Fully persistent B-trees[⋆]

Gerth Stølting Brodal[a], Spyros Sioutas[b], Konstantinos Tsakalidis[c], Kostas Tsichlas[b]

[a]*Department of Computer Science, Aarhus University, Denmark*
[b]*Computer Engineering and Informatics Department, University of Patras, Greece*
[c]*Department of Computer Science, University of Liverpool, UK*

Abstract

We present efficient fully persistent B-trees in the I/O model with block size B that support range searches on t reported elements at any accessed version of size n in $O(\log_B n + t/B)$ I/Os and updates at any accessed version in $O(\log_B n + \log_2 B)$ amortized I/Os, using $O(m/B)$ disk blocks after m updates. This improves both the query and update I/O-efficiency of the previous fully persistent B-trees of Lanka and Mays (ACM SIGMOD ICMD 1991).

To achieve the result, we introduce an implementation for ephemeral B-trees that supports searches and updates in $O(\log_B n)$ I/Os, using $O(n/B)$ blocks, where moreover every update makes a worst-case constant number of modifications to the structure. We make these B-trees fully persistent using an I/O-efficient method for full persistence, inspired by the *node-splitting* method of Driscoll et al. (JCSS 1989). Interesting in its own right, the method is generic enough to be applied to *any* external memory pointer-based data structure with maximum in-degree d_{in} and out-degree $O(B)$, where every node occupies a constant number of blocks on disk. For a user-specified parameter $\pi = \Omega(d_{in})$, we achieve $O\left(\frac{\pi}{B} + \log_2 \pi\right)$ I/O-overhead per access to a field of an ephemeral block and amortized $O\left(\frac{\pi}{B} + \log_2 \pi + \frac{d_{in}}{\pi} \log_2 B\right)$ I/O-overhead and $O(1/B)$ block space-overhead per modification to the ephemeral structure.

Keywords: persistent data structures, external memory, range searching, big data, versioned data bases
2010 MSC: 68P05, 68P10, 68P15, 68P20, 68Q25, 68W40

[⋆]A preliminary version of this article was presented in the Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012). The work of K. Tsakalidis has been partially supported by NSF grants CCF-1319648 and CCF-1533564.

Email addresses: `gerth@cs.au.dk` (Gerth Stølting Brodal), `sioutas@ceid.upatras.gr` (Spyros Sioutas), `K.Tsakalidis@liverpool.ac.uk` (Konstantinos Tsakalidis), `ktsichlas@ceid.upatras.gr` (Kostas Tsichlas)

1. Introduction

B-trees are the most common dynamic dictionary data structures used for external memory [4, 8, 17]. We study the problem of making B-trees fully persistent in external memory.

Persistent indices enable advanced functionalities over the history of data modifications. Their applications to massive data-sets attract increasing attention in practical fields, including database management [30, 25, 28], data stream sketches [35], historical graphs [20], bloom filters [29] and the cloud [23, 21]. More applications are found in computational geometry [34] and biology [15].

External memory model. We study the problem in the I/O model [1], which is usually used for computational settings where the amount of stored data is too large to fit in internal memory, and thus devices of external storage (such as hard disks) are necessary. The model abstracts the computational bottlenecks in the time and space complexities that occur between two consecutive levels of a memory hierarchy, named "internal memory" and "external memory", respectively. Input data resides in external memory, split into consecutive *blocks* of size B. Space-complexity is measured in the number of blocks that the data occupies in external memory. Time-complexity is measured in terms of *I/O-operations* (or *I/Os*) that transfer one block from external to internal memory, and vice versa. Computation on blocks in internal memory occurs "for free".

Persistent data structures. 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; 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; versions form a directed acyclic graph (*version DAG*). A survey on persistence is presented by Kaplan [18].

1.1. Previous results

Full & confluent persistence. The currently most efficient fully persistent B-trees [22] 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. Specifically, Lanka and Mays [22] present fully persistent B$^+$-Trees (FPBT), which can also be used for confluent persistence. They support range queries in $O\left((\log_B n + t/B)\log_B m\right)$ I/Os and updates in $O\left(\log_B^2 n\right)$ amortized I/Os, using $O\left(m/B\right)$ blocks of space, where t is the size of the range query's output.

		Update I/Os	Range Query I/Os
Partial	TSB [26]	$\log_B n \log_B m^\dagger$	n/B
	MVBT [5]	$\log_B^2 n$	$\log_B m + t/B$
	MVAS [33]	$\log_B^2 n$	$\log_B m + t/B$
	ADT [2]	$\log_B m$	$\log_B m + t/B$
Full	FPBT [22]	$\log_B^2 n$	$(\log_B n + t/B) \log_B m$
	New	$\log_B n + \log_2 B$	$\log_B n + t/B$
	New	$\log_B n \log_2 \log_2 B$	$(\log_B n + t/B) \log_2 \log_2 B$

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.

Davoodi et al. [9] present a mechanism for confluent persistence in external memory with logarithmic I/O-overhead per access and update step. Hence, applying their mechanism to B-trees matches the I/O-efficiency of confluently persistent FPBTs.

Partial persistence. I/O-optimal partially persistent B-trees [2] achieve $O(1)$ I/O-overhead per operation. Multiple variants of B-trees have been made partially persistent [4, 8, 27, 16, 17]. Salzberg and Tsotras' [30] 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 [33] and the Time-Split B-trees (TSB) of Lomet and Salzberg [26]. Moreover, the authors in [14] acquire partially persistent *hysterical* B-trees [27] 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 $O(m/B)$ blocks. The $O(\log_B m)$ update I/Os can be replaced with $O(\log_B n)$ I/Os by using standard global rebuilding. In Table 1 we summarize the partially and fully persistent B-trees that can be used in an online setting.

Discussion. All the previous persistent B-trees follow an approach similar to those of Driscoll et al. [12] 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-overhead and $O(\log_2 m)$ amortized time-overhead, whenever an update changes $O(1)$ elements in a node (*update step*); and $O(\log_2 n)$ worst case time-overhead whenever $O(1)$ elements in a node are accessed (*access step*). Lanka and Mays [22] 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 they 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 [11]. 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.

1.2. 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 $O(m/B)$ blocks.

Persistent mechanism. In Section 2 we present a method for making an external memory data structure fully persistent, inspired by the node-splitting method of Driscoll et al. [12]. We require that the ephemeral external data structure is pointer-based, and that 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 $O(B)$. In the preliminary version of this work [7], we also had the requirement that the in-degree of any node is bounded by a constant. In this full version we waive this assumption, allowing our method to handle ephemeral structures with maximum in-degree (of any node) d_{in}.

Specifically, we introduce the user-specified parameter $\pi = \Omega(d_{in})$ in order to provide a trade-off between the I/O-overhead of the access and the update step (for the I/O model). Access to a block of the ephemeral data structure (*access step* for the I/O model) causes in the worst case an overhead of $O\left(\frac{\pi}{B} + \log_2 \pi\right)$ I/Os. Whenever an update operation makes a constant number of modifications to a node (*update step* for the I/O model), the update overhead is $O\left(\frac{\pi}{B} + \log_2 \pi + \frac{d_{in}}{\pi} \log_2 B\right)$ amortized I/Os and the space overhead is $O\left(\frac{1}{B}\right)$ amortized blocks.

