Cache-Oblivious Persistence[⋆]

Pooya Davoodi[1,⋆⋆], Jeremy T. Fineman[2,⋆⋆⋆], John Iacono[1,†], and Özgür Özkan[1]

[1] New York University
[2] Georgetown University

Abstract. Partial persistence is a general transformation that takes a data structure and allows queries to be executed on any past state of the structure. The cache-oblivious model is the leading model of a modern multi-level memory hierarchy. We present the first general transformation for making cache-oblivious model data structures partially persistent.

1 Introduction

Our result is a general transformation to make a data structure partially persistent in the cache-oblivious model. We first review both the persistence paradigm and the cache-oblivious model before presenting our result.

Persistence. Persistence is a fundamental data structuring paradigm whereby operations are allowed not only on the current state of the data structure but also on past states. Being able to efficiently work with the past has become a basic feature of many real world systems, including consumer-oriented software such as Apple's Time Machine, as well as being a fundamental tool for algorithm development. There are several types of persistence which vary in power as to how new versions can be created from existing ones. One can also view persistence as a transformation that extends the operations of an underlying data structure ADT. Data structures, generally speaking, have two types of operations, queries and updates. A query is an operation that does not change the structure, while updates do[1]. A *version* is a snapshot of the data structure at the completion of an update, and updates generate new versions. In all variants of persistence queries are allowed in any version, past or present. The simplest form of persistence is *partial persistence* whereby updates can only be performed on the most recently generated version, called the *present*. The ensuing collection of versions models a linear view of time whereby one can execute operations only in the present and have a read-only view back into the past. A more complicated form of persistence is known as *full* persistence and allows new versions to be generated by performing an

---

[1] Note that, for example, in self-adjusting structures (e.g. [11, 14, 15]) operations considered to be queries are not queries by this definition since rotations are performed to move the searched item to the root, thus changing the structure.

update operation on any of the previous versions; the new version is branched off from the old one and the relationship among the versions form a tree. The most complicated form of persistence is when the underlying data structure allows the combination of two data structures into one. Such operations are called *meld*-type operations, and if meld-type operations are allowed to be performed on any two versions of the data structure, the result is said to be *confluently persistent*. In confluent persistence, the versions form a DAG. In [8], the retroactive model was introduced, which differed from persistence by allowing insertion and deletion of operations in the past, with the results propagated to the present; this is substantially harder than forking off new versions.

Much of the work so far on persistence has been done in the pointer model. The landmark result was when in [9, 13] it was shown that any pointer based structure with constant in-degree can be made fully persistent with no additional asymptotic cost in time and space; the space used was simply the total number of changes to the structure. The first attempt at supporting confluent operations was [10], but the query runtimes could degrade significantly as a result of their transformation. This was largely because by merging a structure with itself, one can obtain an implicit representation of an exponential sized structure in linear steps. In [6] the situation was improved by showing that if a piece of data can only occur once in each structure, then the process of making a structure confluently persistent becomes reasonable.

The cache-oblivious model. The *Disk-Access Model (DAM)* is the classic model of a two-level memory hierarchy; there is a computer with internal memory size M, and an external memory (typically called for historical reasons the *disk*), and data can be moved back and forth between internal memory and disk in blocks of a given size B at unit cost. The underlying premise is that since disk is so much slower than internal memory, counting the number of disk-block transfers, while ignoring all else, is a good model of runtime. A classic data structure in the DAM model is the B-tree [1].

However as the modern computer has evolved, it has become not just a two-level hierarchy but a multi-level hierarchy, ranging from the registers to the main storage, with several levels of cache in between. Each level has a smaller amount of faster memory than the previous one. The most successful attempt to cope with this hierarchy to date has been employing *cache-oblivious algorithms* [12], which are not parameterized by M and B. These algorithms are still analyzed in a two-level memory hierarchy like the DAM, but the values of M and B are used only in the analysis and not known to the algorithm. By doing the analysis for an arbitrary two-level memory hierarchy, it applies automatically to all the levels of a multi-level memory hierarchy.

Cache-oblivious algorithms are analyzed in the ideal-cache model [12], which is the same as the DAM model but for one addition: the system automatically and optimally decides which block to evict from the internal memory when loading a new block. We use the term *cache-oblivious model* to encapsulate both the ideal-cache model and the restriction that the algorithm be cache oblivious. Some DAM-model algorithms are trivially cache oblivious, for example scanning an array of size N has cost $\mathcal{O}(N/B+1)$ in both models. Other structures, such as the B-tree, are parametrized on the value of B and thus cache-oblivious alternatives to the B-tree require completely different methods (e.g. [3, 4]).

