Traceable Data Structures

Umut A. Acar[*] Guy E. Blelloch[†] Srinath Sridhar[‡] Virginia Vassilevska[§]

September 17, 2006

Abstract

We consider the problem of tracking the history of a shared data structure so that a user can efficiently view any previous version of the structure (persistence), and efficiently recover information about all previous operations performed on the data structure, including both reads and writes (traceability). We present a mechanism that works for any bounded-degree linked structure. The mechanism supports any sequence of m operations in $O(m)$ time assuming a RAM, and in $O(m\alpha(m, m))$ assuming a pointer machine. We show that the bound is tight for a pointer machine.

Applications of traceable data structures are copious. For example, one could implement the technique for protecting privacy, for auditing, error notification and even to automatically dynamize static algorithms. In the case of privacy, the approach could be used for storing data structures with sensitive information. If some information is leaked or improperly used, then it is possible to go back and see who read that data or even to trace through how the data was read. This gives some protection, or at least a deterrent, against improper access to the data. With traceable structures it is also possible to track when an error was introduced into a data structure, or to identify the readers of any erroneous data so they can be notified. In algorithm dynamization, any change to the input of the algorithm changes the output only of the functions that read the change. A traceable data structure allows for all these functions to be found efficiently so they can be reexecuted on the new input.

Classification: Data structures

[*]`umut@tti-c.org`. Toyota Technological Institute, Chicago, IL.
[†]`blelloch@cs.cmu.edu`. Carnegie Mellon University, Pittsburgh, PA.
[‡]`srinath@cs.cmu.edu`. Carnegie Mellon University, Pittsburgh, PA.
[§]`virgi@cs.cmu.edu`. Carnegie Mellon University, Pittsburgh, PA.

1 Introduction

We consider the problem of tracking the history of a data structure so that a user can efficiently view any previous version of the structure (persistence), and efficiently recover information about all operations performed on the data structure, including both reads and writes (traceability). We imagine the data structure is shared by many users who can read and update it. We present a general mechanism that works for any bounded-degree linked structures.

This work extends the seminal work of Driscoll et al. [7] on persistent data structures. In a persistent data structure new versions of the structure can be created which effectively copy the previous version. Although writes (updates) can only be applied to the latest versions (the leaves of a version tree), the user can go back and read the data from any version. Driscoll et al. distinguish between partially and fully persistent structures. In a partially persistent structure the versions have to be in a linear order, and in a fully persistent structure the versions can form a version tree. Driscoll et al. describe an implementation technique that can be applied to general bounded-degree linked structures and show that using their technique all operations (creating new versions, creating new nodes, reading and writing) take constant amortized time for both partial and full persistence. Later work by Dietz extends the results by relaxing the bounded degree constraint [3]. Dietz and Raman [4] and Brodal [1] show how to obtain partial persistence in worst case constant time. Other work on persistence includes work on confluently persistent data structures (e.g. Fiat and Kaplan [8]), purely functional data structures (e.g. Okasaki [15]) and persistent data structures for a specific application (e.g. [2, 13, 6]).

We generalize persistent structures by allowing the tracking of previous operations, including both reads and writes. The interface allows a user, perhaps just a privileged user, to find the next read or write operation that was applied to any node of any version of the structure. For example, the user can go back in time to a particular version and for any node ask to see the next version at which it was read (or written). It can also extract any auxiliary information from that operation—*e.g.*, the user who executed that read or write. The interface can be used to go back and trace how a user traversed a structure. Furthermore, the modifications or accesses of a particular node can be traced efficiently in order.

Algorithmically, the techniques used to achieve efficient traceability differ from those of partial persistence. This is because unlike the partially persistent writes we allow reads to be performed at any time in the past. Writes in the past could affect the future and full persistence tackles this problem by simply assuming non-linear ordering of time stamps. For (partial) traceability however, we show that prior techniques can be adapted to maintain a linear order of time while supporting accesses in the past.

As with previous work on persistent data structures, our framework supports operations on general bounded-degree linked data structures. We assume standard operations that create new nodes, and read and write any field of a node. We distinguish between partially and fully traceable data structures which, just as in the framework of Driscoll et al., differ in that a write to a partially traceable structure can only be performed at the latest version, and a write for a fully traceable structure creates a new leaf of the version tree and writes at that version. For the purpose of tracing we assume all operations are tagged with the user who performed it, either implicitly or explicitly. We assume the linked structure is accessed through a root node.

We present a new abstraction of the node splitting method of Driscoll et al., which we call *interval splitting*. The notion of node splitting is separated from that of writing to a persistent data structure. We view splitting as an operation in itself on the nodes of an *interval data structure* which we call *intervals*. An interval data structure is a persistent data structure which supports a single operation—interval splitting, and whose intervals represent versions of vertices of a linked structure over time. When performed on an interval, the splitting operation splits the interval into two so that the two new intervals split the timeframe of the previous one at *any* time within that timeframe, as specified by the user. The operation also ensures that the number of in- and out-pointers is constant for any interval in the structure. This view of node splitting has some advantages. For example, it allows us to maintain the constant degree invariant no matter what we are tracking—versions of the data structure, reads, any other information stored on the nodes. As long as only a constant number of interval splitting calls are done per update, an amortized constant bound on interval splitting can guarantee that all operations are efficient. Another advantage of this view of node splitting is that a fully persistent write is easy to implement as an interval split followed by a partially persistent write. Recall that as presented by Driscoll et al. [7] one had to create two new node copies which slightly complicated the analysis.

Intervals in an interval data structure are similar to the node copies of Driscoll et al. The main difference is that one can create an interval with *any* start time after the initial creation of a node. For example, in a traceable data structure for every access of a node there is an interval of the node which starts at the time of the access. This enables us to store the intervals of every node in a Split-Find-Insert [10, 12] data structure. By naming the sets in each Split-Find-Insert data structures properly, we can support efficient operations which given a node and a time can return the next or previous time when the node was read or written.

The technique we present supports any sequence of m operations in $O(m)$ time assuming a RAM, and in $O(m\alpha(m', m'))$ assuming a pointer machine where m' is the maximum number of operations performed on a particular node. We show that the given time is optimal for a pointer machine. The space required is bounded by $O(m)$.

