Replicated Versioned Data Structures for Wide-Area Distributed Systems

Nazmus Saquib, Chandra Krintz, and Rich Wolski

Abstract—In this work, we investigate the integration of replicated versioned data structures and append-only distributed storage systems. Doing so facilitates high availability and scalability while providing developer access to different versions of program data structures across program executions. Modern distributed systems such as the Internet of Things (IoT) often employ multi-tiered (cloud/edge/sensors) architectures consisting of a wide array of heterogeneous devices generating data frequently. Hence system availability is imperative to avoid data loss, while scalability is required for the efficient operation of the system not only within the same tier but across different tiers as well. Our proposed approach replicates, persists, and versions *program data structures* such as binary search trees and linked lists for use in distributed IoT applications. The versioning and persistence of these structures aid failure recovery and facilitate system debugging from its inception instead of making such considerations an afterthought. Moreover, our experiments suggest versioned data structures can perform better in applications performing high volumes of temporal queries versus traditional methods of persisting data (e.g., in a database). We empirically evaluate the overheads associated with versioning and storage persistence of program data structures, present experimental results for multiple end-to-end applications, and demonstrate the scalability of this approach.

Index Terms—Append-only logs, IoT, replication, versioning

1 Introduction

Distributed systems have evolved over the years resulting in new computing paradigms such as cloud computing, edge computing, and the Internet of Things (IoT) [1]. However, programming applications that leverage the services provided by multi-tier deployments (i.e., cloud, edge, and sensor combinations) are complex and error-prone for several reasons. First, there is no unifying set of programming abstractions that are designed to span resource scales consisting of heterogeneous devices – from resource-restricted IoT end devices (e.g., microcontrollers) to the extensive resource pools available from public clouds. Second, many multi-tier applications must tolerate frequent communication disruptions, outages, and failures when operating across tiers. Clouds mask and handle failures via vast scale and infrastructure support, which are not available or feasible at the edge due to resource constraints, intermittent connectivity to the cloud, and vast heterogeneity of IoT devices and networks. Developers of multi-tier applications face significant failure management programming challenges concerning both the number of devices to be managed and their capabilities.

• *The authors are with the University of California, Santa Barbara, CA 93106 USA. E-mail: {nazmus, ckrintz, rich}@cs.ucsb.edu.*

Manuscript received 24 November 2021; revised 25 October 2022; accepted 26 October 2022. Date of publication 28 October 2022; date of current version 16 November 2022.
This work was supported by NSF under Grants CNS-2107101, CNS-1703560, and ACI-1541215.
(Corresponding author: Nazmus Saquib.)
Recommended for acceptance by R. Prodan.
This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2022.3217969, provided by the authors.
Digital Object Identifier no. 10.1109/TPDS.2022.3217969

Hence for the modern distributed system to revolutionize the next era of computing, it must involve high-level programming abstractions that reconcile the heterogeneity of devices while exploiting the failure management approaches that have emerged from large-scale cloud computing. Our work addresses these dual challenges with new programming support (i.e., high-level libraries) for commonly used program data structures (e.g., linked lists and binary search trees) that transparently facilitates versioning, replication, and eventual consistency. By doing so, we attempt to solve the research challenge of realizing systems that can reconcile the heterogeneity of modern distributed environments while making applications capable of withstanding failure. In Section 3, we present multiple IoT applications from existing literature (urban traffic steering [2], inter-device task dispatch [3], smart locks [4], and machine learning at the edge [5]) that can benefit from our approach.

The combination of versioning, replication, and storage persistence using logs provides us with a unique approach to application development for failure-prone, heterogeneous distributed environments. As versioned data structure preserves its previous states, versioning allows us to organically log every state an application goes through. This removes the burden of explicit logging from the application developer and facilitates debugging as previous versions can be accessed to inquire what caused a failure. Unsurprisingly, versioned data structures can be efficient for applications with large volumes of temporal queries as well.

Replication makes program data structures durable, highly available, and concurrently accessible in the presence of resource failures. Strong consistency involves coordination overhead which can get compounded in failure-prone environments due to frequent network failures. Hence we opt for eventual consistency in this work. As it stands,

TABLE 1
A Comparison Among Different Technologies/Platforms With PEDaLS

Technology/ Platform	Immutable/ Versioned	Distributed/ Replicated	Type of Consistency	Supported Data Structures	Complexity Guarantees	Primary Usage
PEDaLS [8]	Yes	Yes	Strong eventual	Linked data structures with constant in-degree	Constant/op step	Versioned and replicated program data
PDS [10]	Yes	No	N/A	Linked data structures with constant in-degree	Constant/op step	Versioned program data
CRDT [11]	No	Yes	Strong eventual	Data structures having join semilattice or having operation commutativity with idempotence support from communication layer	No	Replicated data
Append-only logs [12]	Yes	Yes	Implementation dependent	Raw data	N/A	System logging, data storage, etc.
Paxos [13]/ Raft [14]	N/A	Yes	Strong	N/A	N/A	Replication protocol
Git [15]/ GitHub [16]	Yes	Yes	Eventual	Text, raw data	N/A	Source code management
Blockchain [17]	Yes	Yes	Eventual	Raw data	N/A	Immutable ledger

Only the immutability of blockchain is supported by cryptography, other technology's immutability/versioning is supported by their exposed interfaces. PEDaLS and PDS provide efficient space/time complexity for linked data structures with constant in-degree in a single machine. Similar mechanisms could potentially be used for other data structures without the same space/time complexity guarantees. PEDaLS does not have the join semilattice or operation commutativity requirements as CRDTs. Note that to the best of our knowledge, PEDaLS is the first and only system to provide replication for storage persistent versioned program data structures.

many multi-tier applications can sacrifice strong consistency in favor of high availability and partition tolerance [4], [6], [7].

Append-only logs provide immutability, which facilitates both availability and coordination avoidance. Both versioned data structures and append-only logs provide versioning, the former from a program level and the latter from a storage level. Our work uses logs to provide a generic method of operation reversal as well. If we can express an operation in terms of appends to logs, we can express the reversal of the operation as log rollback. This plays a vital role in conflict resolution during replication (cf. Section 5.6).

Our work builds upon and extends an approach to make versioned data structures log storage persistent called PEDaLS [8]. In particular, we extend PEDaLS in the following ways: We introduce a replication method for data structures that ensures that any two replicas that observe the same set of operations (possibly in a different order) will arrive at the same state. This property is known as strong eventual consistency [9]. Due to the append-only semantics of logs, PEDaLS requires novel and complex methods to efficiently detect and resolve any temporary conflicts that may arise during the replication process. We present an efficient conflict detection method that avoids full log scans in Section 5.5 and the subsequent conflict resolution method in Section 5.6. Notably, the append-only semantics of logs facilitate the rollback of arbitrary data structures that makes conflict resolution feasible.

We demonstrate the scalability of our proposed replication method in Section 6.2, in which we consider different numbers of replicas and workload composition (i.e., read/write percentage). We also demonstrate how a wide range of use cases can leverage our versioning support using multiple real-life applications in Section 6.4. Our results show that PEDaLS performs significantly better in regard to complex temporal queries than popular SQL and NoSQL databases.

Finally, although conceptually, logs are unbounded, in practice they are limited by the physical capacity of the underlying storage. To address this challenge, we develop a probabilistic model in Section 6.5 that determines the storage required to retain a user-specified number of versions in the face of failures. In the sections that follow, we overview related work and then describe each of these contributions in detail.

2 Related Work

In this section, we discuss the foundations that underlie replicated versioned data structures. We focus this section on advances in data versioning, distributed logging, and data replication, which are key for supporting failure-prone, multi-tier applications. We summarize a comparative study of these foundational concepts in Table 1.

2.1 Data Versioning

Versioned data structures maintain past program states that can be accessed programmatically. Such data structures are *immutable* – a new update operation creates a new version that retains the previous states while recording the latest state. Note that for immutable data structures, even deletion operations logically *append* new information. Immutability facilitates coordination avoidance as well as other robustness features described below [18]. Versioned data structures are also referred to as persistent data structures [10] in the literature. Note that the term "persistent" in this context

does not mean "storage persistence" (i.e., in a system context, the ability to persist data during a power-off state), rather it means that the previous versions (i.e., states) of the data structures are preserved. If all versions (past and present) of a data structure can be both accessed and modified it is *fully persistent*. If all versions of a data structure can be accessed but only the latest one can be modified it is *partially persistent*. A *confluently persistent* [19] data structure can merge different versions into one. We focus on partially persistent data structures in this paper, as the version history is sufficient for our use cases. We refer to partially persistent data structures in this paper as PDSs.

In our work, we use the node-copy method [10] proposed by Driscoll, Sleator, Sarnak, and Tarjan for distributed, linked data structures (e.g., trees and lists). Node-copy is a single-machine, in-memory algorithm. In [8], we modify this algorithm to be distributed and *storage persistent* (through the use of append-only logs), while maintaining the original constant per-node traversal time complexity of the node-copy method. That is, the node-copy method maintains a constant traversal cost between all nodes in a PDS. We overview this approach (called PEDaLS) in Section 4. Most past works on PDSs cover applications from theoretical computer science [20], text editing [21], and computer-aided design [22]. Our extension to distributed systems enables their use in providing application robustness, distributed debugging and root cause analysis, and repair/replay for IoT applications.

A technology related to data versioning is git [15]. Git is a version control system typically used for text documents with some support for binary documents. GitHub [16] is an online platform for hosting git repositories. A primary difference between persistent data structures and git is that the former is used for program data, whereas the latter is used mostly for source codes or files. Git provides support for exploring the difference between two versions, working on different versions (similar to fully persistent data structures), and merging different versions (similar to confluently persistent data structures) among others.

2.2 Append-Only Logs

Due to a decline in storage costs, append-only logs are used widely in distributed and cloud computing systems to facilitate immutability, robustness, and scalability. Examples of log-based systems include cloud object stores [23], [24], event systems [12], distributed databases and file systems [25], [26], [27], [28], [29], log-based transaction systems [30], [31], [32], and popular messaging and streaming services [33], [34], [35].

Immutability facilitates robustness and coordination avoidance [18], [36] as well as high availability (through eventual consistency) for cloud storage, gossip protocols, collaborative editing, and revision control, among others [37], [38], [39]. While data versioning provides immutability from a software level, append-only logs provide immutability from a storage level. Entries in a log are ordered and most log storage systems provide a form of *sequence number* (e.g., Kafka [40] provides offsets, and Facebook LogDevice [41] provides log sequence numbers) that reflects the log order. Log storage systems typically provide a simple API for creating a log, appending to a log, and retrieving entries from specific sequence numbers in a log. This generic API facilitates communication among heterogeneous devices. Our work is agnostic of the underlying log storage system as long as it provides the aforementioned functions.