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 space efficiently, we use a packed memory layout for the persistent nodes of small size.

Incremental B-trees. 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 and support range searches in $O(\log_B n + t/B)$ I/Os. They support insertions 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 [32] on *red-black trees* [3] in order to obtain fully persistent red-black trees with $O(\log_2 n)$ amortized time per insertion and deletion, $O(\log_2 n)$ worst case time per access, using $O(m)$ 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, by setting $\pi = O(1)$ it follows that updating the fully persistent incremental B-trees takes $O(\log_B n + \log_2 B)$ amortized I/Os. Levcopoulos and Overmars [24], Fleischer [13] and Kaporis et al. [19] also present (a, b)-Trees that make in the worst case $O(1)$ 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

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 blocks.

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 $c_f > 0$, and has in-degree at most d_{in}. 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.

User interface. 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 and provides access to it. For example, this identifier may be the index of a cell in an array stored inside the node. 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 D_i's unique identifier.

`pointer` p_i `= Entry(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`, we require that 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`, we require that it points to a node 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 `= Clone(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.

Before every update operation, the user has to call `Clone` explicitly in order to create a new version j that corresponds to the ephemeral structure D_j.

2.1. The structure

Our method is inspired by the node-splitting method [12] 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.

2.1.1. Global version list

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 [12], we store the pre-order layout of T in a dynamic list that supports *order-maintenance* queries [10, 31, 11, 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 pre-order 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 in the GVL in a consecutive range, i.e. an *interval* of versions.

Specifically, the interval denoted by $[i, j)$ contains all versions in the GVL between version i and version j, including version i but excluding version j. We denote by i^+ the *successor version* of version i in the GVL, i.e. the version immediately to the right of i. The interval $[i, i^+)$ is a singleton interval that only contains version i.

By implementing the GVL as in [11], 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 reference to the position of the version's parent version in the GVL.

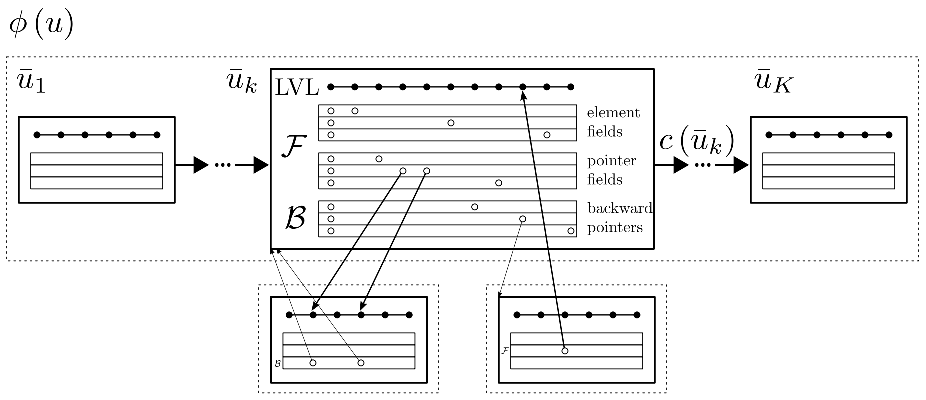


Figure 1: The family $\phi(u)$ records all versions of the ephemeral node u. Black and white dots represent versions and tuples, respectively. Persistent node $\bar{u}_k \in \phi(u)$ points to some persistent node (bottom left). Some other persistent node (bottom right) points to $\bar{u}_k$. The corresponding backward pointers are represented with thin arrows.

2.1.2. Persistent nodes

We record all the changes that occur to an ephemeral node u in a linked list of $K \geq 1$ *persistent nodes* $\bar{u}_k$, called **family** $\phi(u) := \bar{u}_1 \to \ldots \to \bar{u}_K$. To implement the linked list, every persistent node $\bar{u}_k$ contains a pointer $c(\bar{u}_k)$ to the next persistent node $\bar{u}_{k+1}$, where $c(\bar{u}_K) = null$.

Each node $\bar{u}_k \in \phi(u)$ stores the versions of u in some interval of the GVL. Figure 1 illustrates a persistent node $\bar{u}_k$.

Persistent fields. Every persistent node $\bar{u}_k$ stores an ordered set $\mathcal{F}_f(\bar{u}_k)$ for every field f of the corresponding ephemeral node u. If field f stores elements, then the set $\mathcal{F}_f(\bar{u}_k)$ contains *pairs* of the type (version i, value x), where x is the element stored in f at version i.

Forward pointers. Otherwise field f stores pointers, so the set $\mathcal{F}_f(\bar{u}_k)$ contains pairs of the type (version i, pointer $\overrightarrow{p}$), where the *forward pointer* $\overrightarrow{p}$ corresponds to the ephemeral pointer p that is stored in field f at version i. In particular, if p points to the ephemeral node v at version i, then $\overrightarrow{p}$ points[1] to the persistent node $\bar{v} \in \phi(v)$ that corresponds to node v at version i. The set $\mathcal{F}_f(\bar{u}_k)$, where the field f stores pointers, may also contain pairs of the type (version i, *null*) to represent the fact that for an interval of versions that includes version i, the forward pointer does not point to any node.

We denote $\mathcal{F}(\bar{u}_k) = \cup_f \mathcal{F}_f(\bar{u}_k)$.

[1]Paragraphs "local version list" and "pointer invariants" clarify where $\overrightarrow{p}$ points exactly.

Backward pointers. For every pair $(i, \overrightarrow{p}) \in \mathcal{F}_f(\bar{v})$ where the forward pointer $\overrightarrow{p}$ points to $\bar{u}_k$, the persistent node $\bar{u}_k$ contains a *triple*[2] (version i, version j, pointer $\overleftarrow{p}$), where the *backward pointer* $\overleftarrow{p}$ points to $\bar{v}$. We denote by $\mathcal{B}(\bar{u}_k)$ the set of all triples in $\bar{u}_k$ with backward pointers.

Backward pointers do not correspond to ephemeral pointers and they are only used by the persistent mechanism to accommodate updates.

Henceforth, by the term *tuples* we refer to pairs in $\mathcal{F}(\bar{u}_k)$ and triples in $\mathcal{B}(\bar{u}_k)$. The tuples in $\mathcal{F}_f(\bar{u}_k)$ for all f, and in $\mathcal{B}(\bar{u}_k)$ are sorted with respect to the order of their first component (version i) in the GVL.

Local version list. The persistent node $\bar{u}_k$ contains a **Local Version List** LVL($\bar{u}$) that is a linked list containing all the versions in the tuples of $\mathcal{F}(\bar{u}_k)$ and $\mathcal{B}(\bar{u}_k)$ ordered with respect to their order in the GVL. We define the first version in the LVL($\bar{u}_k$) as the *version $i_{\bar{u}_k}$ of the persistent node $\bar{u}_k$.* Every forward pointer to node $\bar{u}_k$ points to some version in the LVL($\bar{u}_k$).