After defining the interface of a traceable data structure in Section 3, we provide a formal specification of interval data structures in Section 4 and then extend this specification to an implementation of traceable data structures in Section 5. In Section 6 we show that our technique is tight for a pointer machine. Possible applications of our data structure are described in Section 2.

2 Applications

Applications of traceable data structures are copious. As examples we can implement the technique for protecting privacy, auditing, error notification and even to obtain dynamic algorithms automatically. We outline some of them here.

Privacy Violation and Auditing. Several organizations maintain large databases that contain sensitive documents linked to one another (e.g. phone and medical records). In the case of phone records, a person could contain links to everyone who has been contacted within a certain time frame. If some information is leaked or improperly used it is then possible to go back and see who read that data or to trace through how they read that data. Furthermore, we would also want to examine all other records that were read by the same person. In the case of auditing, if the data was improperly updated at some point in the past it is possible to go back and trace when and by whom it was updated.

Error Notification. Database transactions are simply a series of read and write operations (e.g. bank transactions). Such databases are themselves linked, for instance a bank account contains several other account numbers of the same user or organization. A typical linked traversal of the database, might query for a certain type of account to obtain an account number (reading a pointer). This can be followed by a query to determine the balance on that account number (reading a value) and finally charging a certain amount (updating a value). However, one might find a mistake in the value stored and therefore all the transactions that happened within a time frame could have read incorrect values. It is therefore important to automatically notify the readers of this value that the transaction was invalid. Our data structure allows for all readers to efficiently be traced through one by one.

Dynamizing Algorithms. Finally, we can obtain practical dynamic algorithms directly from static algorithms. We simply run a static algorithm (say a shortest paths algorithm) on the input. This algorithm can be viewed as performing a series of reads on the input, several functional computations each involving a subset of the input data, followed by writing the output over time. If the input changes by a small amount (say an edge deletion), then only a subset of the functions would be affected. Our technique with some additional implementation details can be used to identify the reader functions so that just those functions need to be rerun.

3 Linked and Traceable Data Structures

Linked Data Structures. We define a *linked data structures* as a set of nodes, each of which consists of a bounded number of fields that contain *pointers* to other nodes or primitive values (integers etc.). In our exposition we will sometimes just consider pointer fields to avoid special casing for primitive values. We distinguish one node as the *root* of the data structure—a constant number of roots can be represented by an extra level of indirection through the root. For all our bounds, we assume that every node v has (in- and out-) degree bounded by a constant C_d. A linked data structures supports `newNode`, `read`, and `write` *operations* for creating new nodes, and reading from and writing into fields.

Traceable Data Structures. A traceable data structure is a linked data structure with multiple versions each derived as a copy of an existing version, and which maintains a traceable history of all `read` and `write` operations ever applied to any version. A particular version of a node is referred to as a *handle* to the node. Whenever a version is created from another, it creates a child of the other in a *version tree*. In addition to the version tree, we maintain a total order on the versions based on a pre-order traversal of the version tree. We allow any operation on any version. A write to a version only applies to that version. To enable the inspection of `read`s and `write`s, the user must provide an *identity*, which we assume to be a positive integer (we use identity zero for initialization), when running these operations.

More precisely, we define a traceable data structure as a tuple (V, F, T, H, R, W), where V is the set of nodes; F is the set of fields; T is a set of *versions* along with a partial order $<_p$ based on the version tree and a total order $<$ based on the pre-order traversal of the tree, $T \supseteq \{t_0, t_\infty\}$; H is the set of *handles* such that $H \subseteq V \times T$; R is the set of *reads* such that $R \subseteq \mathbb{Z}^+ \times H \times F \times (V \cup \{\texttt{null}\})$; W is the *writes* such that $W \subseteq \mathbb{Z}^+ \times H \times F \times (V \cup \{\texttt{null}\})$. The set of handles represent all possible access points to the data structure.

Figure 1 shows the specification of the operations for traceable data structures. The `newNode` operation takes a handle and creates a node in the version specified by the handle. The `read` operation takes the identity of the reader, a handle to a node v in version t, and a field and returns the value stored in the field of v in version t. The `write` operation takes the identity of the writer, a handle to a node v in version t, a field, and a value, and stores the value at the specified field of v in version t. The `newVersion` operation takes a handle to the root node in version t, and creates a new handle to the root in a version t_2 that is greater than t but less than all other existing versions.

The `getNextReadHandle` and `getNextWriteHandle` operations return handles to the earliest read and write after the given handle respectively. Operations `getPreviousReadHandle` and `getPreviousWriteHandle` are symmetric and can be defined similarly. These operations can

```
TDS = Traceable Data Structure

function newTDS ($F$) =
    $h \leftarrow$ (root, $t_0$)
    $W \leftarrow \{(0, h, f, \texttt{null}) \mid \forall f \in F\}$
    TDS $\leftarrow (\{root\}, F, \{t_0, t_\infty\}, \{h\}, \emptyset, W)$
    return $h$

function newNode($h$ as $(\_, t)$) =
    $V \leftarrow V \cup \{v\}$, where $v \notin V$
    $h' \leftarrow (v, t)$
    $H \leftarrow H \cup \{h'\}$
    $W \leftarrow W \cup \{(0, h', f, \texttt{null}) \mid \forall f \in F\}$
    return $h'$

function read ($id, h$ as $(v, t), f$) =
    let $(id, (v, t_1), f, v') \in W$, where
        $t_1 \leq_p t \wedge$
        $(\forall (v, t_2) \in H . t_2 \leq_p t \Rightarrow t_2 \leq_p t_1)$
    $H \leftarrow H \cup (v', t)$
    $R \leftarrow R \cup \{(id, h, f, v')\}$
    return $(v', t)$
```

```
// Requires that $t_1 = t_2$
function write($id, h_1$ as $(v_1, t_1), f, h_2$ as $(v_2, t_2)$) =
    $W \leftarrow W \cup \{(id, h_1, f, v_2)\} \setminus \{(id, h_1, f, v) \mid \forall v \in V\}$

// Requires that $h = (root, t)$
function newVersion($h$ as $(v, t_1)$) =
    $T \leftarrow T \cup \{t_2\}$, where $(t_2 \notin T) \wedge (t_1 <_p t_2) \wedge$
                        $(t_1 < t_2) \wedge (\forall t \in T . t \leq t_1 \vee t > t_2)$
    $h' \leftarrow (v, t_2)$ ; $H \leftarrow H \cup \{h'\}$
    return $h'$

function getNextReadHandle ($h$ as $(v, t_1)$) =
    $R_v \leftarrow \{t \mid (id, (v, t), f, v') \in R \wedge t > t_1\}$
    if $R_v = \{\}$ then return null else return $(v, \min R_v)$

function getNextWriteHandle($h$ as $(v, t_1)$) =
    $W_v \leftarrow \{t \mid (id, (v, t), f, v') \in W \wedge t > t_1\}$
    if $W_v = \{\}$ then return null else return $(v, \min W_v)$

function getReadsWrites($h$) =
    $R_h \leftarrow \{(\_, h, \_, \_) \in R\}$ ; $W_h \leftarrow \{(\_, h, \_, \_) \in W\}$
    return $(R_h, W_h)$
```