Previous persistence results for memory models. In [5] a general method to make a pointer-based DAM model algorithm fully persistent (assuming constant in-degree) was presented where updates incur a $\mathcal{O}(\log B)$ slowdown compared to the ephemeral version. In [2] a partially persistent dictionary for the cache-oblivious model was presented, with a runtime of $\mathcal{O}(\log_B(V + N))$ for all queries and updates, where N is the size of the dictionary and V is the number of updates. Both these results are based on adapting the methods of the original persistence result for the memory model at hand, and are still tied to the view of a structure as being pointer-based. In contrast, we make no assumptions about a pointer-based structure and do not use any of the techniques from [13].

Persistence in the cache-oblivious model. We present a general transformation to make any cache-oblivious data structure partially persistent; this is the first such general result.

In a cache-oblivious algorithm, from the point of view of the algorithm, the only thing that is manipulated is the lowest level of memory (the disk). The algorithm simply acts as if there is one memory array which is initially empty, and the only thing the algorithm can do is look at and change memory cells. As the *ideal cache assumption* states that the optimal offline paging can be approximated within a constant factor online, the algorithm does not directly do any memory management; it just looks at and changes the memory array. Working directly with the memory array as a whole is a necessary feature of efficient cache-oblivious algorithms, and the overriding design paradigm is to ensure data locality at all levels of granularity. We define the space used by a cache-oblivious algorithm, U, to be the highest address in memory touched so far. The runtime of any operation on a particular state of a data structure is expressed as a function of B and M (only), which are unknown to the algorithm and in fact are different for different levels of the memory hierarchy. In partial persistence, another parameter of interest is the number V of updates performed. This is simply defined as the total number of memory cells changed by the algorithm. So, the goal is to keep a history of all previous states of the data structure, and support queries in the past for any cache-oblivious structure while minimizing any additional space and time needed.

Our results. The most straightforward way to present our results is in the case when the ephemeral data structure's runtime does not assume more than a constant sized memory, and does not benefit from keeping anything in memory between operations. These assumptions often correspond to query-based structures, where blocks that happen to be in memory from previous queries are not of use, and are often reflected in the fact that there is no M in the runtime of the structure. In this case our results are as follows: queries in the present take no additional asymptotic cost; queries in the past and updates incur a $\mathcal{O}(\log_B U)$-factor time overhead over the cost if they were executed with a polynomially smaller block size. In particular, a runtime of $t_q(B)$ in the ephemeral structure will become $\mathcal{O}(t_q(\Theta(B^{(1-\epsilon)/\log 3})) \log_B U)$ for any constant $\epsilon > 0$. Thus, as a simple example, a structure of size $\Theta(N)$ and runtime $\mathcal{O}(\log_B N)$ will support queries in the present in time $\mathcal{O}(\log_B N)$, persistent queries in the past and updates in time $\mathcal{O}(\log_B^2 N)$.

Structures where memory plays a role are more involved to analyze. There are two cases. One is where memory is needed only within an operation, but the data structure's runtime is not dependent on anything being left in memory from one operation to the next operation. In this case, as in the above, queries in the present will be run as asymptotically fast as in the ephemeral using a memory of size a constant factor smaller. For queries in the past and updates, our transformation will have the effect of using the runtime of the ephemeral where the memory size is reduced by a $\mathcal{O}(B^{1-(1-\epsilon)/\log 3} \log_B U)$ and a $\mathcal{O}(B^{1-(1-\epsilon)/\log 3})$ factor, respectively. Thus, in this case, a query of time $t_q(M, B)$ in the ephemeral structure will take time $\mathcal{O}(t_q(\Theta(M), \Theta(B)))$ for a query in the present, $\mathcal{O}(t_q(\Theta(M/B^{1-(1-\epsilon)/\log 3} \log_B U), \Theta(B^{(1-\epsilon)/\log 3})) \log_B U)$ for a query in the past, and $\mathcal{O}(t_q(\Theta(M/B^{1-(1-\epsilon)/\log 3}), \Theta(B^{(1-\epsilon)/\log 3})) \log_B U)$ for an update.