Append-only logs also provide a convenient mechanism for operation reversal. If we express update operations of an arbitrary data structure as log appends, we can perform a reversal of those operations via log rollback. This is helpful in cases where a data structure is erroneously updated and later on it is found that the update should not have been applied (or a different one should have been applied). In these cases, logs enable us to rollback all operations starting from the latest up to the undesired one, and to then apply the correct operation if necessary via replay of the operations that follow. This has the same final effect as the undesired operation not being applied at all. Note that this is different than applying a seemingly inverse operation to a data structure, which may result in unwanted side effects.

Similar to append-only logs, blockchain [17] is a continuously growing data storage. However, blockchain has cryptographic backing to ensure the integrity of each added block, whereas the append-only semantics of logs are only ensured through their exposed interfaces. That is, if a malicious user were to change data previously appended to a log through some means, there is no feasible way to detect this anomaly. On the other hand, any such attempt in a blockchain would reveal this modification through cryptographic algorithms. The downside is that in addition, blockchain is resource-intensive. Blockchain is inherently decentralized with algorithms that add to its integrity through this decentralization. Both distributed logs and blockchain are eventually consistent. Distributed logs used in conjunction with consensus protocols such as Paxos [13] and Raft [14] can be strongly consistent as well.

2.3 Data Replication

Distributed systems replicate data to ensure the availability and robustness of the system. However, multi-tier deployments pose unique challenges which make many existing replication protocols inapplicable. First, these deployments include a vast range of heterogeneous devices including resource-constrained systems such as microcontrollers and single-board computers. Thus, protocols which require significant memory or complex computation are not suitable for them. Second, these devices are commonly battery-powered and are connected via unstable networks.

Protocols that require frequent coordination among replicas can experience repeated failure and restart of the replication procedure in such settings. Due to these reasons, protocols such as Paxos [13] and Raft [14] that provide strong consistency and require a quorum through multiple messages among the replicas are precluded for many IoT devices. Multiple works in the literature suggest resorting to a weaker consistency model in favor of availability [42], [43]. Fortunately, many IoT use cases do not require strong consistency semantics and instead can tolerate weaker consistency models (with lower coordination requirements) such as eventual consistency [4], [6], [7]. Thus we can trade off strong consistency for high availability (which many of these use cases do require) in the face of network partitions [44]. Specifically, our approach supports *strong eventual consistency* for

replicated data structures, similar to that used in collaborative environments [45], [46], [47] via Conflict-Free Replicated Data-Types (CRDTs) [11].

Motivated by the various limitations of Paxos, researchers have proposed multiple variations over the years. In Paxos, the leader performs a disproportionately large amount of communication compared to followers. PigPaxos [48] attempts to reduce the load of the leader by distributing some communication responsibilities to follower nodes, called relays. However, for write-heavy workloads (e.g., for sensor-driven IoT applications), the communication load is inherently distributed, limiting PigPaxos' advantage. Moreover, PigPaxos only redistributes the communication workload among nodes; it does not necessarily reduce the total amount of required communication (versus Paxos). Additionally, in failure-prone, heterogeneous environments such as IoT, replicas running Paxos-based protocols often must give up on current progress and start fresh if a quorum is not met. A quorum might not be met for many reasons including multiple contending writers, device/power failure, network latency, etc. Hence the overhead of Paxos-based algorithms is compounded in such scenarios. On the other hand, eventual consistency-based protocols can progress even in failure-prone environments in presence of multiple writers, as each replica is able to execute its operation immediately and incorporate the rest of the operations when the other devices become reachable.

DPaxos [49] proposes a dynamic allocation of quorums to avoid unnecessary wide-area communication. The communication reduction of DPaxos is heavily dependent on the premise that leader election is infrequent. However, if a leader-based protocol is used for write-heavy workloads, the leader election phase will be more frequent. Moreover, in the case of multiple writes, the dynamic allocation of quorums can be prolonged multiple times; effectively being computationally more complex than Paxos without providing any additional advantage. Hence we employ strong eventual consistency for the write-heavy edge deployments to reduce overall communication overhead.

3 Use Cases

In this section, we describe the research question that PEDaLS strives to answer and provide use cases from existing literature that can benefit from the features we adopt in PEDaLS. We also explain why eventually consistent replication is more effective than other forms of replication for each of the use cases. In addition, we identify the advantages of versioning in these settings. Note that the use of append-only logs is a core design feature that is not necessarily dependent on individual applications. Rather it is a design principle that enhances the overall PEDaLS system, as explained in Section 2.2.

Heterogeneous distributed systems such as the IoT are inherently failure-prone due to the deployment environment (e.g., poor network connectivity, unstable power source, devices with limited capacity, etc.). Moreover, in the face of a failure, it is difficult, if not impossible, to debug the system due to the loss of relevant data. PEDaLS provides an approach to alleviate this problem by ensuring that relevant data is preserved and available, possibly in multiple storages, along with a trace of state changes. PEDaLS achieves this through three foundational design choices. First, PEDaLS uses versioned data structures to capture the historical trace of change in data. The immutability of versioned data structures ensures the previous versions are always preserved and the last complete state is always checkpointed. Thus it helps in system debugging in the face of a failure.

Second, PEDaLS uses replication to ensure the availability of data. As strong consistency involves coordination overhead which can get worse in the presence of poor network connectivity, PEDaLS opts for eventual consistency. Eventual consistency is also useful in a decision support context, where a specific decision does not depend on all data items being present.

Third, PEDaLS uses append-only logs for storage. Append-only logs provide robustness and scalability. The immutability of append-only logs facilitates both coordination avoidance and high availability. While versioned data structures provide versioning from a program level, append-only logs provide versioning from a storage level. Next, we present multiple existing use cases from the literature where eventual consistency (as opposed to strong consistency) and versioned programming are beneficial.

Urban Traffic Steering [2]: In smart cities, different sensors are deployed along the streets to gather information regarding crowding, pollution, traffic, etc. To supplement effective decision making, other parameters such as events, attractions, and comments on an area are drawn from distributed or cloud databases. From the collection of this information, smart applications can guide pedestrians and vehicle traffic in complex urban environments. This is a typical example of interaction among heterogeneous devices in an environment which is inherently eventually consistent – numerous devices are continuously updating the state which stabilizes over time (e.g., the traffic in an area). As explained earlier, this is an example of a decision support context where all data points are not necessary to make a decision. Note that strong consistency is not necessary for this application and would only hinder and postpone decision-making due to the communication overhead involved in such protocols. On the other hand, eventual consistency allows each data source to generate and propagate data throughout the system without complex coordination, resulting in a smoother user experience. Moreover, historical data preserved through versioning can provide insightful details on predicting future traffic patterns. In [2], the authors describe eventual consistency as a property of certain IoT applications. The authors further present that such applications not only converge eventually but that the current state can be used to approximate the future states of the environment.

Inter-Device Task Dispatch [3]: It is not uncommon for IoT devices to interact with other nearby devices to complete an objective. Inter-device apps can leverage resources from multiple devices by sharing data and tasks among devices. In [3], the authors point out that the prevalent practices in inter-device app development for IoT are not disciplined and that tasks are dispatched statically with strong consistency. Static dispatch limits the range of interaction among devices while strong consistency imposes restrictions on inter-device apps due to synchronization overhead. Hence

the authors propose Inter-Device Task Dispatch (IDTD), a framework to construct and dispatch tasks into multiple devices dynamically with eventual consistency in a systematic manner. Data versioning in such contexts can store the execution history and as such can be used for system debugging in later phases.

Smart Locks [4]: Smart locks replace traditional door locks with deadbolts that can be electronically controlled by mobile devices or remote servers. In [4] the authors show that an eventual consistency design provides robust revocation and access logging mechanisms for smart locks. At the same time, this design minimizes the system's dependency on external entities, maintains a high level of availability, and reduces the system's vulnerability to remote compromise by allowing devices to forgo direct connections to the internet. Due to lower dependencies on external entities, these locks can be less prone to external attacks, resulting in a more secure system. In smart lock systems, data versioning can provide an implicit log of access control, thus aiding security checks and debugging in case of a breach.

Banking Queue Monitoring [50]: Queue monitoring is used for multiple purposes, such as determining the current load, resource requirement, and expected service time of the system. In [50] authors provide a manually collected dataset of service queues at different banks (cf. Section 6.4.1). Such data collection can be automated by sensors placed at the entrance and exit or even by indoor localization using smartphones and WiFi access points [51]. Data versioning in this application can facilitate predicting future queue conditions. Moreover, as the exact number of occupants is not required to be known at any instant of time, rather an approximate value is sufficient to make relevant decisions, an eventual consistency based-approach works well here as well. Furthermore, adopting an eventual consistency-based approach allows us to avoid coordination overhead, thus ensuring the longevity of battery-powered sensors and better utilization of in-house network bandwidth.

4 Node-Copy Method and PEDaLS

In this section, we present an overview of the node-copy method [10] and PEDaLS [8]. Node-copy method is an efficient single machine, in-memory algorithm to create versioned linked data structures having constant in-degree. The time complexity of update/read operations in this method is constant per operation step, where an operation step is defined as the traversal from one data structure node to the other. A data structure node of a PDS in the node-copy method differs from that of an *ephemeral* (i.e., non-versioned) node in two ways: (i) Each information field and pointer fields in a node are tagged with a version stamp. Version stamps start with one and are monotonically increasing. Every update operation is considered to generate a new version of the data structure, hence the version stamp is increased during each update. (ii) Every data structure node has provisions for a fixed number (user-defined) of extra pointers to accommodate future updates. Once these extra pointers are filled, a copy of the node is made with only the latest pointer fields, essentially freeing up pointer fields for further update operations.

PEDaLS [8] converts the single machine, in-memory node-copy method into a distributed, storage persistent one while maintaining the time complexity of update/read operations of the node-copy method. It exposes data structure versions to developers for use in dependency tracking and program analysis [52], history-aware programming [53], and repair and replay [54] in distributed settings. PEDaLS consists of append-only logs at the storage level, which it can internally interact with simple API calls. Similar to the node-copy method, data structure nodes contain versioned fields with provision for extra pointers. PEDaLS abstracts away all of these complexities from the developers and provides them with functions to create, modify, and access the data structures. PEDaLS does not allow multiple writes (i.e., update operations) at the same replica at the same time, as semantically a new version is obtained by performing an update operation on the latest version – thus requiring an order over the update operations. However, it allows reads during a write operation.