We define the *valid interval* $[i, j)$ *of a tuple* (i, x) (or $(i, \overrightarrow{p})$) in $\mathcal{F}_f(\bar{u}_k)$ to be the set of versions in the GVL for which field f has the particular value x (or contains the forward pointer $\overrightarrow{p}$ to a particular persistent node, respectively). Specifically, version j is the version in the immediately next (*succeeding*) tuple of $\mathcal{F}_f(\bar{u}_k)$, if the tuple exists. Otherwise $j = i_{\bar{u}_{k+1}}$, if $c(\bar{u}_k) \neq null$. Else, j is the last version in the GVL[3]. Similarly, the valid interval of a tuple $(i, j, \overleftarrow{p}) \in \mathcal{B}(\bar{u}_k)$ is $[i, j)$; in this case, version j is explicitly contained in the tuple. We define the *valid interval of a persistent node* $\bar{u_k}$ to be $[i_{\bar{u}_k}, i_{\bar{u}_{k+1}})$, if $c(\bar{u}_k) \neq null$. Otherwise, it is up to the last version in the GVL.

A pair $(i, x) \in \mathcal{F}_f(\bar{u}_k)$ implements the `values` x of `field` f at all the versions in the GVL that belong to the pair's valid interval. A pair $(i, \overrightarrow{p}) \in \mathcal{F}_f(\bar{u}_k)$ with forward pointer $\overrightarrow{p}$ to persistent node $\bar{v}$ implements the `pointers` p_j to node v at all version j in the GVL that belong to the pair's valid interval.

Valid intervals are necessary to implement full persistence within a persistent node. For instance[4], suppose that we write `value` x in `field` f of node u at `version` i. Let i belong to the valid interval of persistent node $\bar{u}_k$ for some k, and in particular to the valid interval $[i', j)$ of pair $(i', y) \in \mathcal{F}_f(\bar{u}_k)$. Simply adding pair (i, x) to $\mathcal{F}_f(\bar{u}_k)$ (succeeding (i', y)) would correctly redefine the valid interval of (i', y) to be $[i', i)$, but would also erroneously define the valid interval of (i, x) to be $[i, j)$. This is a mistake, because the value of f for versions $[i^+, j)$ obtain value x, instead of the previous correct value y. To avoid this inconsistency, we follow the principle that whenever we add a pair (i, x) to $\mathcal{F}_f(\bar{u}_k)$, we also add the succeeding pair (i^+, y) (provided $i^+ \neq j$, i.e. i^+ is strictly before j in the GVL). This defines the valid interval of (i, x) to correctly be $[i, i^+)$, and that of (i^+, y) to be $[i^+, j)$. In other words, the `value` of `field` f

[2]Paragraph "local version list" clarifies version j and paragraph "pointer invariants" clarifies that $\overleftarrow{p}$ does not point anywhere within $\bar{v}$.

[3]To make notation $[i, j)$ coherent, the GVL contains a dummy version $+\infty$ at its end.

[4]The exact details are presented in the paragraph on operation `Write` in Section 2.3.

in node u remains y for all versions in the GVL that belong to $[i', i)$ and $[i^+, j)$, and is set to x only at version i.

Entry array. To provide access to the structure we maintain an **entry array** whose i-th position stores the pair (version i, pointer $\overrightarrow{p}$). Forward pointer $\overrightarrow{p}$ points to the persistent node $\bar{u}_k$ that corresponds to the entry node at version i. Here, i is a reference to the position of version i in the GVL. Also, $\mathcal{B}(\bar{u}_k)$ contains the triple $(i, i^+, \overleftarrow{p})$ with a backward pointer $\overleftarrow{p}$ to the i-th position of the array.

The array contains all versions, since `Clone` is called for every version created. Thus, the valid interval of tuple $(i, \overrightarrow{p})$ in the entry array is defined to be $[i, i^+)$.

Packed memory. We define the *size* of a persistent node $\bar{u}_k$ to be the number of tuples in $\mathcal{F}(\bar{u}_k)$ and $\mathcal{B}(\bar{u}_k)$. This is asymptotically equal to the number of versions in the LVL$(\bar{u}_k)$, since there exists at least one tuple per version. A persistent node is *small*, if its size is at most $\frac{c_f}{2}B - 1$. To utilize space efficiently, we pack in an **auxiliary linked list**, all singleton families consisting of one small persistent node.

2.2. Invariants

Henceforth, without loss of generality, we drop subscript k when referring to a given persistent node $\bar{u} \in \phi(u)$.

Node invariants. The persistent nodes satisfy Invariants 1 and 2.

Invariant 1. *For every field f of the persistent node $\bar{u}$, the first pair in $\mathcal{F}_f(\bar{u})$ contains version $i_{\bar{u}}$.*

Invariant 1 ensures that the contents of an ephemeral node u at any given version i can be retrieved by accessing exactly one persistent node $\bar{u} \in \phi(u)$. Specifically, $\bar{u}$ is the persistent node in $\phi(u)$ whose valid interval contains i.

Invariant 2. *The size of a persistent node $\bar{u}$ (not stored in the auxiliary linked list) is $|\bar{u}| \in \left[\frac{c_f}{2}B, c_{max}B\right]$ for $c_{max} \geq 30\left(c_f + \frac{d_{in}}{B}\right)$.*

Invariant 2 ensures that a persistent node $\bar{u}$ occupies at most a constant number of blocks. Therefore, the out-degree of every persistent node, and thus also of every ephemeral node, is bounded by $O(B)$.

Pointer invariants. Forward and backward pointers satisfy Invariants 3 and 4.

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

The valid intervals of the tuples $(i, \overrightarrow{p})$ and $(i, j, \overleftarrow{p})$ are identical.

Invariant 3 associates every forward pointer $\overrightarrow{p}$ with a *corresponding* backward pointer $\overleftarrow{p}$ and ensures that the valid intervals of their tuples are exactly the same. We set this forward pointer $\overrightarrow{p}$ at version i to particularly point to the version i in the LVL$(\bar{u})$. It suffices for the corresponding backward pointer $\overleftarrow{p}$ to simply point to the persistent node $\bar{v}$.

We describe how to determine the triple with the backward pointer that corresponds to a given forward pointer. Let $(i, \overrightarrow{p}) \in \mathcal{F}_f(\bar{v})$ be the pair containing forward pointer $\overrightarrow{p}$ to persistent node $\bar{u}$, and let $(j, \overrightarrow{q})$ be its succeeding pair in $\mathcal{F}_f(\bar{v})$. We retrieve all triples in $\mathcal{B}(\bar{u})$ with backward pointers to $\bar{v}$, and we locate among them the triple $(i, j, \overleftarrow{p})$. By Invariant 3, at least one such triple exists. More than one such triples exist in the case where there are more than one pointers from the ephemeral node v to the ephemeral node u at all versions in the interval $[i, j)$. If so, we pick an arbitrary one.

We describe how to determine the pair with the forward pointer that corresponds to a given backward pointer. Let $(i, j, \overleftarrow{p}) \in \mathcal{B}(\bar{u})$ be the triple containing backward pointer $\overleftarrow{p}$ to persistent node $\bar{v}$. We retrieve all pairs in $\mathcal{F}(\bar{v})$ with forward pointers to $\bar{u}$, and locate among them, the pair whose valid interval is $[i, j)$. By Invariant 3, at least one such pair exists. If there are more than one such pairs, then we pick an arbitrary one.

We define the *span of a forward pointer* $\overrightarrow{p}$ to persistent node $\bar{u}$ to be the versions in the intersection between the valid interval of the pair containing $\overrightarrow{p}$ and the LVL$(\bar{u})$. The *size* of the span is the number of versions it contains.

