PEDaLS: Persisting Versioned Data Structures

Nazmus Saquib, Chandra Krintz, Rich Wolski
Department of Computer Science
University of California, Santa Barbara
{nazmus, ckrintz, rich}@cs.ucsb.edu

***Abstract*—In this paper, we investigate how to automatically persist versioned data structures in distributed settings (e.g. cloud + edge) using append-only storage. By doing so, we facilitate resiliency by enabling program state to survive program activations and termination, and program-level data structures and their version information to be accessed programmatically by multiple clients (for replay, provenance tracking, debugging, and coordination avoidance, and more). These features are useful in distributed, failure-prone contexts such as those for heterogeneous and pervasive Internet of Things (IoT) deployments. We prototype our approach within an open-source, distributed operating system for IoT. Our results show that it is possible to achieve algorithmic complexities similar to those of in-memory versioning but in a distributed setting.**

***Index Terms*—partially persistent data structures, IoT, portability, append-only storage, distributed programming systems**

I. Introduction

The Internet of Things (IoT), Big Data, and artificial intelligence are coalescing to transform the world around us, making it possible to collect, mine, analyze, and actuate using information from physical objects across geographical locations. To do so, IoT applications amalgamate clouds, edge devices, and sensors as ensemble deployments implementing automation, decision support, and control for objects, devices, and systems in the environment. As a result, IoT deployments are geo-distributed and compose a vast diversity of devices, architectures, resource constraints, and communication systems with highly variable performance, failure, and availability characteristics.

Many popular and important cloud-based services for durable data management and data-driven computation (e.g. AWS S3 [1], HDFS [2], CORFU [3], Kafka [4], AWS Lambda [5], etc.) appear at first to provide the scale and fault-resiliency required for IoT. However, in practice, these services turn out to be ill-suited for IoT deployments because they assume "resource-rich" (e.g. machines with large memories and high clock speeds), cloud-based, and comparatively homogeneous infrastructure connected via high quality and large-capacity networks.

In this paper, we consider how to evolve and extend data services for IoT applications that target tiered edge-and-cloud IoT deployments. Our goal is to tame the heterogeneity of such deployments via new programming abstractions in the form of sophisticated program data structures that simplify and expedite software development for event-triggered and failure-prone settings. Our approach, called *PEDaLS*, combines storage persistence and data structure versioning. First, we transparently store program data structures in non-volatile storage so that they survive unexpected program termination, system failure, and power outage. Second, we use an append-only log as the underlying storage abstraction to facilitate failure recovery as well as data structure versioning and immutability. Doing so enables integration with several existing distributed logging systems [1], [6]–[10].

In this work, we focus on persisting versions of linked program data structures (e.g. linked lists and trees). To do so efficiently, we base our design on algorithmically efficient in-memory, mutable algorithm implementations from Driscoll, Sarnak, Sleator, and Tarjan [11], which embed versions (i.e. nodes and edges) within the original data structure. Note that while the above work proposes a versioning scheme for single-machine, in-memory data structures, our work formulates a versioning scheme for storage in persistent data structures which can span multiple machines in a distributed setting. We use append-only logs for storage as it provides immutability, which in turn provides robustness [12] and helps in debugging systems [13], both of which are highly desirable features in a distributed environment. The use of append-only logs as the backing store introduces new challenges, as any modification of a node in a data structure must be recorded using an append rather than in-place modification. PEDaLS addresses these challenges while maintaining the same time and space complexity of the original work and facilitating distributed storage persistence.

In addition to this new distributed formulation of persistent versioning, we describe a working "real world" implementation of PEDaLS for cloud and IoT settings. We prototype our approach using an open-source, distributed runtime system [6] that supports distributed logs as a first-class storage abstraction. We use this prototype to evaluate empirically the various overheads associated with versioning and persistence for operation workloads on linked lists and binary search trees, and investigate the effect of failures on the logging process. Our results show that PEDaLS maintains the algorithmic complexities of in-memory versioning, and enhances the robustness of distributed data structures with low overhead.

II. Related Work

We first provide background, related work, and context for the contributions we describe in the sections that follow. Specifically, we overview mutable, in-memory versioned data structures (referred to as partially persistent data structures (PDSs) in the literature), non-volatile program object storage, and append-only storage advances that enhance the robustness of distributed systems.

A. Partially Persistent Data Structures (PDSs)

PDSs track version histories for in-memory, program data structures such that versions can be accessed programmatically [11]. Data structures for which all versions can be accessed, but only the latest/newest can be modified, are called *partially persistent*. Those for which all versions can be accessed and modified are called *fully persistent*. Data structures without versioning support are called *ephemeral*. We focus on partially persistent data structures in this work because version histories are immutable and data structures are append-only – properties that are desirable in highly concurrent and failure-prone, distributed settings. We refer to partially persistent data structures simply as PDSs throughout this paper.

PDS update operations result in a new version of the structure. PDSs achieve algorithmic efficiency (in both space and time) by (i) embedding the versions within the original data structure and sharing unmodified sections of the data structure among versions, (ii) by implementing this embedding using *mutable*, pointer-based memory structures behind the scenes, and (iii) by optimizing accesses and updates to versions [11], [14]–[18]. PDSs enable a wide range of programming support including debugging [19], history programming [20], [21], undo and replay [22], [23], lock avoidance [24], referential transparency and functional programming [25]–[28]. Although most works on PDSs are related to the field of theoretical computer science, PDSs have applications in distributed systems as well. For example, sensors in an IoT environment can often malfunction and generate erroneous values. If we have a PDS deployed in the system, we can query the past versions of it to determine at what point we started receiving erroneous values and perform further analysis (e.g. what trend in data it resulted, how it affected the other components of the system that depend on this data, etc.). Systems that experience a high volume of temporal queries, such as the location of items of interest (e.g. delivery tracking, livestock tracking, etc.) throughout the day can also benefit from PDSs.

PEDaLS brings PDS support to programs in a new way, using efficient algorithms that are backed by append-only disk storage exported via logs. For IoT, these efficiencies are necessary to enable battery-powered, resource-restricted platforms (e.g. IoT devices) to use as little power as possible. Often, power consumption is proportional to computation duration and storage access frequency. Append-only semantics are also attractive in this context to mitigate component failures, particularly in the form of communication network partitions.

B. Object Storage

The term *persistence* is also used in computer science to describe the long-term, non-volatile storage of data (e.g. in files or databases on disk), i.e. enabling data to exceed the lifetime of any particular program activation. Related work has investigated (i) tools and language support that automate the process of persisting program (in-memory) data structures and objects to disk (and more recently to non-volatile memory), and (ii) ways of unifying the treatment of transient and persistent objects in programs to simplify programming [29]–[40]. These advances are referred to as persistent storage, persistent object storage, persistent object systems, orthogonal persistence, and persistent programming in the literature. PEDaLS pursues both automation and unification of persistent object storage, but is unique in that it persists *versioned data structures* (i.e. PDSs) to local or remote append-only disk storage. Doing so enables both program data structures and their versions to survive program termination and be accessed by distributed clients.

C. Append-Only Storage Systems

Append-only storage is employed in distributed and cloud computing systems to facilitate immutability, robustness, and scalability, as storage costs have plummeted [12]. It is used by cloud object stores [7], [8], event systems [41], distributed databases and file systems [2], [31], [42]–[44], log-based transaction systems [3], [45], [46], and popular messaging and streaming services [9], [10], [47].

Immutability facilitates robustness and coordination avoidance [12], [48] as well as high availability (through eventual consistency) for cloud storage, gossip protocols, collaborative editing, and revision control, among others [1], [49], [50]. In particular, versioning in distributed systems allows applications to make progress from the last available consistent state [51]. While our evaluation implementation uses the versioning features of a storage system specifically designed for IoT [6], PEDaLS can use any append-only storage system that exports ordered, version information (e.g. sequence numbers) as its backing store for program-level, versioned data structures.