The in-memory node-copy method can reuse previous data for the latter versions and can use extra pointers to accommodate future updates. PEDaLS retains these design choices to create versioned data structures. Implementing the node-copy method using append-only logs has its own challenges. To start with, every pointer manipulation must be expressed as appends to one or more logs instead of being executed in-place. Moreover, in order to maintain the time complexity of the in-memory algorithm, expensive log scans must be avoided. As logs can be remote, PEDaLS must withstand network failures. Finally, atomicity across logs must be guaranteed, i.e., an update operation should either append to all the logs that it is supposed to append to, or keep all of the logs unchanged.

In order to address all of these challenges, PEDaLS uses three types of logs: (i) *DataLog*, (ii) *LinkLog*s, and (iii) *APLog*. The DataLog stores the information field of data structure nodes. A LinkLog stores the pointer fields of a specific data structure node. Each node in a data structure has a dedicated LinkLog. Therefore, a data structure node can be uniquely expressed through a pair of sequence numbers, one for the DataLog and another for the corresponding LinkLog. As each data structure node has a dedicated LinkLog, any pointer modification of a node can be expressed as an append to the corresponding LinkLog. Much like how node-copy method has a fixed number of extra pointers for a node, PEDaLS assumes a fixed number of entries for a node in its LinkLog. As an example, the number of original pointers o is 2 in a bst node (one left and one right). If the number of extra pointers e is 1, the total number of pointers p in a bst node is $p = o + e = 2 + 1 = 3$. Therefore, the first three entries in a LinkLog denote the original node, the next three entries denote the copy of the node, the next three the copy of the copy, so on and so forth. This is the ideal case without any network failure. However, network failures make the scenario complicated with remote logs. PEDaLS considers two types of failures: (i) *Type 1*. An append to a LinkLog fails. This does not change the above calculations. (ii) *Type 2*. An append to a LinkLog succeeds, but the acknowledgment is lost. In this case, PEDaLS retries up to a certain user-defined number of times, which results in appending the same entry multiple times. In this case, the

total number of entries allocated for a node becomes a function of the number of failures f (i.e., $p = o + e + f$). In order to identify these sorts of repeated entries, each entry in a LinkLog embeds the number of remaining pointers for the node after the insertion of the current one. This essentially turns every repeating append to the LinkLog idempotent, barring the side-effect of using up space.

Similar to the in-memory node-copy method, PEDaLS uses the access pointer log (APLog) to store the root node of different versions of the data structure. APLog is always written at the end of an update operation. An append to the APLog denotes a successful completion of a version. If any other log contains a version stamp vs' that is not present in the APLog (possibly due to a system crash), this implies that the version is incomplete. To meet the atomicity requirement of an operation, upon system recovery the tails of all the logs are examined and any entry containing the version vs' is rolled back.

5 Replicating Versioned Data Structures

In this section, we present how we extend PEDaLS to provide replication of versioned data structures. Replication is used to make systems fault-tolerant and to ensure the availability of systems. As most multi-tier deployments have multiple devices spread across different sites, most of which are generating data, we accommodate multi-writer replication.

5.1 System Model

We consider an IoT deployment of N replicas (devices) generating data, i.e, multiple writers can generate data at the same time. Each replica is assigned a node ID from a set S. We represent a replica as $X_s, s \in S$. The underlying network is asynchronous and unreliable; messages may be dropped, duplicated, or reordered. The network may partition and eventually recover. Each replica has local durable storage. We assume replicas may face non-byzantine failures; a replica may crash but will have access to the information recorded in durable storage upon recovery.

5.2 Version Stamps

Although PEDaLS uses monotonically increasing integer values to represent versions, we change the versioning scheme to denote which replica originally executed an operation. Specifically, we use the concatenation of a counter with a node (replica) ID to represent a version stamp (Lamport timestamp [55]). We represent the counter and node ID of a version stamp vs by $vs.counter$ and $vs.nodeID$ respectively. We say version stamp vs_a is less than version stamp vs_b ($vs_a < vs_b$) if and only if (i) the counter of vs_a is less than that of vs_b, or (ii) both the counters are same but node ID of vs_a is less than that of vs_b. When replica X_s executes a new operation in response to a client request, it tags it with version stamp vs ($vs.nodeID = s$), which is greater than all other version stamps it has observed so far (operations that *happened before*). Thus if operation op_a happens before op_b at a replica, $vs_a < vs_b$ where vs_a and vs_b are the version stamps of operations op_a and op_b, respectively.

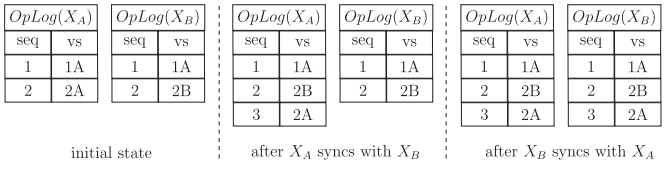
Version stamps of concurrent operations can be ordered arbitrarily but deterministically. Throughout the rest of the paper, we use version stamps to refer both to the version stamp itself and to the operation it tags. The intended use will be clear from the context. We say vs is an operation of X_s (alternatively, X_s is the originator of vs) if $vs.nodeID = s$.

5.3 Ordering Operations

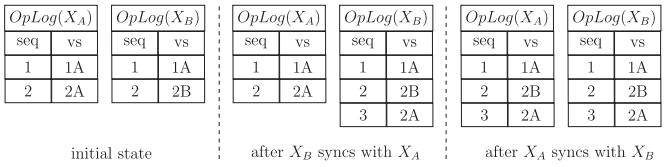
Over time replicas may diverge from each other due to update requests from clients (i.e., processes that can update or query a data structure by sending a request to any replica). To reconcile this divergence each replica periodically performs a round of *merge steps* with the other replicas. A merge step is always between a pair of replicas. Therefore, in a round, there are at most $N - 1$ merge steps. In a merge step, one replica (known as the *reader*) reads entries in the log of operation from another replica (known as the *source*). The goal of the merge step is for the reader to identify and incorporate operations "unknown" to it (i.e., not previously executed at the reader) that the source has already executed. The reader ensures that the causality relationship among the operations is retained while creating this merged list of operations and subsequently executing them.

One way to achieve consistency among replicas is to maintain a log of operations ($OpLog$) in each replica. Each entry in an OpLog is the tuple *(vs,op,val)*. *vs* is the version stamp of the operation, *op* is the type of update operation, and *val* is the operand of *op*. We denote the OpLog of a replica X_s as $OpLog(X_s)$. As long as all replicas execute the same set of operations in the same log order, they will converge. In fact, this is the same principle used in replication protocols such as Raft [14]. However, in order to reduce communication among replicas and latency experienced by clients, we allow temporary divergence of the OpLogs, which is reconciled later on (eventual consistency instead of Raft's strong consistency). This means, at times, a replica must rollback operations in the OpLog along with all other logs used by the underlying data structure to record these rolled back operations, followed by execution of new operations, finally followed by replay of previously rolled back operations. In our work, we model the history of operations (OpLog) as a list CRDT and use an adaptation of the method used in [56], as explained later. Using logs to maintain order adds extra complexity, namely, avoiding log scans. We propose new logs and algorithms to efficiently maintain order in Sections 5.4, 5.5, and 5.6.

Although PEDaLS does not allow multiple concurrent writes at the same replica to preserve the integrity of a version (cf. Section 4), it still allows incoming write requests during merge steps. To do so, the reader makes a backup copy of the underlying data structure up to the version which is certain to remain unchanged even after the merge. A merge step potentially involves reordering and replay of operations that take place on this backup copy. Meanwhile, any direct write request to the reader is serviced by working on the main copy with temporary version stamps. The reader thus has to incorporate these writes on the backup copy at the end of the merge step with updated version stamps, during which new write requests are blocked. As an optional optimization to increase write availability (which we do not adopt in the experiments), the reader can choose to block writes only if the number of new writes not in the backup copy falls below a certain threshold set by the application developer. Otherwise, it repeatedly checks for



(a) X_A merges with X_B then X_B merges with X_A.



(b) X_B merges with X_A then X_A merges with X_B.

Fig. 1. Change in OpLogs as replicas merge with each other. We notice the two different sequence of merge steps results in the same consistent state at the end.

new local writes and incorporates these into the backup copy without blocking. Finally, the reader updates the main copy with the backup copy, which can be done efficiently by simply swapping the references to the two copies.

When a replica X_A executes an operation as a direct request from a client, it appends the operation at the end of $OpLog(X_A)$. Apart from direct client requests, replicas also execute operations that are unknown to them from other replicas' OpLogs (during merge steps). Assume vs_{new} is an operation in $OpLog(X_B)$ that X_A has not yet executed. We denote the operation immediately preceding vs_{new} in $OpLog(X_B)$ as vs_{pred}. As the intention is to maintain a consistent order of operations, X_A tries to place vs_{new} in its own OpLog after vs_{pred} as well. Therefore, to incorporate the unknown operation vs_{new}, X_A first locates vs_{pred} in $OpLog(X_A)$. Let us denote the operation in $OpLog(X_A)$ immediately succeeding vs_{pred} as vs_{succ}. That is, vs_{new} and vs_{succ} are concurrent operations. Now X_A inserts vs_{new} in $OpLog(X_A)$ immediately after vs_{pred} if $vs_{new} > vs_{succ}$. Otherwise, X_A skips over all contiguous version stamps that are greater than vs_{new} and then places vs_{new}. Of course, it might happen that vs_{pred} is not present in X_A to begin with. In that case vs_{pred} must be inserted first. This implies that X_A should start reading $OpLog(X_B)$ from the earliest sequence number that contains an operation unknown to it. We express this whole procedure of inserting operation vs_{new} after vs_{pred} as $insert(vs_{new}, vs_{pred})$.