Invariant 4. *Let $\pi \geq 10d_{in}$. The size of the span of every forward pointer is integer $d \in [1, 2\pi]$.*

Note that the span of a forward pointer to persistent node $\bar{u}$ is an interval of versions in LVL$(\bar{u})$. The span of a forward pointer can also be determined by locating the triple $(i, j, \overleftarrow{p}) \in \mathcal{B}(\bar{u})$ with the corresponding backward pointer, as described above. In particular, it is the interval of versions in the LVL$(\bar{u})$ from version i up to but not including version j.

Invariant 4 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 searching in a set of at most 2π versions.

2.3. Algorithms & correctness

We present the implementation of the user-interface and argue about its correctness. Operations `Write`, `NewNode`, and `Clone` immediately restore Invariants 1 and 3. This may cause $O(d_{in})$ forward pointers to violate Invariant 4 and some persistent nodes to violate Invariant 2. The auxiliary subroutine `Repair()` restores these invariants using an auxiliary *violation queue*.

We define the *predecessor version* of a version i in the LVL to be he rightmost version in the LVL that is not to the right of version i in the GVL. Note that when i belongs to the LVL, it is the predecessor of itself.

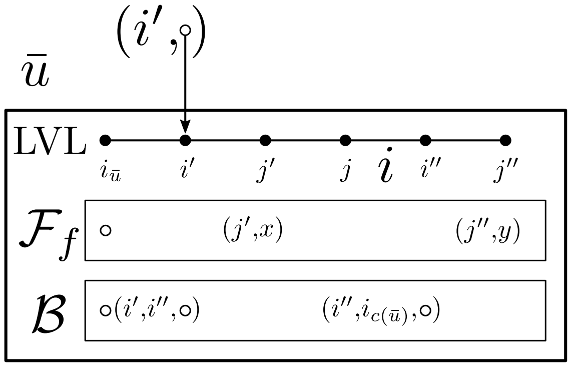


Figure 2: Ephemeral node u contains `value` x in `field` f at `version` i. To `Read`(p_i, f), we access the persistent node $\bar{u} \in \phi(u)$ by the `pointer` p_i, implemented by a forward pointer $\overrightarrow{p}$ in some pair $(i', \overrightarrow{p})$ whose valid interval contains version i. To determine the pair $(j', x) \in \mathcal{F}_f(\bar{u})$ whose valid interval contains version i, we find the predecessor version j of version i in LVL$(\bar{u})$. By Invariant 3 the backward pointer $\overleftarrow{p}$ corresponding to $\overrightarrow{p}$ belongs to a triple $(i', i'', \overleftarrow{p}) \in \mathcal{B}(\bar{u})$ whose valid interval contains i. Therefore versions $j, i \in [i', i'')$ and $i', j, i'' \in$ LVL$(\bar{u})$, although $i \notin$ LVL$(\bar{u})$. Version j is determined by an order-maintenance binary search for i over the span of $\overrightarrow{p}$. Version j' is the predecessor of j among all versions in $\mathcal{F}_f(\bar{u})$.

2.3.1. Operation `Entry`

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

2.3.2. Operation `Read`

`value` x `= Read(pointer` p_i`, field` f`)`. Let `pointer` p_i point to ephemeral node u at version i. Let version i belong to the valid interval of persistent node $\bar{u} \in \phi(u)$. 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 its predecessor version j. 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$(\bar{u})$ and moreover j belongs to the span of $\overrightarrow{p}$.

To locate the output pair in $\mathcal{F}_f(\bar{u})$, we need to find version j. We perform a binary search for i over the versions in LVL$(\bar{u})$ that belong to the span of $\overrightarrow{p}$. Every comparison is implemented by an order-maintenance query between the accessed version in the span and version i. This way, j is the rightmost located version in the span for which the order-maintenance query returns true. At least one order-maintenance query returns true, since (the leftmost considered) version i' lies to the left of version i in the GVL. We find the pair $(j', x) \in \mathcal{F}_f(\bar{u})$ with the predecessor version j' of version j in the LVL$(\bar{u})$ and return its value x.

2.3.3. Operation `Write`

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

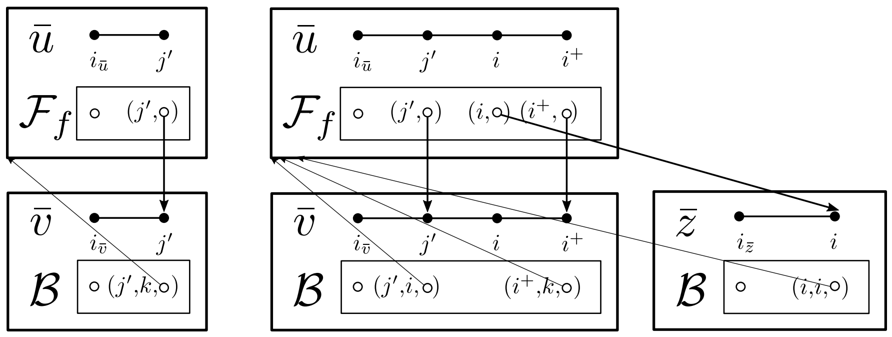
Writing values. If $j' = i$, we merely replace y with x, and add pair (i^+, y) to $\mathcal{F}_f(\bar{u})$, unless a pair with i^+ already exists. In this case, version i is the currently updated version and it belongs to the LVL($\bar{u}$). If $j' \neq i$, we add both the pairs (i, x) and (i^+, y) to $\mathcal{F}_f(\bar{u})$. If there is already a pair in $\mathcal{F}_f(\bar{u})$ with version i^+, it suffices to only add the pair (i, x). In 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 $\mathcal{F}_f(\bar{u})$ 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 $2d_{in}$ forward pointers that point to $\bar{u}$ to violate Invariant 4. That is at most d_{in} forward pointers for each newly inserted version i and i^+ in LVL($\bar{u}$). We denote the set of these forward pointers by $\overrightarrow{P_{\bar{u}}}$. The persistent nodes that contain them have to be inserted to the violation queue, unless they are already there. To locate the forward pointers in the set $\overrightarrow{P_{\bar{u}}}$, we determine the corresponding backward pointers in $\bar{u}$. In particular, we find all the triples $(\ell, \ell', \overleftarrow{p}) \in \mathcal{B}(\bar{u})$ whose valid intervals contain the inserted versions, and we check if there are more than 2π versions in the LVL($\bar{u}$) between versions ℓ and ℓ'. If so, we access the persistent node $\bar{v}$ pointed by $\overleftarrow{p}$ and *mark* the unmarked pair in $\mathcal{F}_f(\bar{v})$ with the corresponding forward pointer. We insert $\bar{v}$ to the violation queue, unless it is already there.

Writing pointers. If x is a pointer then we have to take into account the maintenance of backward pointers and the addition of new versions into LVLs. In particular, if x is a `pointer` to an ephemeral node z at version i, the argument `pointer` x_i is implemented by a forward pointer $\overrightarrow{x}$ in the persistent node $\bar{z} \in \phi(z)$ whose valid interval contains version i. We assume that $\overrightarrow{x}$ points the to predecessor version g of version i in the LVL($\bar{z}$).

