External Memory Fully Persistent Search Trees*

Gerth Stølting Brodal
Aarhus University
Department of Computer Science
Denmark
gerth@cs.au.dk

Casper Moldrup Rysgaard
Aarhus University
Department of Computer Science
Denmark
rysgaard@cs.au.dk

Rolf Svenning
Aarhus University
Department of Computer Science
Denmark
rolfsvenning@cs.au.dk

ABSTRACT

We present the first fully-persistent external-memory search tree achieving amortized I/O bounds matching those of the classic (ephemeral) B-tree by Bayer and McCreight. The insertion and deletion of a value in any version requires amortized $O(\log_B N_v)$ I/Os and a range reporting query in any version requires worst-case $O(\log_B N_v + K/B)$ I/Os, where K is the number of values reported, N_v is the number of values in the version v of the tree queried or updated, and B is the external-memory block size. The data structure requires space linear in the total number of updates. Compared to the previous best bounds for fully persistent B-trees [Brodal, Sioutas, Tsakalidis, and Tsichlas, SODA 2012], this paper eliminates from the update bound an additive term of $O(\log_2 B)$ I/Os. This result matches the previous best bounds for the restricted case of partial persistent B-trees [Arge, Danner and Teh, JEA 2003]. Central to our approach is to consider the problem as a dynamic set of two-dimensional rectangles that can be merged and split.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**.

KEYWORDS

B-trees, range queries, external memory, full persistence, semi-dynamic ray shooting.

ACM Reference Format:
Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. 2023. External Memory Fully Persistent Search Trees. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC '23), June 20–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3564246.3585140

1 INTRODUCTION

The B-tree of Bayer and McCreight [6] is the classic external-memory data structure for storing a dynamic set $\mathcal{S}$ of N totally ordered values. It supports the insertion and deletion of values in $O(\log_B N)$ I/Os and range reporting queries in $O(\log_B N + K/B)$ I/Os, i.e., reporting all values contained in a value range $[x, y]$, where K is the number of values reported and B is the external-memory block size.

In this paper, we consider the problem of storing a dynamic set in external memory *fully persistently*, i.e., all versions of the set are emembered, any previous version can be queried, and any previous version can be updated resulting in a new version of the set. We present the first fully persistent search trees matching the asymptotic I/O bounds of the classic *ephemeral* (i.e., non-persistent) B-trees in the amortized sense.

Throughout this paper, we will assume the I/O model of Aggarwal and Vitter [2] consisting of an internal memory being able to hold M values, and an infinite external memory. One I/O can transfer B consecutive values between internal and external memory, and the I/O complexity of an algorithm is the number of I/Os performed. Computation can only be performed in internal memory, and the only allowed operations on values are comparisons.

1.1 Interface of a Fully Persistent Search Tree

The interface to a fully persistent search tree consists of the below operations. Each version is identified by a unique version identifier v (a positive integer). We let $\mathcal{S}_v$ denote the set of values at version v. The versions form a *version tree* T, where the root (version 1) stores the initial set, and a new version w becomes a child of an existing version v in T, if $\mathcal{S}_w$ is derived as a *clone* of $\mathcal{S}_v$. Updates (insertions and deletions) can only be applied at leaves of the version tree T, i.e., versions that have not been cloned yet. These versions are said to be *unlocked.* A version that has been cloned cannot be updated further and is said to be *locked.* All versions can be queried and cloned. If one needs to update a locked version (an internal node of the version tree), one has to clone the version and apply the update to the new unlocked version (leaf of the version tree).

- Clone(v) Creates a new version w, where $\mathcal{S}_w = \mathcal{S}_v$. Returns the new version identifier w. Version w becomes a child of v in the version tree T, i.e., after the operation version v is locked and version w is unlocked.
- Insert(v, x) Adds x to $\mathcal{S}_v$. Requires version v is unlocked.
- Delete(v, x) Removes x from $\mathcal{S}_v$. Requires version v is unlocked. If x is not contained in $\mathcal{S}_v$, nothing is changed.
- Range(v, x, y) Returns all values in $\mathcal{S}_v \cap [x, y]$ in increasing order.
- Search(v, x) Returns the predecessor of x in $\mathcal{S}_v$.

We let N_v denote the number of values in $\mathcal{S}_v$, and N the total number of updates done to all versions. Figure 1 (top) illustrates a version tree. In the following examples, we illustrate values as integers, but our construction works for any comparison based values.

*Work supported by Independent Research Fund Denmark, grant 9131-00113B.

1.2 Previous Work

For the non-persistent setting in internal memory, a set can be stored using a standard balanced binary search tree, e.g., AVL-trees [1], red-black trees [20], and (a, b)-trees [21]. These all support insertions and deletions in $O(\log_2 N)$ time, and range reporting queries in $O(\log_2 N + K)$ time. In external memory, the B-tree [6] is the classic data structure of choice and supports updates in $O(\log_B N)$ I/Os and range reporting queries in $O(\log_B N + K/B)$ I/Os. A number of papers have since explored the update-query trade-off starting with [10] giving a structure known as the B^ε-tree for any $0 < \varepsilon < 1$ with amortized $O\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\right)$ I/Os for updates and $O\left(\frac{1}{\varepsilon} \log_B N\right)$ I/Os for queries. The B^ε-tree operates similarly to a B-tree but with fanout B^ε and updates are flushed down the tree using buffers, leading to the amortized bound. In [8, 9] the amortized bound was improved to high probability and finally in [15] to worst-case.

The notion of (data structural) full persistence for internal memory data structures was coined by Driscoll, Sarnak, Sleator and Tarjan in [18], where they also described how to achieve a fully persistent version of red-black trees. Previous to this Sarnak and Tarjan [25] had presented partially persistent red-black trees, where partial refers to the limitation that only the most recent version of the red-black tree can be updated, i.e., all versions of the red-black tree form a version tree consisting of a single path with the most recent version of the red-black tree being the single leaf. Demaine, Iacono and Langerman [16] considered the notion of *retroactive* data structures, where updates can be applied to any version in the version tree, and updates are automatically recursively applied to all derived version.

Adopting the notions of partial and full persistence to external-memory search trees has been done in a sequence of papers. See Table 1. The results of Arge, Danner and Teh 2003 [4] and Brodal, Sioutas, Tsakalidis and Tsichlas [11, 12] achieve the best bounds for partial and full persistence, respectively. Both results achieve optimal bounds for range reporting, i.e., $O(\log_B N_v + K/B)$ I/Os (the N in the bounds of [4] for partial persistence can be reduced to N_v by applying global rebuilding after a linear number of updates). The update I/O bound of [4] matches the $O(\log_B N)$ I/Os for B-trees, whereas the full persistence result of [11] has an additive $O(\log_2 B)$ I/Os overhead per update. This paper eliminates this additive term and achieves update bounds matching those of (ephemeral) B-trees and [4] for partial persistence.

1.3 Contribution

The main contribution of this paper are fully-persistent external-memory search trees, achieving the following bounds, matching those of (ephemeral) external-memory B-trees.

Theorem 1.1. *There exist external-memory fully-persistent search trees supporting* Insert *and* Delete *in amortized $O(\log_B N_v)$ I/Os,* Clone *in worst-case $O(1)$ I/Os,* Search *in worst-case $O(\log_B N_v)$ I/Os, and* Range *in worst-case $O(\log_B N_v + K/B)$ I/Os. The space usage is linear in the number of updates.*

It should be noted that for the case where $N_v = B^{O(1)}$, the update bounds are $O(\log_B N_v) = O(1)$ I/Os, whereas the best previous update bound was $O(\log_2 B) = O(\log_2 N_v)$ I/Os [11]. Our range reporting queries can be extended to support an argument k, and to only report in sorted order the first k (or last k) values in a value range $[x, y]$ using $O(\log_B N_v + k/B)$ I/Os, by simply truncating the reporting when k values have been found.

The basic ideas of the construction are the following. As in [18], we use a linearization of the version tree into a version list obtained by a pre-order traversal of the version tree. Values are associated with half-open liveness intervals of the version list. See Figure 1 (bottom). To adapt this to external memory setting, we consider these (value, liveness interval) pairs as vertical segments in a two-dimensional plane, and partition the plane into rectangles, possible splitting segments into multiple smaller disjoint segments. See Figure 2. During insertions and deletions the partition into rectangles is dynamically updated by splitting rectangles in either the version dimension (corresponding to the node splitting idea in [18]) or value dimension (corresponding to splitting a leaf in a B-tree), or by joining two horizontally adjacent rectangles (corresponding to joining two adjacent leaves in a B-tree). See Figure 3. A range reporting query Range(v, x, y) simply identifies all rectangles intersected by the horizontal query segment $[x, y] \times \{v\}$, and reports the values of all vertical segments in these rectangles intersecting the horizontal query segment. Invariants ensure that for all rectangles intersected, except for two, a constant fraction of the values are part of the answer to the query. This is inspired by the filtering search of Chazelle [13].

Essential to our result is the application of a two-dimensional orthogonal point location (vertical ray shooting) structure to identify the rectangle containing a (value, version) point. Point location queries must be supported in worst-case $O(\log_B N)$ I/Os, whereas segment insertions are allowed to be significantly slower, as high as amortized $O(B \log_B^2 N)$ I/Os. The best external-memory bounds for dynamic orthogonal point location are by Munro and Nekrich [24], who support updates and queries with $O(\log_B N \log\log_B N)$ I/Os, and by Arge, Brodal and Rao [3], who support queries with $O(\log_B^2 N)$ I/Os and updates with $O(\log_B N)$ I/Os (in internal memory $O(\log n)$ time updates and queries were achieved by Giora and Kaplan [19]). To achieve queries with $O(\log_B N)$ I/Os, we describe a specialized insertion-only external-memory point location structure, that also crucially avoids the comparison of version identifiers. The internal-memory [18] and external-memory [11] persistent search trees make essential use of the dynamic data structure by Dietz and Sleator [17] to answer order queries among two version identifiers in the version list in $O(1)$ time and I/Os. This is in fact the bottleneck causing the additive $O(\log_2 B)$ I/Os on updates in [11]. We avoid the use of [17] by using an external-memory colored predecessor data structure, that essentially stores multiple predecessor data structures in a single B-tree.

1.4 Outline of Paper

In Section 2 we present our geometric interpretation of the fully persistence search tree problem and describe our data structure for the static case. In Sections 3–5 we present a simplified dynamic result, Theorem 1.2 below, where we assume an upper bound $\overline{N}$ of the total number of updates to be performed and use $\overline{N}$ as a parameter in our construction, and Clone uses $O(\log_B \overline{N})$ I/Os. In

Table 1: I/O bounds for previous and new results for partial and full persistent search trees in external memory. All structures use linear space.