III. PDS Node-Copy Method

Driscoll et. al. introduced a *node-copy method* to version linked data structures (with constant in-degree) by embedding versions efficiently within the structure itself [11]. PEDaLS extends this method using append-only logs. We first overview the original approach and then describe our advances.

Using the node-copy method, a versioned linked data structure consists of nodes and edges, and each node contains a constant value and 1+ edges (i.e. pointer fields). The method adds a fixed number of *extra pointer fields* beyond those required by the original structure. For example, in the case of a binary search tree (BST), a node contains fields for its value and left and right pointer. Additional pointer fields represent versions – i.e. updates to the left or right pointer of the node.

Once all of the extra fields have been used to accommodate update operations, the method makes a copy of the node with only the most recent pointer fields – creating a new set of extra pointer fields for use in future updates. Moreover, the predecessor of the node stores a pointer to this new copy. This way, the method avoids walking through chains of copies of

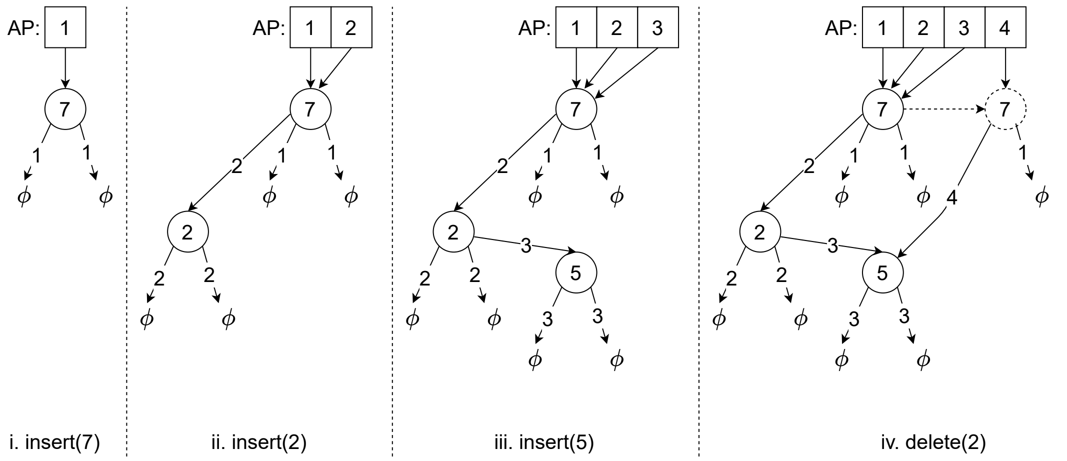


Fig. 1: Partially persistent binary search tree using the node-copy method with one extra pointer. Circles denote node with information field within. Arrows with labels denote pointers with version stamps. Dashed arrows/circles denote that a node has been copied. AP is the access pointer list. ϕ denotes the null node.

the same node to locate a particular version. Note that if the predecessor runs out of extra pointer fields while pointing to a new copy of a node, the predecessor is copied as well. In the worst case, this copying operation and chaining continue to the root node. The method also maintains a list of root nodes indexed by version stamps called the *access pointer* (AP) list. The AP facilitates constant time lookup of the root node for any version.

The number of extra pointers used by the method is a tunable parameter. If the number of extra pointers is small, the time to scan them is short but the number of copies generated may increase, resulting in a higher time and space overhead. If the number of extra pointers is large, it takes more time to scan all the pointers but fewer copies will be needed. We explore this time-space tradeoff for PEDaLS in Section V.

To illustrate how the node-copy method works for linked data structures, consider the binary search tree as shown in Figure 1. We assume that the number of extra pointers is one. We start with the empty tree and insert 7. Both the left and right pointer of the node containing 7 points to null. Assuming version stamps start at one and monotonically increase thereafter, we stamp these pointers with version 1. We also update the AP list (assuming indexing starts at 1) to point to this newly created node.

Next, we insert 2 in the same way. Because 2 is less than 7, we install a new left pointer in the BST using the extra pointer in the node containing 7 and stamp it with the current version (2). The type of the extra pointer (i.e. left or right – in this case, left) is recorded (not shown in the figure for brevity). As the root node does not change, index 2 of AP list points to the same node as AP index 1.

Next, we insert 5 which follows similar insertion steps. Finally, we delete 2. To do this, the node containing 7 must point to the node containing 5 (using a left pointer) and the null node (using a right pointer). However, the node containing 7 has run out of extra pointers and thus must be copied. The original left pointer of this new copy is set to point to the node containing 5 and is stamped with version 4. The original right pointer need not be updated and thus still points to the null node. The extra pointer of this new copy remains unused and is available for future updates. Note that as the node originally containing 7 has been copied, index 4 (i.e. the current version stamp) of the AP list points to the copied node rather than to the original node.

Access operations (i.e. find/search) for a particular version stamp vs traverse pointers with the greatest version stamp less than or equal to vs at each node. Instead of specifying only a value ($find(val)$), a PDS find operation can also include a version stamp ($find(val, vs)$). As an example, consider the operation $find(2, 3)$ – find 2 in version stamp 3, after the execution of all the operations in Figure 1. That is, the current BST is represented by the last column of Figure 1.

The access operation starts from index 3 in the AP list, which points to the node containing 7. As 2 is less than 7, we find the left pointer with the largest version stamp that is less than or equal to 3. In this case, there are two left pointers – one with a version stamp 1 and the other with a version stamp 2. As both are less than the target version stamp 3, we follow the larger one. This leads us to the node containing 2, i.e., 2 is present in version stamp 3. Note that if we search for 2 in version stamp 4 instead, we eventually end up with the null node, indicating that 2 is not present in version stamp 4.

The amortized time complexity for insert and delete using the node-copy method is constant per operation step, where an operation step is defined as the traversal from one node to another [11]. The worst-case time complexity for access using this method is also constant per operation step. Moreover, the worst-case space complexity for insert and delete using the node-copy method is constant per operation step.

IV. PEDaLS

PEDaLS is a set of language and runtime extensions that

- Transparently store immutable and versioned linked data structures in distributed, non-volatile, log-based storage;
- Expose data structure versions to developers for use in dependency tracking and program analysis [52]–[54], history-aware programming [20], [55], and repair and replay [13], [56]–[59] in distributed settings; and
- Enables portability across heterogeneous deployments by requiring only a limited "generic" functionality for generating and accessing storage-persistent logging systems (e.g. [1], [4], [7], [8], [60]) in a distributed setting.

As a result, PEDaLS data structures are able to support versioning and immutability end-to-end as distributed application and systems properties, which are desirable in highly concurrent and failure-prone settings [12], [61].

To enable this, we develop a methodology for realizing PDSs using generic, distributed log structures to facilitate integration with existing systems. A PEDaLS log must

- support append-only updates with ordered entries,
- be network addressable so that they can be co-located or remote relative to the process accessing them, and
- have some mechanism for controlling log length (e.g., size or log entry/element lifetime for automatic garbage collection).

In addition, log elements can be of any type and must be accessible via a comparable index (e.g. a sequence number).

The API functions that PEDaLS expects the storage system to support (or compose to support) are:

- `createLog(log_name)`: create a log with the name `log_name`. Upon completion, this call returns a value that indicates whether or not the log was successfully created.
- `put(log_name,elem)`: append the element `elem` to the log named `log_name`, assigning it the next available sequence number, and return the sequence number to the caller (or an error value if the operation fails).
- `get(log_name,seq_no)`: return the element at sequence number `seq_no` in the log (or an error value if the operation fails or `seq_no` does not exist).
- `getLatestSeqNo(log_name)`: return the latest sequence number of the log named `log_name` (or an error value if the operation fails or the log is empty).

Example systems that support these persistent storage functionalities directly include Kafka [4], Facebook LogDevice [60], and CSPOT [6] among others. Most cloud object stores also support versioning (e.g. Amazon S3 [7] and Google Cloud Storage [8]) and can be integrated into PEDaLS with some additional bookkeeping (e.g. combining version IDs with their timestamp to maintain order). To map PDSs to these distributed storage systems, PEDaLS must overcome multiple challenges that we describe in the subsection that follows.