When the runtime of a structure depends on keeping something in memory between operations, the runtimes of the previous paragraph hold for updates and queries in the present. But, for queries in the past, this is not possible. It is impossible to store the memory of all previous versions in memory. For queries in the past, suppose that executing a sequence of queries in the ephemeral takes time $t_q(M, B)$ if memory was initially empty. By treating the sequence as one meta query, the results of the previous paragraph apply, and our structure will execute the sequence in $\mathcal{O}(t_q(\Theta(M/B^{1-(1-\epsilon)/\log 3} \log_B U), \Theta(B^{(1-\epsilon)/\log 3})) \log_B U)$ time. Having $t_q(M, B)$ represent the runtime in the ephemeral assuming memory starts off empty could cause $t_q(M, B)$ to be higher than without this requirement. Short sequences of operations that have zero or sub-constant cost will have the largest impact, while long or expensive sequences will not.

Data structures often have runtimes that are a function of a single variable, commonly denoted by N, representing the size of the structure. This letter makes no appearance in our construction or analysis, however, as it would be too restrictive—bounds that include multiple variables (e.g. graph algorithms) or more complicated competitive or instance-based analysis can all be accommodated in the framework of our result. For the same reason we treat space separately and can freely work with structures with non-linear space or whose space usage is not neatly expressed as a function of one variable.

All of our update times are amortized, as doubling tricks are used in the construction. The amortization cannot be removed easily with standard lazy rebuilding methods, as in the cache-oblivious model one cannot run an algorithm for a certain amount of time as this would imply knowledge of B.

A lower bound on the space usage is the total number V of memory changes that the structure must remember. If the number of updates is sufficiently long compared to ephemeral space usage (in particular, if $V = \Omega(U^{\log 3})$), then our construction has a space usage of $\mathcal{O}(V \log U)$.

Overview of methods. The basic design principle of a cache-oblivious data structure is to try to ensure as much locality as possible in executing operations. So, our structure tries to maintain locality. It does so directly by ensuring that the data of any consecutive range of memory of size w of any past version will be stored in a constant number of consecutive ranges of memory of size $\mathcal{O}(w^{\log 3})$. Our structure is based on viewing changes to memory as points in two dimensions, with one dimension being the memory address, and the other the time the change in memory was made. With this view,

our structure is a decomposition of 2-D space-time, with a particular embedding into memory, and routines to update and query.

2   Preliminaries

First we define precisely what a cache-oblivious data structure is, and how its runtime is formulated. A cache-oblivious data structure manipulates a contiguous array of cells in memory by performing read and write operations on the cells of this array, which we denote by A. The array A represents the lowest level of the memory hierarchy. By the ideal cache assumption [12], cache-oblivious algorithms need not (and in fact must not) manage transfers between levels of memory; they simply work with the lowest level.

Ephemeral primitives. The ephemeral data structure can be viewed as simply wishing to interact with lowest level of memory using reads and writes:
  - Read(i): returns $A[i]$.
  - Write(i, x): sets $A[i] = x$.

Ephemeral cache-oblivious runtime and space usage. Given a sequence of primitive operations, we can define the runtime to execute the sequence in the cache-oblivious model. The runtime of a single primitive operation depends on the sequence of primitives executed before it, and is a function of M and B that returns either zero if the desired element is in a block in memory or one if it is not. This is computed as follows: for a given B, view A as being partitioned into contiguous blocks of size B. Then, compute the most recent M/B blocks that have had a primitive operation performed in them. If the current operation is in one of those M/B blocks, its cost is zero; if it is not, its cost is one.

Space usage is simply defined as the largest memory address touched so far, which we require to be upper bounded by a linear function in the number of Write operations.

Persistence. In order to support partial persistence, the notion of a version number is needed. We let V denote the total number of versions so far, which is defined to be the number of Write operations executed so far. Let A_v denote the state of memory array A after the vth Write operation. Then, supporting partial persistence is simply a matter of supporting one additional primitive, in addition to Read and Write:
  - Persistent-Read(v, i): returns $A_v[i]$

3   Data Structure

We view the problem in two dimensions, where an integer grid represents space-time. We say there is a point labeled x at (i, v) if at time v a Write(i, x) was executed, setting $A[i] = x$. Thus the x axis represents space and the y axis represents time; we assume the coordinates increase up and to the right. Let P refer to the set of all points associated with all Write operations. In such a view P is sufficient to answer all Read and Persistent-Read queries. A Read(i) simply returns the label of the highest point in

P with x-coordinate i, and Persistent-Read(v, i) returns the label of the highest point in P at or directly below (i, v). We refer to the point (i, v) to be the value associated with memory location i at time v, that is, $A_v[i]$. We denote by V the number of Write operations performed, which is the index of the most recent version.