	Reporting	Update
Partial persistence		
Lomet and Salzberg [23]	$O(N_v/B)$	$O(\log_B N_v \log_B N)$
Becker, Gschwind, Ohler, Seeger, Widmayer [7] / Varman and Verma [26]	$O(\log_B N + K/B)$	$O\left(\log_B^2 N_v\right)$ am.
Arge, Danner, Teh [4]	$O(\log_B N + K/B)$	$O(\log_B N)$ am.
Full persistence		
Lanka and Mays [22]	$O((\log_B N_v + K/B)\log_B N)$	$O\left(\log_B^2 N_v\right)$ am.
Brodal, Sioutas, Tsakalidis, Tsichlas [11]	$O(\log_B N_v + K/B)$	$O(\log_B N_v + \log_2 B)$ am.
This paper	$O(\log_B N_v + K/B)$	$O(\log_B N_v)$ am.

Table 2: I/O bounds of the operations for each improvement of the structure. All structures use linear space.

	Clone	Insert and Delete	Search	Range
Geometric Structure (Sections 2, 3, 4, 5)	$O\left(\log_B \overline{N}\right)$	$O\left(\log_B \overline{N}\right)$	$O\left(\log_B \overline{N}\right)$	$O\left(\log_B \overline{N} + K/B\right)$
Partitioning the Version Tree (Section 6)	$O(\log_B N_v)$	$O(\log_B N_v)$	$O(\log_B N_v)$	$O(\log_B N_v + K/B)$
Lazy Clones (Section 7)	$O(1)$	$O(\log_B N_v)$	$O(\log_B N_v)$	$O(\log_B N_v + K/B)$

Section 3 we describe the dynamic version of our data structure and in Sections 4 and 5 we describe our colored predecessor and point location structures, respectively. In Sections 6–7 we then show how to improve these bounds to those of Theorem 1.1. In Section 6 we show how to make the I/O bounds depend on the size N_v of a version v, instead of the upper bound parameter $\overline{N}$, by splitting the version tree into multiple version trees, where all versions in a version tree have approximately equal sizes. In Section 7 we show how to reduce the cost of Clone from $O(\log_B N_v)$ I/Os to $O(1)$ I/Os by performing *lazy clones*, i.e., the actual cloning is postponed until the first update is performed on a version. An overview of the I/O bounds for each of these improvements is presented in Table 2.

Theorem 1.2. *Let $\overline{N}$ be a parameter giving an upper bound on the total number of updates. Then there exist external-memory fully-persistent search trees that support* Insert, Delete *and* Clone *in amortized $O(\log_B \overline{N})$ I/Os,* Search *in worst-case $O(\log_B \overline{N})$ I/Os, and* Range *in worst-case $O(\log_B \overline{N} + K/B)$ I/Os. The space usage is linear in the number of updates.*

2 STATIC DATA STRUCTURE

Our implementation of a persistent search tree takes a geometric approach. In Section 2.1 we introduce the version tree and version list, concepts that we borrow from [18], and in Section 2.2 we give a geometric interpretation of range reporting queries. In Section 2.3 we introduce our rectangular two-dimensional space partition in the static setting, and in Section 2.4 we summarize the main properties of the rectangular partition that should be maintained by the dynamic structure in Section 3. In Section 2.5 we describe how to support queries at the high level.

2.1 Version Tree and Version List

Following [18], we define the version tree T as follows. The root is version 1. Whenever a new version w is created, it is assigned the smallest unused positive integer value so far. If w is created by cloning v, then w becomes the leftmost child of v. This approach guarantees if T contains s versions, then these are versions $\{1, 2, \ldots, s\}$, that version identifiers are unique, the version identifiers of the children of a node are increasing from right to left, and the version identifiers along a root-to-leaf path are increasing.

Similarly to [18], we derive a *version list* L from T by a preorder left-to-right traversal of T. As version 1 is the root of the version tree T, it is also the first version in L. We assume that L is terminated by version 0. Given two versions v and w, we in the following let $[v, w[$ denote the versions in the version list from v up to w, but excluding w. Note that in the version list $[1, 0[$ includes all nodes of the version list, except for the terminating version 0. With each value x inserted into version v we store the *liveness interval* $[v, \mathrm{succ}(v)[$, where $\mathrm{succ}(v)$ is the successor of v in the version list. If version w is created by cloning version v, then w becomes the leftmost child of v in the version tree, i.e., w should be inserted immediately after v in the version list. Crucial to the definition of the liveness interval, which is also used in [18], is that the later versions w created below v in the version tree are exactly the versions that will be contained in the liveness interval of x, if x was inserted in version v.

In Figure 1 (bottom) the liveness intervals are shown for all values inserted in the example in Figure 1 (top). E.g., the value 2 is inserted into version 1 with liveness interval $[1, 0[$, since only version 1 exists when 2 is inserted. The value 5 is inserted into versions 2 and 8, with liveness intervals $[2, 0[$ and $[8, 6[$, respectively (versions 3, 4 and 5 did not yet exist when 5 was inserted into version 2). Finally, value 7 is inserted in version 1 with liveness interval $[1, 0[$ but again deleted in versions 3 and 7 for liveness intervals $[3, 0[$ and $[7, 6[$, respectively. The result is that value 7 only exists for the remaining version intervals $[1, 7[$ and $[6, 3[$, i.e., versions 1, 2 and 6 in the current version tree.

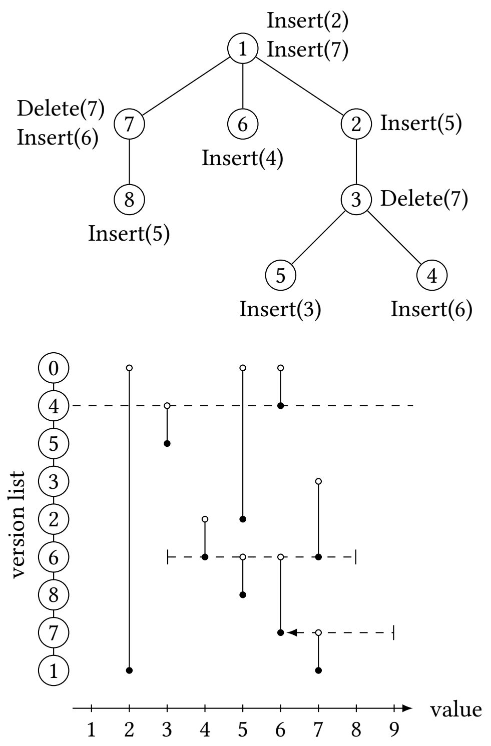


Figure 1: (Top) Version tree illustrating 8 versions of a set after $N = 10$ updates. Versions 1, 2, 3, and 4 contain the sets $\{2, 7\}$, $\{2, 5, 7\}$, $\{2, 5\}$, $\{2, 5, 6\}$, respectively. Updates to a version are shown next to the node of the version tree. (Bottom) The version list, consisting of a left-to-right preorder traversal of the version tree terminated by 0, and the half-open liveness intervals of versions containing a value. Note that the value 7 only is contained in versions 1, 6 and 2 of the set. The topmost dashed line shows that version 4 of the set contains $\{2, 5, 6\}$, the dashed line segment at version 6 shows that the range reporting query $\text{Range}(6, 3, 8)$ has result $\{4, 7\}$, and the bottommost dashed arrow that the query $\text{Search}(7, 9)$ has result 6.

2.2 Geometry of Updates and Queries

Our problem has a natural geometric interpretation in a two-dimensional space, see Figure 1 (bottom). Consider the plane where x-values correspond to values, and the y-value corresponds to the order in the version list. $\text{Insert}(v, x)$ corresponds to inserting the line segment $\{x\} \times [v, \text{succ}(v)[$, whereas $\text{Delete}(v, x)$ to deleting the line segment $\{x\} \times [v, \text{succ}(v)[$, possibly splitting a longer line segment into two disjoint segments. Crucial for the implementation of deletions is that the length of the interval to be deleted (at the time of deletion) is only between two adjacent versions in the version list. The values in $\mathcal{S}_v$ are exactly those vertical line segments intersecting the horizontal line at version v, and the answer to the range reporting query $\text{Range}(v, x, y)$ are exactly the vertical segments intersected by $[x, y] \times \{v\}$, and the answer to $\text{Search}(v, x)$ is the rightmost vertical segment intersected by $[-\infty, x] \times \{v\}$.

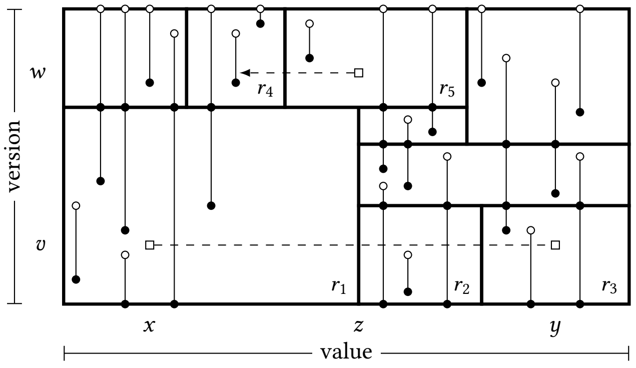


Figure 2: The partitioning of the plane into rectangles, a query $\text{Range}(v, x, y)$ represented by the dashed line between two square endpoints spanning rectangles r_1, r_2, and r_3, and a query $\text{Search}(w, z)$ represented by the dashed arrow originating from a square endpoint, spanning rectangles r_5 and r_4. Note that black dots on vertical segments correspond to the upper endpoint of the segment in the rectangle below and the lower endpoint of the segment in the rectangle above.

2.3 Static Rectangular Partitioning

Assume N updates have been performed on a fully persistent search tree. In the following we present a static structure storing the resulting versions using the geometric representation discussed in Section 2.2 and supports queries in $O(\log_B N)$ I/Os. In Section 3 we extend this structure to support updates.

A key cornerstone of our structure is the parameter

$$R = \Theta(B \log_B N) \ .$$

A crucial property that we will make repeated use of is that, given a list of R segments in external memory stored in $O(R/B)$ blocks, the list can be scanned using $O(R/B) = O(\log_B N)$ I/Os.

We partition the value × version plane into disjoint rectangles $r_1, r_2, \ldots, r_k$, see Figure 2. We denote the full plane by $[-\infty, +\infty[\times [1, 0[$ (by slight misuse of notation we let $[-\infty, +\infty[$ denote the whole value range), and each rectangle r is of the form $[x^r, y^r[\times [v^r, w^r[$, where $x^r < y^r$ are values in the value range, possibly $x^r = -\infty$ and $y^r = \infty$, and v^r and w^r are versions, where w^r is strictly after v^r in the version list. The vertical segments corresponding to liveness intervals of a value (Section 2.1) are split into shorter segments such that segments never overlap with two rectangles, i.e., a segment $\{x\} \times [v, w[$ is repeatedly split into segments $\{x\} \times [v, v^r[$ and $\{x\} \times [v^r, w[$, if a rectangle $r = [x^r, y^r[\times [v^r, w^r[$ is intersected by the segment, i.e., $x^r \leq x < y^r$ and $v^r \in]v, w[$. For a rectangle r, we let S^r denote the set of vertical segments $\{x\} \times [v, w[$ contained in r.