The difference with respect to the addition of pairs in $\mathcal{F}_f(\bar{u})$ is that (instead of adding pair (i^+, y)) we add the pair $(i^+, \overrightarrow{y}')$ with forward pointer $\overrightarrow{y}'$ to the persistent node $\bar{v}$ that is pointed by $\overrightarrow{y}$, i.e. the forward pointer in the found pair $(j', \overrightarrow{y})$. We add to the LVL($\bar{v}$) the versions i and i^+, unless they already belong to it. This is accomplished by a binary search over the span of $\overrightarrow{y}$.

We set $\overrightarrow{y}'$ to particularly point to version i^+ in the LVL($\bar{w}$). Note that $\overrightarrow{y}$ already points to version j' in the LVL($\bar{w}$). The at most $2d_{in}$ forward pointers $\overrightarrow{P_{\bar{v}}}$ that violate Invariant 4, because of the addition of i and i^+, are processed as



(a) Before `Write` (b) After `Write`

Figure 3: To write `pointer` x_i from ephemeral node u to ephemeral node z at `version` i, `Write`(p_i, f, x_i) is called. The internal implementation of `Read`(p_i, f) finds the pair $(j', \overrightarrow{y}) \in \mathbb{F}_f(\bar{u})$ whose valid interval contains version i. Forward pointer $\overrightarrow{y}$ points to some persistent node $\bar{v} \in \phi(v)$, i.e. u contains a `pointer` y_i to ephemeral node v at `version` i.
Pairs $(i, \overrightarrow{x})$ and $(i^+, \overrightarrow{y}')$ are added to $\mathbb{F}_f(\bar{u})$ with forward pointers $\overrightarrow{x}, \overrightarrow{y}'$ to persistent nodes $\bar{z} \in \phi(z)$ (whose valid interval contains i) and $\bar{v}$, respectively. Corresponding backward pointers are added to $\mathcal{B}(\bar{v}), \mathcal{B}(\bar{z})$. Versions i, i^+ are added to LVL$(\bar{u})$ and LVL$(\bar{v})$ and i to LVL$(\bar{z})$.

described for $\overrightarrow{P_{\bar{u}}}$. To restore Invariant 3, we change the triple $(j', k, \overleftarrow{y}) \in \mathcal{B}(\bar{v})$ to $(j', i, \overleftarrow{y})$, where $[j', k)$ was the valid interval of the corresponding pair $(j', \overrightarrow{y}) \in \mathcal{F}(\bar{u})$. Moreover, we add the triple $(i^+, k, \overleftarrow{y}')$ to $\mathcal{B}(\bar{v})$ that corresponds to the pair $(i^+, \overrightarrow{y}') \in \mathcal{F}(\bar{u})$.

To establish the forward pointer $\overrightarrow{x}$ we insert version i to the LVL$(\bar{z})$ succeeding version g, unless i is already there. We set $\overrightarrow{x}$ to particularly point to version i in the LVL$(\bar{z})$. The added pair $(i, \overrightarrow{x})$ implements `pointer` x_i. The at most d_{in} forward pointers $\overrightarrow{P_{\bar{z}}}$ that violate Invariant 4 are processed as described for $\overrightarrow{P_{\bar{u}}}$. We restore Invariant 3 for the added pair $(i, \overrightarrow{x})$ by inserting to $\mathcal{B}(\bar{z})$ the triple $(i, i^+, \overleftarrow{x})$, where the backward pointer $\overleftarrow{x}$ points to $\bar{u}$.

Managing node sizes. The insertions of tuples in the persistent nodes $\bar{u}$, $\bar{v}$ and $\bar{z}$ increases their size. If the nodes are not small any more due to the insertions, 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.

2.3.4. Operation `NewNode`

`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 empty. 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$(\bar{u})$.

2.3.5. Operation `Clone`

`version` j `= Clone(version` i`)`. We find the position of version i in the GVL from the reference stored at the i-th position of the entry array. We insert version j immediately to the right of version i in the GVL. We insert in the j-th position of the entry array a reference to the position of version j in the GVL.

Let $\bar{u}$ be the persistent node pointed by the forward pointer $\overrightarrow{q}$ in the i-th position of the entry array. We insert in the j-th position of the entry array a forward pointer $\overrightarrow{p}$ to $\bar{u}$, add the triple $(j, j^+, \overleftarrow{p})$ to $\mathcal{B}(\bar{u})$ (where the backward pointer $\overleftarrow{p}$ points to the j-th position of the entry array and corresponds to $\overrightarrow{p}$) and replace the triple with $\overrightarrow{q}$ in $\mathcal{B}(\bar{u})$ with $(i, j, \overleftarrow{q})$.

At most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ that point to $\bar{u}$ may violate Invariant 4. We process them as described in `Write`. If $\bar{u}$ violates Invariant 2 we insert it to the violation queue, unless it is already there. Finally, `Repair` is called.

2.3.6. Operation `Repair`

`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. See Section 2.4.2 for the analysis.

Managing spans. To restore Invariant 4 for the forward pointer in the marked pair $(i, \overrightarrow{p}) \in \mathcal{F}_f(\bar{u})$ with valid interval $[i, k)$, we reset the size of its span to π as following. Let $\overrightarrow{p}$ point to the persistent node $\bar{w}$. We find the version j in the span of $\overrightarrow{p}$ that resides π positions to the right of version i in the LVL($\bar{w}$). We set the forward pointer $\overrightarrow{p}'$ to version j in the LVL($\bar{w}$), 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 and insert $\bar{u}$ again to the violation queue.

We restore Invariant 3 by changing the triple $(i, k, \overleftarrow{p}) \in \mathcal{B}(\bar{w})$ to $(i, j, \overleftarrow{p})$ and by adding a new triple $(j, k, \overleftarrow{p})$ to $\mathcal{B}(\bar{w})$ that corresponds to the new pair $(j, \overrightarrow{p}') \in \mathcal{F}(\bar{u})$. We find the predecessor version of 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 predecessor version, unless it already exists. This may cause at most d_{in} forward pointers $\overrightarrow{P_{\bar{u}}}$ to violate Invariant 4. We check them as described in `Write`. Also, if $\bar{u}$ or $\bar{w}$ violate Invariant 2 they are added to the violation queue.

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

We first determine the version j at which we will split $\bar{u}$, by scanning the LVL($\bar{u}$) from right to left. Version j is the leftmost version in the LVL($\bar{u}$), such that the number of tuples whose version succeeds j is less than $\frac{c_{max}}{2}B$, without counting the tuples with *null* values that occur last in $\mathcal{F}_f(\bar{u})$ with respect to the order of the versions. (These tuples represent empty fields in the new node and as such they are not stored in it.)