All of the points lie in a rectangular region bounded by horizontal lines representing time zero at the bottom and the most recent version at the top, and vertical lines representing array location zero (on the left) and the maximum space usage so far (on the right). At a high level, we store a decomposition of this rectangular region into vertically-separated square regions. For each of these square-shaped regions, we use a balanced hierarchical decomposition of the square that stores the points and supports needed queries, which we call the ST-tree. As new points are only added to the top of the structure, only the top square's ST-tree needs to support insertion. As such, the non-top squares' ST-trees are stored in a more space-efficient manner and as new squares are created the old ones are compressed.

3.1   Space-Time Trees

We define the *Space-Time Tree* or ST-tree which is the primary data structure we use to store the point set P and support fast queries on P. This tree is composed of nodes, each of which has an associated space-time rectangle (which we simply call the rectangle of the node). The tree has certain properties:

- Each node at height h in the tree (a leaf is defined to be at height 0) corresponds to a rectangle of width 2^h. This implies all leaves are at the same depth.
- Internal nodes have two or three children. An internal node's rectangle is partitioned by its children's rectangles.
- A leaf is *full* if and only if it contains a point in P. An internal node is full if and only if it has two full children. If an internal node has three children, one must be full.
- Some rectangles may be three sided and open in the upward direction (future time); these are called *open*, while four-sided rectangles are called *closed*. All open rectangles are required to be non-full.

The above conditions imply that a node's rectangle is partitioned by the rectangles of half the width belonging to its children which we call left, right, and if there is a third one, upper. Each node stores:

- Pointers to the three children (left, right and upper) and parent.
- The coordinates of its rectangle.

Additionally, each leaf node (height 0 and thus width 1) stores the following

- The at most 1 point in P that intersects the rectangle of the leaf.
- The results of a Persistent-Read corresponding to the point at the bottom of the rectangle.

Lemma 1. *If a node at height h is full, then its rectangle intersects at least 2^h points in P.*

Proof. Follows directly from the fact that full nodes have at least two full children, full leaves intersect one point in P contained in their rectangle, and the rectangles of all leaves are disjoint and at the same level.   □

3.2   Global Data Structure

- The variable U will be stored and will represent the space usage of the data structure rounded up to the next power of two.
- The data structure will store an array of $\lceil \frac{V}{U} \rceil$ ST-trees. The last one is called the *top* tree, and the others are called *bottom* trees. The tree's root's rectangles will partition the grid $[1..U] \times [1..\infty]$. Bottom tree's roots correspond to squares of size U; the j-th bottom root corresponds to square $[1..U] \times [((j-1)U+1)..jU]$. The top tree's rectangle is the remaining three-sided rectangle $[1..U] \times [((\lceil \frac{V}{U} \rceil - 1)U + 1)..\infty]$.
- An array storing a log of all V Write operations which will be used for rebuilding.
- A current memory array, call it C, which is of size U. The entry $C[i]$ contains the value of $A[i]$ at the present, and a pointer to the leaf containing the highest point in P in column i; which also contains the value of $A[i]$ at the present.
- A pointer p to the leaf corresponding to the most recent Persistent-Read.

3.3   How the ST-Trees are Stored

The roots of all ST-trees correspond to rectangles of width U and thus have height $\log U$, and structurally are subgraphs of the complete 3-ary tree of height $\log U$.

The top tree is stored in memory in a brute force way as a complete 3-ary tree of height $\log U$ using a biased Van Emde Boas layout [12]. This layout depends on a constant $0 < \epsilon < 1$ and can be viewed as partitioning a tree of height h into a top tree of height ϵh and $3^{\epsilon h}$ bottom trees of height $h(1 - \epsilon)$. Each of the nodes of these trees is then placed recursively into memory. This will waste some space as nodes and subtrees that do not exist in the tree will have space reserved for them in memory. Thus the top ST-tree uses space $U^{\log 3}$.

