `Repair` is called.



(a) Before (b) After

Figure 3: Operation `Write` inserts to the ephemeral node u at version i a pointer x that points to the ephemeral node v at version i. The figures show how the corresponding persistent nodes $\bar{u}$, $\bar{v}$ and $\bar{w}$ look before and after the operation. The persistent node $\bar{w}$ is the node that is pointed by the forward pointer of the pair in $\mathcal{F}_f(\bar{u})$ whose valid interval contains version i, where f is the field in $\bar{u}$ that contains the inserted pointer. The local version lists are represented by horizontal lines.

Repair() iteratively pops a persistent node $\bar{u}$ from the violation queue, and restores Invariant 4 for the forward pointers in the marked pairs of $\mathcal{F}_f(\bar{u})$ and Invariant 2 for $\bar{u}$. These invariants may in turn be violated in other persistent nodes, which we insert in the violation queue as well. This iteration terminates when the queue becomes empty.

To restore Invariant 4 for the forward pointer in the marked pair $(i, \overrightarrow{p})$ in $\mathcal{F}_f(\bar{u})$, we reset the size of its span to π as following. Let $\overrightarrow{p}$ point to the persistent node $\bar{v}$. We find the version j in the span of $\overrightarrow{p}$ that resides π positions to the right of version i in the LVL($\bar{v}$). We set the forward pointer $\overrightarrow{p}'$ to version j in the LVL($\bar{v}$), and add the pair $(j, \overrightarrow{p}')$ to $\mathcal{F}_f(\bar{u})$. If the span of $\overrightarrow{p}'$ violates Invariant 4, we mark its pair. We restore Invariant 3 for the added pair as described in `Write`. Node $\bar{v}$ may violate Invariant 2. We find the version that precedes version j in the LVL($\bar{u}$), by a binary search over the *whole* LVL($\bar{u}$). We insert j immediately to the right of its preceding version, unless it already exists. This may cause at most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ to violate Invariant 4. Node $\bar{u}$ may violate Invariant 2.

To restore Invariant 2 for the persistent node $\bar{u}$, we split it into two persistent nodes, such that the right one has size approximately $\frac{c_{max}}{2}B$. We first determine the

version j at which we will split $\bar{u}$, by scanning LVL($\bar{u}$) from right to left. Version j is the leftmost version in the LVL($\bar{u}$), such that the number of pairs whose version succeeds j is less than $\frac{c_{max}}{2}B$. Unless $j' = j$, for every pair (j', x) in $\bar{u}$ whose valid interval contains version j, we add a pair (j, x) in $\bar{u}$. If x is a forward pointer to a persistent node $\bar{v}$, we restore Invariant 3 as described in `Write`. If x is a backward pointer to $\bar{v}$, restoring Invariant 3 involves a binary search for the version that precedes j' in the LVL($\bar{v}$). Node $\bar{v}$ may violate Invariant 2. Moreover, at most d_{in} forward pointers $\overrightarrow{P_{\bar{v}}}$ may violate Invariant 4. We create a new persistent node $\bar{u}'$ that succeeds $\bar{u}$ in the family $\phi(u)$, by setting $c(\bar{u}')$=$c(\bar{u})$ and $c(\bar{u})$=$\bar{u}'$. We split the LVL($\bar{u}$) at version j. The right part becomes LVL($\bar{u}'$). Version j becomes the version of $\bar{u}'$. All the pairs in $\bar{u}$ with a version in LVL($\bar{u}'$) are moved to $\bar{u}'$. We traverse all forward and backward pointers in $\mathcal{F}(\bar{u}')$ and $\mathcal{B}(\bar{u}')$ in order to update the corresponding backward and forward pointers to point to $\bar{u}'$, respectively. Version j becomes the version of $\bar{u}'$. The node $\bar{u}'$ satisfies Invariant 1 due to the addition of the pairs (j, x).