Figure 2 provides a high-level overview of our approach, which models the PDS node-copy method described previously, using persistent logs. Each node and original field has a version stamp (vs) that denotes the version at which the node was created. Each node also has a constant number of extra fields (1 is used/shown in the figure), which hold

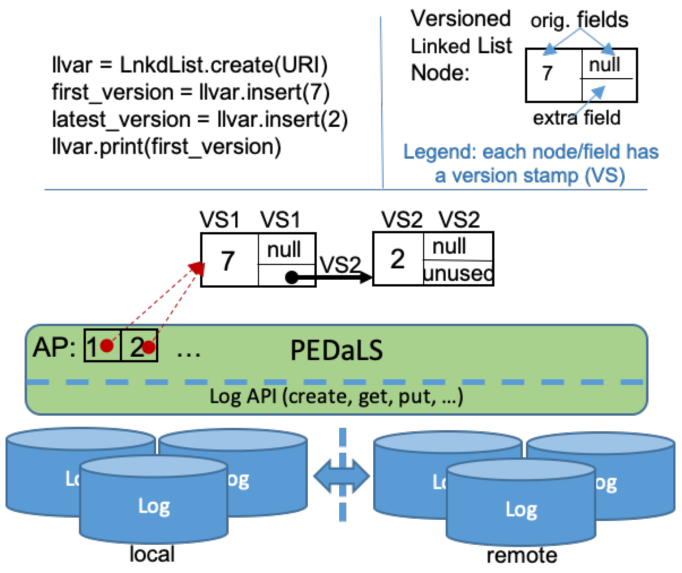

llvar = LnkdList.create(URI)
first_version = llvar.insert(7)
latest_version = llvar.insert(2)
llvar.print(first_version)
Versioned Linked List Node:
orig. fields
7
null
extra field
Legend: each node/field has a version stamp (VS)
VS1
VS1
7
null
VS2
VS2
VS2
2
null
unused
AP: 1 2 ...
PEDaLS
Log API (create, get, put, ...)
Log
Log
Log
local
Log
Log
Log
remote

Fig. 2: High-level architecture of PEDaLS that uses node-copy to version linked data structures. The top left shows user code using PEDaLS library operations. Each node in this list (e.g. top right) has an integer value and next pointer as original (developer-defined) fields. PEDaLS embeds versions/modifications within a single data structure using "extra" fields in each node. Also, each node and field has a version stamp representing their creation "time". The PEDaLS AP represents the access pointer and indexes the root node of each version for fast access. PEDaLS persists the structure and its versions using 1+ non-volatile, append-only, distributed logs.

version stamps that track the version at which original field updates occur in each node. As in the original method, we assume that information (value) fields are constant and that pointer fields (e.g. the next pointer of a linked list) can change across versions. End-users interact with the data structure using library calls as shown in the top left corner of Figure 2. Any modification to a data structure node is translated to low-level API functions of the underlying log storage, possibly affecting multiple geographically distributed logs. PEDaLS hides this translation from the end-user.

A. Challenges Using Logs to Implement Node-Copy

To map node-copy to logs, we must preserve the original time and space complexity of the original, in-memory, algorithm. Although the AP list of the node-copy method (cf. Section III) can be modeled as a separate log for efficiency, doing so imposes undue complexities. First, it is not clear how to represent both the information field and pointer fields of a node using logs. Moreover, logs are append-only – pointer manipulations must be expressed as appends (e.g. we cannot have an entry representing a pointer to a null node and later update that entry to point to a different node).

This leads to a challenge that is even more intricate – if we represent an updated node link by appending to a single log

TABLE I: Logs used by PEDaLS .

Log	Field	Description
DataLog	vs	version stamp during node creation
	val	information field of the node
	link	name of link log for the node
LinkLog	vs	version stamp during pointer creation
	dseq	DataLog seq. no. where the information field of the node being pointed to is stored
	lseq	LinkLog seq. no. of the node being pointed to where the first pointer among the contiguous pointers of the required copy is stored
	rem	number of extra pointers remaining after the insertion of the current pointer
	type	type of pointer, e.g., left/right for binary search tree
APLog	vs	version stamp of the data structure
	dseq	DataLog seq. no. where the root node's information field is stored
	lseq	LinkLog seq. no. of the root node where the first pointer of the required copy is stored

repeatedly, we potentially require a full log scan to find an arbitrary link – defeating our goal of maintaining the original time complexity of node-copy. To avoid this, we use multiple logs to represent nodes and their connections. This, however, leads to a new challenge – although an append to a single log is atomic, appends to multiple logs are not. Moreover, logs can be distributed across a network, so a network failure could potentially leave an underlying data structure semantically inconsistent.

To summarize, there are four primary challenges in implementing versioning via node-copy using logs:

- *C1*: Logs are append-only and thus we cannot perform any updates in place (as we do for in-memory structures as described above – specifically, creating links on the fly).
- *C2*: A scan of a log with an arbitrary number of entries will violate the amortized and the worst case time complexity of the operations guaranteed by the node-copy method.
- *C3*: Updates to a single log are atomic, however, PEDaLS must also guarantee that multi-log updates are also atomic if used to manage versioning.
- *C4*: Because the persistent backing store can be local to the function or on a host across a network, we must consider the impact of failures in our algorithms and analyses.

We next describe a design that allows us to efficiently implement node-copy using logs while addressing these challenges.

B. Implementing Node-Copy Method using Logs

Our log mapping design, which avoids log scans (addressing *C2*), derives from two primary observations. First, data structure updates modify node pointers and these updates can be interleaved. We thus use a separate log per node to avoid scanning entries from unrelated updates. Second, when we copy a node (when it runs out of extra pointers), the information (e.g. value) does not change. We thus use a shared log (across nodes) to hold node information. This combination allows versioned data structure updates to occur independently, while maintaining the efficiency of find/search operations, avoiding copy overhead, and conserving space.

Specifically, PEDaLS represents a node in a linked data structure by a pair of log sequence numbers: one for the shared information log – the *DataLog*, and another for the node-specific pointer log – a *LinkLog*. When a pointer is added to a node, we append an entry to the *LinkLog* of the node (addressing *C1*). The second sequence number is used to distinguish node copies.

The efficiency of the in-memory node-copy method lies in the fact that every predecessor node points to the required copy of the successor node. To traverse a copy of a node we need only scan a *fixed* number of pointers. That is, we scan $(p = o + e)$ pointers, where o is the number of original pointers and e is the number of extra pointers. Therefore, to achieve similar time complexity, we must restrict (i.e. fix) the number of entries in the LinkLog that we need to scan in order to traverse a node. This is relatively straightforward to do: because copies of a node are not interleaved (i.e. a node is copied only when the previous copy becomes full), we can use contiguous log entries of a LinkLog to represent a particular copy.

Initially, it appears that we can use contiguous p LinkLog entries to store a copy of a node and denote a copy using the first sequence number among these entries. That is, for the n-th copy of a node (considering the original node to be the "first" copy), the p entries starting from the sequence number $((n-1)*p+1)$ store that copy. This is indeed the case – in absence of network failures.

However, failures alter the situation. We consider two types of failures. (i) *Type 1*: an append to a log fails. (ii) *Type 2*: an append to a log succeeds, but the acknowledgment (which returns the sequence number where the entry was appended) is lost. In both cases PEDaLS retries. However, note that Type 2 failures violate the boundary conditions discussed above. Copies of a node do not strictly end at multiples of p anymore. This implies that we must embed the information regarding where a copy ends within an entry of the LinkLog.

To account for failures, we record the number of extra pointers left after the insertion of that entry in a dedicated field in the LinkLog. This way, once this field reads 0 while scanning entries of a copy in the LinkLog, we know we have reached the end of the current copy.

In general, we embed sufficient information in an entry of a log so that append to that log becomes idempotent (this addresses *C4*). Note that in presence of failures the number of entries we need to scan is bounded by the number of failures f $(p = o + e + f)$.