Figure 1: A specification for operations on traceable data structures.

be composed to list through all reads and writes of a given node in order. The `getReadsWrites` operation returns the reads and the writes of the given handle.

Traceable data structures are called *partially* traceable if `newVersion` and `write` can only be applied to the greatest existing version (excluding t_∞) and *fully* traceable otherwise. Note that the structure does not maintain a version tree explicitly, but the tree is implicit in the way the versions were created and the total order of the versions.

4 Interval Data Structures and Interval Splitting

An interval data structure explicitly represents the different values (or states) that each node of a linked data structure takes over time in the form of (time) *intervals*. The data structure is an abstraction of the node splitting method of Driscoll et al. [7]. An interval data structure supports a single operation, interval splitting, while maintaining key invariants which allow us to efficiently extend the data structure to both a persistent and a traceable data structure. We show an amortized $O(1)$ time bound for interval splitting and a linear space bound in the number of (explicit) interval splits.

An interval data structure relies on an **auxiliary** data structure to store intervals. This data structure provides `init_aux` and `insert_aux` operations for initialization and insertion (see Figure 3). By choosing an auxiliary data structure, interval data structures can be used to implement both persistent and traceable data structures (Sections 5 and 5.2).

4.1 The interval data structure

Given a set of nodes V, a set of fields F, and a set of time stamps $T \supseteq \{t_0, t_\infty\}$ drawn from a totally ordered universe U, we define an *interval data structure* as a tuple (V, F, T, I, ρ), where $I \subseteq V \times T \times T$ is a set of *intervals*, and $\rho : I \times F \rightarrow \{I \cup \texttt{null}\}$ is a *points-to* function. Each *interval* $i \in I$ is a tuple consisting of a node and a *time interval* defined by two time stamps. The *time interval* of an interval $i = (v, t, t')$, denoted by $\tau(i)$, is the half open interval $[t, t')$. We assume that always $t' > t$. We say that an interval i *points-to* another interval j, if for

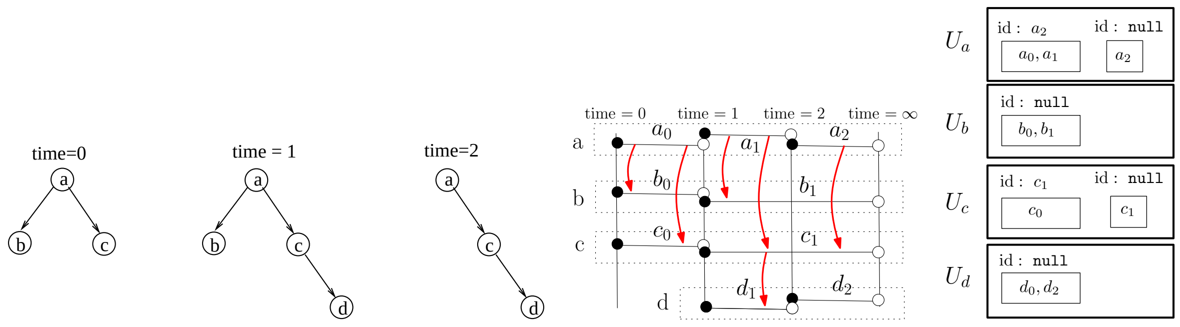

Figure 2: Three versions of a tree, a corresponding interval data structure and the update Split-Find-Insert data structures for the nodes.

some field $f \in F$, $\rho(i, f) = j$. The interval data structure maintains the *time stamps* T in an order-maintenance structure [5], which supports the insert, successor, and compare operations in constant time—we write $\texttt{insertTS}(\cdot)$, $\texttt{successorTS}(\cdot)$, and $t < t'$ for these operations.

The versions used in the specification of traceable data structures (Section 3) are represented by the time stamps of the interval data structure as follows. For partially traceable data structures, the versions are totally ordered and there is a one-to-one correspondence between them and the time stamps. In the case of fully traceable data structures, the versions are partially ordered in the form of a version tree. As in previous work (e.g. [7]), we map this partial order into a total order by a pre-order traversal of the version tree. The time stamps follow this total order.

To elucidate the maintained invariants of interval data structures, we use the following definitions. We say two time intervals, $[t_i, t'_i)$ and $[t_j, t'_j)$ *overlap* if $[t_i, t'_i) \cap [t_j, t'_j) \neq \emptyset$. We say that two intervals i and j *overlap* if their time intervals overlap. We define an order on non-overlapping time intervals by comparing their start times $[t_i, t'_i) < [t_j, t'_j)$ if $t_i < t_j$ assuming $[t_i, t'_i) \cap [t_j, t'_j) = \emptyset$. Given a node v, we write I_v to denote the set of intervals of v; formally $I_v = \{(v, t, t') \mid (v, t, t') \in I\}$. If $i \in I_v$, then we say that i is an interval of (or belongs to) v. Given an interval $i = (u, t, \_)$ and any interval $j \in I_v$, we define $\texttt{image}(i, j)$ as the interval $j' \in I_v$ which covers the start time of i, i.e. $t \in \tau(j')$.

We say that an interval i *dominates* j (and j is *dominated by* i) if $i \in I_u$ and $j \in I_v$ overlap and i points to its image on j, i.e., $\rho(i, f) = \texttt{image}(i, j)$ for some $f \in F$. We write $\texttt{dominators}(j)$ to denote the set of dominators of j. Similarly, we write $\texttt{dominated}(j)$ to denote the set of intervals dominated by j. We write $N(i) = \texttt{dominated}(i) \cup \texttt{dominators}(i)$ for the set of *neighbors* of j. For convenience, we define $\texttt{dominated}_f(i) = \{j \in \texttt{dominated}(i) | \rho(i, f) = \texttt{image}(i, j), f \in F\}$.