2.3 Analysis In this subsection we prove the following theorem.

THEOREM 2.1. *Let D be a pointer-based ephemeral data structure that supports queries in $O(q)$ worst case I/Os and where updates make $O(u)$ modifications to the structure in the worst case. Given that every node of D occupies at most $\lceil c_f \rceil$ blocks, for a constant c_f, D can be made fully persistent such that a query to a particular version is supported in $O(q(c_{max}+\log_2 \pi))$ worst case I/Os, and an update to any version is supported in $O(d_{in}(c_{max} + \log_2(c_{max}B)) + \log_2 \pi)$ amortized I/Os, where $d_{in} = O(1)$ is the maximum in-degree of any node in D, $c_{max} = \Omega\left(c_f\left(d_{in} + \frac{d_{in}^2}{B}\right)\right)$ is the number of blocks occupied by a node of $\bar{D}$, and $\pi \geq d_{in}+9$ is a constant.After performing a sequence of m updates, the fully persistent structure occupies $O(u\frac{m}{B})$ blocks of space.*

The following remarks are necessary for the analysis. A version in the LVL($\bar{u}$) belongs to the span of at most d_{in} forward pointers that point to $\bar{u}$, and thus it belongs to the valid interval of at most d_{in} pairs in $\mathcal{B}(\bar{u})$.

LEMMA 2.1. *After splitting a persistent node $\bar{u}$ the size of $\bar{u}'$ is within the range $[(\frac{c_{max}}{2}-c_f)B, \left(\frac{c_{max}}{2}+c_f\right)B+d_{in}-1]$*

Proof. The number of pairs with version j and with versions that succeed version j in LVL($\bar{u}$) before the split is at least $(\frac{c_{max}}{2} - c_f)B$. This sets the lower bound. The number of pairs with a version that succeeds j in the LVL($\bar{u}$) is at most $\frac{c_{max}}{2}B-1$. There are at most c_fB pairs with a single version in $\mathcal{F}(\bar{u})$, and at most d_{in} pairs in $\mathcal{B}(\bar{u})$ whose valid interval contains version j. We add one pair for each of them to $\bar{u}'$. This sets the upper bound. □

First we analyze the worst case cost of every operation. `Access` performs $O(1)$ I/Os to the access array. `Read` performs $O(c_{max})$ I/Os to load the persistent node $\bar{u}$ into memory, and $O(\log_2 2\pi)$ I/Os for the order maintenance queries to the data structure of [8] in order to determine the appropriate version j. `Write` performs $O(c_{max})$ I/Os to load $\bar{u}$ as well as $O(\log_2 2\pi)$ I/Os to locate the proper version as in `Read`. $O(1)$ I/Os are needed to access version i^+ in the GVL, and $O(d_{in}c_{max})$ I/Os are needed to access the forward pointers of $\overrightarrow{P_{\bar{u}}}$. If the written value is a pointer, then we also need $O(\log_2 2\pi)$ I/Os to process $\bar{v}$ and $\bar{w}$, and $O(d_{in}c_{max})$ I/Os to access the forward pointers of $\overrightarrow{P_{\bar{v}}}$ and $\overrightarrow{P_{\bar{w}}}$. In total, without taking into account the call to `Repair`, the worst case cost is $O(d_{in}c_{max}+\log_2 2\pi+c_{max}+1)$ I/Os. `NewNode` makes$O(c_{max})$ I/Os to access the entry node. `NewVersion` spends $O(1)$ I/Os to update the GVL, and $O(c_{max})$ I/Os to process $\bar{u}$ and to insert $\bar{u}$ to the violation queue. To restore Invariant 4 for one forward pointer `Repair` makes $O(c_{max})$ I/Os to load $\bar{u}$ and $\bar{v}$ into memory, $O(\log_2(c_{max}B))$ I/Os to insert version j to the LVL($\bar{v}$), and $O(d_{in}c_{max})$ I/Os to insert the nodes with $\overrightarrow{P_{\bar{v}}}$ to the violation queue. In total the worst case cost is $O(c_{max}(d_{in}+2)+\log_2(c_{max}B))$ I/Os. To restore Invariant 2 for a persistent node, `Repair` makes $O(c_{max})$ I/Os to load the node into memory, $O(c_fB(c_{max}+\log_2 2\pi))$ I/Os to restore Invariant 3 for the added pairs in $\bar{u}$ with forward pointers, $O(d_{in}(c_{max}+\log_2(c_{max}B)))$ I/Os to restore Invariant 3 for the added pairs in $\bar{u}$ with backward pointers, $O((c_fB+d_{in})d_{in}c_{max})$ I/Os to insert the nodes with $\overrightarrow{P_{\bar{v}}}$ to the violation queue, and $O(((\frac{c_{max}}{2})B-1)c_{max})$ I/Os to set the forward and backward pointers to point to $\bar{u}'$. In total the worst case cost is $O\Big(B\big(\frac{c_{max}^2}{2}+c_f(c_{max}(d_{in}+1) + \log_2 2\pi)\big) + d_{in}\big(c_{max}(d_{in}+1)+\log_2(c_{max}B)\big)\Big)$ I/Os.