There will be a level of the van Emde Boas layout that includes the leaves and has size in the range $3^{\log_3 B} = B$ to $3^{(1-\epsilon)\log_3 B} = B^{1-\epsilon}$ nodes. Any path of length k in an ST-tree will be stored in $\mathcal{O}(1 + \frac{k}{\log B})$ blocks. Additionally, we have the following lemma:

Lemma 2. *Any induced binary tree of height h will be stored in $\mathcal{O}(1 + \frac{2^h}{B^{(1-\epsilon)/\log 3}})$ blocks.*

Proof. There is a height h' in the range $\log_3 B$ to $\log_3 B^{1-\epsilon}$ whereby all induced subtrees of nodes at that height will fit into a block. A binary tree of height h will have $2^{h-h'}$ trees of height h', and $2^{h-h'} - 1$ nodes of height greater than h'. Each tree of height h' is stored in memory in B consecutive locations and therefore intersects at most two blocks, thus, even if each node above height h' is in a different block, the total number of blocks the subtree is stored in is $\mathcal{O}(2^{h-h'}) = \mathcal{O}(2^{h-\log_3 B^{1-\epsilon}}) = \mathcal{O}\left(\frac{2^h}{2^{(1-\epsilon)\log_3 B}}\right) = \mathcal{O}\left(\frac{2^h}{B^{(1-\epsilon)\log_3 2}}\right) = \mathcal{O}\left(\frac{2^h}{B^{(1-\epsilon)/\log 3}}\right).$ □

The bottom trees are stored in a more compressed manner; the nodes appear in the same order as if they were in the top tree, but instead of storing all nodes in a complete 3-ary tree, only those nodes that are actually in the bottom tree are stored. Thus the size of the bottom tree is simply the number of nodes in the tree. The facts presented for the top trees and Lemma 2 also hold for the compressed representation.

3.4   Executing Operations

Read(i). We simply return $C[i]$.

Persistent-Read(v, i). The answer is in the tree leaf containing (i, v), the point associated with this operation. We find this leaf by moving p in the obvious way: move the pointer p to parent nodes until you reach a rectangle containing (i, v) or the root of a tree. If you reach a root, set p the root of the appropriate tree, the $\lfloor v/U \rfloor$th one. Then move the pointer p down until you hit a leaf. The answer is in the leaf.

Write(i, x). We call a node a top node if its rectangle is open. We will maintain the invariant that no top node is full. Since Write modifies P by adding a point above all others, this guarantees that the new point will intersect only non-full nodes.

The insertion begins by checking for two special cases. The first is if the memory location i is larger than U, which is an upper bound on the highest memory location written so far. In this case, we apply the standard doubling trick and U is doubled, all the trees are destroyed and re-built by re-executing all Writes which are stored in the log as mentioned in Section 3.2. The current memory array C is also doubled in size.

The other special case is when once every U operations the point associated with the Write is on the $(U + 1)$-th row of the rectangle of the top tree and a new top tree is needed. In this case, the representation of the existing top tree is compressed and copied as a new bottom tree, removing any unused memory locations, and the top tree is reinitialized as a binary tree where the leafs contain values stored in the corresponding cells of C.

Then the main part of the Write operation proceeds as follows.
 - Sets $C[i] = x$
 - Increments V
 - Follows the pointer to the leaf containing (i, V). Note that this is a top node. Add the data to this leaf and mark it full. This means that the point set P now contains the new Write, as it must. However, it is a top node and is full, which violates the previously stated invariant. We then use the following general reconfiguration to preserve that fact that top nodes can not be full.

Reconfiguration. When a node becomes full we mark it as such and proceed according to one of the following two cases. 1) The parent of the node already has three children, in which case the parent also becomes full. We recurse with the parent. 2) The parent of the node has only two children, which implies the parent is still not full. At this point, we close all open rectangles in the subtree of the current node and add a third child to the parent. (These procedures are explained below in more detail.) The rectangle of the new child is on top of the rectangle of the current node.

In other words, when a leaf node becomes full this leads to that leaf node and 0 or more of its ancestors becoming full. We recurse until the highest such ancestor node p and close every open rectangle in its subtree, and add as the third child to the parent of p a sibling node whose open rectangle is on top of p's newly closed rectangle.

Closing rectangles. We close the open rectangles in the subtree of a node by traversing the 2 children of each node that correspond to open rectangles and changing the top side of each open rectangle from ∞ to V.

Adding the third child. In order to add a third child to a node at height $h+1$, we create a complete binary tree of height h whose root becomes the third child. Note that the leafs of this tree contain the answers to the Persistent-Read queries at the corresponding points. We copy this information from C.