To illustrate how *insert* works, we refer to the OpLogs in Fig. 1 (only the sequence numbers and version stamps are shown for brevity). We consider two replicas in our system, X_A and X_B. Let us assume X_A executed operation $1A$ that X_B became aware of during the latter's merge step. At this point, both X_A and X_B executed one operation independently but concurrently, operation $2A$ and $2B$ respectively. Now we consider two different scenarios. (i) Fig. 1a. X_A (reader) merges with X_B (source). For now, we assume readers start comparing the two OpLogs from the beginning (we show in Section 5.5 how full log scans can be avoided). Both OpLogs have $1A$ as the first entry, so no action is needed. However, X_B has $2B$ in the second entry whereas X_A has $2A$. This is equivalent to the insert operation $insert(2B, 1A)$ i.e., insert $2B$ after $1A$ in $OpLog(X_A)$ (as $2B$ comes after $1A$ in $OpLog(X_B)$). We note how the insertion operation is implicitly embedded in the log order. As X_A currently has $2A$ after $1A$ and $2B > 2A$, it can place $2B$ after $1A$. "Placing $2B$ after $1A$" is a multi-step process: X_A trims its OpLog up to sequence number 1, append the entry containing $2B$, and finally re-append the entry containing $2A$. Additionally, it trims/(re-)appends to any logs used by the underlying data type. When X_B (reader) merges with X_A (source) after this, X_B can simply append $2A$ after $2B$ in its OpLog. (ii) Fig. 1b. X_B (reader) merges with X_A (source). Starting comparison from the top of the OpLogs as before reveals different entries in the second entry: $OpLog(X_A)$ has $2A$ as the second entry whereas $OpLog(X_B)$ has $2B$. This translates to the operation $insert(2A, 1A)$ to be executed in OpLog of X_B. As the version stamp after $1A$ at X_B is $2B$ and $2A < 2B$, $2A$ is placed after $2B$. Merging the other way follows the steps similar to the previous scenario. We see that in both scenarios we end up with the same final state in both the replicas.

Note that we could have forgone this relatively complex ordering following [56] and instead chosen a strict ascending order of version stamp counters, breaking ties through lexicographical order of node IDs. However, this approach would have resulted in interleaving sequences of operations made by different replicas concurrently. Our current choice of the method in [56] on the other hand makes sure concurrent sequence of operations executed by different replicas are not interleaved. In the chosen method, the contiguous operations performed by a replica at a time are placed together with minimal breaks. Also, note that to break tie between concurrent operations we choose the greater version stamp to take precedence over the smaller one (e.g., $2B$ appears before $2A$) to maintain similarity with existing work [56]. In practice, we could have chosen the reverse.

In a merge step between a reader X_i and a source X_j, the reader performs two tasks: (i) *Conflict detection*: The reader detects whether it is in conflict with the source, i.e., whether the source has operations that the reader does not know of. Note that we are concerned with unidirectional conflict, i.e., if the reader has operations that are unknown to the source no extra steps are taken (this is resolved during some other merge step when the current source becomes a reader). A simple way to detect conflict is to scan the OpLogs of the reader and the source from the top until a mismatch is found in the corresponding sequence numbers. However, full log scans can get prohibitively expensive. Hence we explain a mechanism to avoid log scans in Section 5.5. (ii) *Conflict resolution*: In case of conflict, the reader resolves this conflict, possibly by reordering the operations which require rollback and replay of some operations. The conflict detection stage finds the sequence numbers of the two OpLogs from where the comparison should be started ($reader_{start}$ and $source_{start}$ for OpLog of the reader and the source respectively) to guarantee that the reader encounters all the operations it has not seen that have been already executed by the source. These two sequence numbers are used by the conflict resolution stage to incorporate all the unknown operations in the reader's OpLog.

We next introduce a new log that helps us to avoid a full log scan (Section 5.4). We show how the conflict detection stage uses this log to detect the presence and point of

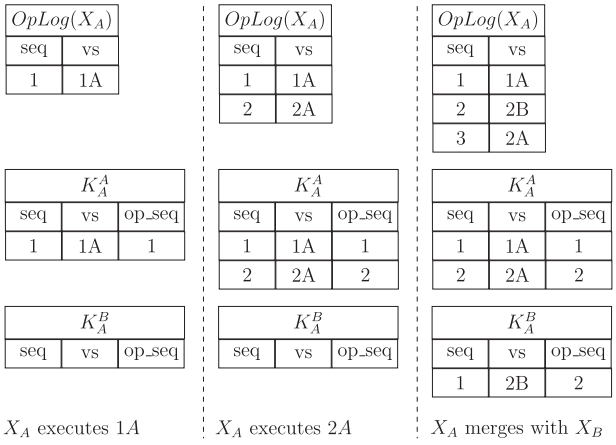


Fig. 2. Mapping from version stamps to sequence number of OpLog in KnowledgeLogs.

conflict (Section 5.5). Section 5.6 then describes how the conflict resolution stage takes this information and uses $insert$ operations to execute a list of ordered operations.

5.4 KnowledgeLogs

OpLogs grow over time and the merge steps become costly if we must scan from the top. To avoid a full scan of the OpLog of the source by the reader, a replica maintains a map of the last observed version stamp from each replica to a sequence number in its OpLog using one KnowledgeLog for each replica. Each entry in a KnowledgeLog contains the tuple *(vs, op_seq)*. *vs* denotes the version stamp of the operation. *op_seq* denotes the sequence number of OpLog where the operation with *vs* was first appended. More precisely, each entry of KnowledgeLog K_i^j on host X_i contains tuples that map each version stamp vs whose node ID is j to a sequence number in $OpLog(X_i)$. Although the position of a version stamp might change due to later merge steps, note that a version stamp can only be pushed down in order but never pulled up due to the way $insert$ works. Thus, the sequence numbers stored in KnowledgeLogs provide us a starting point to search for a version stamp. The version stamp might be at that sequence number, or at a later one, but never at an earlier one. For improved performance, we can also opt to cache a fixed number of entries from the end of KnowledgeLog in memory, which must be a tunable parameter depending on the device capabilities. As all the information needed to maintain a KnowledgeLog are present in the OpLog, KnowledgeLogs can be reconstructed after a system crash.

We refer to Fig. 2 as an example of interactions among OpLogs and KnowledgeLogs. Operation $1A$ is inserted in $OpLog(X_A)$ at sequence number 1. To record the mapping from version stamp $1A$ to sequence number 1, X_A appends $(1A, 1)$ to K_A^A. Similarly, X_A appends $(2A, 2)$ to K_A^A to record that the operation with version stamp $2A$ was inserted in $OpLog(X_A)$ at sequence number 2. A merge step with X_B results in the operation with version stamp $2A$ to be pushed down in order i.e., at sequence number 3. As we have already recorded $2A$ in K_A^A and we can reach $2A$ in $OpLog(X_A)$ even if we start scanning from the recorded op_seq value (in this case 2), we can keep it unchanged. We only append the entry $(2B, 2)$ to K_A^B. Now if X_A (reader) performs a merge step with an arbitrary replica X_s (source) and wants to know whether X_s has any operation originating from replica X_B that the reader does not know of, it can simply compare the tails of K_A^B and K_s^B. If the last entry of K_A^B contains a version stamp that is less than that of the version stamp contained in the last entry of K_s^B, then X_s has operation originating from X_B that X_A does not know of (as two version stamps with same node ID follow happens-before relationship and version stamps are written to the KnowledgeLog in increasing order). This process is explained in detail in the next section.

5.5 Conflict Detection

In the conflict detection stage during a merge step between reader X_i and source X_j, the reader X_i compares the last entries of K_i^m and K_j^m, $\forall m \in S$. We represent the last entry of a log L by $tail(L)$ and a field f in entry e by $e.f$. If $tail(K_i^m).vs < tail(K_j^m).vs$, this means X_j (source) has executed operations that X_i has not. This holds as the operations in a KnowledgeLog have the same node ID and are executed in increasing order of their counter. The counter captures the happens-before relationship between two version stamps with the same node ID. We say X_i lags behind X_j with respect to X_m when $tail(K_i^m).vs < tail(K_j^m).vs$. X_i might lag behind X_j with respect to more than one replica. Let us represent the set of all replicas with respect to which X_i lags behind X_j as X_{lag}.

We represent the set of node IDs of the replicas in X_{lag} as S_{lag}. We find the replica X_p in X_{lag} such that $tail(K_j^p).op\_seq < tail(K_j^l).op\_seq, \forall l \in S_{lag} \wedge l \neq p$. That is, X_p is the replica whose operation is at the earliest point of conflict between X_i and X_j. However, X_i might not know about operations of X_p that have version stamps less than $tail(K_j^p).vs$. To ensure X_i can detect all unknown operations, it scans backward from the tail of K_j^p until it finds the entry e such that the entry before it has a version stamp equal to $tail(K_i^p).vs$. Then $e.op\_seq$ is the sequence number from which the reader start scanning the source's OpLog (i.e., $source_{start} = e.op\_seq$). In other words, $e.vs$ is the earliest operation in $OpLog(X_B)$ that X_A has not yet executed.

Let the version stamp of the sequence number $source_{start} - 1$ in $OpLog(X_j)$ be vs_{prev}. To incorporate $e.vs$, X_i executes $insert(e.vs, vs_{prev})$ in $OpLog(X_i)$. To do this, X_i first finds the sequence number of $e.vs$ in $OpLog(X_i)$ – the value of $reader_{start}$ is this sequence number plus one. Note that all operations in $OpLog(X_j)$ from sequence number 1 to $source_{start} - 1$ must be present in $OpLog(X_i)$, otherwise there is some operation between these two sequence numbers in $OpLog(X_B)$ that X_A has not seen, and the value of $source_{start}$ found by the previous steps would have been different. Therefore, $reader_{start}$ must be greater than or equal to $source_{start}$. To find the value of $reader_{start}$, X_i starts scanning $OpLog(X_A)$ from the sequence number $source_{start} - 1$. It stops scanning if the currently scanned entry's version stamp is equal to vs_{prev}. The required value of $reader_{start}$ is the sequence number where we stop scanning plus one.

To illustrate the conflict detection stage, we consider the scenario in Fig. 3. Let us assume there are three replicas in our system, X_A, X_B, and X_C. The OpLog of X_A has $1A$ and

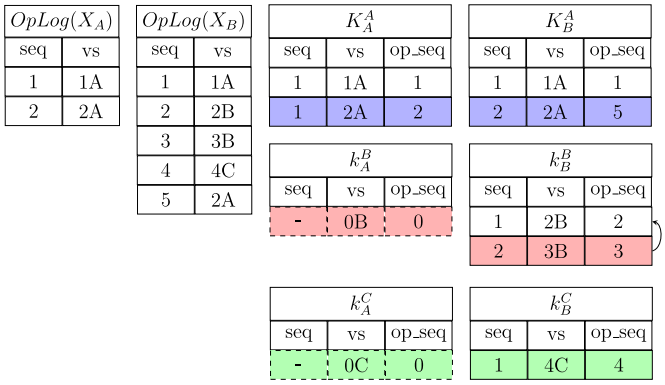


Fig. 3. OpLogs and KLogs of replicas X_A and X_B in a system with three replicas. Dashed entries represent placeholders used during computation when a knowledge log is empty. During conflict detection, the reader X_A compares the same colored entries with each other to find the earliest possible point of conflict. The arrow from the second entry to the first entry of K_B^B, represents X_A's backward scan to find the earliest version stamp with node ID B that it does not know of.