Unless $j' = j$, if x is not a pointer, then for every pair $(j', x) \in \mathcal{F}(\bar{u})$ whose valid interval contains version j we add a pair (j, x) to $\mathcal{F}_f(\bar{u})$ for the particular field f. For the pair $(j', \overrightarrow{x}) \in \mathcal{F}_f(\bar{u})$, where $\overrightarrow{x}$ is a forward pointer to a persistent node $\bar{w}$, we first add version j to $LVL(\bar{w})$ by binary searching in the span of $\overrightarrow{x}$ and then we add to $\mathcal{F}_f(\bar{u})$ the pair $(j, \overrightarrow{x}')$ where $\overrightarrow{x}'$ points to $\bar{w}$ at version j. Then to restore Invariant 3 we change the triple $(j', \ell, \overleftarrow{x})$ that corresponded to the pair $(j', \overrightarrow{x}) \in \mathcal{F}_f(\bar{u})$ to $(j', j, \overleftarrow{x})$, and we also add the triple $(j, \ell, \overleftarrow{x})$ which correponds to the new pair $(j, \overrightarrow{x}')$. We do similarly for the at most d_{in} triples with backward pointers in $\mathcal{B}(\bar{u})$ that contain version j in their valid interval. In particular, we replace all triples $(k, \ell, \overleftarrow{x})$ that contain j in their valid interval with two triples $(k, j, \overleftarrow{x})$ and $(j, \ell, \overleftarrow{x})$. By following $\overleftarrow{x}$ we locate the pair $(k, \overrightarrow{x}) \in \mathcal{F}(\bar{v})$ with a forward pointer to $\bar{u}$ at version k whose valid interval is $[k, \ell)$. We add the pair $(j, \overrightarrow{x}')$, where $\overrightarrow{x}'$ is a forward pointer to $\bar{u}$ at version j. We also add j to LVL($\bar{v}$) by making a binary search over the whole LVL($\bar{v}$). This may in turn violate Invariant 4 for forward pointers in $\overrightarrow{P_{\bar{v}}}$, which we process as described above.

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($\bar{u}'$) and version j becomes the version of $\bar{u}'$. All the tuples in $\bar{u}$ with valid interval $[k, \ell)$, such that $k \geq j$ are moved to $\bar{u}'$. Then, all forward and backward pointers in $\mathcal{F}(\bar{u}')$ and $\mathcal{B}(\bar{u}')$ are traversed in order to update the corresponding backward and forward pointers to point to $\bar{u}'$ instead of $\bar{u}$. Note, that this step changes existing pointers and does not introduce new tuples or versions anywhere in the structure. Node $\bar{u}'$ satisfies Invariant 1 due to the addition of the pairs (j, x) for each non-empty field.

Finally, all nodes that had their size increased because of the introduction of new tuples are checked with respect to Invariant 2 and each node that violates it is inserted into the violation queue, unless it is already there.

2.4. Analysis

In this subsection we prove the following theorem.

Theorem 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 and an update to a particular version of the persistent structure $\bar{D}$ is supported, respectively, in:*

$O\left(q\left(c_{max} + \log_2 \pi\right)\right)$ *worst case I/Os and*

$O\left(u\left(c_{max} + \log_2 \pi + \frac{d_{in}}{\pi} \log_2\left(c_{max} B\right)\right)\right)$ *amortized I/Os,*

where d_{in} is the maximum in-degree of any node in D, $\pi \geq 10 d_{in}$ is the size of the span and where any node of $\bar{D}$ occupies at most $c_{max} \geq 30\left(c_f + \frac{\pi}{B}\right)$ blocks. After performing a sequence of m updates, $\bar{D}$ occupies $O\left(u\frac{m}{B}\right)$ 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 1. *After splitting a persistent node $\bar{u}$ the size of the new persistent node $\bar{u}'$ is within the range* $\left[\left(\frac{c_{max}}{2} - c_f\right)B - d_{in}, \left(\frac{c_{max}}{2} + c_f\right)B + d_{in} - 1\right]$.

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

2.4.1. Worst-case analysis

First, we list the worst-case I/O-cost of the basic subroutines called by the operations. Accessing the entry array takes $O(1)$ I/Os. Querying and updating the GVL takes $O(1)$ time [11] and thus $O(1)$ I/Os. Loading a persistent node in memory or inserting it to the violation queue takes $O(c_{max})$ I/Os. After having accessed a persistent node by a forward pointer, the fields of a particular version are retrieved by $O(\log_2 \pi)$ calls to the order-maintenance structure. After having accessed a persistent node by a backward pointer, the fields of a particular version are retrieved by $O(\log_2(c_{max}B))$ calls to the order-maintenance structure.

During navigation, operations `Entry` and `Read` incur worst-case I/Os. Specifically, `Entry` takes $O(1)$ worst-case I/Os in total, since it only accesses the entry array. `Read` accesses the persistent node $\bar{u}$ by a forward pointer, and thus it takes $O(c_{max} + \log_2 \pi)$ worst-case I/Os in total to load $\bar{u}$ into memory and to determine the appropriate version j.

During update, only operation `NewNode` incurs worst-case I/Os. Specifically, `NewNode` takes $O(1)$ worst-case I/Os in total, since it creates a persistent entry node and inserts it into the auxiliary linked list.

2.4.2. Amortized analysis

During update, the remaining operations incur amortized I/Os. Specifically, an update operation commenced by the user consists of a call to `Write` or to `Clone` that is followed by a `Repair` operation. To derive the amortized I/O-cost of the update operations, we count the amortized number of *modifications* that the operations cause to the structure and we charge every modification with an amortized number of I/Os. A modification particularly consists of adding a tuple or a version to a persistent node, or of changing the value in a tuple. To unify the above types of modifications, we assume that a version and a tuple fit in a field of a block, and we count the number of modified fields. Note that this number bounds asymptotically the number of persistent nodes updated by the operations.

Potential function. Let D_i be the persistent structure after the i-th update operation has been performed and without loss of generality let it be performed on persistent structure D_{i-1}. The amortized number of modifications caused by the i-th update operation is $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, where c_i is the actual number of modifications to D_{i-1}, and where $\Phi(D_i)$ is the potential of the persistent structure D_i. In particular, 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

$$\Xi(\overrightarrow{p}) = \max\left\{0, \frac{|\overrightarrow{p}| - \pi}{d_{in}}\right\}$$

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 $\bar{u}$. The amortized number of modifications is an upper bound on the actual number of modifications of a worst-case sequence of operations, since the potential function is always non-negative.

Change in potential. We calculate how much a single update operation changes the potential. Operation `Write`, without the call to `Repair`, increases the potential by at most $\Delta\Psi = 12$. In particular, it increases $\Psi(\bar{u})$, $\Psi(\bar{z})$ and $\Psi(\bar{w})$, since at most two pairs are added in $\bar{u}$, one triple is added in $\bar{w}$ and one in $\bar{z}$, or because at most three pairs are added in $\bar{u}$ and one in $\bar{z}$. Moreover, it increases the potential by at most $\Delta\Xi = 5$. In particular, the potential of the at most d_{in} forward pointers of $\overrightarrow{P_{\bar{z}}}$, the potential of the at most $2d_{in}$ forward pointers of $\overrightarrow{P_{\bar{u}}}$ and the potential of at most $2d_{in}$ forward pointers $\overrightarrow{P_{\bar{w}}}$ is increased by $\frac{1}{d_{in}}$. Note that although the span of a forward pointer may contain both new versions, and as such its potential is increased by $\frac{2}{d_{in}}$, in total the increase in potential cannot be larger than 2 for each such node. In total, the potential increases by at most $\Delta\Phi = 17$.

Operation `Clone`, without the call to `Repair`, increases the potential by at most $\Delta\Psi = 3$, because of the addition of the new triple in $\bar{u}$. Moreover, it increases the potential by at most $\Delta\Xi = 1$, since the potential of at most d_{in} forward pointers of $\overrightarrow{P_{\bar{u}}}$ is increased by $\frac{1}{d_{in}}$. In total, the potential increases by $\Delta\Phi = 4$.