Figure 2 shows the versions of a tree at three successive times and the representation of the versions as an interval data structure. The interval data structure consists of intervals drawn as horizontal lines. The intervals of each node are enclosed within a dotted rectangle representing the node. For instance, the set of ordered intervals corresponding to node a is a_0, a_1 and a_2. The pointers (drawn as curved arrows) correspond to the tree pointers. For example, interval a_0 points to and dominates b_0 and c_0. Interval d_2 is defined over time $[2, \infty]$, while c_1 is defined over $[1, \infty]$. Intervals c_1 and d_2 therefore overlap. Since c_1 contains a pointer to d_1 it is also a dominator of d_2. Similarly $\texttt{dominated}(a_1)$ is $\{b_1, c_1\}$. The image of c_1 on d_2 is interval d_1. Note that there are many different interval data structures for representing the same set of trees—e.g., node d need not be represented by 2 intervals here—d_1 and d_2 can be merged.

An interval data structure maintains the following correctness and performance invariants to enable correct and fast access to the history.

```
$P = (V, F, T, I, \rho)$                                        // The data structure

function iSplit $(i)$ =
    if $N(i) > C$ then  choose $t_{med} \in T$ s.t.  $N(i)$ is split in half
                        return (iSplitAtTime $(i, t_{med})$)
    else return $i$

function iSplitAtTime $(i, t)$ =                                // Precondition:  $t \in T \wedge t \in \tau(i)$
    $(v, t_i, t'_i) \leftarrow i$
    if $t = t_i$ then return $i$
    $j \leftarrow (v, t, t'_i)$
    for each $f \in F$ do if $((i, f), k) \in \rho$ then $\rho \leftarrow \rho \cup \{((j, f), \mathtt{image}(j, k))\}$
    for each $k \in I$ and $f \in F$ s.t. $((k, f), i) \in \rho$ do $\rho \leftarrow \rho \cup \{((k, f), \mathtt{image}(k, i))\} \setminus \{((k, f), i)\}$
    $i \leftarrow (v, t_i, t)$; $I \leftarrow I \cup \{j\}$; insert_aux$(i, j)$
    for each $k \in N(i)$ do iSplit$(k)$
    return $j$
```

Figure 3: The `iSplit` and `iSplitAtTime` operation.

Invariant 1 (Correctness)

1. *The intervals of a node do not overlap and cover a continous time interval ending at t_∞.*
2. *If an interval i points to another interval j, then j is the image of i on j.*

Notice that 2 implies that if i points to j, then i and j overlap and hence i dominates j.

Invariant 2 (Performance)
An interval dominates and is dominated by a constant number of intervals (i.e., the number of neighbors is constant).

Motivation for an interval data structure The interval data structure formalizes the node splitting method as given in the Driscoll et al. paper [7] which associates node splitting directly with the write operation. Although this formulation does not offer much new technical insight it has several advantages. By allowing split to be called directly, one can use it to maintain the invariants with any operation which stores new timed information on nodes. Furthermore, by supporting an efficient splitting operation which splits an interval *at a particular time*, reads and writes (partially or fully traceable) are relatively easy to implement and analyze. The ability to create intervals with any start time also allows us to later store read and write information in an interval starting at the read or write time. This makes it possible to use a Split-Find-Insert data structure directly on the node intervals for tracing reads or writes. Moreover, with an explicit split operation, a fully persistent (or traceable) write becomes an easy extension of a partially persistent (or traceable) write as seen in Section 5.

The iSplit and iSplitAtTime operations. An interval data structure supports two types of interval splitting, `iSplit` and `iSplitAtTime`. Given an interval data structure that satisfies the correctness and performance invariants, the `iSplit` and `iSplitAtTime` operations increase the number of intervals while preserving the invariants. For completeness we include pseudo-code (Figure 3) for these operations.

The `iSplit` operation takes an interval $i = (v, t_i, t'_i)$ and splits i, if it has more than C neighbors, where C is a constant to be specified later in the analysis. To split i, the operation finds a suitable time t_{med} that splits the neighbors roughly in half—we show later in this section that picking t_{med} as the median of the starting times of the neighbors suffices. After determining t_{med}, `iSplit` calls `iSplitAtTime` at t_{med}. The `iSplitAtTime` operation takes an interval $i = (v, t_i, t_{i'})$ and a time $t \in \tau(i)$, creates a new interval j for v with time interval $[t, t_{i'})$ and copies the fields of i to j. The end time of i is updated to be t. The operation then updates the intervals pointing to i, which may now point to j, and inserts j into the auxiliary data structure. Splitting i into two can cause the performance invariant to be violated by increasing the number of intervals that the neighbors dominate or are dominated by. To restore the performance invariant, the operation calls `iSplit` on all neighbor intervals of i.

Splitting the time interval of i between j and (the new) i at t ensures the first correctness invariant. The second correctness invariant is ensured by making each new field point to its image and updating the pointers into i. In the next section we show that the performance invariant is also maintained.

4.2 Analysis

Consider an implementation of an interval data structure as a graph, where each node represents an interval. Each node is tagged with the start time of its interval and has bidirectional pointers to its neighbors (defined by the union of its dominators and the intervals that it dominates). This implementation requires maintaining the neighbor pointers consistently when an interval is split. This can be achieved by extending `iSplitAtTime` so that the neigbors with start time greater than t (the time of the split) are now changed to point only to the new interval and all neighbors that contain t point to both intervals. Because of the performance invariant, maintaining the neigbor pointers takes constant time.

We prove that this implementation enables splitting one interval in amortized constant time. We begin by showing that splitting an interval produces two new intervals that each have at most half the number of neighbors as the split interval. Furthermore, we show that the number of new pointers created is bounded by $2 \cdot C_d$ (C_d is the bound on the degree of a node in the underlying linked data structure).

Lemma 1
Suppose `iSplit` splits an interval i into intervals i_1 and i_2. Then, we have 1) $|N(i_1)| \geq \lfloor |N(i)|/2 \rfloor - C_d$, 2) $|N(i_2)| \geq \lfloor |N(i)|/2 \rfloor - C_d$, and 3) $|N(i_1)| + |N(i_2)| \leq |N(i)| + C_d$.