If the rectangle partition is constructed such that each rectangle contains $O(R)$ segments, each rectangle is stored in $O(R/B)$ blocks in external memory, and we have a point location structure that can report the rectangle containing a given a query point using $O(\log_B N)$ I/Os (e.g., the structure in Section 5), then all segments in the rectangle containing a query point can be reported using $O(\log_B N + R/B) = O(\log_B N)$ I/Os.

As seen in Figure 2 a range reporting query may need to report values from multiple rectangles. Assuming that we for each rectangle can output $\Theta(R)$ values, except for the at most two rectangles containing the endpoints of the range reporting query, then each rectangle can be found using $O(\log_B N)$ I/Os by the point location structure, and the I/O cost for querying each of these rectangles can be charged to the output. The total number of I/Os for a range reporting query becomes $O(\log_B N + K/B)$ I/Os, where K is the total number of values reported by the range reporting query.

We require that each non-rightmost rectangle contains $\Theta(R)$ segments spanning all versions of the rectangle. We denote these *spanning segments*, i.e., segments $\{x\} \times [v^r, w^r[$ contained in a rectangle $r = [x^r, y^r[\times[v^r, w^r[$, where $x^r \leq x < y^r$. This ensures that all rectangles considered by a Search and Range query for a given version contain $\Theta(R)$ values, except for possibly the rightmost rectangle. Note that for Search queries, the predecessor must be reported, and by the spanning requirement, if the predecessor does not exist in the current rectangle, then it must either exist in the left neighboring rectangle for this version, or no predecessor exists. For each rectangle the segments are stored in increasing value order, so that range reporting queries can report values in increasing order.

To be able to efficiently compare versions within a rectangle $r = [x^r, y^r[\times[v^r, w^r[$, we maintain the *local version list* $L^r \subseteq L \cap [v^r, w^r]$ for r, that contains the two versions v^r and w^r and all versions that define endpoints of segments in S^r. Since $|S^r| = O(R)$, we also have $|L^r| = O(R)$. The versions in L^r are stored in the same order as in the global version list L. With each version $v \in L^r$ we store the *local version* $\pi^r(v)$ of v in L^r, that just is the position of v in L^r, i.e., an integer in the range $1..|L^r|$. The important property is that v is before w in L^r if and only if $\pi^r(v) < \pi^r(w)$, where the later is a simple comparison between two integers.

We store the segments in S^r only using the local versions. Segment $\{x\} \times [v, w[$ in S^r is stored as the triple $(x, \pi^r(v), \pi^r(w))$, and S^r is stored in increasing value order in $O(|S^r|/B)$ blocks in external memory. When a new version v is added to L^r, all versions w after v in L^r get their local version $\pi^r(w)$ increased by one, and similarly all segments in S^r need to be checked if the local version of their endpoints must be increased by one.

If the version v of a query point (x, v) is not the endpoint of any of the segments in the rectangle r containing (x, v), i.e., $v \notin L^r$, let u be the predecessor of v and w the successor of v in the local version list. Since no version between u and w are endpoints of any segments and all segments are inclusive in the first version and exclusive in the second version, all versions between u and w must have the same values in the value range of r. Since $u \in L^r$, we can use $\pi^r(u)$ to compare against the local versions of the segment endpoints in S^r during a query. In Section 4 we describe a structure that can find the predecessor version in a local version list using $O(\log_B N)$ I/Os, which thus does not increase the I/O cost of queries.

2.4 Structural Requirements

The below list summarizes the main properties required by our rectangular partition from the previous sections. In Section 3 we explain how to maintain them dynamically.

- A rectangle stores $O(R)$ segments.
- The number of spanning segments in a rectangle is $\Theta(R)$, except for rightmost rectangles.
- The segments are stored in increasing value order.
- For each rectangle and segment in a rectangle we store the local versions.

Additionally, we need two more structures. The first structure we call a *colored predecessor* structure which relates global and local versions. The second structure is a *point location* structure which allows us to navigate the rectangular partitioning. Crucial to both structures is that they should avoid comparing the relative order of versions in the global version list, as we also do internally in the rectangles by using local version lists. That is, we need data structures for the following two problems:

- Colored predecessor problem (Section 4): Given a version v and a rectangle r find the predecessor version u of v in the local version list L^r using $O(\log_B N)$ I/Os. For this problem, we think of each rectangle as a unique color.
- Point location (Section 5): Given a query point (x, v) find the rectangle containing the query point using $O(\log_B N)$ I/Os.

2.5 Queries

To perform Range(v, x, y) we first perform a point location query to find the rectangle r containing the point (x, v) and perform a colored predecessor query to find the predecessor u of v in L^r. We find the vertical segments in r intersected by the horizontal segment $[x, y] \times \{u\}$ by scanning through all triples (z, i, j) in S^r and report those z where $x \leq z \leq y$ and $i \leq \pi^r(u) < j$, i.e., the local version interval contains $\pi^r(u)$. If $y^r < y$ we recursively call Range(v, y^r, y). The properties of the partitioning ensure that for each rectangle visited, we will report $\Theta(R)$ values, except possibly the leftmost and rightmost rectangles considered.

To perform Search(v, x), similarly to Range(v, x, y) we first locate the rectangle containing the point (x, v), but now we instead traverse left until the first segment intersection with $[-\infty, x] \times \{v\}$ is identified, which will be the predecessor of x in $\mathcal{S}_v$ (how to move to the rectangle to the left of another rectangle is described in Section 3.2). It is guaranteed that at most two rectangles are required to be scanned due to spanning segments requirement.

Assuming the I/O bounds for the colored predecessor structure in Section 4 and the point location structure in Section 5 are $O(\log_B N)$, we from the discussion in this section get that Range and Search queries can be performed in $O(\log_B N + K/B)$ and $O(\log_B N)$ I/Os, respectively.

3 UPDATES

In Section 2.3 we described how to efficiently perform queries given a rectangular partition of the plane, assuming some structures exist to report rectangles and version predecessors, where each rectangle had to meet a list of requirements as summarized in Section 2.4. In this section, we discuss how to turn these requirements into invariants, such that the structure can be made dynamic, i.e., allow for cloning versions and performing insertions and deletions of values in the different versions. We assume, that on initialization we are given a constant $\overline{N}$, which is an upper bound on the total number of updates to be performed on the structure, and let

$$R = B \log_B \overline{N} \ .$$

When performing a Clone operation, the geometric view remains the same, apart from a horizontal part being "stretched" to make room for the new version. All information stored in the rectangles remains the same, as the new version is not contained in any segment endpoints. The new version must therefore only be inserted in the global version list L and the colored predecessor structure from Section 4. Assuming that a direct pointer to the version cloned in L is provided, the new version can be inserted into L using worst-case $O(1)$ I/Os. A new version can be inserted in the colored predecessor structure using amortized $O(\log_B \overline{N})$ I/Os. In total the Clone operation requires amortized $O(\log_B \overline{N})$ I/Os.

The Insert operation is performed as discussed in Section 2.2 by inserting a value x into the structure at version v is equivalent to adding the segment $\{x\} \times [v, \mathrm{succ}(v)[$. Similarly, to perform a Delete operation of a value x at version v, then the segment $\{x\} \times [v, \mathrm{succ}(v)[$ is to be removed. Note that this potentially means partitioning an already present segment $\{x\} \times [u, w[$, where $u \leq v < w$, into two new segments, where the piece of the segment between versions v and $\mathrm{succ}(v)$ is removed.

As $\mathrm{succ}(v)$ is the version immediately after v in the version list, and all rectangles are exclusive the top version, then the segment $\{x\} \times [v, \mathrm{succ}(v)[$ exists entirely within one rectangle. This rectangle r can be found as the rectangle containing the point (x, v) using the point location structure of Section 5 with $O(\log_B \overline{N})$ I/Os. L^r can then be updated with versions v and $\mathrm{succ}(v)$, which are either inserted or deleted to make L^r reflect the current segment endpoints. The locations in L^r can be found using the colored predecessor structure in Section 4 with $O(\log_B \overline{N})$ I/Os. As new versions are inserted into L^r, this may require the colored predecessor structure to be updated using $O(\log_B \overline{N})$ I/Os for each newly inserted version. Next, the segments of S^r must be updated for the segment $\{x\} \times [v, \mathrm{succ}(v)[$, as well as the local version of all segments in S^r. Note that the local version of all versions present after v in L^r is incremented or decremented by either 0, 1 or 2, depending on how many versions were inserted or deleted from L^r. The segments of S^r can thus be updated in a single scan using $O(\log_B \overline{N})$ I/Os by the requirement on the size of S^r.

Updates can thus be performed using $O(\log_B \overline{N})$ I/Os. They may however break the requirements that were placed on the rectangles to ensure that queries are efficient. In order to make sure that these requirements are met, they are listed in Section 3.1 as invariants, where the asymptotic bounds are replaced by defined constants.

3.1 Invariants

For a rectangle $r = [x^r, y^r[\times [v^r, w^r[$ we define a *spanning segment* to be a segment $\{x\} \times [v^r, w^r[\in S^r$, i.e., the segment represents a value x, which is presented in all versions r spans. The following invariants are then placed on the rectangle:

I_1: $|S^r| \leq 2R$, i.e., a rectangle stores at most $2R$ segments.

I_2: The number of spanning segments in r is at least $\frac{R}{4}$, except if $y^r = +\infty$, i.e., r is a rightmost rectangle, where there is no lower bound on the number of spanning segments.

I_3: The segments in S^r are stored in sorted order by the values the segments represent.

I_4: The list L^r contains precisely the two versions v^r and w^r, and the versions of all endpoints of segments in S^r.

Invariants I_3 and I_4 can easily be maintained upon an update (see Section 3.7). Invariants I_1 and I_2 may however be broken by an update. For insertions, only invariant I_1 may break, and for deletions, both invariants may break. Note that deletions can break both invariants at once.

In order to handle invariant I_1, the issue is the rectangle now contains too many segments. This can be fixed by repeatably splitting the rectangle into two smaller rectangles, until they all do not contain too many segments. These splits may be done either vertically or horizontally, as illustrated in Figure 3.