Let D_i be the persistent structure after the i-th operation. We define the potential of D_i to be $\Phi(D_i)=\sum_{\overrightarrow{p}\in\mathcal{P}} \Xi(\overrightarrow{p})+\sum_{\bar{u}\in\overline{\mathcal{U}}}\Psi(\bar{u})$, where $\mathcal{P}$ is the set of all forward pointers and $\overline{\mathcal{U}}$ is the set of all persistent nodes in D_i. The function $\Xi(\overrightarrow{p})=\max\{0, |\overrightarrow{p}|-\pi\}$ provides the potential to the forward pointer $\overrightarrow{p}$ for the splitting of its span. By $|\overrightarrow{p}|$ we denote the size of the span of $\overrightarrow{p}$. Function $\Psi(\bar{u})$ =

$\max\left\{0, 3\left(|\bar{u}| - \left(\left(\frac{c_{max}}{2}+c_f\right)B+d_{in}\right)\right)\right\}$ provides the potential to the persistent node $\bar{u}$ for its split. By $|\bar{u}|$ we denote the size of the persistent node $\bar{u}$.

Operation `Write`, without the call to `Repair`, increases $\Psi(\bar{u})$, $\Psi(\bar{v})$ and $\Psi(\bar{w})$ by at most 12 in total. The potential of the at most $2d_{in}$ forward pointers of $\overrightarrow{P_{\bar{u}}}$ and $\overrightarrow{P_{\bar{v}}}$ is increased by at most 2. The potential of the at most d_{in} forward pointers and $\overrightarrow{P_{\bar{w}}}$ is increased by at most 1. In total the potential increases by $12+5d_{in}$. Operation `NewVersion` increases $\Psi(\bar{u})$ by 3 and the potential of at most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ by 1. In total the potential increases by $3+d_{in}$. When operation `Repair` restores Invariant 4 for one marked pair it increases $\Psi(\bar{u})$ and $\Psi(\bar{v})$ by 6 in total, and the potential of the at most d_{in} forward pointers $P_{\bar{u}}$ by at most 1. The potential of the forward pointer in the pair is decreased by π. In total the potential changes by $d_{in}+6-\pi$. When operation `Repair` restores Invariant 2 for the persistent node $\bar{u}$, it increases $\Psi(\bar{u})$ by at most $3c_fB$ due to the added pairs with elements and forward pointers, and by at most $3d_{in}$ due to the added pairs with backward pointers. The addition of the corresponding backward and forward pointers increases the potential of at most c_fB+d_{in} persistent nodes by 3. The potential of at most $d_{in}(c_fB+d_{in})$ forward pointers to these persistent nodes is increased by 1. After the split, the potential decreases by $3(\frac{c_{max}}{2}-c_f)B$ by the lower bound in Lemma 2.1. In total the potential changes by $B(c_f(9+d_{in})-\frac{3}{2}c_{max})+d_{in}(d_{in}+6)$.

Let D_i be the result of executing `Write` or `NewVersion` on D_{i-1}, followed by a `Repair` operation. We assume that a version and a pair fit in a field of a block. A modification of a field involves adding a pair or a version in a persistent node, or changing the value in a pair. The number of modifications caused by `Repair` bounds asymptotically the number of persistent nodes it accesses. The amortized number of fields modified by `Repair` is $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, where c_i is the real number of modifications in D_{i-1}. If `Repair` restores Invariant 4 for α forward pointers, the amortized number of modified fields is

$$\alpha\left(3+(d_{in}+6-\pi)\right).$$

This is because we add one version and two pairs. It is non-positive for $\pi \geq d_{in}+9$. If `Repair` restores Invariant 2 for β persistent nodes, the amortized number of modified fields is

$$\beta\Bigg(3c_fB + 3d_{in} + 2B\left(\frac{c_{max}}{2}+c_f\right) + 2d_{in} - 2 + \\ +B\Big(c_f(9+d_{in}) - \frac{3}{2}c_{max}\Big) + d_{in}(d_{in}+6)\Bigg).$$