Proof: For the proof, we will show how to pick t_{med} to ensure the two properties. Consider two nodes u and v and two intervals i and j such that $i \in I_u$ and $j \in I_v$. We know that if i dominates j then, there is an edge from u to v. Therefore, an interval can dominate or be dominated by at most C_d intervals containing any particular time stamp.

Consider now some set of intervals I^i corresponding to the neighbors of interval i. Let t_{med} be the median of their starting times. Since at most C_d of these intervals contain t_{med}, at least $\lfloor |I^i|/2 \rfloor - C_d$ intervals have start times less than t_{med} and at least $\lfloor |I^i|/2 \rfloor - C_d$ intervals have start times greater than t_{med}. The number of neighbors of the two intervals obtained by splitting i is therefore half that of i, within a constant of C_d. Furthermore, both i_1 and i_2 could be neighbors of all intervals of I^i containing t_{med}, but the rest of the intervals in I^i are split among $N(i_1)$ and $N(i_2)$. Hence $|N(i_1)| + |N(i_2)| \leq |N(i)| + C_d$. ■

We now show that the amortized cost of a sequence of `iSplit` and `iSplitAtTime` operations is $O(1)$ assuming that the operations start with a interval data structure with one (or a constant number of) intervals, and that each call to `insert_aux` takes amortized constant time. For the proof, we assume that the performance invariants hold before `iSplit`, i.e. that the number of neighbors of an interval is bounded by a constant C. We show that the operation maintains the performance invariant.

Theorem 2 (Amortized cost for `iSplit`)
The amortized space and time cost of an `iSplit` or `iSplitAtTime` operation is $O(1)$. In particular, the size of an interval data structure is linear in the number of explicit `iSplit` and

`iSplitAtTime` operations. Moreover, every interval of the data structure after the operation has at most C neighbors.

Proof: In this proof, we assign credits to each pointer at its creation.

The time it takes to find t_{med}, together with the time to reassign neighbor pointers when splitting an interval is at most some constant since by the performance invariant every interval has at most a constant number of neighbors. Suppose this time is bounded by γC, for a constant $\gamma > 1$. Then it is sufficient to add 4γ credits to each new pointer created.

We maintain the invariant that for every interval i, the sum of credits on its neighbor pointers is at least $4\gamma(|N(i)| - C/2 - C_d)$.

If an interval on C neighbors is to be split, then it has at least $4\gamma(C/2 - C_d)$ credits on its neighbor pointers. Then to find t_{med}, split the interval into two and to fix up the pointers, use γC of the credits. There are at least $\gamma C - 4\gamma C_d$ credits left. Since creating the new interval could cause the creation of at most $2C_d$ new pointers, use $8\gamma C_d$ to give each of those 4γ credits. There are at least $\gamma C - 12\gamma C_d$ credits left on the two intervals caused by the split. The number of neighbors on each of these two intervals is at most $C/2 + C_d$, so as long as $C \geq 12C_d$, the number of credits suffices to ensure the invariant. As a base case, when an interval is created, it has at most $C/2 + C_d$ creditless pointers. Whenever a new pointer is added to it, it gets an extra 4γ credits which are only used when it is split.

An `iSplitAtTime` operation needs to pay γC credits for splitting and updating pointers and $C_d \times 4\gamma$ to add credits to the new pointers it creates. The amortized runtime is hence $O(1)$. Because the credits only come from `iSplitAtTime`s, and since each such operation only spends a constant number of credits, the number of pointers in the data structure is linear in the number of called `iSplitAtTime`s. Furthermore, since if an interval gets more than C neighbors, it is split, no interval has more than C neighbors. This also implies that the number of intervals in the data structure (hence its size) is linear in the number of `iSplitAtTime`s called by the user. ■

5 Traceable Data Structures

We describe how to represent fully and partially persistent and traceable data structures with interval data structures. The representation relies on keeping different values of the vertices of the data structure in intervals and uses Split-Find-Insert data structures [9, 10, 12] to provide fast access to the history of the `read` and `write` operations. We prove that all operations take amortized constant time; we show in Section 6 that the bound is optimal.

5.1 Split-Find-Insert Data Structures

A set splitting problem is defined on a totally ordered universe U. One is given a subset $X \subseteq U$ on n elements partitioned into a collection of named sets of consecutive elements. A Split-Find-Insert data structure supports the following operations:

- **initSFI**(x, id): given an element $x \in X$ and a name id, create a data structure containing just one set $\{x\}$ with name id;
- **find**(x): given an element $x \in X$, find the set in which x is contained and return its name;
- **split**(x, id): given an element $x \in X$, and a name id, split the set S in which x is contained into two sets $S_1 = \{s \in S | s \leq x\}$ and $S_2 = \{s \in S | s > x\}$ and rename them so that S_2 has the original name of S and S_1 is named id. If S_2 is empty, then S is renamed to id.
- **insert**$(x,\ x')$: given an element $x \in X$ and a new element $x' \in U \setminus X$, such that $x' > x$ and for every $y \in X \setminus \{x\}$, $x < y \implies x' < y$, insert x' in the set containing x immediately after x.

- **nextElement**(x): returns `null` if x is the last element and returns the smallest element of U greater than x.

A Split-Find data structure only supports the operations **find** and **split**. Hopcroft and Ullman [11] described the first Split-Find data structure and gave an $O((m+n)\log^* n)$ bound for m interleaved **split** and **find** operations on an n element universe. Gabow [9] gives an $O(n + m\alpha(n,n))$ time algorithm for the same problem, thus guaranteeing amortized $\alpha(n,n)$ time per operation. If m is known then this bound can be brought down to $\alpha(m,n)$. Both these algorithms can be extended to also support the **insert** operation in amortized $O(\log^* n)$ and $O(\alpha(n,n))$ time respectively.

The Hopcroft and Ullman, and Gabow data structures work on a pointer machine. La Poutre [14] shows that Gabow's construction is essentially optimal by giving an $\Omega(n+m\alpha(m,n))$ lower bound for the Split-Find problem on general pointer machines.

Assuming a RAM model, Gabow and Tarjan [10] give a linear time $O(m+n)$ algorithm for m interleaved **split** and **find** operations on n elements, thus showing that in the RAM model one can obtain amortized $O(1)$ operations. Imai and Asano [12] further extend Gabow and Tarjan's approach to also support **insert** in amortized $O(1)$ time, in the RAM model. All data structures cited take $O(m+n)$ space.