Next, the node-copy method uses an AP list for constant time access to the root node of a particular version of the data structure. Similarly, PEDaLS maintains an *APLog* to store version root nodes for the data structure. PEDaLS writes the APLog last (the order of write is DataLog>LinkLog(s)>APLog). Therefore, an append to the APLog denotes the successful completion of a version.

That is, APLog acts as a checkpoint denoting the complete versions currently present in the data structure. This design choice addresses *C3*: if there are rogue entries in LinkLogs/DataLog with version stamps vs that are greater than the latest version stamp recorded in the APLog (this can be identified after a system crash or network failure), we know that the last operation did not complete and can either trim these entries or retry the operation (we log requested operations before we start execution for the latter).

Table I summarizes the different types of logs used by PEDaLS along with a description of the fields stored in each of their entries. Note that *type* field in LinkLog is used for the sake of generalization; data structures that have only one type of entry (e.g. singly linked list) ignore this field.

Most log storage systems have some form of built-in retention policy which prevents logs from growing without bounds. For example, Kafka [4] provides retention policies based both on time (messages older than a configured time are deleted) and on space (once a log reaches a configured space limit messages are deleted from the end). CSPOT [6] provides rollover where once a log reaches a specified number of entries, newer entries start overwriting the older ones. Therefore, to ensure all the required versions are preserved in their entirety, an end-user has to specify the number of versions K he/she wants to retain. PEDaLS then allocates enough log space for each type of log based on the value of K. Currently, PEDaLS refuses update operations once it reaches K versions. This sort of policy where service is refused based on the unavailability of space is not uncommon (e.g. Redis [62]). Note that PEDaLS continues to service read operations even after it reaches K versions.

C. Node-Copy using Logs: Step by Step Example

Figure 3 shows the contents of the different logs used by PEDaLS across multiple operations on a PEDaLS binary search tree (BST). As in Figure 1, we start with the empty BST and assume the number of extra pointers is 1. For the simplicity of exposition, we name the LinkLog of a node as $link\_val$, where val is the node value (information).

To $insert(7)$, we append the entry *(1,7,link_7)* to the DataLog which returns the sequence number 1. This sequence number will be used later to record the root in the APLog. We append two entries in $link\_7$, one for the BST left pointer and one for the right pointer. Note that for both of these entries, the *rem* field is 1, denoting the number of extra pointers after the insertion. As both of these pointers point to the null node, we use an invalid sequence number (0) for *dseq* and *lseq*. The first append to LinkLog returns the sequence number 1. Therefore, we conclude the insertion of 7 by recording the tuple *(1,1,1)* in the APLog.

Operation $insert(2)$ follows a similar approach with two added steps. First, we find the current version of the data structure. To do this, we read the tail of the APLog. This reveals that the latest version is 1 (this is the first field of the last entry in the APLog in the top left corner of Figure 3), so the current working version is 2. Second, we add a pointer from the node containing 7 to the node containing 2. As the data for the node containing 2 was inserted at sequence number 2 of the DataLog and the first pointer of the node was inserted at sequence number 1 of its LinkLog, *dseq* and *lseq* values of this pointer are 2 and 1 respectively.

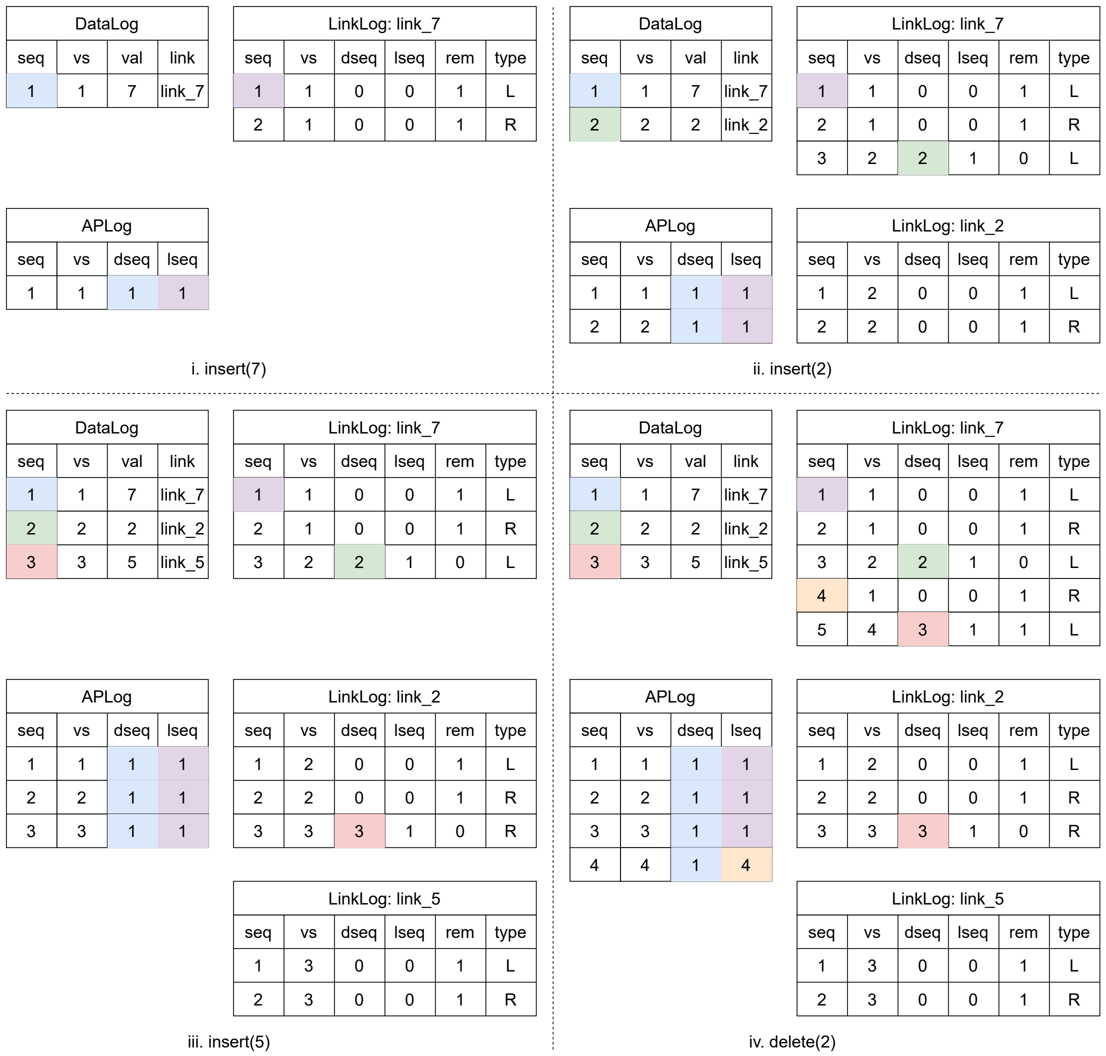
After recording this pointer, we decrement the number of extra pointers (recorded as 0 in the *rem* field). The value to be decremented comes from the tail of $link\_7$, which is 1 at this point. Note that the APLog entry for version 2 is identical to that of version 1, as the root node does not change. Execution of $insert(5)$ follows similar steps.

Execution of $delete(2)$ involves some additional steps, as no more extra pointers are left in the node containing 7 but we need to add 5 to the left of 7. Therefore, we make a copy of the node containing 7. Since the right pointer does not change, we copy over only the latest right pointer. This is done in sequence number 4 of $link\_7$ (bottom right corner of Figure 3). Next, we install the new left pointer with $dseq$ and $lseq$ values set to 3 (5 was inserted in sequence number 3 of DataLog) and 1 (node containing 2 pointed to 5 using $lseq$ value of 1), respectively. As the root node is copied in this case, we record the node in the APLog by appending $(4, 1, 4)$.