When operation `Repair` restores Invariant 4 for one marked pair with a forward pointer, it increases the potential by at most $\Delta\Psi = 6$, since a pair is added to $\bar{u}$ and a triple is added to $\bar{w}$. Moreover, it increases the potential by at most $\Delta\Xi = 1 - \frac{\pi}{d_{in}}$. This is because the potential of the at most d_{in} forward pointers of $P_{\bar{u}}$ is increased by $\frac{1}{d_{in}}$, due to the addition of version j. In addition, the potential of the forward pointer in the pair is decreased by $\frac{\pi}{d_{in}}$, due to the split of the span. In total, the potential changes by $\Delta\Phi = 7 - \frac{\pi}{d_{in}}$.

When operation `Repair` restores Invariant 2 for a persistent node $\bar{u}$, it increases the potential by at most $\Delta\Psi = \left(9c_f - \frac{3}{2}c_{max}\right) B + 5d_{in}$. This is because it adds at most $c_f B$ pairs and at most d_{in} triples in $\bar{u}$, and at most $c_f B + d_{in}$ triples and pairs with the corresponding backward and forward pointers in other persistent nodes. After the split $(\frac{c_{max}}{2} - c_f)B - d_{in}$ pairs and triples have been moved to the new persistent node $\bar{u}'$, as implied by the lower bound in Lemma 1. Moreover, it increases the potential by at most $\Delta\Xi = c_f B + d_{in}$, since at most $d_{in}\,(c_f B + d_{in})$ forward pointers have the size of their span increased by $\frac{1}{d_{in}}$, due to the addition of version j. In total the potential changes by $\Delta\Phi = \left(10c_f - \frac{3}{2}c_{max}\right) B + 6d_{in}$.

Number of modifications. We calculate the amortized number of modifications caused by the `Repair` operation. When `Repair` restores Invariant 4 for α forward pointers, then the actual number of modifications is $c_i = 3\alpha$, since for each such forward pointer we add a new version in an LVL as well as a pair and its corresponding triple. Thus, the amortized number of modifications is

$$\tilde{c_i} = \alpha\left(3 + \left(7 - \frac{\pi}{d_{in}}\right)\right)$$

which is non-positive for $\pi \geq 10d_{in}$.

When `Repair` restores Invariant 2 for β persistent nodes, then the actual number of modifications is $c_i = \beta\left(3c_f B + 5d_{in} + 2B\left(\frac{c_{max}}{2} + c_f\right) - 2\right)$. That is because we add at most $c_f B$ new pairs in $\bar{u}$ with forward pointers, and we add one corresponding new triple with a backward pointer and a new version at every persistent node pointed by each of these forward pointers. Moreover, we add at most d_{in} triples with backward pointers in $\bar{u}$, and we add one corresponding forward pointer and version at every persistent node pointed by each such backward pointer. Finally, we transfer at most $\left(\frac{c_{max}}{2} + c_f\right) B + d_{in} - 1$ pairs to $\bar{u}'$ and we update at most an equal number of pointers so that they point to $\bar{u}'$ insteaf of $\bar{u}$. Thus, the amortized number of modifications is

$$\tilde{c_i} = \beta\Bigg(3c_f B + 5d_{in} + 2B\left(\frac{c_{max}}{2} + c_f\right) - 2 + \\ (10c_f - \frac{3}{2}c_{max})B + 6d_{in}\Bigg)$$

which is non-positive for $c_{max} \geq 30c_f + \frac{22d_{in}-4}{B}$. Moreover $c_{max} = \Omega\left(\frac{\pi}{B}\right)$, since the span of a forward pointer cannot be larger than the size of a persistent node. Hence, we choose the constant 30 for the lower bound of c_{max} in Theorem 1. It follows that by choosing π and c_{max} appropriately we guarantee that the amortized number of modifications caused by the updates is not affected by the `Repair` operation.

Finally, we calculate the amortized number of modifications caused by operations `Write` and `Clone`. The actual number of modifications caused by `Write` is $c_i = 8$. This is because we add at most 4 versions and 4 pairs. Thus the amortized number of modifications caused by `Write` is $\tilde{c_i} = 8 + \Delta\Phi = 25$. The actual number of modifications caused by `Clone` is $c_i = 2$. This is because we add at most one version and one pair. Thus the amortized number of modifications caused by `Clone` is $\tilde{c_i} = 2 + \Delta\Phi = 6$.

I/O-cost per modification. We calculate the amortized I/O-cost per modification. A sequence of m updates incurs $\Theta(m)$ modifications to the structure, since the amortized number of modifications made by `Repair` is 0 for a suitable choice of π and c_{max}, and the amortized number of modifications made by `Write` and `Clone` is $\Theta(1)$. Every modification takes $O(c_{max} + \log_2 \pi)$ I/Os since the modified persistent node is loaded into memory to add the appropriate tuples, and the $O(\log_2 \pi)$ queries are performed to the GVL to add the appropriate versions to the LVL of the persistent node. Moreover, every modification increases the span of at most d_{in} forward pointers by 1. Thus, in a sequence of m modifications the spans increase by $O(md_{in})$ in total. Without loss of generality we assume that m is large enough, such that every span initially has size π, and thus it must suffer π version insertions in order to be split. Splitting a span takes $O(\log_2(c_{max}B))$ I/Os. Therefore, the total cost for splitting all spans in a sequence of m updates is $O\left(\frac{d_{in}}{\pi}\log_2(c_{max}B)\right)$ I/Os. It follows that the total amortized I/O-cost for processing one modification is $O\left(\frac{d_{in}}{\pi}\log_2(c_{max}B) + c_{max} + \log_2 \pi\right)$. This also upper bounds the amortized I/O-cost of every update operation, since the amortized number of modifications made by `Write` and `Clone` is $\Theta(1)$.

Note that depending on the in-degree of the given ephemeral structure, we can set the parameters c_{max} and d_{in} appropriately, in order to obtain different trade-offs between the I/O-overheads. For example, if $d_{in} = O(1)$ we can obtain an I/O-efficient mechanism for full persistence with $O(1)$ I/O-overhead per access step and $O(\log_2 B)$ I/O-overhead per update step, by setting $\pi = \Theta(1)$ and $c_{max} = \Theta(1)$. Moreover, if $d_{in} = O(\log_2 B)$, we can also obtain an I/O-efficient mechanism for full persistence with $\Omega(\log_2\log_2 B)$ I/O-overhead per access and update step by setting $\pi = \log_2^k B$ and $c_{max} = \Theta(1)$, for a constant $k \geq 2$. Finally, if $d_{in} = B$, we can also obtain an I/O-efficient mechanism for full persistence with $O(\log_2 B)$ I/O-overhead per access and update step by setting $\pi = 11B$ and $c_{max} = \Theta(1)$. However, in this case it would be better to apply the naive solution.

Space cost. The space usage after m operations with $O(u)$ modifications each in the worst case is $\Theta\left(u\frac{m}{B}\right)$ blocks, since an update operation makes $O(1)$ amortized number of fields modifications and since all small blocks are packed in the auxiliary linked list.

3. Incremental B-trees

In this section we design B-trees [4, 8, 17] that use $O(n/B)$ 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.

Ordinary B-trees. Before we describe our B-trees, we briefly recall the properties of ordinary B-trees [4, 8, 17]. 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.

Node marks. 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*[5].