This is because we add at most c_fB pairs with forward pointers in the node and one corresponding backward pointer and version at the node pointed by each of these forward pointers. We add at most d_{in} pairs with backward pointers in the node and one corresponding forward pointer and version at the node pointed by each backward pointer. We transfer at most $(\frac{c_{max}}{2}+c_f)B+d_{in}-1$ pairs to $\bar{u}'$ and update as many pointers. It is non-positive for $c_{max} \geq c_f(28+2d_{in})+2\frac{d_{in}^2+11d_{in}-2}{B}$. The amortized number of fields modified by `Write` is $8+12+5d_{in}$. This is because we add at most 4 versions and 4 pairs. The amortized number of fields modified by `NewVersion` is $2+3+d_{in}$. This is because we add at most one version and one pair. Thus, the amortized number of fields modified by `Write` and `NewVersion` is $O(d_{in})$, which implies an $O(d_{in}(c_{max}+\log_2\pi))$ amortized I/O-cost per modification, since $O(c_{max})$ I/Os are needed to load a node in memory and $O(\log_2 \pi)$ I/Os are needed to insert a version to the span. Moreover, when we insert a version to the local version list of a persistent node that is accessed by a backward pointer, we need $O(\log_2(c_{max}B))$ I/Os. Thus, we charge every modified field with $O(\log_2 c_{max}B)$ I/Os. The total cost for an update step is $O(d_{in}(c_{max}+\log_2 c_{max}B)+\log_2\pi)$ amortized I/Os. Since an update operation makes $O(d_{in})$ amortized number of fields modifications and since all small blocks are packed in the auxiliary linked list, it follows that the space usage after m operations is $\Theta(d_{in}\frac{m}{B})$.

3 Incremental B-Trees

In this section we design B-Trees [4, 7, 14] that use $O(n/B)$ disk blocks of space, support insertions and deletions of elements in $O(\log_B n)$ I/Os, and range queries in $O(\log_B n + t/B)$ I/Os. They are designed such that an update makes in the worst case $O(1)$ modifications to the tree. This is achieved by marking unbalanced nodes and by incrementally performing the expensive rebalancing operations of ordinary B-Trees over the sequence of succeeding updates.

Before we describe our B-Trees, we briefly recall the properties of ordinary B-Trees [4, 7, 14]. All the nodes of a B-Tree, except possibly the root, have degree $\Theta(B)$. The tree has height $O(\log_B n)$ when n elements are stored in it. Range searching is supported in $O(\log_B n + t/B)$ I/Os and inserting and deleting an element in $O(\log_B n)$ I/Os. The latter operations might cause some nodes to exceed the upper and lower bounds of the degree. Thus, updates perform *rebalancing operations* to restore the bounds. These are *splitting* a node into two nodes of almost equal degree, *fusing* two low degree nodes into one, and moving children from a high degree node to a low degree node (*share*). In

ordinary implementations of B-Trees, a single update might cause the rebalancing operations to cascade up on a path of the tree, causing $O(\log_B n)$ I/Os in the worst case. In particular, insertions of elements in the leaves might cause cascaded splits on a leaf-to-u path, where u is an ancestor node of the leaf in the tree. Similarly deletions might cause cascaded fusions on a leaf-to-u path, possibly followed by a share at the parent of u.

3.1 The Structure An *Incremental B-Tree* is a rooted tree with all leaves on the same level. Each element is stored exactly once in the tree, either in a leaf or in an internal node. In the latter case it acts as a *search key*. An internal node u with k children stores a list $[p_1, e_1, p_2, \ldots, e_{k-1}, p_k]$ of $k-1$ elements $e_1, \ldots, e_{k-1}$ stored in non-decreasing order and k children pointers $p_1, \ldots, p_k$. The discussion that follows shows that $\frac{B}{2}-1 \leq k \leq 2B+1$. If x_i is an element stored in the i-th subtree of u, then $x_1 < e_1 < x_2 < e_2 < \cdots < e_{k-1} < x_k$ holds.

To handle the rebalancings of the tree incrementally, we mark the nodes to be rebalanced. In particular, each node can either be *unmarked* or it contains one of the following marks:

Overflowing mark: The node should be replaced by two nodes.

Splitting mark: The node w is being *incrementally split* by moving elements and children pointers to its unmarked right sibling w'. We say that nodes w and w' define an *incremental splitting pair*.

Fusion mark: The node w is being *incrementally fused* by moving elements and children pointers to its unmarked right sibling w'. In case w is the rightmost child of its parent, then w' is its unmarked left sibling and elements and children pointers are moved from w' to w. We say that nodes w and w' define an *incremental fusion pair*.