Note that for update operations, the crux of the algorithm is in node connectivity. We present the $AddNode$ routine for BST in Algorithm 1 (the routine for linked list is similar and simpler as there is only one original link and hence no link must be copied during node copy). The routine adds a child node ($cNode$) to the desired parent node ($pNode$). As successive predecessors may run out of extra pointers to accommodate this addition (cf. Section III), the full path from the root of the tree to the desired parent node is supplied to the routine. We assume the node representation in the algorithm is a structure containing $dseq$, $lseq$, and the $link$ fields (cf. Table I). The last entry in a log L is represented by $tail(L)$, the entry at sequence number i in log L is represented by $L[i]$, and a field f in an entry e of a log is $e.f$. Note that the versions of a data structure are strictly ordered and a new version is obtained by modifying the previous version. Therefore, we need to know the previous version in its entirety before we can generate the next version of a data structure. Thus, PEDaLS does not allow concurrent updates.

Access operations follow a similar pattern to that of node-copy. As an example, we consider the operation $find(2, 3)$ – find 2 in version stamp 3, after executing the above operations (i.e. BST has the representation shown in the bottom right corner of Figure 3). We start by first locating the root node for version 3 from the APLog. In this case, the root node for version 3 is recorded in the entry at sequence number 3 in the APLog. From this entry, we know that the root node's data is stored in sequence number 1 ($dseq = 1$) of the DataLog.

The entry at sequence number 1 of DataLog provides us with the name of the LinkLog. As $lseq = 1$ in the APLog entry, we start scanning $link\_7$ from sequence number 1. Scanning the top three entries is enough to fully traverse the current copy of the node (as the third entry has $rem = 0$, denoting the end of the current copy). This reveals two left



i. insert(7)

DataLog			
seq	vs	val	link
1	1	7	link_7

LinkLog: link_7					
seq	vs	dseq	lseq	rem	type
1	1	0	0	1	L
2	1	0	0	1	R

APLog			
seq	vs	dseq	lseq
1	1	1	1

ii. insert(2)

DataLog			
seq	vs	val	link
1	1	7	link_7
2	2	2	link_2

LinkLog: link_7					
seq	vs	dseq	lseq	rem	type
1	1	0	0	1	L
2	1	0	0	1	R
3	2	2	1	0	L

APLog			
seq	vs	dseq	lseq
1	1	1	1
2	2	1	1

LinkLog: link_2					
seq	vs	dseq	lseq	rem	type
1	2	0	0	1	L
2	2	0	0	1	R

iii. insert(5)

DataLog			
seq	vs	val	link
1	1	7	link_7
2	2	2	link_2
3	3	5	link_5

LinkLog: link_7					
seq	vs	dseq	lseq	rem	type
1	1	0	0	1	L
2	1	0	0	1	R
3	2	2	1	0	L

APLog			
seq	vs	dseq	lseq
1	1	1	1
2	2	1	1
3	3	1	1

LinkLog: link_2					
seq	vs	dseq	lseq	rem	type
1	2	0	0	1	L
2	2	0	0	1	R
3	3	3	1	0	R

LinkLog: link_5					
seq	vs	dseq	lseq	rem	type
1	3	0	0	1	L
2	3	0	0	1	R

iv. delete(2)

DataLog			
seq	vs	val	link
1	1	7	link_7
2	2	2	link_2
3	3	5	link_5

LinkLog: link_7					
seq	vs	dseq	lseq	rem	type
1	1	0	0	1	L
2	1	0	0	1	R
3	2	2	1	0	L
4	1	0	0	1	R
5	4	3	1	1	L

APLog			
seq	vs	dseq	lseq
1	1	1	1
2	2	1	1
3	3	1	1
4	4	1	4

LinkLog: link_2					
seq	vs	dseq	lseq	rem	type
1	2	0	0	1	L
2	2	0	0	1	R
3	3	3	1	0	R

LinkLog: link_5					
seq	vs	dseq	lseq	rem	type
1	3	0	0	1	L
2	3	0	0	1	R

Fig. 3: A versioned binary search tree (BST) using node-copy with one extra pointer implemented on logs (this mirrors the in-memory tree in Figure 1). 0 is assumed to be an invalid sequence number and hence is used to denote null nodes. DataLog sequence numbers are color-coded to represent the links. LinkLog sequence numbers are color-coded only if the entry denotes the start of a root node.

pointers, one with version stamp 1 and the other with version stamp 2. As $1 < 2$, we follow the latter one, i.e., the link at sequence number 3 of $link\_7$. This leads to the node whose information is stored in the entry at sequence number 2 of the DataLog. Reading this entry reveals the value stored here is indeed 2, completing the access operation.

V. Evaluation

In this section, we empirically evaluate the performance of PEDaLS. We implement PEDaLS over CSPOT [6], an open-source, distributed runtime system that runs on edge, cloud, and sensor systems, and uses memory-mapped files for its log abstraction. We evaluate linked lists and binary search trees (BST) as representative linked data structures since both are used by developers as building blocks for more complex structures (e.g. stacks, queues, ordered collections, etc.).

A. *Experimental Methodology*

To evaluate PEDaLS, we have devised a set of update (insert/delete) workloads for linked lists and BSTs. We execute insert for linked lists at the end of the list. We present average workload time, which includes scan/find time (for both linked list and BST). We construct 100 different workloads (combinations of insert and delete operations), each with 1000 operations.

Our workload generator uses a uniform probability distribution to select operations. For insertion, the generator randomly chooses an integer between 1 to 100 with uniform distribution. If the integer is already present, it selects the next integer not already present in the data structure. For deletion, the generator randomly chooses an integer already present in the data structure with uniform distribution. This way the generator guarantees all operations will execute to

Algorithm 1 Node Copy: AddNode (BST)

Require: childNode $cNode$; stack of nodes leading from root of BST to parent of $cNode$, S; number of links per node $linksPerNode$; working version stamp vs
Ensure: $cNode$ is added to its parent

```
1:  while $S \neq \phi$ do
2:      $pNode \leftarrow S.pop()$
3:      $lastLink \leftarrow tail(pNode.link)$
4:      $newLink \leftarrow \{\}$
5:      $newLink.vs \leftarrow vs$
6:      $newLink.dseq \leftarrow cNode.dseq$
7:      $newLink.lseq \leftarrow cNode.lseq$
8:      $childType \leftarrow getType(pNode.dseq, cNode.dseq)$   ▷ getType returns type of link i.e. left or right
9:      if $lastLink.rem > 0$ then   ▷ node not full
10:         $newLink.rem \leftarrow lastLink.rem - 1$
11:         $pNode.link.append(newLink)$
12:         break
13:     else
14:         $leftLink \leftarrow \{\}$
15:         $rightLink \leftarrow \{\}$
16:         $i = pNode.leq$
17:         $iteratorLink \leftarrow pNode.link[i]$
18:         while $iteratorLink \neq \phi$ & $iteratorLink.rem \geq 0$ do
19:             if $iteratorLink.type = L$ then   ▷ iterator is a left pointer
20:                 $leftLink \leftarrow iteratorLink$
21:             else
22:                 $rightLink \leftarrow iteratorLink$
23:             end if
24:             $i \leftarrow i + 1$
25:             $iteratorLink \leftarrow pNode.link[i]$
26:         end while
27:         if $childType = L$ then   ▷ copy right child
28:             $linkLogSeq \leftarrow pNode.link.append(rightLink)$
29:         else
30:             $linkLogSeq \leftarrow pNode.link.append(leftLink)$
31:         end if
32:         $newLink.rem \leftarrow linksPerNode - 2$
33:         $pNode.link.append(newLink)$
34:     end if
35:     $cNode.dseq \leftarrow pNode.dseq$
36:     $cNode.lseq \leftarrow linkLogSeq$
37:     $cNode.link \leftarrow pNode.link$
38: end while
```

completion. [1] Unless otherwise specified, our results present the average across 100 workloads.

In addition to microbenchmarks, we evaluate the performance of PEDaLS for an end-to-end distributed application. The application (presented in Section V-E) implements a simple clone of Amazon Simple Storage Service (S3) for storing and serving images using PEDaLS.