Lemma 3. *The amortized number of times a node at height $h + 1$ gains a third child following an insertion into its subtree is $\mathcal{O}(\frac{1}{2^h})$.*

Proof. We use a simple potential function where adding a third child on top of a node p at height h has cost 1, and all top nodes at height $l \leq h$ in the subtree of p have a potential of $\frac{1}{2^{h-l}}$ for each full child they have.

Observe that each step during the handling of an insert, a top node with two full children becomes no longer a top node and it is marked as full. Thus, since the potential difference at the level of p matches the cost, the amortized cost at any other level is zero except for at the leaf level where the amortized cost is $\frac{1}{2^h}$. □

4   Analysis

4.1   Space Usage

Lemma 4. *The space usage of a bottom tree is $\mathcal{O}(U \log U)$.*

Proof. The proof is deferred to the full version [7]. □

Lemma 5. *The space usage of the entire structure is $\mathcal{O}(U^{\log 3} + V \log U)$.*

Proof. The top tree uses space $\mathcal{O}(U^{\log 3})$. From Lemma 4, the $\mathcal{O}(V/U)$ bottom ST-trees use space $\mathcal{O}(U \log U)$ each. The log of Write operations is size $\mathcal{O}(V)$. The current memory array is size $\mathcal{O}(U)$. Other global information takes size $\mathcal{O}(1)$. □

4.2   Comments on Memory Allocation

According to the ideal cache assumption, in the cache-oblivious model the runtime is within a constant factor of that which has the optimal movement of blocks in and out of cache. Thus, we can assume a particular active memory management strategy and a runtime of using this strategy is an upper bound of the runtime in the cache-oblivious model. In particular, we can base our assumed memory management strategy on the one an ephemeral structure would use on the same sequence of operations, with particular memory and block sizes.

Globally, we assume the memory M is split into three equal parts, with each part dedicated to the execution of each of the three primitive operations.

4.3   Analysis of Read

Theorem 6. *Suppose a sequence X of Read operations takes time $T(M, B)$ to execute ephemerally in the cache-oblivious model on a machine with memory M and block size B and an initially empty memory. Then in our structure, the runtime is $\mathcal{O}(T(M/3, B))$.*

Proof. Since executing Read operations is simply done using the current memory buffer, and there is a memory of size $M/3$ allocated for Reads, this gives the result. $\square$

4.4   Analysis of Persistent-Read

Lemma 7. *Consider the leaves of the ST-tree corresponding to a subarray of ephemeral memory of size w at a fixed time v ($A_v[i \ldots i + w - 1]$ for some i). These leaves and all of their ancestors in the ST-tree are contained in $\mathcal{O}\left(\frac{w}{B^{(1-\epsilon)/\log 3}} + \log_B U\right)$ blocks*

Proof. Let L be the set of all points corresponding to the memory locations $A_v[i \ldots i + w - 1]$. There are two disjoint rectangles corresponding to nodes in the ST-tree with width at most $2w$ such that the two rectangles contain and partition all elements of L. Let x_l and x_r be the nodes corresponding to these two rectangles. Those leaves in subtrees of x_l and x_r corresponding to points in L are the leaves of two binary trees with roots x_l and x_r. Since the height of x_l and x_r are at most $1 + \log w$, by Lemma 2 any binary tree rooted at them will occupy at most $\mathcal{O}\left(1 + \frac{w}{B^{(1-\epsilon)/\log 3}}\right)$ blocks. The nodes on the path between x_l and x_r are stored in $\mathcal{O}(\log_B U)$ blocks. $\square$

Theorem 8. *Suppose a sequence X of Persistent-Read operations executed on the same version takes time $T(M, B)$ to execute ephemerally in the cache-oblivious model on a machine with memory M and block size B and an initially empty memory. Then in our structure, the runtime is*

$$\mathcal{O}\left(T\left(\frac{\frac{1}{3}M}{B^{1-\frac{1-\epsilon}{\log 3}}(1+\log_B U)}, B^{\frac{1-\epsilon}{\log 3}}\right)(1+\log_B U)\right)$$