A Split-Find (-Insert) data structure does not typically include the **nextElement** operation. That operation however is easy to support by storing the elements in an ordered list.

5.2 Traceable Data Structures via Intervals

A traceable data structure is maintained on top of an interval data structure. Notice that unlike in [7] we assume interval splitting for both partial and full traceability. Driscoll et al. use node copying instead of node splitting for partial persistence. In the case of traceability, because reads can be performed on any version, copying a node may cause both parent and child intervals to have too many pointers, and hence the node copying method is not sufficient for partial traceability.

Handles are implemented as intervals. More precisely, handle (v,t) (as in the specification Section 3) is represented as i, where i is an interval that starts at t, i.e., $\tau(t) = (t,t')$ for some t'. For simplicity, the implementation does not maintain a separate handle set. For completeness, we include detailed pseudo-code (Figure 4) in Appendix A.

For each interval i the implementation maintains a set of reads and writes, written R_i and W_i respectively. Each read in R_i is a tuple (id,i,f) where id is the identity of the reader and f is the field whose contents is read. Each write in W_i is a tuple (id,i,f,j) where id is the identity of the writer, f is the field being written to and j is another interval. To provide fast access to the history of the `read` and `write` operations, the implementation uses Split-Find-Insert data structures to store the reads from and writes of each interval. For any handle (interval) i of node v, these data structures enable finding the first set of `reads` and `writes` of v that takes place after t. A key assumption behind the approach is that each read and write take place at the start time of an interval. The implementation satisfies this assumption by creating an interval for each handle (since all reads and writes take place via a handle, this is sufficient). This is possible because of the `iSplitAtTime` operation supported by interval data structures.

For each node v, the implementation maintains two Split-Find-Insert data structures U_v (for Updates) and A_v (for Accesses). The elements of A_v and U_v are both drawn from the universe of the intervals of v ordered by their start times. Figure 2 shows an example of how U_v is maintained. For example, a write was performed on c_1 which resulted in c_1 pointing to d_1. This created a split on U_c resulting in a set containing c_0 with id c_1 and the original set which now contains c_1 with id `null`. The implementation ensures the following invariants for the access and update data structures.

Accesses: The implementation names each set $I \in A_v$ by either `null` or an interval. If I is named by `null`, then for any interval $j \in I$, there are no reads of v that take place after j. If I is named by an interval $i = (v, t, t')$, then i is read and I consists of all intervals of v that come before i but after the interval j that was last read before i. More precisely, $k \in I$ if $k = (v, t_k, \_)$, $t_k < t$, and there is no read $(\_, k', \_)$ with $k' = (v, t'_k, \_)$ and $t_k < t'_k < t$.

Updates: The implementation names each set I, $I \in U_v$ by either `null` or an interval. If I is named by `null`, then for any interval $j \in I$, there are no writes to v that take place after j. If I is named by an interval $i = (v, t, t')$, then i is written to and I consists of all intervals of v that come before i but after the interval j that was last written before i. More precisely, $k \in I$ if $k = (v, t_k, \_)$, $t_k < t$, and there is no write $(\_, k', \_, \_)$ with $k' = (v, t'_k, \_)$ and $t_k < t'_k < t$.

The implementation assumes a one-to-one correspondence between intervals and the elements of U_v and A_v. Since the data structures by Gabow [9] and by Imai and Asano [12] both store the elements explicitly, this can be achieved by maintaining pointers between each interval and the corresponding element in U_v and A_v.

The `initAux` and `insertAux` functions operate on the Split-Find-Insert data structures. The `initAux` function creates the access and the update data structures for the vertex specified by the interval. The `insertAux` operation inserts the interval j into the access and the update data structures of the specified vertex after i. The operation assumes that i and j both belong to the same vertex v.

We now explain how the operations are supported. The `new` operation creates a traceable data structure by creating an initial interval and handle for the root, initializing the access and update data structures for the root, initializing the interval data stucture, and returning the handle. The `newNode`, and the `newVersion` operations perform initialization of new nodes and new versions respectively. The `newNode` operation allocates a new node, creates an interval for it, initializes the auxiliary data structure for the node. The `newVersion` operation creates a new version by adding a new interval for the root immediately after the given handle (the operation requires that the provided handle belong to the root).

Given an interval i of a node v, a field f and a reader id, the `read` operation starts by extending the read set for i with the read (id, i, f). It then splits the set of intervals containing i in the access data structure A_v at i (the interval being read). This ensures that the intervals for which this read is the next read are assigned their own set and named by i. If i has already been read, then the split operation has no effect. Let j be the interval pointed by field f of i. To return a handle to j at time t, the operation splits j at t by calling `iSplitAtTime`; this ensures that the returned handle has its own interval.

The `write` operation takes a field f and two intervals i and j belonging to nodes v and u respectively. Then `write` has the task of making v have value u at the start time t of i. The implementation of `write` differs in the partially and fully traceable cases. The partially traceable `write` first updates the given interval i to point to j by extending the points-to function and the write set of i appropriately. It then calls `iSplit` on j to maintain the performance invariant. To ensure that the set of intervals for which the write of i becomes the next write, the operation splits the update set U_v at i. The fully traceable `write` can be viewed as a split followed by a partially traceable write. It first finds the next version t_2 after t in the version-tree pre-order. If this version is contained in $\tau(i)$, then i is split at t_2. This ensures that the original fields of the node before the next write are preserved at the versions after t in the pre-order. Notice that the invariants are still maintained after this step, unlike in [7]. If t_2 is not in $\tau(i)$, then there is an interval i' starting at t_2 which already contains the correct field values. After this, the operation proceeds as the `write` operation for partial traceability and updates the points-to function, the write set of i, and U_v. The `write` operations for both partially and fully traceable data structures require that the start time of the first interval is contained in the interval of the second. This ensures the second correctness invariant of interval data structures.

The `getNextReadHandle`(i) operation for $i = (v, t, t')$ first calls **nextElement** on A_v to find the interval j that succeeds i. It then finds the earliest interval after j that is read and returns it, if it exists, and returns `null` otherwise. Similarly, the `getNextWriteHandle`(i) operation for $i = (v, t, t')$ first calls **nextElement** on U_v to find the interval j that succeeds i. It then finds the earliest interval after j that is written and returns it, if it exists and returns `null` otherwise. The `getReadsWrites` operation simply returns all reads and writes of i.