To the best of our knowledge, no other system provides general-purpose versioning and storage persistence of program data structures. Moreover, we want to explore the cost of providing versioning and storage support to systems relying on in-memory ephemeral data structures. Thus, we compare PEDaLS data structures against in-memory ephemeral and in-memory persistent data structures (denoted as simply *ephemeral* and *persistent* in the results). We use memory-mapped files as a backing store for the in-memory data structures.

To evaluate the trade-off in space and time using extra pointers, we consider 1, 5, and 10 extra pointers for PDSs (both in-memory and PEDaLS). We refer to the PEDaLS implementation using n number of extra pointers as PEDaLS-n. Similarly, we refer to the in-memory persistent implementation using n number of extra pointers as persistent-n.

We perform our experiments using virtual machine instances in a private cloud running Eucalyptus [63]. Each instance has two 2GHz CPUs and 2GB of memory. Unless otherwise specified, we co-locate the logs and workload for this study.

B. Space Analysis

We first evaluate PEDaLS space usage. Figures 4a and 4b show the average space in bytes used by linked list and BST respectively to execute 1 to 1000 operations. The results show that PEDaLS space requirements are linear with respect to the number of operations for versioned data structures (persistent-n and PEDaLS-n). The space requirements for ephemeral data structures are linear in the number of nodes present at any instant of time (due to scaling it appears to be constant in Figure 4).

We expect that when the number of extra pointers is small, the node-copy method will copy more nodes (and consume more space). This is evident in the results. The average slope of the lines for PEDaLS-1 BST, PEDaLS-5 BST, and PEDaLS-10 BST are respectively 352, 300, and 296 (Figure 4b). This implies that, on average, each update operation in PEDaLS-1 BST requires 352 bytes, whereas each update operation in PEDaLS-10 BST requires 296 bytes.

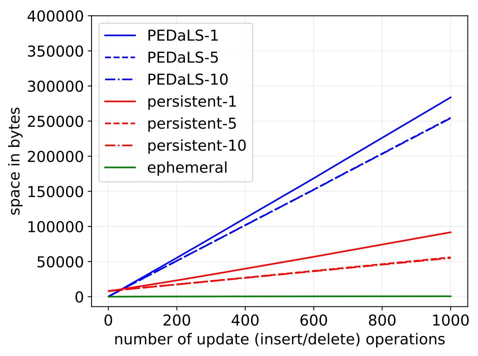
Note that although the difference in average slope between PEDaLS-10 BST and PEDaLS-1 BST is more than 50, this difference reduces to 4 when considering PEDaLS-5 BST and PEDaLS-10 BST. That is, we only reduce space consumption via extra pointers up to a point. For linked list PEDaLS-10 saves roughly one byte of space per operation as compared to PEDaLS-5. When compared to persistent-n BSTs, PEDaLS-n BSTs require $1.50x$, $1.75x$, and $1.80x$ more space for $n = 1, 5$, and 10 respectively. That is, the space overhead to map the in-memory node-copy method to logs is quite low.

Similar observations for linked list from Figure 4a reveal space overhead is as low as $2.00x$. Ephemeral data structures can free the corresponding memory once a node is deleted. Moreover, they do not have to perform bookkeeping related to maintaining versioning information. Therefore, we expect their space requirement to be lower. Unsurprisingly, the space overhead to maintain PEDaLS-10 BST as opposed to ephemeral BST is $178x$. The same overhead is $202x$ for linked list.

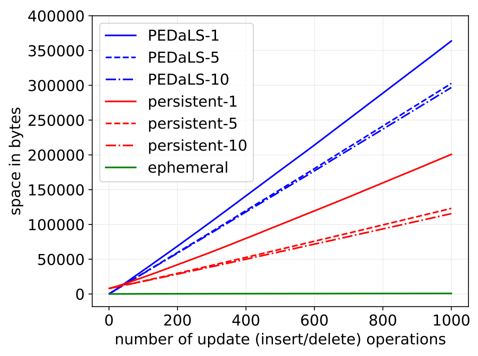
C. Time Analysis

We next consider the additional time needed for versioning and disk persistence. Figure 5 shows the average time taken by

[1] Our workloads are available (for reproducibility purposes) as part of our open-source release of PEDaLS at https://github.com/MAYHEM-Lab/PEDaLS.

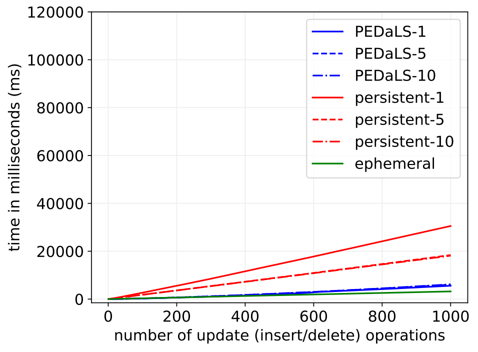


(a) Linked list.

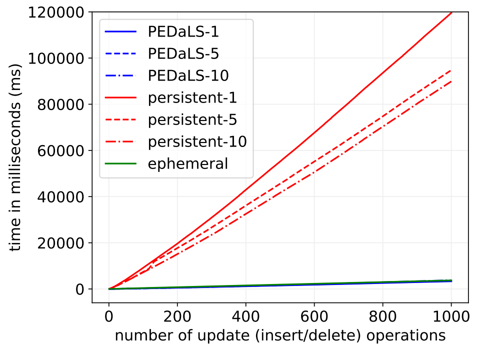


(b) BST.

Fig. 4: Average space usage.



(a) Linked list.



(b) BST.

Fig. 5: Average execution time.

the different data structures to execute a number of operations ranging from 1 to 1000. The time requirements (shown on the y-axis) are linear with respect to the number of operations (shown on the x-axis).

For ephemeral linked list, the average time taken to execute an update operation is 3.30 milliseconds. This value is 4.11 milliseconds, 4.47 milliseconds, and 4.51 milliseconds for PEDaLS-1 linked list, PEDaLS-5 linked list, and PEDaLS-10 linked list respectively. That is, even the slowest PEDaLS-n linked list implementation introduces only $1.35x$ overhead. The persistent-n implementations are the slowest, requiring 28.85 milliseconds, 18.28 milliseconds, and 18.18 milliseconds for $n = 1, 5$, and 10 respectively.

These results are surprising. We expect the performance order would be ephemeral>persistent-n>PEDaLS-n. However, in the case of linked list the experiments show PEDaLS-n has a better performance than persistent-n. Moreover, the small overhead in PEDaLS-n when compared to ephemeral, shows that it is possible to use a log-based approach to implement storage-persistent PDS linked list *without a significant performance penalty*, relative to the standard, mutable, and unversioned pointer-based implementations.

These experiments indicate that the performance of the memory allocator plays a key role in the performance of versioned persistence. The underlying memory allocator that CSPOT (runtime system over which PEDaLS is currently implemented) uses is trivial: it simply appends to a fixed-size, pre-allocated, circular log buffer that is mapped to a Linux file. Further, there is no deallocation – the log "wraps" to automatically garbage collect log entries [6]. For the ephemeral and persistent implementations, the memory allocator uses a first-fit dynamic allocation algorithm with eager coalescing of adjacent free blocks on deallocation. Thus, the implementations that use a dynamic allocator cause it to "chase" an internal free list of blocks from time to time during allocation and coalescing.

Additionally, both allocators flush a modified memory region to the backing store (i.e. a Linux file) to prevent corruption in case of a system crash. The dynamic allocator versions (ephemeral and persistent) use the Linux `msync()` system call to write back modified mapped memory. For CentOS 7, this call causes either one or two (due to alignment) 8 kilobyte pages to be flushed to the backing file. Moreover, it must flush the modified memory to the backing store (i.e. a Linux file) each time the data changes in an allocated buffer or in the internal memory allocated data structures.