All kinds of marks can be stored in the nodes explicitly. However, we cannot afford to explicitly mark all unbalanced nodes since an update operation may unbalance more than a constant number of them. We can also store overflowing and fusion marks implicitly, based on the observation that the unbalanced nodes occur consecutively in a path of the tree. In particular, for a $u \rightarrow v$ path in the tree, where u is an ancestor of v and all nodes in the path have overflowing marks, we can represent the marks implicitly, by marking u with an overflowing mark and additionally storing in u an element of v. The rest of the nodes in the path have no explicit mark. This defines an *overflowing path*. Similarly, we can represent paths of nodes with fusion marks, which defines a *fusion path*.

Unmarked nodes that do not belong to incremental pairs are called *good* nodes. We define the *size* s_u of an internal node u to be the number of good children plus twice the number of its children with an overflowing mark minus the number of its children with a fusion mark. The size of a leaf is the number of elements in it. Conceptually, the size of an internal node is the degree that the node would have, when the incremental rebalancing of its children has been completed. The advance of the incremental rebalancing is captured by the following invariants.

INVARIANT 5. *An incremental splitting pair (w, w') with sizes s_w and $s_{w'}$ respectively satisfies $2 \cdot |s_w + s_{w'} - 2B - 1| \leq s_{w'} < s_w$. Node w is explicitly marked with a splitting mark and node w' is unmarked.*

The left inequality of Invariant 5 ensures that the incremental split terminates before the resulting nodes may participate in a split or a fusion again. In particular, it ensures that the number of the transferred elements and children pointers from w to w' is at least twice the number of insertions and deletions that involve the nodes of the splitting pair since the beginning of the incremental split. This allows for the transfer of one element and one child pointer for every such insertion and deletion. The right inequality of Invariant 5 ensures that the incremental split terminates, since the size of w' increases and the size of w decreases for every such insertion and deletion.

INVARIANT 6. *An incremental fusion pair (w, w') with sizes s_w and $s_{w'}$ respectively, where elements and children pointers are moved from w to w', satisfies $0 < s_w \leq \frac{B}{2} + 3 - 2 \cdot |s_w + s_{w'} - B + 1|$. Node w is explicitly marked with a fusion mark and node w' is unmarked.*

Conversely, the right inequality of Invariant 6 ensures that the incremental fusion terminates before the resulting node may participate in a split or a fusion again. The left inequality of Invariant 6 ensures that the incremental fusion terminates, since the size of w decreases for every insertion and deletion that involve the nodes of the incremental pair.

INVARIANT 7. *Except for the root, all good nodes have size within $[B/2, 2B]$. If the root is unmarked, it has at least two children and size at most $2B$.*

It follows from the invariants that the root of the tree cannot have a splitting or a fusion mark, since no sibling is defined. It can only have an overflowing mark or be unmarked. The following invariants are maintained with respect to the incremental paths.

INVARIANT 8. *Let $u \rightarrow v$ be an overflowing path. All nodes of the path have size $2B+1$. Node u is explicitly marked with an overflowing mark, and the rest of the nodes are implicitly marked with an overflowing mark.*

Invariant 8 implies that a node with an overflowing mark has size $2B+1$.

INVARIANT 9. *Let $u \rightarrow v$ be a fusion path. All nodes of the path have size $B/2$. Node u is explicitly marked with a fusion mark, and the rest of the nodes are implicitly marked with a fusion mark.*

Invariant 9 implies that a node with an implicit fusion mark has size $B/2$. A node with an explicit fusion mark may have size $B/2$ or belong to an incremental fusion pair.

INVARIANT 10. *All overflowing and fusion paths are node-disjoint.*

LEMMA 3.1. *The height of the Incremental B-Tree with n elements is $O(\log_B n)$.*