$2A$, whereas the OpLog of X_B has $1A$, $2B$, $3B$, $4C$, and $2A$. One possible sequence of actions that might lead to this state: X_A executed operation $1A$. X_B merged with X_A, and then executed two operations $2B$ and $3B$. X_C (not shown in the figure) merged with X_B and executed $4C$. X_B merged with X_C. X_A executed operation $2A$. Finally, X_B merged with X_A again. Now let us consider X_A performs a merge step with X_B. Comparing the tails of K_A^m and K_B^m, $m \in \{A, B, C\}$, we see that X_A lags behind X_B with respect to X_B and X_C, i.e., $X_{lag} = \{X_B, X_C\}$ (we assume the absence of entry in a KnowledgeLog to be equivalent to having a placeholder entry with a version stamp with minimum possible invalid counter value, in this case, 0). As the op_seq value of $tail(K_B^B)$ (i.e., 3) is smaller than that of $tail(K_B^C)$ (i.e., 5), $X_p = X_B$. However, X_A is not yet certain $tail(K_B^B).vs$ is the earliest unknown version stamp. X_A scans K_B^B backwards to find the earliest unknown version stamp, which in this case is $2B$. The corresponding op_seq value is 2, therefore $source_{start} = 2$. The entry immediately preceding $2B$ in $OpLog(X_B)$ has the version stamp $1A$. X_A reads the entry at sequence number $source\_start - 1 = 1$ in $OpLog(X_A)$ and finds that the entry contains $1A$. Therefore $reader\_start$ is equal to $1 + 1 = 2$ as well. The conflict detection algorithm is presented in Algorithm 1.

5.6 Conflict Resolution

Conflict resolution is triggered when a conflict is detected, to find and execute a merged order of operations between the reader and the source. When there are one or more conflicts between the reader and the source, it rolls back the OpLog of the reader to the earliest point where the reader does not lag behind the source with respect to the version stamps before it and then replays the operations at the reader (adjusting the OpLog of the reader) to reflect the merged order. At the start of conflict resolution, X_i knows both $source_{start}$ and $reader_{start}$, i.e., the sequence number of $OpLog(X_i)$ and the sequence number of $OpLog(X_j)$ at which X_i should start comparing the two OpLogs. X_i creates an ordered list, R_i, of the operations in $OrdLog(X_i)$ starting from the sequence number $reader_{start}$ up to its latest sequence number. X_i creates a second ordered list, R_j, of the operations in $OrdLog(X_j)$ starting from the sequence number $source_{start}$ up to its latest sequence number.

Algorithm 1. Conflict Detection

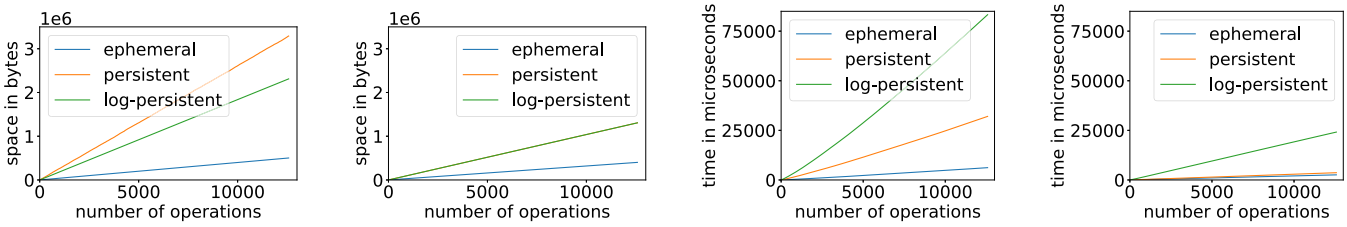
Require: reader replica X_i, source replica X_j, and set of node IDs S
Ensure: earliest point of conflicts $source_{start}$ and $reader_{start}$

$X_{lag} \leftarrow \phi$
for $m \in S$ **do**
  if $tail(K_i^m).vs < tail(K_j^m).vs$ **then**
    $X_{lag} \leftarrow X_{lag} \cup \{X_m\}$
  end if
end for
if $X_{lag} = \phi$ **then**
  return
end if
$S_{lag} \leftarrow \phi$
for $X_m \in X_{lag}$ **do**
  $S_{lag} \leftarrow S_{lag} \cup \{m\}$
end for
$p \leftarrow \text{argmin}_m(tail(X_j^m).op\_seq), m \in S_{lag}$
$idx \leftarrow latest\_seq(X_j^p)$
while $idx > 0$ **do**
  if $tail(K_i^p).vs < K_j^p[idx].vs$ **then**
    $source_{start} \leftarrow K_j^p[idx].op\_seq$
    $idx \leftarrow idx - 1$
  else
    break
  end if
end while
$vs_{prev} \leftarrow OpLog(X_j)[source_{start} - 1].vs$
$idx \leftarrow source_{start} - 1$
while $idx \leq latest\_seq(OpLog(X_i))$ **do**
  if $OpLog(X_i)[idx].vs = vs_{prev}$ **then**
    $reader_{start} \leftarrow idx + 1$
    break
  else
    $idx \leftarrow idx + 1$
  end if
end while
ResolveConflict($reader_{start}, source_{start}$)

To incorporate the operations unknown to itself, X_i first includes those operations from R_j to R_i by invoking $insert$ procedures: for each entry e in R_j, X_i first finds the entry e_{pred} in R_i which contains the version stamp immediately preceding e in R_j. If the version stamp of the entry following e_{pred} in R_i is smaller than $e.vs$, X_i inserts e immediately after e_{pred} (provided e is not already present there). Otherwise, it skips over all contiguous entries where the version stamp is greater than $e.vs$, and then inserts e (provided that e is not already present there). Once X_i has all the operations in R_i, it rolls back, i.e., prunes, $OpLog(X_i)$ starting from $reader_{start}$ and then replays all operations in R_i at $OpLog(X_i)$. The conflict resolution algorithm is presented in Algorithm 2.

6 Experimental Results

In this section, we evaluate multiple aspects of data versioning and PEDaLS. We start with an empirical evaluation of the overhead associated with introducing versioning to in-



(a) Bst space. (b) Linked list space. (c) Bst time. (d) Linked list time.

Fig. 4. Average space and time requirement for bst and linked list. Both the space and the time requirements are linear with respect to the number of operations for the two data structures.

memory ephemeral data structures and subsequent integration of in-memory logs to represent versioning. Note that only this experiment is conducted completely in memory. The rest of the experiments use memory-mapped files as the log abstraction. To demonstrate the scalability of storage persistent PEDaLS, we then show the effect of increasing number of replicas on throughput for workloads with varied composition, i.e., different percentage of read and write operations. Next, we explore the effect of Knowledge logs on the time to detect conflicts. Then, we describe a number of end-to-end applications that leverages the efficiency of PEDaLS in answering complex temporal queries. Finally, we present a probabilistic study on bounding log sizes to ensure a user provided minimum number of versions can be fully recorded by PEDaLS. Unless otherwise specified, we set the number of extra pointers to 1 for versioned data structures in all cases. This captures the worst case scenario in regard to the number of times nodes must be copied.

Algorithm 2. Conflict Resolution

Require: reader replica X_i, source replica X_j, $source_{start}$ and $reader_{start}$ value obtained from Conflict Detection stage
Ensure: X_i is not lagging behind X_j
1: **procedure** ResolveConflict$source_{start}, reader_{start}$
2: $R_i \leftarrow \{OpLog(X_i)[reader_{start}], \ldots, tail(OpLog(X_i))\}$
3: $R_j \leftarrow \{OpLog(X_j)[source_{start}], \ldots, tail(OpLog(X_j))\}$
4: **for all** $e \in R_j$ **do**
5: $e_{pred} \leftarrow$ the entry before e in R_j ▷fixed dummy value assumed for first element
6: $insert(e.vs, e_{pred}.vs)$ in R_i
7: **end for**
8: Prune $OpLog(X_i)$ starting from sequence number $reader_{start}$
9: **for all** $e \in R_i$ **do**
10: Replay/Execute $e.op$ and append e to $OpLog(X_i)$
11: $q \leftarrow$ sequence number of e in $OpLog(X_i)$
12: $k \leftarrow e.vs.nodeID$
13: **if** $e.vs > tail(K_i^k).vs$ **then**
14: append $(e.vs, q)$ to K_i^k
15: **end if**
16: **end for**
17: **end procedure**

We perform our experiments using virtual machine instances in a private cloud running Eucalyptus [57]. Each instance has two 2GHz CPUs and 2GB of memory. Unless otherwise specified, we perform each experiment 100 times and present the average values. We also present the standard deviation when applicable along with the mean, either as raw values (for tabular data) or as error bars (for bar charts).

6.1 Versioning Overhead

In this experiment, we investigate the overhead associated with making in-memory ephemeral data structures versioned, i.e., in-memory persistent. We also investigate the overhead associated with implementing versioned data structures using in-memory logs (i.e., same as PEDaLS but without storage persistence), which we term as *log-persistent*. Note that in this experiment we are concerned exclusively with the overhead in these two cases: (i) making data structures versioned (i.e., persistent as described in [10]) and (ii) making data structures persistent using logs as described in [8]. Hence we keep the storage (in this case, main memory) the same for all three types of data structures: ephemeral, persistent, and log-persistent. Moreover, we design the experiment to retain how a storage persistent log implementation would perform – by making an extra copy from a data structure node to the in-memory log. Our results show that log-persistent algorithms preserve the same time and space complexity as persistent algorithms.

We choose linked list and binary search tree (bst) as representative linked data structures for this experiment since both are used by developers as building blocks for more complex structures, e.g., stacks, queues, ordered collections, etc. Note that the original work on persistent data structures [10] provides space/time complexity guarantees only for linked data structures having constant in-degree. Hence, although a similar mechanism can be employed for other data structures, a similar space/time complexity is not guaranteed. We use CityPulse [58] temperature dataset containing 12579 data points collected from the city of Aarhus in Denmark between February-June 2014 for this experiment. In case of bst, we use the UNIX timestamp at the time of the collection of data point as the key and the temperature as the value, whereas linked list stores both as the value.