The invariant I_2 ensures that each rectangle contains sufficiently many spanning segments. If there are too few spanning segments, denoted as an *underflowing* rectangle, and the rectangle is not a rightmost rectangle, then there must exist some rectangles, which are the right neighbors of the underflowing rectangle. These rectangles must then either contain sufficiently many spanning segments or be rightmost rectangles themselves. By performing horizontal splits to align the rectangles, then the underflowing rectangle can be joined with the right neighbors in order to become a rightmost rectangles or to then contain sufficiently many spanning segments. See Figure 4 for an illustration of how this works. This figure only covers the basic idea of merging, and it later turns out, that more cases for merging is needed, which can be seen in Figure 5. Note that merge is performed by making multiple primitive rectangle transformations.

A merge operation takes time proportional to its number of neighbors. Therefore, we place an upper bound t_N on the number of right neighbors, for some integer constant $t_N \geq 2$. When making a merge operation, the top and bottom right neighbors are split, which if they are close to underflowing will not be an issue, as only one of the two rectangles they are split into will remain close to underflowing. If however there only is one right neighbor, and it is close to underflowing, it may be split twice, and make two rectangles, which are close to underflow. In this case, one of the resulting rectangles may be merged with all of its left neighbors, which is symmetrical to merging to the right, see Figure 5b. Note that there must then also be an upper bound t_N on the number of left neighbors.

This thus adds an extra invariant for each rectangle r, with N_r^R and N_r^L denoting the number of neighbors to the right and left of rectangle r respectively.

I_5: $N_r^R \leq t_N$ and $N_r^L \leq t_N$, i.e., the number of neighboring rectangles to rectangle r is at most t_N on both sides.

Note that horizontal splits may increase the number of neighbors in some rectangles. However, if some rectangle has too many neighbors, then it can be split horizontally, to distribute the number of neighbors between two rectangles, and thus decrease the number of neighbors each rectangle has.

3.2 Finding the Neighbors

For the construction, we must be able to find the neighbors of a given rectangle for merging and detecting neighbor overflow. To

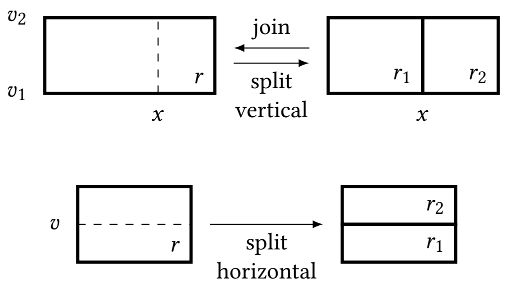


Figure 3: Primitive rectangle transformations: Vertical split and join at value x, and horizontal split at version v.

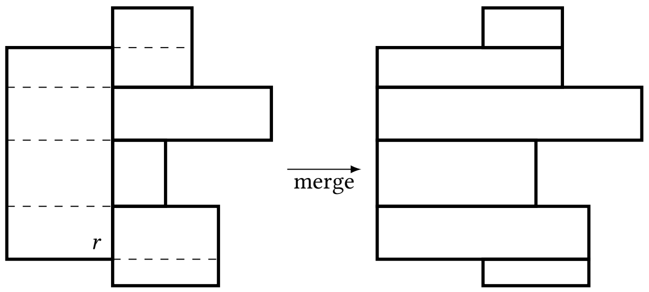


Figure 4: Merging of a rectangle r with its right neighboring rectangles. The dashed lines represent the location that the rectangles are horizontally split, before they are vertically joined.

find the neighbors to the right of a rectangle, we use that the rectangles are exclusive in the top and right coordinates. By repeatedly querying the endpoints using the point location structure from Section 5, all right rectangles can be found from the bottom up. To find the left rectangles, we need a point inside the left neighbor. Such a point can be found by maintaining a search tree over all rectangle start points and searching for the predecessor of the current left side of the rectangle. Each query uses $O(\log_B \overline{N})$ I/Os, and as the number of neighbors is bounded by the constant t_N, the total to find all neighbors is $O(\log_B \overline{N})$ I/Os.

3.3 Potential Functions

Updates to the persistent structure, when ignoring the potential cost of updating the partition of rectangles to restore the rectangle invariants, can be done with $O(\log_B \overline{N})$ I/Os. Splitting and merging rectangles, in order to maintain the invariants, then uses further I/Os, which affect the update time. We amortize the cost of this rebalancing over the updates. In this section, we state the potential function Φ driving the analysis of our structure.

Horizontal and vertical splits are triggered by the rectangle having too many segments. Vertical splits are performed if there are many spanning segments and horizontal splits if there are many non-spanning segments. There must thus be a large potential when there are many spanning and non-spanning segments. The merge is triggered by an underflow in the number of spanning segments. The potential must therefore be large when there are few spanning segments. Finally, rectangles are split when there are too many neighbors on one side, so the potential must be large when there are many neighbors on the same side.

This leads to the following potential function Φ, which is the weighted sum of four potential functions, with S_r denoting the number of spanning segments in rectangle r, and I_r denoting the number of internal endpoints of segments in r, i.e., endpoints of segments, which are not equal to the bottom or top version of r. The constants c_O, c_U, c_I, and c_N are determined in Section 3.6.

$$\Phi = (c_O \cdot \Phi_O + c_U \cdot \Phi_U + c_I \cdot \Phi_I + c_N \cdot \Phi_N) \cdot R \cdot \log_B \overline{N}$$

The four subpotential functions are again defined as $\Phi_O = \sum_r \Phi_O^r$, $\Phi_U = \sum_r \Phi_U^r$, $\Phi_I = \sum_r \Phi_I^r$, and $\Phi_N = \sum_r \Phi_N^r$, where

$$\Phi_O^r = \max\{0, S_r/R - 1/2\}$$

$$\Phi_U^r = \begin{cases} 0 & \text{if } y^r = +\infty \\ \max\{0, 1/2 - S_r/R\} & \text{otherwise} \end{cases}$$

$$\Phi_I^r = \begin{cases} \max\{0, I_r/R - 1/2\} & \text{if } y^r = +\infty \\ \max\{0, (I_r/R - 1/2)(9 - 16 \cdot \min\{S_r/R, 1/2\})\} & \text{otherwise} \end{cases}$$

$$\Phi_N^r = \max\left\{0, N_r^L - t_N/2\right\} + \max\left\{0, N_r^R - t_N/2\right\}$$

For a rectangle r, we denote the four potentials Φ_O^r, Φ_U^r, Φ_I^r and Φ_N^r for *overflow*, *underflow*, *internal endpoint* and *neighbor potentials*, respectively.

The max functions ensure that the potential is never negative. E.g., for spanning segments, subtracting $R/2$ from S_r insures that each rectangle does not have any potential for the first $R/2$ spanning segments, and then linear potential on the rest. Similarly for the rest of the potential functions.

The potential function Φ_I^r for internal points contains the same base function on the internal point count, but then depending on if the rectangle can contain underflow potential, this base function is multiplied by a function dependent on the amount of underflow. This leads to the value of the function being scaled larger when there is more underflow. As for all non-rightmost rectangles, it must hold that $S_r \geq R/4$, then the *scale value* $9 - 16 \cdot \min\{S_r/R, 1/2\}$ is in the interval $[1, 5]$.

Note that cloning and queries do not alter the potential. The potential must therefore be analyzed only for Insert and Delete updates. Any update can be split into two parts: updating a segment in some rectangle and a potential re-partitioning of the rectangles.

For each of the rebalancing operations, the colored predecessor structure and point location query structure need to be updated for the new rectangles. In Section 4 we describe that inserting a new rectangle into the colored predecessor structure requires amortized $O(R \log_B \overline{N})$ I/Os, as each rectangle contains $\Theta(R)$ versions, and each version can be inserted in amortized $O(\log_B \overline{N})$ I/Os. Further, we describe in Section 5 that inserting new rectangles into the point location query structure requires amortized $O(R \log_B \overline{N})$ I/Os. As both operations must be performed with the released potential, then a rebalancing operation needs to release $R \log_B \overline{N}$ potential to pay for the work. In Section 3.5 and 3.6 we show that the factor preceding $R \log_B \overline{N}$ in Φ decreases by at least 1 for every rebalancing operation.

For each update operation without rebalancing, Φ may increase by no more than $O(\log_B \overline{N})$ for updates to use amortized $O(\log_B \overline{N})$ I/Os. We achieve this by showing in Section 3.4 that the factor preceding $R \log_B \overline{N}$ in Φ increases by $O(1/R)$. Table 3 summarizes how the potential changes with and without rebalancing.

3.4 Update without Rebalancing

When performing update operations without rebalancing the rectangles, the partitioning of the rectangles remains unaltered, and thus the neighbors of all rectangles are unchanged. This results in $\Delta\Phi_N = 0$ for both update operations.

Insert. For an insert operation, a segment $\{x\} \times [v, \mathrm{succ}(v)[$ is inserted into a rectangle $r = [x^r, y^r[\times[v^r, w^r[$. This may create a spanning segment if the rectangle only spans a single version, or the inserted value is inserted into the only version in the rectangle, where it was non-present, e.g., r already contains segments $\{x\} \times [v^r, v[$ and $\{x\} \times [\mathrm{succ}(v), w^r[$. This leads to $\Delta\Phi_O \leq 1/R$. By creating a spanning segment, the underflow of spanning segments may decrease but not increase, resulting in $\Delta\Phi_U \leq 0$. The inserted segment may also create at most two new internal points in the rectangle. As the scale value is at most 5, then $\Delta\Phi_I \leq 10/R$.

Delete. For a delete operation, a segment $\{x\} \times [v, \mathrm{succ}(v)[$ is removed from a rectangle $r = [x^r, y^r[\times[v^r, w^r[$. This can then break or delete a spanning segment, which increases the underflow of segments, leading to $\Delta\Phi_U \leq 1/R$. As the number of spanning segments can only decrease, then $\Delta\Phi_O \leq 0$. By breaking a segment, up to two new internal endpoints may be created in the rectangle. The two new internal points then increase the potential function by at most $10/R$. The decrease of spanning segments in the internal point potential function then leads to a further increase, since the scale value of Φ_I^r can increase. As the number of segments is at most $2R$, and at least $R/4$ are spanning, then the number of internal points is at most $7/2 \cdot R$, leading to an increase of at most $(7/2 - 1/2) \cdot 16/R = 48/R$. In total $\Delta\Phi_I \leq 58/R$.