Proof. We transform the Incremental B-Tree into a tree where all incremental operations are completed and thus all nodes are unmarked. We process the marked nodes bottom-up in the tree and replace them by unmarked nodes, such that when processing a node all its children are already unmarked. A node with an overflowing mark that has size $2B+1$ is replaced by two unmarked nodes of size B and $B+1$ respectively. The two nodes in an incremental splitting pair (w, w') are replaced by two nodes, each containing half the union of their children. More precisely, they have sizes $\lfloor \frac{s_w + s_{w'}}{2} \rfloor$ and $\lceil \frac{s_w + s_{w'}}{2} \rceil$ respectively. By Invariant 5 we derive that $\frac{8}{5}B \leq s_w + s_{w'}$, i.e. each of the nodes has degree at least $B/2$. The two nodes in an incremental fusion pair (w, w') are replaced by a single node that contains the union of their children and has size $s_w + s_{w'}$. By Invariant 6 we derive that $\frac{3}{4}B - 1 \leq s_w + s_{w'}$. In all cases the nodes of the transformed tree have degree at least $B/2$, thus its height is $O(\log_B n)$. The height of the transformed tree is at most the height of the initial tree minus one. It may be lower than that of the initial tree, if the original root had degree two and its two children formed a fusion pair. □

3.2 Algorithms The insertion and deletion algorithms use the *explicit mark* and the *incremental step* algorithms as subroutines. The former maintains Invariant 10 by transforming implicit marks into explicit marks. The latter maintains Invariants 5 and 6 by moving at most four elements and child pointers between the nodes of an incremental pair, whenever an insertion or a deletion involve these nodes.

In particular, let $u \rightarrow v$ be an implicitly defined overflowing (resp. fusion) path where u is an ancestor of v in the tree. That is, all marks are implicitly represented by marking u explicitly with an overflowing (resp. fusion) mark and storing in u an element e of v. Let w be a node on $u \rightarrow v$, and w_p, w_c be its parent and child node in the path respectively. Also, let e_p be an element in w_p. The subroutine *explicit mark* makes the mark on w explicit, by breaking the $u \rightarrow v$ path into three node-disjoint subpaths $u \rightarrow w_p$, w, and $w_c \rightarrow v$. This is done by replacing the element at u with e_p, explicitly setting an overflowing mark on w, and explicitly setting an overflowing mark together with the element e in w_c. If $u = w$ or $w = v$, then respectively the first or the third subpath is empty.

The *incremental step algorithm* is executed on a node w that belongs to a fusion or a splitting pair (w, w'), or on an overflowing node w. In the latter case, we first call the procedure *explicit mark* on w. Then, we mark it with an incremental split mark and create a new unmarked right sibling w', defining a new incremental splitting pair. The algorithm proceeds as in the former case, moving one or two children from w to w', while preserving consistency for the search algorithm. Note that the first moved child causes an element to be inserted to the parent of w, increasing its size.

In the former case, the rightmost element e_k and child p_{k+1} of w are moved from w to w'. If the special case of the fusion mark definition holds, they are moved from w' to w. Let w_p be the common parent of w and w', and let e_i be the element at w_p that separates w and w'. If p_{k+1} is part of an overflowing or a fusion path before the move, we first call *explicit mark* on it. Next, we delete e_k and p_{k+1} from w, replace e_i with e_k, and add p_{k+1} and e_i to w'. If p_{k+1} was part of a splitting or fusion pair, we repeat the above once again so that both nodes of the pair are moved to w'. We also ensure that the left node of the pair is marked with an incremental fusion mark, and that the right node is unmarked. Finally, if the algorithm causes $s_{w'} \geq s_w$ for a splitting pair (w, w'), the incremental split is complete and thus we unmark w. It is also complete if it causes $s_w = 0$ for a node w of a fusion pair. Thus, we unmark the nodes of the pair, possibly first marking them explicitly with a fusion mark and dismissing the empty node w from being a child of its parent.

Insert The insertion algorithm inserts one new element e in the tree. Like in ordinary B-Trees, it begins by searching down the tree to find the leaf v in which the element should be inserted, and inserts e in v as soon as it is found. If v is marked, we perform two incremental

steps at v and we are done. If v is unmarked and has size at most $2B$ after the insertion, we are done as well. Finally, if v has size $2B+1$ it becomes overflowing. We define an overflowing path from the highest ancestor u of v, where all the nodes on the $u \rightarrow v$ path have size exactly $2B$, are unmarked and do not belong to an incremental pair. We do this by explicitly marking u with an overflowing mark and inserting element e in it as well. This increases the size of u_p, the parent of u. We perform two incremental steps to u_p, if it is a marked node or if it is an unmarked node that belongs to an incremental pair. Otherwise, increasing the size of u_p leaves it unmarked and we are done. Note that in order to perform the above algorithms, the initial search has to record node u_p, the topmost ancestor and the bottommost node of the last accessed implicitly marked path, and the last accessed explicitly marked node.