We present the space requirements for storing these data points in Figs. 4a and 4b. Note that this space is for the storage of the data structure, i.e., any auxiliary storage used (and subsequently freed) for intermediate computation is not included.

As evident from the figures, the space complexity is linear in number of operations for all three types of data structures – and differ only in the value of the constant. This signifies that log-persistent data structures are able to maintain a similar space complexity as persistent data structures. As expected, log-persistent and persistent data structures require more space than ephemeral data structures due to their retainment of information regarding past versions. In case of bst, persistent and log-persistent data structures

require 6.55x and 4.60x space respectively of that of ephemeral data structure. In case of linked list, this overhead is 3.25x for both persistent and log-persistent data structures.

Interestingly, log-persistent bst requires less space than persistent bst. This is because copying a node in persistent data structures involves copying both the data field and pointer fields [10]. On the other hand, copying a node in log-persistent data structures (as in PEDaLS) involves copying only the pointer fields. In the case of linked list, both the persistent and log-persistent versions require the same amount of space. This is because in this experiment we are only inserting values in our linked list at the head, which never triggers copying of a node, thus keeping the storage space the same for both persistent and log-persistent data structures.

We present the time requirements of storing the temperature data in Figs. 4c and 4d. We see that the time complexity is also linear in the number of operations, signifying that log-persistent data structures are able to maintain a similar time complexity as their in-memory counterparts (persistent data structures). Log-persistent and persistent data structures require more time than ephemeral data structures due to their relatively complex traversal rules. In the case of bst, persistent and log-persistent data structures require 5.17x and 13.44x times that of ephemeral data structure, respectively. In the case of linked list, these overheads are 1.42x and 9.38x for persistent and log-persistent data structures, respectively. The greater time required by log-persistent data structures is expected, as it involves additional non-trivial steps in its operations, such as complex methods to find boundaries among copies of nodes instead of having direct pointers to those copies.

6.2 Scalability

To evaluate the scalability of PEDaLS, we use a subset of the CityPulse temperature dataset used in Section 6.1. As updates are more expensive than reads in general, we augment the subset with random reads to capture the performance of PEDaLS under diverse workloads consisting of different percentages of update and read operations. We consider update percentages (read % is in parenthesis) of 1 (99), 25(75), and 50(50) to observe the impact of workloads with different update/read compositions on scalability over 10000 operations (update+read). We also vary the number of replicas the client sends requests to among 1, 5, and 10. The round trip latency among the instances (replicas+client) as determined through the *ping* utility varies between 0.53ms to 0.93ms asymmetrically on average. The network bandwidth among the instances is approximately 1Gbits/second. To simulate the effect of horizontal scaling where clients located in different regions access different replicas, a client process evenly distributes operations across replicas using round-robin without delay.

Fig. 5 shows the throughput of the system in operations per second for PEDaLS bst. As evident from the figure, throughput decreases with the increase in write percentage. This is expected, as update operations involve appends to multiple logs in addition to searching for a value. Moreover, we observe that although throughput of PEDaLS increases with an increase in the number of instances, this increase is not linear. This is due to the processing required for background merge steps.

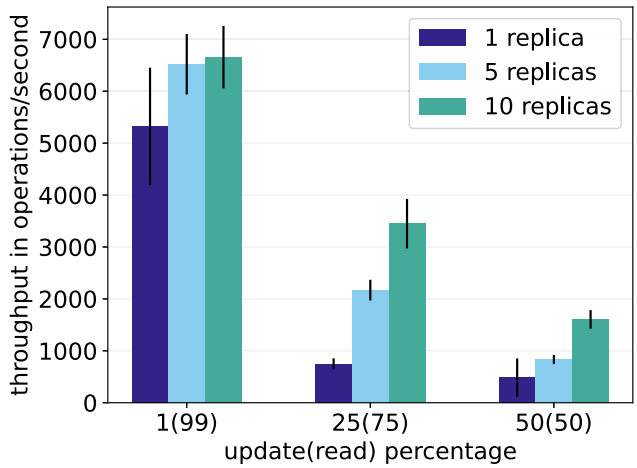


Fig. 5. Scalability of PEDaLS.

6.3 Effect of Knowledge Logs in Conflict Detection

To reduce the time required for conflict detection, PEDaLS uses Knowledge logs. In this section, we explore the time benefit provided by Knowledge logs by isolating the time to detect conflict between two replicas. Specifically, we run two different sets of experiments. In one set of experiments, as has the case been so far, PEDaLS uses Knowledge logs to optimize conflict detection. In the other set of experiments, PEDaLS performs naive conflict detection, i.e., reads the OpLogs of the reader and the source from the top until a point of mismatch is found or one of the logs is exhausted.

For this experiment, we vary the number of update operations among 100, 200, and 300. We send half of the operations to one replica X_A and the other half to another replica X_B, both chosen from the pool of replicas used in Section 6.2. As the execution time of conflict detection can depend on the role (source or reader) of a replica, we calculate the time for conflict detection in both ways and take the average. The number of operations already present in a replica can affect the time for conflict detection as well. Hence, we consider four cases: (i) both X_A and X_B each have half of the operations, (ii) X_A has half of the operations and X_B has all of the operations (due to a merge step), (iii) X_A has all of the operations and X_B has half of the operations, and finally (iv) both X_A and X_B have all of the operations.

We show the total time taken for conflict detection for different workloads in Fig. 6. Our results show that conflict detection with Knowledge log can be 11.34x as fast as conflict detection without Knowledge log for 300 update operations. In general, irrespective of whether Knowledge log is used or not, the time for conflict detection increases with the number of update operations. This is expected, as more update operations result in more entries in both OpLog and Knowledge log, which in turn leads to more log scans.

6.4 End-to-End Applications

In this section, we evaluate the performance of PEDaLS for three different applications: (i) banking queue monitoring, (ii) room occupancy detection, and (iii) livestock tracking.

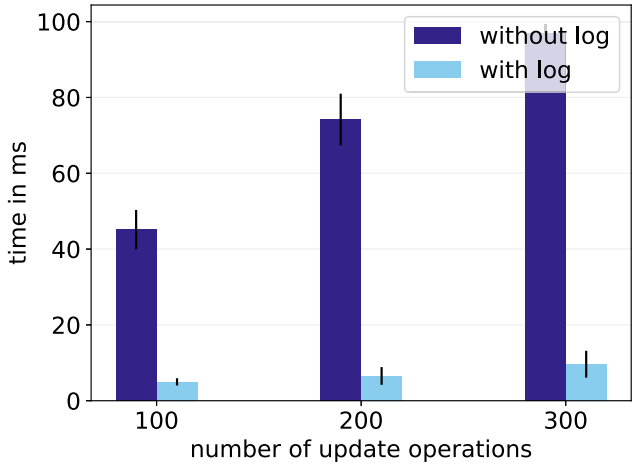


Fig. 6. Conflict detection with and without Knowledge logs.

We compare the performance of PEDaLS with a relational database (PostgreSQL) and a NoSQL database (MongoDB) for each of the above-mentioned applications. We summarize these results in Table 2 and provide a visual representation of the results in Fig. 7.

Our results show that PEDaLS outperforms the other two storage systems when it comes to answering complex temporal queries for all applications. Although PEDaLS is outperformed by the other two storage systems for update operations in two applications, the speedup provided by PEDaLS in executing temporal queries far outweighs the slowdown in updates. Moreover, PEDaLS provides programmatic access to versioned data, while other systems do not. In essence, PEDaLS makes a tradeoff between update performance and version accessibility in the favor of the latter.

6.4.1 Banking Queue Monitoring

In mathematics, queuing theory involves the analysis of several related events such as arriving at a queue, the wait time at a queue, and departure from the queue. The models created through such analysis can be used to make decisions regarding increasing servers, optimizing queue length, and approximating heavy and light traffic [50]. One practical example of the application of these mathematical concepts is to determine the efficiency of a system as indicated by the average wait time of a request/person in a queue. Data related to physical queues can also help in providing location information for individuals (e.g., was a person in the queue at a certain time) and describing individual behavior (e.g., when does a person generally show up for a service).

In [50], authors provide a queue dataset for three banks in Ogun State of Nigeria collected over four weeks for each bank. The dataset contains the wall-clock time when a user enters the bank and the time in minutes for the user to reach the front of the queue. The dataset assigns a monotonically increasing integer as a user ID to every user entering the bank each day (starting with 1) but does not contain the information whether a user ID x of one day corresponds to the same person having user ID y on some other day. Hence, for the purpose of this experiment, we assume every user is assigned a new ID upon each entry. We select the data for one bank and augment it by calculating the timestamp of departure by adding the wait time in queue to the timestamp of arrival. Thus each data point contains three information: a user id, a timestamp, and an operation (enqueue or dequeue). This information is recorded in a storage system every time a user enters and leaves the queue. The final dataset contains 35534 data points.

We assume we want to answer the query Q_1: "which users are present in the queue at time X"? This is an example of a temporal query that can be efficiently answered using a versioned queue. Although the storage of the data is simple for both of the databases under consideration (i.e., a single insertion for both PostgreSQL and MongoDB) and PEDaLS queue (i.e., a single enqueue or dequeue operation with the timestamp as the version and user ID as the value), answering Q_1 requires complex queries on part of the databases as shown in Fig. 8. On the other hand, PEDaLS only requires a versioned traversal of the queue to answer Q_1.

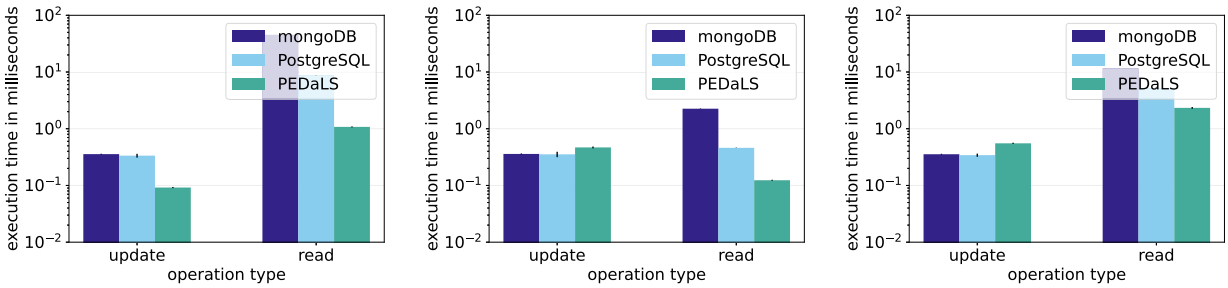
Our results show that the average update times for PostgreSQL, MongoDB, and PEDaLS are 0.335ms, 0.357ms, and 0.092ms respectively. That is, PEDaLS is 3.64x as fast as PostgreSQL and 3.88x as fast as MongoDB in regard to updates for the application under consideration. Our results further reveal that the average query time for PostgreSQL, MongoDB, and PEDaLS are 8.734ms, 45.184ms, and 1.078ms respectively. That is, PEDaLS is 8.10x as fast as PostgreSQL and 41.91x as fast as MongoDB in regard to queries for the application under consideration.