A cursory examination of the CSPOT source code [2] shows that it unmaps each storage log after each append operation, causing dirty pages to be flushed back. Thus it is likely that ephemeral performance is dominated by backing-store synchronization traffic. As a result, the additional computational overhead associated with versioning using the node-copy method is negligible.

Figure 5b shows the average time taken by the different implementations of BST to perform the workloads. The time requirements are again linear with respect to the number of operations. Ephemeral BST requires 3.82 milliseconds to execute an operation. PEDaLS-n BSTs require slightly lower – 2.88 milliseconds, 3.04 milliseconds, and 3.24 milliseconds for $n = 1, 5$, and 10 respectively.

The additional complexities inherent in a PDS implementation of BST put additional performance pressure on the dynamic memory allocator. This is reflected in the per-operation times for persistent-n, which are 106.71 milliseconds, 88.48 milliseconds, and 81.26 milliseconds for $n = 1, 5$, and 10 respectively. That is, persistent-n can have an overhead of as much as $28x$ when compared to ephemeral.

D. Search Performance

Note that update operations (insert/delete) require traversals of existing nodes (e.g. to find leaf node to which a new node is inserted). We next break out the time to perform access (find/search) operations alone, i.e. data structure traversal. We consider different access operations for the two data structures: (i) finding the last node in linked list (Figure 6a) and (ii) finding the maximum value in BST (Figure 6b). Figure 6 shows traversal time as a function of the number of nodes.

Note that although many workloads resulted in having > 10 nodes in linked list, We do not show the complete results for the sake of visual comparability (e.g. no workload resulted in a depth of > 10 BST nodes). Because we search from the latest version, we simply follow the last link (i.e. at most 2 links for BST) of each node. We find that varying the number of extra pointers for this experiment has no significant performance difference.

Figure 6 shows that the access time is linear in the number of nodes for PEDaLS. On average, PEDaLS requires 0.23 milliseconds and PEDaLS BST requires 0.29 milliseconds. Both the persistent and ephemeral data structures are three orders of magnitude faster than PEDaLS for access, with no significant difference between each other. This again emphasizes the importance of the performance of the memory allocator. For updates, PEDaLS is faster than persistent for both linked list and BST and is on par with ephemeral.

E. End-to-End Application: Image Server

Finally, we evaluate the use of PEDaLS for an end-to-end distributed application commonly found in IoT settings, e.g. [64]. The program implements an image server, which sensors and/or users can use to upload and download images for analysis.

[2]https://github.com/MAYHEM-Lab/cspot

We compare two different implementations of this image server. For the first implementation, we use an Amazon S3 bucket located in the us-west-2 region as the server. The client process is located in a private cloud in UCSB and interacts with the bucket using the boto3 library (the instance has the same specifications as the ones used so far – 2GHz CPU, 2GB RAM). For the second implementation, we use a t3.small (2GHz CPU, 2GB RAM) EC2 instance, also located in the us-west-2 region. We employ a PEDaLS -1 BST in this instance that acts as an image indexer. The average RTT between the client instance and the EC2 instance that we observe is approximately 30 milliseconds. Note that our goal is not to outperform the S3 image server, rather explore the utility of PEDaLS in an IoT setting.

For this experiment, we first upload 500 images (256 KB each) to the server, followed by the retrieval of the images from the server. Surprisingly, PEDaLS based server outperforms s3 based server in both upload time and download time. Figure 7 shows the average upload and download times per image for the two servers. The average upload time is 153 milliseconds for the S3 based server, whereas this value is 144 milliseconds for PEDaLS. That is, the latter is $1.1x$ faster. The average download time is 146 milliseconds for S3 based server, whereas this value is 83 milliseconds for PEDaLS based server. In this case, PEDaLS is $1.8x$ faster.

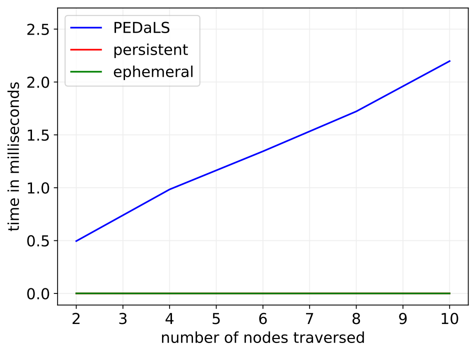
VI. Conclusion

Both partially persistent data structures (PDSs) and append-only storage systems provide immutability and history-based programming – albeit at different "levels" (program versus systems). These features are useful at both levels in distributed, large-scale, and failure-prone contexts such as those for heterogeneous and pervasive Internet of Things (IoT) deployments. In this paper, we investigate how to combine the two so that high-level, linked program data structure operations with versioning support, automatically and transparently map to append-only persistent storage – enabling, for the first time, survivability and programmatic access by distributed clients to both the data structures and their version histories.

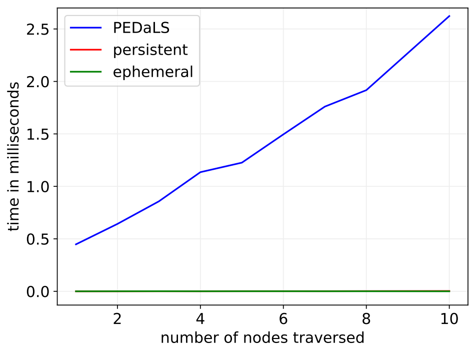
To enable this, we present a new approach for efficiently supporting versioned, linked data structures in programs by leveraging algorithmic advances from partially persistent data structures. We use these methods to design a mapping and library implementation of version-aware data structure operations that are backed by append-only storage. We implement this approach using an append-only storage abstraction from a portable, open-source event system for IoT. We use this system to evaluate the algorithmic complexities and performance overhead for operation workloads for linked list and binary search tree (BST) structures as well as end-to-end using a multi-tier image processing application. Our results show that we are able to achieve the algorithmic complexities of the original PDSs and low overhead for storage-persistent versioning.

References

[1] "Amazon S3," 2021, https://aws.amazon.com/s3/ [Online; accessed 11-Apr-2021].



(a) Linked list.



(b) BST.

Fig. 6: Access (node traversal) time vs number of nodes traversed.

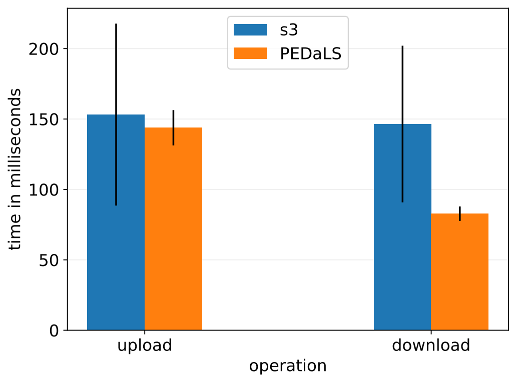


Fig. 7: Average upload and download time per image for the Amazon S3 and PEDaLS image servers.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[3] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. Davis, "Corfu: A shared log design for flash clusters," in *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[4] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[5] "AWS Lambda," https://aws.amazon.com/lambda/, 2021, [Online; accessed 11-Apr-2021].

[6] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT," in *ACM Symposium on Edge Computing*, 2019.

[7] Amazon, "S3 Object Versioning," 2019, https://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html [Online; accessed 28-Sep-2019].

[8] "Google Cloud: Versioned Object Storage," 2018, https://cloud.google.com/storage/docs/object-versioning [Online; accessed 12-Sep-2018].

[9] "Apache Kafka," 2019, http://kafka.apache.org [Online; accessed Sep. 2019].

[10] "Amazon kinesis streams service," 2020, https://docs.aws.amazon.com/kinesis/index.html Accessed 15-Apr-2020.

[11] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, "Making data structures persistent," *J. Comput. Syst. Sci.*, vol. 38, no. 1, 1989.

[12] P. Helland, "Immutability changes everything," in *Conference on Innovative Data Systems Research*, 2015, http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf Accessed 15-Sep-2019.