Proof. Consider executing X ephemerally with memory $\dfrac{M}{3(1+\log_B U)B^{1-\frac{1-\epsilon}{\log 3}}}$ and block size $B^{\frac{1-\epsilon}{\log 3}}$. It keeps in memory $\dfrac{M}{3(1+\log_B U)B^{1-\frac{1-\epsilon}{\log 3}}} \cdot \dfrac{1}{B^{\frac{1-\epsilon}{\log 3}}} = \dfrac{M}{3(1+\log_B U)B}$ ephemeral blocks. Each ephemeral block is stored in $\mathcal{O}\left(B^{\frac{1-\epsilon}{\log 3}}/B^{\frac{1-\epsilon}{\log 3}}\right) = \mathcal{O}(1)$ blocks; including the ancestors of the block this becomes $\mathcal{O}(1 + \log_B U)$. Now, by Lemma 7 the memory blocks needed to keep the leaves representing $\dfrac{M}{3(1+\log_B U)B}$ consecutive memory locations of length B in the persistent structure and their ancestors is $\dfrac{M}{3(1+\log_B U)B} \cdot (1 + \log_B U) = \frac{M}{3B}$. Thus they can all be stored in the third of memory allocated to Persistent-Read operations ($\frac{M}{3B}$ blocks), and thus moving p to any location in the memory in the ephemeral structure will involve walking it entirely though locations in memory in the persistent structure, and thus will have no cost. Now suppose the ephemeral structure moves one block into memory at unit cost. This could require moving $\mathcal{O}\left(B^{\frac{1-\epsilon}{\log 3}}/B^{\frac{1-\epsilon}{\log 3}} + \log_B U\right) = \mathcal{O}(1 + \log_B U)$ blocks to move the associated

leaves and their ancestors in the persistent structure into memory by Lemma 7. Thus, this is the slowdown factor the persistent structure will incur relative to the ephemeral.

□

4.5 Analysis of Write

We charge the cost of rebuilding to Write operations. This only increases their cost by a multiplicative constant factor since we double U after $\Omega(U)$ Writes.

Every U Writes, we make the existing top tree a bottom tree and create a new top tree. Three steps involved in this process are closing the rectangles of the top tree, compression of the top tree to a bottom tree, and initializing a new top tree and populating the leafs.

The following lemmas bound the cost of these steps.

Lemma 9. *Copying performed to populate the leafs of a newly created subtree of height h, or closing the open rectangles in the subtree of a node at height h costs* $\mathcal{O}\left(\frac{2^h}{B^{(1-\epsilon)/\log 3}}\right)$ *block transfers.*

Proof. We only close rectangles that are open and we copy nodes from the array C to leafs of the new subtree which only has open rectangles. If a node is closed all of its children are closed and do not need to be traversed. Given any node, since at most 2 out of its potentially 3 children can be open, the tree we traverse to close rectangles or copy nodes is a binary tree. By Lemma 2, each of these tasks costs $\mathcal{O}\left(\frac{2^h}{B^{(1-\epsilon)/\log 3}}\right)$ block transfers.   □

Lemma 10. *Compression of the top tree into a bottom tree costs* $\mathcal{O}\left(\frac{U \log U}{B^{(1-\epsilon)/\log 3}}\right)$ *block transfers.*

Proof. The top tree is initially a complete binary tree and is stored in $\mathcal{O}\left(\frac{U}{B^{(1-\epsilon)/\log 3}}\right)$ blocks by Lemma 2. We have to account for the additional blocks used to store the elements inserted and nodes created in the top tree after it was created.

By Lemma 3, $\mathcal{O}(\frac{U}{2^k})$ nodes are added to the top tree at height k. When a node at height k is added to the top tree, it is the root of a complete binary search tree and its subtree is stored in $\mathcal{O}\left(2^k/B^{(1-\epsilon)/\log 3}\right)$ blocks by Lemma 2. This implies that added nodes take an additional $\sum_{j=1}^{\log U} \frac{U}{2^j} \cdot \frac{2^j}{B^{(1-\epsilon)/\log 3}} = \mathcal{O}\left(\frac{U \log U}{B^{(1-\epsilon)/\log 3}}\right)$ blocks.   □

Theorem 11. *Suppose a sequence of k Write operations takes time $T(M, B)$ to execute ephemerally in the cache-oblivious model on a machine with memory M and block size B and an initially empty memory. Then in our structure, the runtime is*

$$\mathcal{O}\left(T\left(\frac{\frac{1}{3}M}{B^{1-\frac{1-\epsilon}{\log 3}}}, B^{\frac{1-\epsilon}{\log 3}}\right) + k\frac{\log U}{B^{\frac{1-\epsilon}{\log 3}}}\right)$$

Proof. To find the item, the analysis is similar to that of Persistent-Read, except we can use the pointers in C to directly go to the item. This removes the $\log_B U$ terms.