Several works in the literature suggest that NoSQL databases can outperform SQL databases in the case of a high volume of unstructured data [59], [60]. SQL databases need complex design and multiple writes at different tables for unstructured data, making the updates slower. On a similar note, NoSQL databases can store denormalized data resulting in many cases in a simple single query to retrieve relevant information. However, the data under consideration is structured and the query, although complex, does not involve reading from multiple tables. Hence we do not

TABLE 2
The Mean Update and Query Execution Time for Different End-to-End Applications

	Banking Queue		Room Occupancy		Livestock Tracking	
	update in ms	query in ms	update in ms	query in ms	update in ms	query in ms
PostgreSQL	0.335 (0.026)	8.734 (0.045)	**0.354 (0.039)**	0.460 (0.003)	**0.342 (0.022)**	5.186 (0.141)
MongoDB	0.357 (0.006)	45.184 (0.295)	0.359 (0.007)	2.255 (0.019)	0.355 (0.007)	11.687 (0.222)
PEDaLS	**0.092 (0.002)**	**1.078 (0.022)**	0.468 (0.017)	**0.124 (0.003)**	0.553 (0.014)	**2.333 (0.083)**

The standard deviation is shown in parentheses. The best (fastest) execution time for each type of operation for each application is presented in bold. PEDaLS outperforms the other two storage systems in regard to query execution time for all applications.



(a) Banking queue. (b) Room occupancy. (c) Livestock tracking.

Fig. 7. The mean update and query execution time for different end-to-end applications as presented in Table 2. As the execution time varies in order of magnitude among the different applications, we present the execution time in log scale for a better and uniform visualization.

observe the advantages generally associated with NoSQL databases over SQL databases for the application under consideration. This is supported by the similar update time for both PostgreSQL and MongoDB. However, the query time for MongoDB is higher than PostgreSQL. This is expected, as the MongoDB *aggregation pipeline* [61] (i.e., the pipeline responsible for processing documents in stages such as matching, grouping, projection, etc.) is still relatively immature and thus is not as well performant as the query engine of PostgreSQL. In fact, the aggregation pipeline is not as expressive as SQL queries either, which is apparent from the complex query of Fig. 8. PEDaLS on the other hand is the fastest for both update and query. During update, PEDaLS must either add a node to the end of the queue or delete a node from the head of the queue. As the queue records both the head and the tail, both of the above operations can be performed efficiently. Moreover, none of these operations result in copying of a node, as a node can have at most one child which can be deleted at most once. As a result, the update performance of PEDaLS is better than both PostgreSQL and MongoDB. To answer Q_1, PEDaLS must perform versioned traversal of the queue, which is inherently fast due to the node-copy method. In contrast, both PostgreSQL and MongoDB has to perform complex queries (cf. Fig. 8). Hence PEDaLS outperforms the others in regard to read as well.

6.4.2 Room Occupancy Detection

Indoor positioning systems are used to locate, track, and identify individuals/objects in indoor settings where technologies like GPS cannot perform with desired precision. A common application of such a system is to detect the occupancy of a room at different times of the day. Apart from eliciting information pertinent to security (e.g., was user x present in room y during some event z), it can also provide historical data to aid in scheduling of events depending on the occupancy of different rooms throughout different times of the day.

```
SELECT userid
FROM bankingqueue
WHERE ts <= X
AND op = 1
AND userid NOT IN (
SELECT userid
FROM bankingqueue
WHERE ts <= X
AND op = 0)
```

```
db.bankingqueue.find({
'ts': {'$lte': X},
'op': 1,
'userid': {
'$nin': db.bankingqueue.distinct(
'userid',
{'ts': {'$lte': X}, 'op': 0}
)}},
{ 'userid': 1, '_id': 0 })
```

Fig. 8. SQL (left) and NoSQL (right) queries to find the set of users in the queue at time X. Here ts stands for timestamp and op stands for enqueue/dequeue operation. Note that $distinct$ is not necessary for the NoSQL query to get the correct result as a user is issued a unique ID during each entrance to the queue. However, it is faster than the alternative which involves converting the subquery result into an array and subsequently mapping it to a function that picks out each user ID in the array from the JSON object enclosing it. The $distinct$ function extracts the value from the JSON object automatically and presents an array readily usable with the not in (nin) operator.

SmartBench [62] is a benchmark focusing on queries resulting from (near) real-time applications and longer-term analysis of IoT data. For this experiment, we use the data generation tool of SmartBench to generate room occupancy dataset based on seed data collected from a real system. The generated dataset contains periodic location data including the room in which a user is. However, the data does not contain the explicit timestamp at which a user leaves a room. Hence, we augment the data with information that denote a user has left the old room by inserting a new record with a timestamp that falls between the timestamp when the user was last seen in the old room and the timestamp when the user was first seen in the new room. We assign an invalid room number for such records, which essentially denotes the user is on its way from the old room to the new room. The preprocessed dataset contains 4002 timestamped data points with information regarding the location (i.e., room numbers) of users at different times of the day.

The storage of this data for databases involves the insertion of a single row containing the timestamp, user ID, and the location of the user at that timestamp. For PEDaLS, we maintain a bst with user IDs as the keys and room numbers as the values. We treat the timestamp as the version stamp for the bst. A record in the dataset containing an invalid room number corresponds to a deletion from the bst and an insertion into the bst otherwise. We assume we want to answer the query Q_2: "what is the location of user with ID uid at time X"? PEDaLS bst performs a simple versioned search with X as the timestamp and uid as the key to answer this query. However, PostgreSQL and MongoDB require complex queries as shown in Fig. 9.

Our results show that the average query time for PostgreSQL, MongoDB, and PEDaLS are 0.460ms, 2.255ms, and 0.124ms respectively. That is, PEDaLS is 3.71x as fast as PostgreSQL and 18.19x as fast as MongoDB in regard to queries for the application under consideration. As in the banking queue application of Section 6.4.1, MongoDB does not demonstrate any advantage over PostgreSQL for structured data of the application under consideration. PEDaLS

```
SELECT room
FROM roomoccupancy
WHERE ts <= X AND
userid = uid
ORDER BY ts
DESC LIMIT 1
```

```
db.roomoccupancy.find(
{'ts': { '$lte': X }, 'userid': uid },
{ 'room': 1, '_id': 0 }
).sort('ts', -1).limit(1)
```

Fig. 9. SQL (left) and NoSQL (right) queries to find the room at which user with ID uid is at time X. Here ts stands for timestamp.

must perform a versioned search of a key in the bst to answer Q_2. This is relatively simpler than the complex queries required by PostgreSQL and MongoDB as presented in Fig. 9. Hence PEDaLS outperforms both PostgreSQL and MongoDB in regard to read for this application as well. Our results also indicate that both PostgreSQL (0.354ms) and MongoDB (0.359ms) perform better than PEDaLS (0.468ms) for update operations (approximately 1.32x times). This is expected, as the databases need only to insert a single entry to record an update, whereas PEDaLS has to search the tree for the appropriate position of a new node to be inserted or an old node to be deleted. In contrast, PEDaLS queue for Section 6.4.1 did not require an extra read per update operation to find the existence and/or position of a data structure node. This explains the superior performance of PEDaLS update in the banking queue monitoring application.

6.4.3 Livestock Tracking

Livestock tracking is a popular application of IoT in smart farms to study animal behavior and animal-ecosystem interaction [63], [64]. In a livestock tracking system, each animal is fitted with a tracking device (e.g., a GPS collar) that records the location information at a pre-defined regular interval. This information can then be used to analyze different behavior of the cattle, such as grazing patterns. As the tracking and data processing devices are typically battery-powered in these deployments, we must minimize the number of times a location is recorded/communicated over the network; without losing valuable information.

In [63], the authors provide a dataset containing daytime (approx. 8 hours per day) grazing locations of cattle collected from 6 Alpine farms in the summer of 2011. 2 to 4 cows from each of these farms (in total 15) were equipped with a GPS collar and a logger box. Although data were collected at 20 seconds interval in this work, an analysis of the dataset shows that even a larger interval of 3 minutes shows an average movement (i.e., distance between the locations at the beginning and at the end of the interval) of 12 meters for a cow (cf. Table 3). Depending on the use case, this might be an acceptable distance.

We consider a scenario where a new patch of land B is opened up for grazing beside an old patch of land A. We want to know which are the cows that prefer the new patch of land B, along with at what time of the day. Specifically, we want to answer the query Q_3: "which cows are present in B at time X"? This is an example of a temporal query that can be efficiently answered using a versioned bst. As the dataset in [63] contains data for at most 4 cows from any one farm, we use synthetic data generated using random walk for 100 cows. The one-dimensional random walk starts from 0. Whenever the random variable has a value $>= 0$, we assume the location is in land B, otherwise land A. As in [63], we assume each cow is equipped with a GPS collar.

TABLE 3
Average Distance Between the Locations at the Start and the End of an Interval for an Individual Cow

interval (seconds)	distance (meters)
20	1.62
60	4.31
100	6.81
140	9.22
180	11.56

From Table 3, we see that the point-to-point distance covered by a cow on average is approximately only 12 meters for a 3 minute interval. Hence, for energy-efficiency we assume the GPS collar collects location information at 3 minutes (180 seconds) interval instead of 20 seconds over the duration of 8 hours ($8 * 60/3 = 160$ data points for each cow). We further assume the timestamped location data is sent to a nearby edge server for storage in two methods. In the first method, as common to many IoT deployments, the server stores each location data into a database upon reception. Each data point contains a cattle id and its position at the time the data was collected. Note that in this kind of setup, the collection of data is fast as it involves only a single insertion to a database table. However, retrieving answers to complex queries such as Q_3 can be time-consuming. We present the SQL and NoSQL queries to answer Q_3 in Fig. 10.