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.*

Definitions. Unmarked nodes that do not belong to incremental pairs are called *good* nodes. We define the *size* of an internal node u to be $s_u = u_g + 2u_o - u_f$, where u_g, u_o and u_f are the number of the children of u that are good, have an overflowing mark and have a fusion mark, respectively. 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.

Invariants. 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.*

[5]The difference from splitting pairs is that elements and pointers are moved from the "small" node w to the "large" node w', whereas in a splitting pair, w is "larger" than w'.

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* $u \to v$ *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* $u \to v$ *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 2. *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, when an insertion or deletion involve these nodes.

Explicit mark. 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 $w_c \rightarrow v$. Hence, the element e at u is replaced with e_p, an overflowing mark is explicitly set on w, and an overflowing mark together with element e are explicitly set in w_c. If $u = w$ or $w = v$, then the first or the third subpath is empty, respectively.

Incremental step. 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 an in-order 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 2. *Incremental B-trees on n elements 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 on t reported elements in $O(\log_B n + t/B)$ I/Os. Moreover, every update makes a constant number of modifications to the tree.*

Proof. By Lemma 2 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 $O(n/B)$ 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 $O(1)$ blocks of space and can be accessed in $O(1)$ I/Os.

In conclusion, 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 of the fully persistent mechanism 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 1 we can choose some constants $\pi \geq 10$ and $c_{max} \geq 90$. A range search operation on the i-th version of the fully persistent incremental B-tree is implemented by an `Entry(`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 `Clone(i)` operation that creates a new version identifier j for the structure after the update operation. Then, an `Entry(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 1 we get the following corollary.

Corollary 1. *Fully persistent B-trees support range searches that report t elements at any version of size n 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 $O(m/B)$ blocks after m updates.*

Applying to the incremental B-trees, the I/O-efficient mechanism for full persistence with parameter $\pi = \log_2^2 B$ and constant $c_{max} \geq 90$, by Theorem 1 we obtain an implementation of fully persistent B-trees that support range searches at any version in $O((\log_B n + t/B)\log_2\log_2 B)$ I/Os and updates at any version in $O((\log_B n)\log_2\log_2 B)$ amortized I/Os, using space $O(m/B)$ blocks.

4. Conclusion

Our results are an important step forwards towards addressing the open problem of Vitter [34], which asks to make B-trees fully persistent with $O(1)$ amortized update time. In this work we presented access-I/O-optimal fully persistent B-trees with a mere additive $O(\log_2 B)$ amortized I/O-overhead per update step.

An important problem left open is to improve this overhead to $O(1)$ I/Os or, conversely, to show that this is not doable. One direction is to investigate whether our generic method for full persistence can be improved any further, or if it is tight to any superconstant "access step"-to-"update step" I/O-complexity trade-off inherent in the dynamic indexability model [36]. On the other hand, one may attempt to relax the conditions on incremental B-trees in pursuit of *sub*constant amortized modifications per update operation (perhaps by exploiting bit-packing techniques on the word-RAM variant of the I/O model). Finally, contrary to our approach that focuses on real-time (i.e. worst-case optimized) implementations of ephemeral B-trees, making ordinary (i.e. with amortized update I/O-cost) B-tree implementations fully persistent remains unaddressed, even in internal memory models.

References

[1] Alok Aggarwal and S. Vitter, Jeffrey. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

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

[3] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[4] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):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 Rolf Möhring and Rajeev Raman, editors, *Algorithms — ESA 2002*, pages 152–164. Springer, 2002.

[7] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. Fully persistent B-trees. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 602–614. SIAM, 2012.

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

[9] Pooya Davoodi, Jeremy T. Fineman, John Iacono, and Özgür Özkan. Cache-oblivious persistence. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014*, pages 296–308. Springer, 2014.

[10] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, STOC '82, pages 122–127. ACM, 1982.

[11] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, STOC '87, pages 365–372. ACM, 1987.

[12] 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.

[13] Rudolf Fleischer. A simple balanced search tree with O(1) worst-case update time. *International Journal of Foundations of Computer Science*, 07(02):137–149, 1996.

[14] Michael T. Goodrich, Jyh-Jong Tsay, Darren E. Vengroff, and Jeffrey S. Vitter. External-memory computational geometry. In *Proceedings of the IEEE 34th Annual Foundations of Computer Science*, FOCS '93, pages 714–723. IEEE, 1993.

[15] Wing-Kai Hon, Lap-Kei Lee, Kunihiko Sadakane, and Konstantinos Tsakalidis. Compressed persistent index for efficient rank/select queries. In Frank Dehne, Roberto Solis-Oba, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, pages 402–414. Springer, 2013.

[16] Scott Huddleston and Kurt Mehlhorn. Robust balancing in B-trees. In Peter Deussen, editor, *Theoretical Computer Science*, pages 234–244. Springer, 1981.

[17] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17(2):157–184, 1982.

[18] Haim Kaplan. *Handbook on data structures and applications*, chapter Persistent data structures, pages 241–246. CRC Press, 2004.

[19] Alexis Kaporis, Christos Makris, George Mavritsakis, Spyros Sioutas, Athanasios K. Tsakalidis, Kostas Tsichlas, and Christos Zaroliagis. ISB-tree: A new indexing scheme with efficient expected behaviour. *Journal of Discrete Algorithms*, 8(4):373 – 387, 2010.

[20] Andreas Kosmatopoulos, Kostas Tsichlas, Anastasios Gounaris, Spyros Sioutas, and Evaggelia Pitoura. HiNode: An asymptotically space-optimal storage model for historical queries on graphs. *Distrib. Parallel Databases*, 35(3-4):249–285, 2017.

[21] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. Everyone loves file: File storage service (FSS) in Oracle cloud infrastructure. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 15–32. USENIX Association, July 2019.

[22] Sitaram Lanka and Eric Mays. Fully persistent B+-trees. *SIGMOD Rec.*, 20(2):426–435, 1991.

[23] Pierre Bernard le Roux, Steve Kroon, and Willem Bester. DSaaS: a cloud service for persistent data structures. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, CLOSER 2016, pages 37–48. SCITEPRESS, 2016.

[24] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with O(1) worst-case update time. *Acta Informatica*, 26(3):269–277, 1988.

[25] David Lomet, Roger Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Immortal DB: Transaction time support for SQL server. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 939–941. ACM, 2005.

[26] David B. Lomet and Betty Salzberg. Exploiting a history database for backup. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 380–390. Morgan Kaufmann Publishers Inc., 1993.

[27] David Maier and Sharon C. Salveter. Hysterical B-trees. *Information Processing Letters*, 12(4):199–202, 1981.

[28] Apostolos N. Papadopoulos, Kostas Tsichlas, Anastasios Gounaris, and Yannis Manolopoulos. Access methods. In *Computing Handbook, 3rd Edition: Information Systems and Information Technology*, pages 1–18. CRC Press, 2014.

[29] Yanqing Peng, Jinwei Guo, Feifei Li, Weining Qian, and Aoying Zhou. Persistent bloom filter: Membership testing for the entire history. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1037–1052. ACM, 2018.

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

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

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

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

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

[35] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. Persistent data sketching. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 795–810. ACM, 2015.

[36] Ke Yi. Dynamic indexability and the optimality of B-trees. *J. ACM*, 59(4):21:1–21:19, 2012.