Delete The deletion algorithm removes an element e from the tree. Like in ordinary B-Trees, it begins by searching down the tree to find node z that contains e, while recording the topmost and the bottommost node of the last accessed implicitly marked path and the last accessed explicitly marked node. If z belongs to an overflowing or a fusion path, it explicitly marks it. If z is not a leaf, we then find the leaf v that stores the successor element e' of e. Next, we swap e and e' in order to guarantee that a deletion always takes place at a leaf of the tree. If v belongs to an overflowing or a fusion path, we mark it explicitly as well. The explicit markings are done in order to ensure that e and e' are not stored as implicit marks in ancestors of z or v.

We then delete e from v. If v is good and has size at least $B/2$ after the deletion, then we are done. If v is overflowing or belongs to an incremental pair, we perform two incremental steps on v and we are done. Otherwise, if leaf v is unmarked and has size $B/2-1$ after the deletion, we check its right sibling v'. If v' is overflowing or belongs to an incremental pair, we perform two incremental steps on v', move the leftmost child of v' to v and we are done. Only the move of the leftmost child suffices when v' is good and has degree more than $B/2+1$. Finally, if v' is good and has size at most $B/2+1$, we begin a search from the root towards v in order to identify all its consecutive unmarked ancestors u of size $B/2$ that have a good right sibling u' of size at most $B/2+1$. We act symmetrically for the special case of the fusion pair.

Let u_p be the node where the search ends and u be its child that was last accessed by this search towards v. We implicitly mark all the nodes on the $u \rightarrow v$ path as fusion pairs by setting a fusion mark on u and storing an element of v in u. We next check node u_p. If it is unmarked and has size greater than $B/2$, defining the fusion path only decreases the size of u_p by one, hence we are done. If node u_p is marked, we additionally apply two incremental steps on it and we are done. If u_p is good and has size $B/2$, and its sibling u'_p is good and has size bigger than $B/2+1$, we move the leftmost child of u'_p to u_p. This restores the size of u'_p back to $B/2$ and we are done. Finally, if node u_p is good and has size $B/2$, but its sibling is marked or belongs to an incremental pair, we explicitly mark u_p and move the leftmost child of u'_p to u_p. Next, we apply two incremental steps on u'_p.

Range Search A range search is implemented as in ordinary B-Trees. It decomposes into two searches for the leaves that contain the marginal elements of the range, and a linear scan of the leaves that lie in the range interleaved with a inorder traversal of the search keys in the range.

3.3 Correctness & Analysis

We show that the update algorithms maintain all invariants. Invariant 8 follows from the definition of overflowing paths in the insert algorithm. The insert and delete algorithms perform two incremental steps, whenever the size of a node that belongs to an incremental pair increases or decreases by one. This suffices to move at least one element and pointer and thus to preserve Invariants 5 and 6. Invariant 7 is a corollary of Invariant 8, 5 and 6. With respect to Invariant 10, the insert and delete algorithms define node-disjoint incremental paths. Moreover, each incremental step ensures that two paired nodes remain children of a common parent node. Finally, the moves performed by the delete algorithm excluding incremental steps, explicitly mark the involved node preventing the overlap of two paths.

THEOREM 3.1. *Incremental B-Trees can be implemented in external memory using $O(n/B)$ blocks of space and support searching, insertions and deletions of elements in $O(\log_B n)$ I/Os and range searches in $O(\log_B n + t/B)$ I/Os. Moreover, every update makes a constant number of modifications to the tree.*

Proof. By Lemma 3.1 we get that the height of the tree is $O(\log_B n)$. We now argue that the tree has $O(n/B)$ nodes, each of which has degree $O(B)$, i.e. the space bound follows and each node can be accessed in $O(1)$ I/Os.

From Invariants 5 - 9, it follows that all overflowing and the good leaves have size at least $B/2$. Also two leaves in an incremental pair have at least $B/2$ elements combined. Thus the leaves consume $O(n/B)$ disk blocks of space, which dominates the total space of the tree. The same invariants show that all nodes have at most $\frac{8B+4}{3}$ elements in them and their degree is upper bounded by $\frac{16B+8}{3}$. Thus every node consumes $O(1)$

disk blocks of space and can be accessed in $O(1)$ I/Os.