3.5 Rebalancing Rectangles

If an update in rectangle r causes the invariants to be violated, we perform three phases of rebalancing operations to restore them. First, if the update is a deletion that causes the number of spanning segments to fall below $R/4$ then we perform a single merge to restore invariant I_2. The resulting rectangles contain up to $6R+1$ segments as at most 3 rectangles are joined. Second, invariant I_1 is restored in all rectangles containing more than $2R$ segments by performing $O(t_N)$ vertical and horizontal splits as described in Section 3.1. The first two phases may cause rectangles to have many neighbors. In phase three, we fix this by making neighbor splits until all rectangles have fewer than t_N neighbors, which restores invariant I_5. If the update is an insertion or deletion that causes the rectangle to contain more than $2R$ segments but not too few spanning segments, rebalancing starts from the second phase. In the remainder of this section, we show how the different subpotential functions change for every rebalancing operation.

Vertical split. We perform a vertical split only if the number of segments is $|S^r| > 2R$, and the number of spanning segments is $S_r \geq R$. The split is performed at the median value of the spanning segments. This evenly partitions the spanning segments, such that both new rectangles contain $\geq R/2$ spanning segments. Thus, $\Delta\Phi_U = 0$. For Φ_O, due to the max function and that there now are two rectangles, the potential is released from the threshold in both the new rectangles, i.e., $\Delta\Phi_O = -1/2$.

For the internal points, there is no guarantee as to how they are distributed, but in any case, then $\Delta\Phi_I \leq 0$, since the number of internal points is unchanged.

When splitting the rectangle, all left neighbors become left neighbors of the left rectangle, and all right neighbors become right neighbors of the right rectangle, thus not altering the neighbor potential. Since $t_N \geq 2$, the two new rectangles being neighbors do not yield any increase in potential, and thus $\Delta\Phi_N = 0$.

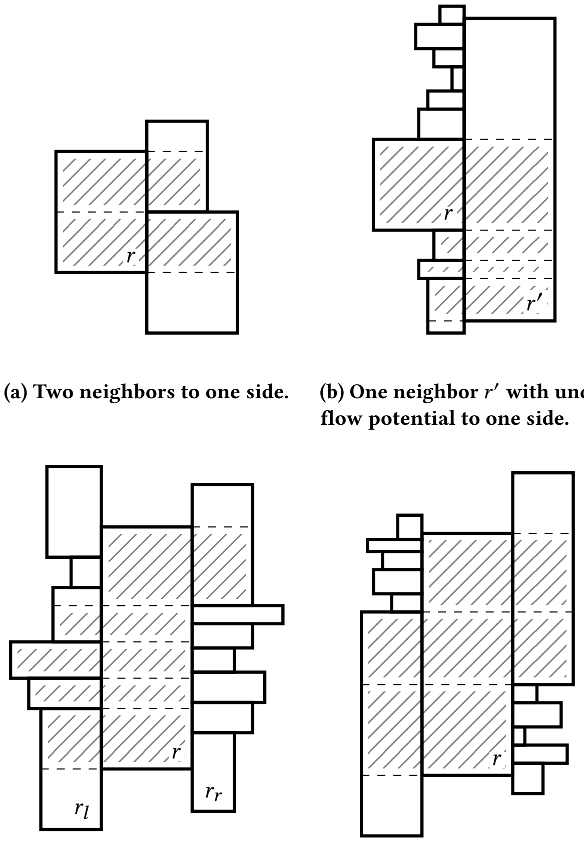
Horizontal split. We perform a horizontal split only if the number of segments is $|S^r| > 2R$, and the number of spanning segments is $S_r < R$. Since each of the at least R non-spanning segments contributes at least one internal endpoint, then $I_r \geq R$. The median version among the internal endpoints is then found using the local versions, at which the split will occur. This evenly partitions the internal points into the new top and bottom rectangles, with some endpoints landing on the split line, which will no longer be internal points. If the rectangle has no underflow potential, a similar analysis as for vertical splits leads to $\Delta\Phi_I = -1/2$ and $\Delta\Phi_U = 0$.

If the rectangle has some underflow potential, then the scale value of the internal point potential function is larger than one. During the split, new spanning segments may be created, but worst-case no new spanning segments are created, and the underflow potential is transferred to both new rectangles. This results in $\Delta\Phi_U \leq 1/2 - S_r/R$ and $\Delta\Phi_I \leq -1/2\,(9 - 16S_r/R)$. As the rectangle, in this case, has underflow potential, then it must be a non-rightmost rectangle, and thus $R/4 \leq S_r < R/2$.

When splitting horizontally, all spanning segments in the original rectangle become spanning in both new rectangles. Any other segment may become spanning in at most one of the two rectangles. In total, each segment may yield one extra spanning segment compared to before the split. As there are at most $6R+1$ segments, and there is no potential for the first $R/2$ new spanning segments, then $\Delta\Phi_O \leq (6R + 1 - R/2)/R \leq 6$, assuming that $R/2 \geq 1$.

Consider one side of neighbors. The split distributes these neighbors among the top and bottom rectangle, with at most one becoming the neighbor of both. None of the two new rectangles can have more neighbors than the original rectangle. If one rectangle has the same number of neighbors as the original rectangle, then the other has one neighbor. The neighbor potential thus does not increase internally among the rectangles. However, the split leads to the one neighboring rectangle at the split point, getting an additional neighbor. Considering both sides, this leads to $\Delta\Phi_N \leq 2$.

Neighbor split. A neighbor split occurs when the number of neighbors on one side of a rectangle is strictly larger than t_N, the number of segments is $\leq 2R$ and the number of spanning segments is $S_r \geq R/4$ if the rectangle is non-rightmost. The median version is selected among the versions where the neighbor changes and a horizontal split is made at this version. This distributes the neighbors evenly among the new top and bottom rectangle, thus decreasing the potential. However, for the other side, one split is made, leading



r
r
r'
r
r_l
r_r
r
(a) Two neighbors to one side.
(b) One neighbor r' with underflow potential to one side.
(c) Multiple neighbors to both sides, with no overlap.
(d) Multiple neighbors to both sides, with overlap.

Figure 5: The different merge cases, with the underflowing rectangle being r, apart from when r is a leftmost rectangle. The dashed lines represent the location where the rectangles are horizontally split. The grayed areas indicate the newly joined rectangles.

to a potential increase of at most one neighbor. In total, this results in $\Delta\Phi_N \leq -t_N/2 + 1$.

By similar arguments as for horizontal splits, the at most $2R$ segments lead to $\Delta\Phi_O \leq 2R/R - 1/2 = 3/2$ and $\Delta\Phi_U \leq 1/2 - {R/4}/R = 1/4$. Note, however, that they cannot both increase. Since the internal points are distributed among the top and bottom rectangle and the scale value can only decrease, then $\Delta\Phi_I \leq 0$.

Merge. A merge occurs, when a rectangle r is underflowing, i.e., when $S_r < R/4$ and the rectangle is not a rightmost rectangle. Note that the number of segments in the rectangle is at most $2R + 1$ and that all neighbors that are non-rightmost rectangles contain at least $R/4$ spanning segments, and at most $2R$ segments. There are then multiple cases for how the merge is performed, in order to make sure not too many new neighbors are created. These cases can be seen in Figures 4 and 5.

Two neighbors on some side. This case is seen on Figure 5a. If one side of r contains exactly two neighbors, the neighbors on this side can be split, as well as r itself, to align the rectangles for joining. The three splits result in $\Delta\Phi_N \leq 3$. One half of the split neighbors are not joined, and using similar arguments as earlier, they cannot increase the potential of the internal points or underflow. The joined rectangles must contain at least $R/2$ spanning segments and thus have no underflow potential. This removes the underflow of r, resulting in $\Delta\Phi_U \leq -1/4$.

There are at most $R/2$ internal points in r with no internal endpoint potential, which now potentially get potential. However, as they are placed into rectangles with at least $R/2$ spanning segments, the scale value is 1. The potential on the already existing internal points may decrease, due to the increase in spanning segments and implied decrease in scale value, however, it cannot increase. Therefore $\Delta\Phi_I \leq 1/2$.

The neighbor rectangles contain at most $2R$ spanning segments, and their overflow potential each increase by at most $2/3$. Rectangle r contains at most $2R+1$ segments. As these segments are distributed into two rectangles, each of which may contain $\geq R/2$ spanning segments, then all distributed segments may be spanning. The spanning segments of r, of which there are $< R/4$, may then become spanning in both new rectangles. The rest of the segments can only be spanning in one of the new rectangles. The extra +1 segment arrives from some old spanning segment that was broken and has a gap of at least one version. Therefore at most $2R - R/4$ of the remaining non-spanning segments may become new spanning segments. This results in $\Delta\Phi_O \leq 5 + 1/4$.

Note that this case is a special case of the basic merge seen in Figure 4.

One neighbor on some side. This case is shown in Figure 5b. If some side has one neighbor r', and the other does not have two neighbors, then it is favorable to join with r', as this limits the neighbor increase in r'. To join r with r', r' may need to be split twice (at the top and bottom of r), leaving two additional rectangles with the same underflow as r'. If r' has no underflow, this is not a problem, and the merge can stop by simply joining r into the split middle section of r'. For the following, assume that r' has underflow potential.

In this case, one of the two created rectangles from r' must have its underflow potential removed to release the potential needed for the operation. Consider the same side of r' as the side r is on. This side must contain some neighbors above and below r. Choose the half, which contains the least number of neighbors, and split the part of r' to this half, and join all of the rectangles, which then must remove the underflow. It must hold that this half has at most $t_N/2$ neighbors, leading to at most $t_N/2 + 2$ splits when factoring in the extra splits to align the neighbors with r' and the split from r. By a similar argument as above, it must hold that $\Delta\Phi_U \leq -1/4$, as the underflow potential of r is released.

The at most $2R$ segments of r' are split into at most $t_N/2 + 2$ rectangles, where they may be new spanning segments in all but one of them, increasing the overflow potential by $\leq t_N + 2$. Rectangle r was underflowing and had less than $R/4$ spanning segments, and therefore the first $R/4$ spanning segments added to r from r' do not increase the potential. The last split rectangle increases the potential by $3/2$ by a similar argument as above. In total resulting in $\Delta\Phi_O \leq t_N + 2 - 1/4 + 3/2 = t_N + 3 + 1/4$.

Consider the internal points of r', where some are added to r and the rectangles below or above r. By a similar argument as above, it also holds that $\Delta\Phi_I \leq 1/2$.

Assume that r' has k neighbors on the side r is on, and only consider the change in the neighbor potential for these rectangles. Then at most $\lceil k/2 \rceil$ neighbors will change from being a neighbor of r' into being a neighbor of other rectangles. The rounding is when k is odd, as there may be $\lfloor k/2 \rfloor$ neighbors on either side of r, and r is one of the rectangles which no longer is a neighbor of r'. Worst case, they all become neighbors of the same rectangle, increasing the neighbor potential in this rectangle by at most $\lceil k/2 \rceil$. The difference in potential of r' is at most $\max\{0, \lfloor k/2 \rfloor - t_N/2\} - \max\{0, k - t_N/2\}$. Lastly, an aligning split increases the neighbor potential by at most 1. In total, the potential increase is maximized for $k = t_N/2$ resulting in $\Delta\Phi_N \leq t_N/4 + 2$.