We need to bound the cost of operations performed to maintain the ST-tree. Recall that after each insertion, we go up the tree until we find an ancestor p such that its parent is not full. Then, we create a new sibling node q whose leafs are populated with the copies of values stored in the corresponding top leaf nodes of p, and close the rectangles in the subtree of p. Letting the height of p's parent be h, we refer to this sequence of operations as the expansion of a node at height h.

The creation of the new top tree occurs once every U Writes, and thus the amortized cost per Write is $\mathcal{O}(\log U/B^{(1-\epsilon)/\log 3})$ by Lemma 10. Since the expansion of a node at height h happens because there have been at least 2^h insertions since the node was created, by Lemma 9, the amortized cost of a Write operation is at most

$$\frac{\log U}{B^{\frac{1-\epsilon}{\log 3}}} + \sum_{j=1}^{\log U} \frac{1}{2^j} \cdot \frac{2^j}{B^{\frac{1-\epsilon}{\log 3}}} = \mathcal{O}\left(\frac{\log U}{B^{\frac{1-\epsilon}{\log 3}}}\right). \qquad \square$$

We also consider the case when all Write operations are executed in unique memory cells. This is a reasonable assumption when we have update operations involving numerous Write operations where for each memory cell we only need to remember the last Write executed during that update operation.

Theorem 12. *Suppose a sequence of* Write *operations performed on unique cells of A takes time $T(M, B)$ to execute ephemerally in the cache-oblivious model on a machine with memory M and block size B and an initially empty memory. Then in our structure, the runtime is*

$$\mathcal{O}\left(T\left(\frac{\frac{1}{3}M}{B^{1-\frac{1-\epsilon}{\log 3}}\log B}, \frac{B^{\frac{1-\epsilon}{\log 3}}}{\log B}\right)\log_B U\right)$$

Proof. Observe that given M', M'', B', B'' if $B' \leq B''$ and $M'/B' = M''/B''$, then $T(M', B') \geq T(M'', B'')$. Thus, since no two Write operations overlap, we have in the worst case that $k \leq T\left(\frac{\frac{1}{3}M}{B^{1-\frac{1-\epsilon}{\log 3}}\log B}, \frac{B^{\frac{1-\epsilon}{\log 3}}}{\log B}\right) \cdot \frac{B^{\frac{1-\epsilon}{\log 3}}}{\log B}$. The result then follows by substituting k in Theorem 11. $\square$

References

1. Bayer, R., McCreight, E.M.: Organization and Maintenance of Large Ordered Indices. Acta Inf. 1, 173–189 (1972)
2. Bender, M.A., Cole, R., Raman, R.: Exponential Structures for Efficient Cache-Oblivious Algorithms. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, p. 195. Springer, Heidelberg (2002)
3. Michael, A., Bender, E.D.: Demaine, and Martin Farach-Colton. Cache-Oblivious B-Trees. SIAM J. Comput. 35(2), 341–358 (2005)
4. Michael, A.: Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary 53(2), 115–136 (2004)
5. Brodal, G.S., Tsakalidis, K., Sioutas, S., Tsichlas, K.: Fully persistent B-trees. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 602–614 (2012)
6. Collette, S., Iacono, J., Langerman, S.: Confluent persistence revisited. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 593–601 (2012)

7. Davoodi, P., Fineman, J.T., Iacono, J., Özkan, Ö.: Cache-oblivious persistence. CoRR, abs/1402.5492 (2014)

8. Demaine, E.D., Iacono, J., Langerman, S.: Retroactive data structures. ACM Transactions on Algorithms 3(2) (2007)

9. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. 38(1), 86–124 (1989)

10. Fiat, A., Kaplan, H.: Making data structures confluently persistent. J. Algorithms 48(1), 16–58 (2003)

11. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The Pairing Heap: A New Form of Self-Adjusting Heap. Algorithmica 1(1), 111–129 (1986)

12. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. ACM Transactions on Algorithms 8(1), 4 (2012)

13. Sarnak, N., Tarjan, R.E.: Planar Point Location Using Persistent Search Trees. Commun. ACM 29(7), 669–679 (1986)

14. Sleator, D.D., Tarjan, R.E.: Self-Adjusting Binary Search Trees. J. ACM 32(3), 652–686 (1985)

15. Tarjan, R.E.: Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM 22(2), 215–225 (1975)