For brevity, we do not describe the `getPreviousReadHandle` and `getPreviousWriteHandle` operations, which by symmetry are very similar to `getNextReadHandle` and `getNextWriteHandle` respectively. Also note that the approach can be extended to support `getNextReadHandle` and `getNextWriteHandle` operations for any *field* of an interval to find the next `write` or `read` of that field. One interesting application of a `getPreviousWriteToField` operation is that one can support nonpointer fields using little space. In particular, one can store a value in a non-pointer field only in the interval that it was written to. When this field is to be read later from a different interval, one can use the `getPreviousWriteToField` operation to recover the value.

Analysis. Since all operations for traceable and persistent data structures each perform a constant number of calls to `iSplit` and `iSplitAtTime` and perform constant additional work, we immediately obtain amortized $O(1)$ bounds for partially and fully persistent data structures. Furthermore, an amortized linear space bound for persistent data structures in the number of `newVersion`, `write` and `newNode` calls. For traceable data structures, the number of intervals is linear in the number of calls to `newVersion`, `write`, `newNode`, and `read`. An amortized linear space usage follows from the linear space usage of the Split-Find-Insert data structures. For the running time of the operations we establish the following correspondence.

Lemma 3
Consider a a traceable data structure with m interleaved calls to `newVersion`*,* `write`*,* `newNode`*, and* `read` *performed on it. The following holds:*

- `write`*,* `newVersion`*, and* `newNode` *take (amortized) time proportional to an* **insert***,*
- `getNextReadHandle` *and* `getNextWriteHandle` *take (amortized) time proportional to a* **find***,*
- `read` *takes (amortized) time proportional to a* **split** *followed by an* **insert**

on a Split-Find-Insert data structure with $O(m)$ elements.

Theorem 4
On a traceable data structure on a RAM, m interleaved calls to `newVersion`*,* `write`*,* `newNode`*, and* `read` *take $O(m)$ time. On a pointer machine, m calls take $O(m\,\alpha(m,m))$ time. In both cases the space usage is $O(m)$.*

Persistent Data Structures Since (fully and partially) persistent data structures require a subset of the operations of (fully and partially) traceable data structures, they can also be implemented using interval data structures. Appendix A describes such an implementation. Since persistent data structures do not require storing reads and writes their implementation is reasonably straightforward.

6 Optimality

We show that the bounds we obtain for traceable data structures are optimal for pointer machines. We give two reductions from a Split-Find problem to (1) a data structure on multiple versions supporting operations `read`, `getNextReadHandle` and `getReads`, and (2) a data structure on multiple versions supporting `write`, `getNextWriteHandle` and `getWrites`, where `getReads` and `getWrites` return the reads and writes of a handle respectively. For simplicity, we assume that the reads and writes are done on data fields, which of course can be simulated by adding extra nodes.

Consider an instance of a Split-Find data structure P on a universe U with w elements, such that r **split** and **find** operations were performed. We first describe the reduction to the `read` data structure. For every element $t \in U$, create a new version of v with start time t. Every name of a set in P is associated with a read of handle of v. A $\mathbf{split}(t, id)$ operation is modeled by a `read` of v by identity id at time t' which is the time right after t. A $\mathbf{find}(t)$ operation is modeled by a `getNextReadHandle` of v at time t, followed by an immediate `getReads` which returns the identity of the earliest reader of v after time t. This coincides with the name of the set in which t lies in P. A `read` at time t' splits the intervals of v whose first read handle was after t' into two — those whose next read handle should now be t' (and hence whose time stamps are $< t'$), and the remaining ones, whose next read handle does not change. This corresponds exactly to splitting the set of t in P since t is the last time stamp before t'.

The reduction to the `write` data structure is similar. First, for every element $t \in U$ we do a `write` on v_1 at time t. After this, a $\mathbf{split}(t, id)$ operation is modeled by a `write` to v_2 at time t' right after t by identity id. (we add an extra time stamp t_∞ greater than all elements in U). A $\mathbf{find}(t)$ operation is modeled by a `getNextWriteHandle` on v_2 at time t, followed by an immediate `getWrites` on the returned handle. This returns the identity of the earliest write to v_2 after t, and hence the name of the set in the Split-Find data structure including t.

Hence we have a correspondence between data structures on multiple versions supporting `read`, `getNextReadHandle` and `getReads`, or `write`, `getNextWriteHandle` and `getWrites` and Split-Find data structures. By La Poutre's $\Omega(n + m\alpha(m, n))$ time lower bound [14] for m interleaved **split** and **find** operations on a pointer machine, we obtain

Theorem 5
For any pointer machine there exists a constant $d > 0$ such that for any $w > 1$ and $r \geq 0$ there is an instance of a traceable data structure on one node with w versions (or intervals) with

- *a sequence of $w - 1$ `read`, r `getNextReadHandle` and r `getReadsWrites` operations, and*
- *a sequence of $2w - 1$ `write`, r `getNextWriteHandle` and r `getReadsWrites` operations,*

so that the execution of either of these by the pointer machine requires $d(w + r\alpha(r, w))$ steps.

7 Conclusion

This paper presents efficient techniques for tracing and inspecting read and write operations performed on a data structure, while also allowing these operations to be performed on current and previous versions. The work extends previous work on persistent data structure with mechanisms for tracing and inspecting operations. We show that the mechanisms can be supported in amortized constant time on the RAM model, and in amortized $O(\alpha(m, m))$ time on a pointer machine, where m is the number of operations. We show that the bound for pointer machines is tight.

References

[1] G. S. Brodal. Partially persistent data structures of bounded degree with constant update time. *Nordic J. of Computing*, 3(3):238–255, 1996.

[2] B. Chazelle. How to search history. *Inf. Control*, 64(1-3):77–99, 1985.

[3] P. F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74. Springer-Verlag, August 1989.

[4] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *SODA '91: Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 78–88, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[5] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 365–372, 1987.

[6] D. P. Dobkin and J. I. Munro. Efficient uses of the past. In *Journam of Algorithms*, volume 6, pages 455–465, 1985.

[7] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, Feb. 1989.