Note that this case uses the basic merge seen in Figure 4 twice, once for merging r into r', and once to merge the top or bottom rectangle of r' into its neighbors.

Multiple neighbors on both sides. In this case, many splits must be made. If r is joined with the neighbors to the left, this may increase the number of neighbors for the rectangles on the right side. If, however, all of these right neighbors only originally have r as a left neighbor, then the neighboring potential for these rectangles cannot increase in total, as the neighbor potential r has on the right is released.

This holds for all right neighbors, except for the top and bottom. If r is split to align with these, it can then freely be split in between. To not increase the number of neighbors of the top and bottom rectangle on the left, the splits are made to align with the side, which covers most of r. This can be seen in Figure 5c, where r is split at the top of r_l instead of at the top of r_r, as r_l covers most of r. The top of both rectangles cannot however be directly compared, apart from checking if they are identical, but by making a point query at (x^{r_r}, w^{r_l}), i.e., at the top value of r_l, but at the left side of r_r, then this point will be inside r_r only if the top of r_l is below the top of r_r. Similarly for the top neighbors.

The selected top and bottom neighbors may overlap with the part of r they cover, as seen in Figure 5d, further creating two cases. This happens only when the selected neighbors are from each side. If they do not overlap but exactly cover r, then this reduces to the case with two neighbors. Note that overlapping can be checked with a point query, like above. For both cases, it holds that $\Delta\Phi_U \leq -1/4$, as the underflow potential of r is removed, and no new underflow potential is introduced, and $\Delta\Phi_I \leq 1/2$ by similar arguments as above, as the internal points of r are distributed into rectangles with no underflow potential.

No overlap. This is the case illustrated in Figure 5c.In this case, four splits are initially made, two on the selected neighbors and two in r, and potentially a fifth split to align the space between the top and bottom split. All joined rectangles in between become neighbors of rectangles that previously only had r as a neighbor, therefore not increasing the potential. In total, the neighbor potential may only increase for the 3 splits outside of r, resulting in $\Delta\Phi_N \leq 3$. Rectangle r is split at most $t_N - 1$ times. The increase in overflow potential for the other three split rectangles is at most $3/2$ each, by a similar argument as above. In r there are at most $R/4$ spanning segments, which may be spanning at most t_N rectangles, and the remaining segments may be spanning in at most $t_N - 1$ rectangles, by similar argument as above. In total resulting in $\Delta\Phi_O \leq 2t_N + 3 + 3/4$.

Overlap. This is the case illustrated in Figure 5d. In this case, the selected neighbors overlap vertically. By further splitting these neighbors, r can be joined into three rectangles. The splits in the neighbors result in $\Delta\Phi_N \leq 4$. By similar arguments as above, the increase in overflow potential for r is at most $4+1/4$. In the neighboring rectangle, there are at most $2R$ segments, which all may become spanning in two new rectangles. Each of the three rectangles does not have potential for the first $R/2$ spanning segments, resulting in total $\Delta\Phi_O \leq 10 + 3/4$.

Leftmost rectangles. Rectangles at the rightmost side do not have underflow potential and thus cannot trigger a merge. Rectangles at the leftmost side, however, only have neighbors on the right side. In this case, r must be merged with the right neighbors, however, as it has no left neighbors, the neighbor potential on this side cannot increase. This is done as seen in Figure 4. By similar arguments as earlier, it can be shown that $\Delta\Phi_U \leq -1/4$, $\Delta\Phi_I \leq 1/2$, $\Delta\Phi_N \leq 2$ and $\Delta\Phi_O \leq 2t_N + 1 + 1/4$.

3.6 Potential Function Constants

The differences in the subpotential functions, as calculated in Section 3.4 and Section 3.5 can be seen summarized in Table 3. For the insert and delete rows at the top, the differences are without rebalancing.

In Φ the subpotential functions are a linear combination. In the table, each rebalancing operation row must decrease by 1 to release $R \log_B \overline{N}$ potential. In order to find working constants c_N, c_O, c_U and c_I, the rows of the table can be viewed as constraint for a linear program, where each row must be less than -1. Some values are dependent on t_N, which is a constant. In order to find a solution, the linear program is tried solved for increasing values of t_N, until a solution is found. For the constraint from horizontal split with underflow potential, $\Delta\Phi_U$ and $\Delta\Phi_I$ are not constants since the scale value can change. However, they are linear bounds in S_r . By creating two constrains for the minimum and maximum value of S_r, which is $R/4$ and $R/2$ respectively as there must be underflow potential in this case, the program may be solved.

The objective function to minimize in the linear program, is chosen to be the sum of the constants. This yields the following constants

$$t_N = 22 \quad c_N = 573 \quad c_O = 2 \quad c_U = 22904 \quad c_I = 2754 \ .$$

These constants can then be used to check if the constraint from horizontal split with underflow potential is satisfied, by calculating the left side of the constraint:

$$573 \cdot 2 + 2 \cdot 6 + 22904 \cdot (1/2 - S_r/R) + 2754 \cdot (-1/2 \cdot (9 - 16 S_r/R))$$
$$= 217 - 872 S_r/R \ .$$

This value is large, when S_r is small. The smallest value of S_r is $R/4$, resulting in the maximum value of the left side being -1. The constraint is therefore satisfied, and the solution to the constants is valid.

Table 3: Difference in the subpotential functions for the different update operations and rebalancing operations.

Operation	$\Delta\Phi_N$	$\Delta\Phi_O$	$\Delta\Phi_U$	$\Delta\Phi_I$
Insert	0	$\leq 1/R$	≤ 0	$\leq 10/R$
Delete	0	≤ 0	$\leq 1/R$	$\leq 58/R$
Vertical split	0	$-1/2$	0	≤ 0
Horizontal split ($\Phi_U^r = 0$)	≤ 2	≤ 6	0	$-1/2$
($\Phi_U^r > 0$)	≤ 2	≤ 6	$\leq 1/2 - S_r/R$	$\leq -1/2 \cdot (9 - 16S_r/R)$
Neighbor split	$\leq -t_N/2 + 1$	$\leq 3/2$	$\leq 1/4$	≤ 0
Merge (Figure 5a, $N_r^L = 2 \vee N_r^R = 2$)	≤ 3	$\leq 5 + 1/4$	$\leq -1/4$	$\leq 1/2$
(Figure 5b, $N_r^L = 1 \vee N_r^R = 1$)	$\leq t_N/4 + 2$	$\leq t_N + 3 + 1/4$	$\leq -1/4$	$\leq 1/2$
(Figure 5c, $N_r^L \geq 3 \wedge N_r^R \geq 3$, no overlap)	≤ 3	$\leq 2t_N + 3 + 3/4$	$\leq -1/4$	$\leq 1/2$
(Figure 5d, $N_r^L \geq 3 \wedge N_r^R \geq 3$, overlap)	≤ 4	$\leq 10 + 3/4$	$\leq -1/4$	$\leq 1/2$
(Figure 4, $x^r = -\infty \wedge N_r^R \geq 3$)	≤ 2	$\leq 2t_N + 1 + 1/4$	$\leq -1/4$	$\leq 1/2$

3.7 Maintaining Structure inside a Rectangle

The remaining invariants to maintain are I_3 and I_4. For update operations without rebalancing, the updated segments can be inserted into S^r to maintain the order. An update may only add new versions to r, and using the colored predecessor structure, the new versions can be inserted into L^r at the right position. The segments can then be scanned, to update the local versions of the endpoints.

For the rebalancing operations, first note that S^r can maintain the value order during horizontal and vertical splits, as the segments may be partitioned in order, and for vertical joins all segments in one rectangle are before all rectangles in the other, and the lists can then simply be concatenated.

For a horizontal split L^r is split at some version, and the resulting segments in the upper rectangle can be scanned to update the local versions. For a vertical split, L^{r_1} and L^{r_2} must be reconstructed from the endpoints of the segments in the rectangles. As binary merge-sort on $O(R)$ elements uses $O(R/B \log_2 R) = O(R \log_B \overline{N})$ I/Os, then $O(R)$ elements can be sorted a constant number of times using the released potential. This allow for constructing L^{r_1} and L^{r_2} for the resulting rectangles from the endpoints of the segments, by sorting them by the local version in L^r.

When vertically joining two rectangles, two version lists L^{r_1} and L^{r_2} needs to be merged into L^r. Firstly, the start and end of the two list must be equal, as the top and bottom version of the rectangles are equal. Using the colored predecessor structure, before it is updated, the versions may be merged, as the predecessor query can be used to determine the ordering. As the list contains $O(R)$ versions, and each predecessor uses $O(\log_B \overline{N})$ I/Os, the resulting $O(R \log_B \overline{N})$ I/Os can be payed by the released potential. The local ordering on the segments can then be done by a constant number of sorts and scans.

3.8 Space Usage

Each update operation increases the value of the subpotential functions by $O(1/R)$. Each rebalancing operation decreases the subpotential functions by at least 1, and creates $O(1)$ new rectangles. There are at most $\overline{N}$ updates performed on the structure, resulting in $O(\overline{N}/R)$ rectangles. Each rectangle uses $O(R/B)$ blocks of space, i.e., the total space usage is $O(\overline{N}/R) \cdot O(R/B) = O(\overline{N}/B)$ blocks.

4 COLORED PREDECESSOR QUERIES

In this section we sketch our solution to the colored predecessor problem (details are available in the full version of the paper). That is, given a version v and a rectangle r, find the predecessor version of v in the local version list L^r. We consider each rectangle to represent a unique *color*. The global version list L is ordered from left-to-right such that the predecessor of a version is to the left of it. We overload the notation of r such that it also defines the color corresponding to the rectangle r and let $C = \{1, 2, 3, \ldots\}$ be the set of colors. In the point location structure in Section 5 we use the simple case with only a single color ($|C| = 1$) to find the predecessor of a version among the bottom versions of the rectangles. The following theorem states our result for the colored predecessor problem.