[13] P. Alvaro, S. Galwani, and P. Bailis, "Research for practice: Tracing and debugging distributed systems; programming by examples," in *CACM*, Jan. 2017.

[14] P. F. Dietz and R. Raman, "Persistence, amortization and randomization," in *ACM-SIAM Symposium on Discrete Algorithms*, 1991.

[15] G. S. Brodal, "Partially persistent data structures of bounded degree with constant update time," *Nord. J. Comput.*, vol. 3, no. 3, 1996.

[16] A. Fiat and H. Kaplan, "Making data structures confluently persistent," in *Symposium on Discrete Algorithms*, 2001.

[17] H. Kaplan, "Persistent Data Structures," 2004.

[18] F. Pluquet, S. Langerman, A. Marot, and R. Wuyts, "Implementing partial persistence in object-oriented languages," in *Meeting on Algorithm Engineering & Expermiments*, 2008.

[19] L. Ceze, C. von Praun, C. Cascaval, P. Montesinos, and J. Torrellas, "Programming and Debugging Shared Memory Programs with the Data Coloring," in *Workshop on Compilers for Parallel Computing*, 2009.

[20] E. D. Demaine, J. Iacono, and S. Langerman, "Retroactive data structures," *ACM Trans. Algorithms*, vol. 3, no. 2, May 2007.

[21] ——, "Retroactive data structures," in *ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[22] H. Mannila and E. Ukkonen, "The set union problem with backtracking," *International Colloquium on Automata, Languages and Programming*, vol. 226, 1986.

[23] J. Westbrook and R. E. Tarjan, "Amortized analysis of algorithms for set union with backtracking," *SIAM J. Comput.*, vol. 18, 1989.

[24] Y. Zhan and D. E. Porter, "Versioned programming: A simple technique for implementing efficient, lock-free, and composable data structures," in *ACM International on Systems and Storage Conference*, 2016.

[25] "Haskell," 2019, "https://www.haskell.org" Accessed 17-Sep-2019.

[26] "Immutable.js," 2019, "https://immutable-js.github.io/immutable-js/" Accessed 20-Sep-2019.

[27] John McClean, "Java Persistent Collections," 2019, "https://medium.com/@johnmcclean/the-rise-and-rise-of-java-functional-data-structures-63782436f93b" Accessed 20-Sep-2019.

[28] C. Okasaki, "Purely Functional Data Structures," Carnegie Mellon University, Tech. Rep. CMU-CS-96-177, 2019, https://www.cs.cmu.edu/ rwh/theses/okasaki.pdf Accessed 20-Sep-2019.

[29] A. Chien, P. Balaji, P. Beckman, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, R. Schreiber, J. Hammond, J. Dinan, I. Laguna, D. Richards, A. Dubey, B. van Straalen, M. Hoemmen, M. Heroux, K. Teranishi, and A. Siegel, "Versioned distributed arrays for resilience in scientific applications," *Procedia Comput. Sci.*, vol. 51, no. C, Sep. 2015.

[30] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *USENIX Conference on File and Stroage Technologies*, 2011.

[31] A. Twigg, A. Byde, G. Milos, T. Moreton, J. Wilkes, and T. Wilkie, "Stratified b-trees and versioned dictionaries," in *USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'11, 2011.

[32] 1995.

[33] Oracle, "Java Persistence API," 2019, "https://docs.oracle.com/cd/E19798-01/821-1841/6nmq2cpag/index.html" Accessed 18-Sep-2019.

[34] Oracle, "JDBC," 2021, https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdbc/toc.htm Accessed 2-Apr-2021.

[35] R. Agarwal, "The c++ interface in objectivity," *SIGPLAN Not.*, vol. 29, no. 12, Dec. 1994.

[36] T. Kelly, "Persistent Memory Programming on Conventional Hardware," *ACMQUEUE*, vol. 17, no. 4, 2019.

[37] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *ACM Conference on Programming Language Design and Implementation*, ser. PLDI 2018, 2018.

[38] M. Atkinson, P. Bailey, K. Chisholm, W. Cockshott, and R. Morrison, "An Approach to Persistent Programming," *Computer Journal*, vol. 26, no. 4, 1983.

[39] M. Atkinson, L. Daynes, M. Jordan, T. Printezis, and S. Spence, "An orthogonally persistent Java," in *SIGMOD*, 1996.

[40] S. Balzer, "Contracted Persistent Object Programming," in *PhD Workshop, ECOOP*, 2005.

[41] B. Stopford, *Designing Event Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka*. O'Reilly Media, 2018, https://drive.google.com/file/d/1NGst29pUjZwtn8pXTKvlSSuau2-to5dD/view Accessed 15-Sep-2019.

[42] R. Kotla, L. Alvisi, and M. Dahlin, "Safestore: A durable and practical storage system," in *USENIX Annual Technical Conference*, 2007, pp. 129–142.

[43] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li, "Storage management in asterixdb," *VLDB*, vol. 7, no. 10, Jun. 2014.

[44] C. Gong, S. He, Y. Gong, and Y. Lei, "On integration of appends and merges in log-structured merge trees," in *International Conference on Parallel Processing*, 2019.

[45] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Chariots: A scalable shared log for data management in multi-datacenter cloud environments." in *EDBT*, 2015, pp. 13–24.

[46] H. Vo, S. Wang, D. Agrawal, G. Chen, and B. Ooi, "Logbase: a scalable log-structured database system in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1004–1015, 2012.

[47] "Apache Samza," 2019, http://samza.apache.org [Online; accessed Sep. 2019].

[48] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *ACM Queue*, vol. 11, no. 3, Mar. 2013.

[49] S. Burckhardt, "Principles of eventual consistency," *Foundations and Trends in Programming Languages*, vol. 1, no. 1-2, 2014.

[50] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed Data Structures over a Shared Log," in *Symposium on Operating System Principles*, Nov. 2013.

[51] P. Helland, "Data on the outside versus data on the inside," in *Conference on Innovative Data Systems Research*, 2015, http://cidrdb.org/cidr2005/papers/P12.pdf Accessed 15-Sep-2019.

[52] W. Lin, C. Krintz, R. Wolski, M. Zhang, X. Cai, T. Li, W. Xu, and R. Zhou, "Tracking Causal Order in AWS Lambda Applications," in *IEEE International Conference on Cloud Engineering*, Jun. 2018.

[53] W.-T. Lin, C. Krintz, and R. Wolski, "Tracing Function Dependencies Across Clouds," in *IEEE Cloud*, 2018.

[54] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, Dec. 2018.

[55] D. Meissner, B. Erb, F. Kargl, and M. Tichy, "Retro-lambda: An event-sourced platform for serverless applications with retroactive computing support," in *Intl. Conf. on Distributed and Event-based Systems*, 2018.

[56] W.-T. Lin, F. Bakir, C. Krintz, R. Wolski, and M. Mock, "Data repair for Distributed, Event-based IoT Applications," in *ACM International Conference on Distributed and Event-Based Systems*, 2019.

[57] I. Beschastnikh, P. Wang, Y. Brun, and M. Ernst, "Debugging distributed systems," in *CACM*, Jun. 2016.

[58] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining temporal invariants from partially ordered logs," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 3, Jan. 2012.

[59] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay," in *NSDI*, 2007.

[60] Facebook, "LogDevice," 2020, https://engineering.fb.com/core-data/logdevice-a-distributed-data-store-for-logs/ Accessed 29-Feb-2020.

[61] D. Bailis, "Coordination avoidance in distributed databases," 2015, ph.D. Dissertation, University of California, Berkeley, http://www.bailis.org/papers/bailis-thesis.pdf Accessed 15-Sep-2019.

[62] "Redis," "http://redis.io".

[63] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*. IEEE, 2009, pp. 124–131.

[64] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, "Where's The Bear? – Automating Wildlife Image Processing Using IoT and Edge Cloud Systems," Computer Science Department at the University of California, Santa Barbara, Tech. Rep. UCSB-CS-2016-07, October 2016.