[8] A. Fiat and H. Kaplan. Making data structures confluently persistent. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 537–546, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[9] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 90–100, 1985.

[10] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 246–251, New York, NY, USA, 1983. ACM Press.

[11] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM J. Comput.*, 2(4):294–303, 1973.

[12] H. Imai and T. Asano. Dynamic orthogonal segment intersection search. *J. Algorithms*, 8(1):1–18, 1987.

[13] H. Kaplan and R. E. Tarjan. Purely functional, real-time deques with catenation. *J. ACM*, 46(5):577–603, 1999.

[14] J. A. LaPoutre. Lower bounds for the union-find and the split-find problem on pointer machines. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 34–44, New York, NY, USA, 1990. ACM Press.

[15] C. Okasaki. Amortization, lazy evaluation, and persistence: lists with catenation via lazy linking. In *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, page 646, Washington, DC, USA, 1995. IEEE Computer Society.

```
IDS = $(V, F, T, I, \rho)$

function initAux ($i$ as $(v,\_,\_)$) =
    $A_v \leftarrow$initSFI($i$,NONE) ; $U_v \leftarrow$initSFI($i$,NONE)

function insertAux ($i$ as $(v,\_,\_), j$) =
    $A_v$.insert($i,j$) ; $U_v$.insert($i,j$)

function new ($F$) =
    $i \leftarrow (root, t_0, t_\infty)$
    initAux($i$)
    $\rho \leftarrow \{((i,f),\texttt{null}) \mid \forall f \in F\}$
    $W_{root} \leftarrow \{\}$; $R_{root} \leftarrow \{\}$
    IDS $\leftarrow (\{root\}, F, \{t_0, t_\infty\}, \{i\}, \rho)$
    return $i$

function newNode($i$ as $(\_, t, \_)$) =
    $V \leftarrow V \cup \{v\}$, where $v \notin V$
    $j \leftarrow (v, t, t_\infty)$
    initAux($j$)
    $I \leftarrow I \cup \{j\}$
    $W_j \leftarrow \{\}$; $R_j \leftarrow \{\}$
    $\rho \leftarrow \rho \cup \{((j,f),\texttt{null}) \mid \forall f \in F\}$
    return $j$

function newVersion($i$ as $(root, t_1, \_)$) =
    $T \leftarrow T \cup \{t_2\}$, where $(t_2 \notin T) \wedge (t_1 < t_2) \wedge$
                                $(\forall t \in T. t \leq t_1 \vee t > t_2)$
    $j \leftarrow$ splitAtTime($i, t_2$)
    return $j$

function read ($i$ as $(v, t, t'), id, f$) =
    $R_i \leftarrow R_i \cup \{(id, i, f)\}$
    $A_v$.split($i,i$)
    $j \leftarrow \rho(i,f)$
    return iSplitAtTime($j,t$)
```

```
// write for partially traceable data structures
// requires that $t \in \tau(i) \cap \tau(j)$
function write ($i, id, f, j$) =
    $\rho \leftarrow (\rho \setminus \{((i,f),\_)\}) \cup \{((i,f),j)\}$
    $W_i \leftarrow W_i \cup \{(id,i,f,j)\} \setminus \{(id',i,f,j') \in W\}$
    iSplit($j$)
    $U_v$.split($i,i$)

// write for fully traceable data structures
// requires that $t \in \tau(i) \cap \tau(j)$
function write ($i$ as $(v,t,t_i), f, j$) =
    $t_2 = T$.successorTS($t$)
    if $t_2 \neq t_i$ then iSplitAtTime($i, t_2$)
    $\rho \leftarrow (\rho \setminus \{((i,f),\_)\}) \cup \{((i,f),j)\}$
    iSplit($j$)
    $W_i \leftarrow W_i \cup \{(id,i,f,j)\} \setminus \{(id',i,f,j') \in W\}$
    $U_v$.split($i,i$)

function getNextReadHandle($i$ as $(v,\_,\_)$) =
    $j \leftarrow A_v$.nextElement($i$)
    if ($j \neq$ null) then
        return $A_v$.find($j$)
    else
        return null

function getNextWriteHandle($i$ as $(v,\_,\_)$) =
    $j \leftarrow U_v$.nextElement ($i$)
    if ($j \neq$ null) then
        return $U_v$.find($j$)
    else
        return null

function getReadsWrites($i$) = return $(R_i, W_i)$
```

Figure 4: Operations on traceable data structures

```
function init_aux (i,f) = no-op
function insert_aux (i,f) = no-op

function read $((i,t),n,f)$ =
    for all $j \in$ dominated$_f(i)$
        if $t \in \tau(j)$, then return $j$

// partially persistent write
function write $((i,t),f,j)$ =
    $i_t \leftarrow$iSplit($i,t$)
    $\rho \leftarrow \rho \setminus \{((i,f),\_)\} \cup \{((i,f),j)\}$
    return iSplit($j$)
```

```
// fully persistent write
    function write $(i,t,f,j)$
    $i' =$ iSplitAtTime($i,t$)
    $(v,t,t_i) \leftarrow i'$
    $t_2 = T$.successorTS($t$)
    if $t_2 \neq t_i$ then iSplitAtTime($i', t_2$)
    $\rho \leftarrow (\rho \setminus \{((i',f),\_)\}) \cup \{((i',f),j)\}$
    return iSplit($j$)
```

Figure 5: The operations particular to persistent data structures.

A Pseudocode for implementing traceable data structures via intervals

Figure 4 gives set-notation pseudocode for our implementation of traceable data structures. Figure 5 shows the code for the operations particular to the persistent data structures. These operations when combined with the `new`, `newVersion`, and `newNode` operations from traceable data structures (Figure 4) provide all required operations. Since persistent data structures do not require tracking the `read` and `write` operations, they do not require an auxiliary data structure for storing intervals—the `init_aux` and `insert_aux` perform no work. The `read` operation relies on the observation that a pointer from an interval to another is implicitly represented by the dominators. The partially persistent write splits the interval being written to and performs the write. For full persistence, we require a mapping of the version tree into a total order; this mapping is described by Driscoll et al. [7]. The fully persistent `write` splits the interval once, and splits it again if there is not a subsequent interval—this split ensures that only the current version is affected by this write. The implementation maintains the *invariant* that for all handles (i, t), $t \in \tau(i)$. This makes reading and writing to be performed always on the correct version of the node.