We conclude that searching, inserting and deleting an element costs $O(\log_B n)$ I/Os. A range search costs $O(\log_B n)$ I/Os for the search and $O(t/B)$ I/Os for the traversal. Finally, the rebalancing algorithms are defined such that they perform at most a constant number of modifications (incremental steps and definitions of paths) to the structure. □

3.4 Application to Incremental B-Trees The interface in Section 2 can be used to make Incremental B-Trees fully persistent. The marks of every node are recorded by an additional field. Since d_{in}=1, $c_f \leq 3$, by Theorem 2.1 we get constants $\pi \geq 10$ and c_{max}=96. A range search operation on the i-th version of the fully persistent incremental B-Tree is implemented by an `Access(`i`)` operation to determine the root at version i, and a `Read` operation for every node at version i visited by the ephemeral algorithm. Since every node at version i is accessed in $O(1)$ I/Os, the range search makes $O(\log_B n + t/B)$ I/Os. An update operation on the i-th version is implemented first by a `NewVersion(`i`)` operation that creates a new version identifier j for the structure after the update operation. Then, an `Access(`j`)` operation and a sequence of `Read` operations follow in order to determine the nodes at version j to be updated. Finally, a sequence of $O(1)$ `Write` operations follows in order to record the modifications made by the insertion and the deletion algorithms described in Section 3.2. By Theorem 2.1 we get the following corollary.

COROLLARY 3.1. *There exist fully persistent B-Trees that support range searches at any version in $O(\log_B n + t/B)$ I/Os and updates at any version in $O(\log_B n + \log_2 B)$ amortized I/Os, using space $O(m/B)$ disk blocks, where n denotes the number of elements in the accessed version, m the total number of updates, t the size of the query's output, and B the disk block size.*

4 Conclusions

By further parametrizing the amortized analysis with respect to π we can achieve a method for I/O-efficient full persistence with $O(c_{max} \log_2 \pi)$ I/O-overhead per access step and $O(d_{in}(c_{max} + d_{in}\frac{\log_2 c_{max} B}{\pi}) + \log_2 \pi)$ amortized I/O-overhead and $O(\frac{1}{B})$ amortized space-overhead per update step. For example, setting $\pi = \log_2 B$ in the case of Incremental B-Trees, we obtain an access and update overhead of $O(\log_2 \log_2 B)$ I/Os. Obtaining fully persistent Incremental B-Tress with constant I/O- and space-overhead per access and update step remains an open problem. This paper is a first step towards solving the open problem posed by Vitter [23] that asks to make B-Trees with $O(1)$ amortized update time fully persistent.

References

[1] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, September 1988.

[2] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *J. Exp. Algorithmics*, 8:1.2, 2003.

[3] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[4] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[5] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[6] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Ann. European Symp. on Algorithms*, LNCS 2461, pages 152–164, 2002.

[7] Douglas Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[8] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Ann. ACM Symp.on Theory of Computing*, pages 365–372, 1987.

[9] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Ann. ACM Symp. on Theory of Computing*, pages 122–127, 1982.

[10] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[11] Rudolf Fleischer. A simple balanced search tree with O(1) worst-case update time. *Int. J. Found. Comput. Sci.*, 7(2):137–150, 1996.

[12] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Ann. Conf. on Foundations of Computer Science*, pages 714–723, 1993.

[13] Scott Huddleston and Kurt Mehlhorn. Robust balancing in B-trees. In *In Proceedings of the 5th GI-Conf. on Theoretical Computer Science*, LNCS 104, pages 234–244, 1981.

[14] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–184, 1982.

[15] Haim Kaplan. Persistent data structures. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 31, pages 31-1–31-26. CRC Press, 2004.

[16] Sitaram Lanka and Eric Mays. Fully persistent B+-trees. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 426–435, 1991.

[17] David B. Lomet and Betty Salzberg. Exploiting a history database for backup. In *Proceedings of the 19th Int. Conf. on Very Large DataBases*, pages 380–390, 1993.

[18] David Maier and Sharon C. Salveter. Hysterical B-trees. *Inf. Process. Lett.*, 12(4):199–202, 1981.

[19] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31(2):158–221, 1999.

[20] Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Inf.*, 21(1):101–112, 1984.

[21] Athanasios K. Tsakalidis. AVL-trees for localized search. *Inf. Control*, 67(1-3):173–194, 1986.

[22] Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Trans. Knowl. Data Eng.*, 9(3):391–409, 1997.

[23] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, 2008.