THEOREM 4.1. *Let N be a parameter giving an upper bound on the total number of updates to the global version list L and all local version lists L^r. Then there exists a data structure that given a version $v \in L$ and a color r, can find the predecessor u of v in L^r in worst-case $O(\log_B N)$ I/Os. The structure supports insertions of versions in both L and L^r and deletions from L^r in amortized $O(\log_B N)$ I/Os. The space usage is linear in the number of updates.*

We store all versions from the global version list L and all local version lists L^r at the leaves of a B-tree ordered by the global version list, i.e., the same version can appear multiple times. For every local version $v_r \in L^r$, we conceptually color the path from the leaf containing v_r to the root by the color r, i.e., an internal node can have up to $|C|$ colors. The predecessor of a version v among the versions in L^r can now be found by following the path from v towards the root until a node of color r is found that has child of color r that is to the left of the search path, from which the search reverses towards the leaves following the rightmost nodes of color r (starting with a child to the left of the path). To avoid storing a color at every node on the path to the root we observe that when considering the colored paths from the root towards the leaves it suffices to only color the nodes where paths branch. The number of branches is $O(N)$. To achieve query complexity $O(\log_B N)$ we use the idea of *down pointers* [27] to avoid searching in every node among its colors. Finally, we make the structure dynamic using a weight-balanced B-tree [5].

5 POINT LOCATION

In this section, we sketch how to find the unique rectangle containing a query point (v, x), where v is a version and x is a value (details are available in the full version of the paper). Recall that we consider a disjoint rectangular partition of the plane as described in Section 2.3. At first, this appears to be an orthogonal planar point location problem. However, a crucial difference in our setting is that versions (corresponding to the vertical axis) are ordered according to their position in the version list. Because of this, it is hard to simply apply existing algorithms. Instead, we formulate it as a *dynamic ray shooting* problem on the bottom segments of each rectangle. Throughout this section, we assume that given a version v we can quickly find the predecessor of v among the versions represented by bottom segments. This is a special case of the colored predecessor problem we solve in Section 4 and allows us to assume that v is a version corresponding to a bottom segment of some rectangle and that for insertions of new segments, we know its correct position among the existing segments, with regards to the global ordering of versions. The bounds we get are summarized in Theorem 5.1

Theorem 5.1. *There exists a semi-dynamic ray shooting data structure for non-overlapping horizontal segments, supporting ray shooting queries in worst-case $O(\log_B S)$ I/Os and insertions in amortized $O(B \log_B^2 S)$ I/Os, where S is the number of segments inserted. The space usage is $O(S \log_B S)$ blocks. Furthermore, segments are only compared on the vertical axis for equality, that is testing if they are at the same height.*

In Section 3.8 it was shown that $S = O(\overline{N}/R)$ resulting in the promised bound for point location queries in Section 2.4, and the insertions are within the potential released, as described in Section 3.3. Note that the space usage is $O(\overline{N}/B)$ blocks.

To see how this theorem applies to our problem we first observe that it suffices to consider insertions in the structure. Each segment corresponds to the bottom of a rectangle and thus the primitive rectangle transformations (Figure 3) and consequently the merge rectangle transformations (Figures 4 and 5) define how the set of segments change. Crucially, in all cases any point covered by a segment must still be covered, i.e., the area covered by segments only increases. Thus, using only insertions we can make sure that a given ray shooting query finds some segment at the correct version, by only inserting the segments that cover previously uncovered areas. For every version we maintain the real set of segments created by rectangle transformations in a B-tree, sorted on the horizontal axis. Since there are at most S segments we can now find the correct segment to report with an overhead of only $O(\log_B S)$. The overhead for insertions is similar.

Our approach is to store the segments in a segment tree T. For every node u in T, we have a secondary structure with a subset of the versions corresponding (primarily) to segments that span the horizontal interval defined by u. For a query (v, x) we find the predecessor of v in all the secondary structures on the path down to the leaf in T which contains x in its interval, as one of these predecessors will be the segment we are looking for. To determine the real predecessor among the candidate predecessors from the path we perform consecutive predecessor queries efficiently using a variation of fractional cascading [14]. Finally, since we need a dynamic version of the structure we use a weight-balanced B-tree [5].

6 PARTITIONING THE VERSION TREE

In this section, we prove the following theorem which describes how to improve the I/O bounds in Theorem 1.2, such that they depend on the size N_v of the accessed version v instead of the upper bound $\overline{N}$ on the total number of updates.

Theorem 6.1. *Assume we have a data structure for external-memory fully-persistent search trees supporting a sequence of at most $\overline{N}$ updates, for a constant $\overline{N}$, that supports* Insert, Delete *and* Clone *in amortized $O(\log_B \overline{N})$ I/Os,* Search *in worst-case $O(\log_B \overline{N})$ I/Os, and* Range *in worst-case $O(\log_B \overline{N} + K/B)$ I/Os, and uses space linear in the number of updates. Then there exist external-memory fully-persistent search trees supporting* Insert, Delete *and* Clone *in amortized $O(\log_B N_v)$ I/Os,* Search *in worst-case $O(\log_B N_v)$ I/Os, and* Range *in worst-case $O(\log_B N_v + K/B)$ I/Os, and uses space linear in the number of updates.*

The basic idea of our approach is to split the version tree into smaller version trees by cutting out subtrees after a certain number of updates, such that all versions in a version tree have approximately the same size.

To be more precise, let T be a version tree rooted at version v_0. We let $N_0(T)$ be the initial number of insertions in version v_0 before other updates are performed at version v_0. We let $upd_T(v_0)$ denote the number of insertions, deletions and clones (i.e., $|T| - 1$) in T, excluding the initial $N_0(T)$ insertions in version v_0. For a subtree T_v rooted at a non-root node v, we let $upd_T(v)$ denote the number of insertions, deletions and clones (i.e., $|T_v|$) in T_v.

Let c be a constant, where $0 < c < 1$. We maintain the invariant $upd_T(v_0) \leq cN_0(T)$. Crucially, this ensures that $|N_v - N_0(T)| \leq cN_0(T)$ for any version v in T. We split T when $upd_T(v_0) = \lfloor cN_0(T) \rfloor$. We say that a version v is *small* if $N_v < \frac{2(2-c)}{c(1-c)}$. Small versions can be maintained naively as a list of values. If a new version tree arising from a split has a small version as the root, we get rid of the tree and instead represent all small versions in the tree naively. Each version in the tree that is not small becomes a version tree by itself. As soon as a version that is maintained naively is not small, it is converted into a version tree.

Choosing $\overline{N} = (1+c)N_0(T)$ ensures $\overline{N} \leq \frac{1+c}{1-c} N_v$ for all versions v in T, i.e., $O(\log_B \overline{N}) = O(\log_B N_v)$ and the bounds in Theorem 6.1 follow, ignoring the cost for splitting version trees. The split of T will result in four version trees T', T_H, T_L and T_R, some of which may be empty, and we rebuild each of them by simply performing all the updates again to an initial empty version tree. We assume that the version tree and the history of all insertions and deletions for each version are maintained explicitly so that we can easily repeat all operations. This introduces a constant space overhead for each update.

To split T, we first find a *heavy subtree* T_v rooted at a node v with many updates, but where none of the subtrees at the children are heavy. This ensures that by cutting T_v from T many updates must be performed before the next split of T. More formally, the node v must satisfy the following,

(1) Node v is *heavy*, that is $upd_T(v) > \frac{c}{2}N_0(T)$, and
(2) $upd_T(w) \leq \frac{c}{2}N_0(T)$ for all children w of v.

We can always find a node v satisfying the conditions above by following a path of heavy nodes from the root v_0 towards the leaves since the leaves always satisfy the second condition. We now describe the four resulting trees and show that a linear number of updates must be performed in each of them before they are split again.

We construct the subtree $T' = T \setminus T_v$ only if $v_0 \neq v$, where T_v is the subtree of T rooted at v. In version v_0 of T, there might be values that have been inserted but have been canceled again by deletions. In T' the initial insertions in version v_0 are all the insertions in version v_0 of T, including the initial insertions, that have not been canceled by a deletion. Version v_0 of T' contains no deletions. For all other versions in T', the updates are the same as in T. The number of insertions left in v_0 in T' defines the new $N_0(T') \geq N_0(T) - (upd_T(v_0) - upd_T(v)) > N_0(T) - (cN_0(T) - \frac{c}{2}N_0(T)) = (1 - \frac{c}{2})N_0(T)$, as all updates, but the updates in v may be deletions in v_0. Similarly, the number of updates in T' is now $upd_{T'}(v_0) = upd_T(v_0) - upd_T(v) < cN_0(T) - \frac{c}{2}N_0(T) = \frac{c}{2}N_0(T)$. Therefore

$$\frac{upd_{T'}(v_0)}{N_0(T')} < \frac{\frac{c}{2}N_0(T)}{(1-\frac{c}{2})N_0(T)} = \frac{c}{2-c} ,$$

and a gap of at least $\lfloor cN_0(T') \rfloor - upd_{T'}(v_0) \geq \left(c - \frac{c}{2-c}\right) N_0(T') - 1$ future updates must occur in T' before we need to split T'. Note $c - \frac{c}{2-c} > 0$ for all $0 < c < 1$. If v_0 i s not small, i.e., $\frac{2(2-c)}{c(1-c)} \leq N_0(T')$, we always have a gap of at least one update. For $c = \frac{1}{2}$ the gap is of size $\frac{1}{6}N_0(T') - 1$ and $12 \leq N_0(T')$.

We split the subtree T_v into at most three version trees T_H, T_L, and T_R, depending on the number of children of v. Version v exists in all three version trees, but when an operation subsequently refers to version v, we let it refer to version v in T_H. The version tree T_H consists of v and the subtree at the child of v containing the most updates. If v has no children, T_H is still created but only contains v. Next, we greedily partition the remaining children of v into two sets L and R, such that the total number of updates in each is at most $\frac{c}{2}N_0(T)$. We then create the two subtrees T_L and T_R, which are rooted at v with children L and R, respectively. These version trees are only created if they have at least one child. Similarly to the case for the root v_0 of T', for the root v of T_H, T_L, and T_R, we examine all the insertions and deletions on the path from v_0 to v in T and keep only the insertions that are not canceled by a deletion on the path. The resulting set of insertions is the initial set of insertions for v of size $N_0(T_H) = N_0(T_L) = N_0(T_R)$. We define d to be the number of deletions on the path from v to v_0. Since $upd_{T_H}(v) \leq \frac{c}{2}N_0(T)$ by the second condition in the definition of v, and $upd_{T_L}(v) = \sum_{w \in L} upd_T(w) \leq \frac{c}{2}N_0(T)$ (similarly for T_R) by the choice of L and R, the analysis becomes the same for all three version trees with root v. Here we consider the analysis for T_L and note that $d \leq upd_T(v_0) - \sum_{w \in L} upd_T(w)$. The number of initial insertions in v in T_L is exactly the number of initial insertions in v_0 in T, that are not deleted by the d deletions on the path from v to v_0, i.e., $N_0(T_L) = N_0(T) - d$. Therefore

$$\begin{aligned}\frac{upd_{T_L}(v)}{N_0(T_L)} &\leq \frac{\sum_{w \in L} upd_T(w)}{N_0(T) - \left(upd_T(v_0) - \sum_{w \in L} upd_T(w)\right)} \\ &\leq \frac{\frac{c}{2}N_0(T)}{(1-c)N_0(T) + \frac{c}{2}N_0(T)} = \frac{c}{2-c} .\end{aligned}$$

Thus, we get the same gap as for T'.

Finally, we argue about the resulting I/O and space bounds. Each of the four resulting version trees created by a split contains at most $\left(1 + \frac{c}{2}\right) N_0(T)$ updates, i.e., by assumption can be constructed using $O\left(N_0(T) \log_B N_0(T)\right)$ I/Os. Since there have been performed at least $\left(c - \frac{c}{2-c}\right) N_0(T) - 1$ updates since T was created, we can charge the cost of splitting T to these updates, yielding an additional amortized $O\left(\log_B N_0(T)\right)$ cost per update. Since all versions in T have size at least $\left(1 - \frac{c}{2}\right) N_0(T)$, we can restate the I/O cost as amortized $O\left(\log_B N_v\right)$. Similarly, the additional $O(N_0(T_H)/B)$ blocks of space overhead when splitting T for introducing v and all $N_0(T_H)$ initial updates to v in all three versions trees T_H, T_L and T_L, can be charged to the at least $\left(c - \frac{c}{2-c}\right) N_0(T) - 1$ updates to T since T was constructed, i.e., the space usage remains linear.

In the above analysis we showed that by setting $c = \frac{1}{2}$, we are guaranteed for each of the resulting trees, e.g., T', at least $\frac{1}{6}N_0(T') - 1$ updates to T' are required before T' is required to be split. Any choice of $0 < c < 1$ works, and at least $\left(c - \frac{c}{2-c}\right) N_0(T') - 1$ updates are required before a split. The best choice is $c = 2 - \sqrt{2}$, where the lower bound becomes $(3 - 2\sqrt{2})N_0(T') - 1 = 0.1715N_0(T') - 1$, that is slightly better than the $\frac{1}{6}N_0(T') - 1 = 0.1667N_0(T') - 1$ lower bound achieved by setting $c = \frac{1}{2}$.

7 LAZY CLONES

In this section we prove the following theorem which describes how to improve the amortized I/O bound on Clone operations in Theorem 6.1, such that Clone operations use worst-case constant I/Os.

Theorem 7.1. *Assume we have a data structure for external-memory fully-persistent search trees that supports* Insert, Delete *and* Clone *in amortized $O\left(\log_B N_v\right)$ I/Os,* Search *in worst-case $O\left(\log_B N_v\right)$ I/Os, and* Range *in worst-case $O\left(\log_B N_v + K/B\right)$ I/Os, and uses space linear in the number of updates. Then there exist external-memory fully-persistent search trees supporting* Insert *and* Delete *in amortized $O\left(\log_B N_v\right)$ I/Os,* Clone *in worst-case $O(1)$ I/Os,* Search *in worst-case $O\left(\log_B N_v\right)$ I/Os, and* Range *in worst-case $O\left(\log_B N_v + K/B\right)$ I/Os. The space usage is linear in the number of updates.*

The idea of the construction to reduce the I/O bound for Clone operations, is to postpone the actual cloning to the first update to the version and to charge the cost for the cloning to the later update instead. Any version v, which has not been the subject of any updates, must contain the same values as the (locked) parent version u it has been cloned from. Therefore, any clone or query performed on version v can be performed on version u. Hence version v does not need to be explicitly created in the structure. When version v is cloned, it will become locked, and a new unlocked version w is created that is identical to both v and u. Thus, the same result is obtained as if w was cloned from u.

In this way, *lazy clones* can be implemented using a single layer of references. Each version is either a *real* version existing in the underlying structure or a *lazy* version pointing to a locked real version containing the same values. When making a clone w of a lazy version v pointing to a real version u, version v can no longer receive updates and is locked. Version v remains lazy and is never explicitly constructed. The new version w becomes lazy, pointing to the real version u. As a Clone operation always creates a lazy version from some real version, and a real version can be found by traversing at most 1 pointer, then Clone operations use worst-case $O(1)$ I/Os. Any Insert or Delete operation must first check if the operation is performed on a real or lazy version, which uses $O(1)$ additive I/Os. If the version is lazy, it must first be made real, which uses amortized $O(\log_B N_v)$ I/Os, by performing a Clone operation on the underlying data structure given to the construction. After this, the update can be performed in amortized $O(\log_B N_v)$ I/Os, resulting in total amortized $O(\log_B N_v)$ I/Os. Similarly, Search and Range operations on lazy versions are instead performed on the equivalent real versions. The overhead is $O(1)$ additive I/Os. Finally, storing the single layer of references from lazy versions to real versions requires only additive linear space. This concludes Theorem 7.1.

8 CONCLUSION AND FUTURE WORK

This paper presents external-memory fully-persistent B-trees with I/O bounds (Theorem 1.1) matching those of classical B-trees [6]. A natural open question is whether this result can be extended to the update-query trade-off regime by buffering updates as was done for classical B-trees. Adopting our solution seems plausible but nontrivial since it likely requires buffering the point location structure. In the B^ε tree, queries can be performed efficiently since all the relevant buffered updates are on the path in the tree from the root to the query position. However, that would not be the case in the segment tree, which our point location structure is built upon. Another direction is to improve the amortized bounds to instead hold with high probability or worst-case. The main obstacle here seems to be how to handle rectangle rebalancing. These open questions are similar to the improvements to the classical B-tree mentioned in Section 1.2, and likely some of those techniques can also be deployed here.

REFERENCES

[1] Georgy M. Adelson-Velsky and Evgenii M. Landis. 1962. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)* 146 (1962), 263–266. English translation by Myron J. Ricci in Soviet Mathematics - Doklady, 3:1259–1263, 1962..

[2] Alok Aggarwal and Jeffrey Scott Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127. https://doi.org/10.1145/48529.48535

[3] Lars Arge, Gerth Stølting Brodal, and S. Srinivasa Rao. 2012. External Memory Planar Point Location with Logarithmic Updates. *Algorithmica* 63, 1 (2012), 457–475. https://doi.org/10.1007/s00453-011-9541-2

[4] Lars Arge, Andrew Danner, and Sha-Mayn Teh. 2003. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics* 8 (2003), 22 pages. https://doi.org/10.1145/996546.996549

[5] Lars Arge and Jeffrey Vitter. 2003. Optimal External Memory Interval Management. *SIAM J. Comput.* 32 (09 2003), 1488–1508. https://doi.org/10.1137/S009753970240481X

[6] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Informatica* 1 (1972), 173–189. https://doi.org/10.1007/BF00288683

[7] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. 1996. An Asymptotically Optimal Multiversion B-Tree. *The VLDB Journal* 5, 4 (1996), 264–275. https://doi.org/10.1007/s007780050028

[8] Michael A. Bender, Rathish Das, Martin Farach-Colton, Rob Johnson, and William Kuszmaul. 2020. Flushing Without Cascades. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, Shuchi Chawla (Ed.). SIAM, 650–669. https://doi.org/10.1137/1.9781611975994.40

[9] Michael A. Bender, Martín Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. 2017. Write-Optimized Skip Lists. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 69–78. https://doi.org/10.1145/3034786.3056117

[10] Gerth Stølting Brodal and Rolf Fagerberg. 2003. On the Limits of Cache-Obliviousness. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing* (San Diego, CA, USA) *(STOC '03)*. Association for Computing Machinery, New York, NY, USA, 307–315. https://doi.org/10.1145/780542.780589

[11] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. 2012. Fully Persistent B-trees. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*. SIAM, 602–614. https://doi.org/10.1137/1.9781611973099.51

[12] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. 2020. Fully persistent B-trees. *Theoretical Computer Science* 841 (2020), 10–26. https://doi.org/10.1016/j.tcs.2020.06.027

[13] Bernard Chazelle. 1986. Filtering Search: A New Approach to Query-Answering. *SIAM J. Comput.* 15, 3 (1986), 703–724. https://doi.org/10.1137/0215051

[14] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica* 1, 2 (1986), 133–162. https://doi.org/10.1007/BF01840440

[15] Rathish Das, John Iacono, and Yakov Nekrich. 2022. External-memory dictionaries with worst-case update cost. arXiv:2211.06044 [cs.DS]

[16] Erik D. Demaine, John Iacono, and Stefan Langerman. 2007. Retroactive Data Structures. *ACM Transactions on Algorithms* 3, 2, Article 13 (May 2007), 20 pages. https://doi.org/10.1145/1240233.1240236

[17] P. Dietz and D. Sleator. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. ACM, New York, NY, USA, 365–372. https://doi.org/10.1145/28395.28434

[18] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1989. Making Data Structures Persistent. *J. Comput. System Sci.* 38, 1 (1989), 86–124. https://doi.org/10.1016/0022-0000(89)90034-2

[19] Yoav Giora and Haim Kaplan. 2009. Optimal Dynamic Vertical Ray Shooting in Rectilinear Planar Subdivisions. *ACM Transactions on Algorithms* 5, 3, Article 28 (July 2009), 51 pages. https://doi.org/10.1145/1541885.1541889

[20] Leonidas J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. IEEE Computer Society, 8–21. https://doi.org/10.1109/SFCS.1978.3

[21] Scott Huddleston and Kurt Mehlhorn. 1982. A New Data Structure for Representing Sorted Lists. *Acta Informatica* 17 (1982), 157–184. https://doi.org/10.1007/BF00288968

[22] Sitaram Lanka and Eric Mays. 1991. Fully Persistent B+-trees. *SIGMOD Records* 20, 2 (April 1991), 426–435. https://doi.org/10.1145/119995.115861

[23] David B. Lomet and Betty Salzberg. 1993. Exploiting A History Database for Backup. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 380–390. https://dl.acm.org/doi/10.5555/645919.672672

[24] J. Ian Munro and Yakov Nekrich. 2019. Dynamic Planar Point Location in External Memory. In *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA (LIPIcs, Vol. 129)*, Gill Barequet and Yusu Wang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 52:1–52:15. https://doi.org/10.4230/LIPIcs.SoCG.2019.52

[25] Neil Sarnak and Robert Endre Tarjan. 1986. Planar Point Location Using Persistent Search Trees. *Commun. ACM* 29, 7 (1986), 669–679. https://doi.org/10.1145/6138.6151

[26] Peter J. Varman and Rakesh M. Verma. 1997. An Efficient Multiversion Access Structure. *IEEE Transactions on Knowledge and Data Engineering* 9, 3 (1997), 391–409. https://doi.org/10.1109/69.599929

[27] Dan E. Willard. 1985. New Data Structures for Orthogonal Range Queries. *SIAM J. Comput.* 14, 1 (1985), 232–253. https://doi.org/10.1137/0214019

Received 2022-11-07; accepted 2023-02-06