In the second method, the server maintains a PEDaLS bst to store the IDs of cows that are in land B using timestamps as the version stamps. Although this requires some preprocessing during insertion to determine whether a cow is already in land B, this makes query Q_3 faster compared to the former method; as we can now perform a tree traversal at version X in the bst to retrieve the full set of cows present in land B. This also removes the onus on part of the developer to write complex queries, as the relevant information is readily available through a versioned access operation (in this case, tree traversal) in PEDaLS bst.

As described above, there are 160 data points for each cow, i.e., 160 unique timestamps. We perform $160 * 100$ insertions and 160 queries (one for each unique timestamp) in one iteration of the experiment. The average update times for PEDaLS and PostgreSQL are 0.553ms and 0.342ms respectively, whereas the average query times are 2.333ms

```
SELECT lt1.cattle_id
FROM livestocktracking AS lt1
JOIN
(SELECT cattle_id, MAX(ts) as
maxts
FROM livestocktracking
WHERE ts <= X
GROUP BY cattle_id) AS lt2
ON lt1.cattle_id = lt2.cattle_id
AND lt1.ts = lt2.maxts
WHERE lt1.pos >= 0
```

```
db.livestocktracking.aggregate([
{'$match': {'ts': {'$lte': X}}},
{'$sort': {'ts': -1}},
{'$group':{
'_id': '$cattle_id',
'maxTS': {'$max': '$ts'},
'currentPos': {'$first': '$pos'}
}},
{'$match': {'currentPos': {'$gte': 0}}},
{'$project': { '_id': 1}}
])
```

Fig. 10. SQL (left) and NoSQL (right) queries to find the set of cows present in plot B at time X. Here ts and pos stand for timestamp and position respectively.

and 5.186ms respectively. That is, although for insertion PostgreSQL is 1.62x as fast as PEDaLS, for query PEDaLS is 2.22x as fast as PostgreSQL. From a power consumption perspective, as long as the number of updates to the number of queries ratio is less than $(5.186 - 2.333)/(0.553 - 0.342) = 13.52$, PEDaLS is more efficient than PostgreSQL. The higher time requirement for PEDaLS update is expected, as every update operation involves searching for a node in the bst as well. On the other hand, the databases only have to insert a single row for an update. However, due to the efficient versioning scheme of PEDaLS bst, answering query Q_3 simply amounts to traversing a particular version of the bst. In contrast, the databases have to perform complex queries as presented in Fig. 10. Hence PEDaLS outperforms both PostgreSQL and MongoDB in regard to read. Although the update time (0.355ms) for MongoDB is comparable to that of PostgreSQL, much like Sections 6.4.1 and 6.4.2, MongoDB requires significantly more time for reads (11.687ms). This is again due to the inefficient aggregation pipeline of MongoDB.

6.5 Bounding Log Size

One challenge with PEDaLS is that the underlying append-only, persistent backing storage abstraction may have a size limit due to host resource constraints or storage service design. For example, CSPOT implements logs of fixed size as a circular buffer [65]. Although this approach provides fast and automatic garbage collection of log entries, if a PEDaLS log "wraps" it may lose entries that it requires to maintain a particular version in the version history. A PEDaLS user specifies the maximum number of data structure versions (i.e., the version history) she wishes to maintain. PEDaLS ensures that at least this length of version history is available.

To do so, PEDaLS provides a probabilistic guarantee that a program never accesses a version for which only partial information is available due to log wrap. It uses the probability that it must maintain a given history length to compute the log sizes that are necessary to ensure that this guarantee is met. In the case where this size is attempted to be exceeded due to an update operation performing a log append, PEDaLS refuses further update operations. This practice of refusing update requests in absence of adequate space is not uncommon (e.g., Redis [66]). However, read requests can still be serviced.

Our failure model assumes that storage failures occur only during log-append operations and these failures take two forms. Either the data is not written at all, or it is written but the sequence number associated with the write is not returned. In both cases, PEDaLS assumes that the failure can occur silently and thus must be remediated by a timeout and a retry. As stated in Section 4, we term the first type of failure as a *Type 1* failure and the second a *Type 2*.

As node copy is most frequent when the number of extra pointers e is one, we consider the required log size for the different logs when $e = 1$. Then for any case where $e > 1$, the required log sizes will not be greater than the ones calculated for $e = 1$. We begin by determining the log sizes assuming a fault-free system and then consider a distributed system with failures. Let K be the maximum number of versions in the version history specified by the PEDaLS user, K_i the number of inserts, and K_d the number of deletions ($K = K_i + K_d$). The APLog which tracks all operations (each one creating a version) thus requires K entries while the DataLog only grows as a result of insert operations and thus it must minimally contain K_i entries. However, worst case, all of the operations are inserts so the DataLog must contain K entries.

We consider the required size of the LinkLog separately for linked list and BST. In the case of linked list insert or delete, PEDaLS requires one new link. As linked list has only one type of pointer, even if PEDaLS must copy a node, it only introduces one new link. Therefore, we require the log size for the LinkLog for linked list to be at most K. Even in the worst-case scenario when we are inserting to and deleting from the end of the same node K times, a LinkLog of size K will not experience rollover and wrap.

In the case of BST insert, PEDaLS adds one new link to a node. However, this might result in node copying. Unlike linked list, for BST, PEDaLS must copy both pointers in a node during node copy. That is, if we are changing the left (or right) pointer, PEDaLS must copy over the old right (or left) pointer apart from introducing the new left (or right) pointer. Thus every second modification of a node forces it to make one extra entry into the LinkLog. Aside from the fact that the first version of a node requires two entries in the LinkLog, the number of links in the LinkLog is at most 1.67 times that of the number of versions K when all operations are inserts. In practice, this factor is much lower. As an example, over 100 iterations of 12579 insertions (different order for each iteration) for the CityPulse temperature dataset used in Section 6.1 revealed the maximum links required by any LinkLog is only 26, i.e., 0.002 times of $K(12579)$.

However, in the case of BST delete, PEDaLS might have to introduce at most two entries to the same node when the deleted node has two children. If node copying is necessary (the additional node pointer is occupied), the total number of new links to be introduced is three. Therefore, assuming the worst-case scenario where every operation is a deletion and where the target node has both of its children present, we can say that the number of entries at each LinkLog will not exceed $3K$. Thus PEDaLS can preserve K versions of BST when the LinkLog is set to $3K$ in the absence of Type 2 log failures.

Now that we have set a baseline for the size of the history of each log in absence of failures, we next consider how this changes with the introduction of our failure modes. We are specifically concerned about Type 2 failures where an entry is appended but the resulting log sequence number is lost (since a Type 1 failure does not grow any log).

Let us assume that a log append fails with probability q and that the probability of any such failure is independent of any other. Given that we need to insert i entries into a log ($i = K$ for APLog, $i = K$ for DataLog, $i = K$ for linked list LinkLog, and $i = 3K$ for BST LinkLog) and we must tolerate at most F failures (i.e., F additional entries are needed in each log), the total number appends that we can make is $i + F$. As an append either fails or succeeds, we can model the probability of failure of f failures in at most $i + F$ appends as a binomial distribution having a probability mass function λ parameterized by the failures f:

TABLE 4
Required Log Size to Keep the Probability of Roll Over Below 0.000001

i	q	$i+F$	overhead
1000	0.1	1168	1.168
	0.2	1339	1.339
	0.3	1553	1.553
	0.4	1834	1.834
	0.5	2223	2.223
2000	0.1	2301	1.151
	0.2	2624	1.312
	0.3	3030	1.515
	0.4	3566	1.783
	0.5	4311	2.155
3000	0.1	3429	1.143
	0.2	3901	1.300
	0.3	4496	1.499
	0.4	5283	1.761
	0.5	6379	2.126

i denotes the required number of successful appends, F denotes the maximum allowable number of unsuccessful appends, and q denotes the probability of failure of an append. The overhead is calculated as $\frac{i+F}{i}$.

$$\lambda(f) = \binom{i+F}{f} q^f (1-q)^{i+F-f} \tag{1}$$

Given that we set the log size to $i+F$, rollover happens only when $f > F$. Thus, the probability of rollover is 1.0 minus the probability that $f \leq F$ expressed as:

$$P_{rollover} = 1 - \sum_{f=0}^{f=F} \lambda(f) \tag{2}$$

Given probability q that any append fails and a history version size i, we can programmatically find the number of additional entries F (in turn the size of a log) required to make the probability of rollover arbitrarily low. As a worst-case example, for an append failure probability $q = 0.5$ and a version history of $K = 1000$ versions, the APLog and DataLog for BST would need to contain 2223 entries each and the LinkLog would require 6379 entries to make the rollover probability for any of these logs < 0.0000001. That is, with a failure probability of 0.5 the APLog, DataLog, and LinkLog are approximately 2.2 times as large as they would be without failures.

This overhead decreases with failure probability. For example, when the append failure probability is $q = 0.1$ this overhead reduces to a factor of 1.2 compared to the failure-free case, and when $q = 0.001$ the overhead factor is 1.009 (9 extra entries in APLog and DataLog and 14 extra entries in the LinkLog). As a practical matter, the failure probability q for any given append is almost certainly well below 0.001 (e.g., we observed no failures in tens of millions of distributed logging events over the several months this paper has been in preparation) making the additional space required to tolerate failures in a real-world setting negligible. Table 4 shows the log size required to keep the probability of rollover below 0.000001 for different values of q.

7 Conclusion and Future Directions

Partially persistent data structures (PDSs) provide data structure immutability by implementing versioning. The original works on PDSs are in-memory only, use mutable data structures internally, and work only on a single system. PEDaLS presents a PDS implementation that is crash-resistant, provides immutability from both storage level (i.e., append-only logs) and software level, and works in a distributed environment. It is the first system to provide versioned program linked data structures in a distributed setting that can facilitate system debugging.

In this work, we extend PEDaLS by introducing data replication, thus ensuring availability and fault tolerance. Our results show that program data structures replicated by PEDaLS are scalable. Moreover, we explore multiple end-to-end applications to demonstrate use cases for PEDaLS data structures. Finally, we show how to determine the maximum needed log size as a function of stored history length. These contributions make PDSs utilitarian in a modern, distributed computing context while preserving their original low algorithmic complexities.

Currently, PEDaLS guarantees convergence of an arbitrary linked data structure with constant in-degree. However, convergence itself does not ensure the satisfaction of the user expectations on the semantics of the underlying data structure, e.g., add-wins set [11], [67] instead of conventional set. A possible future direction is to introduce mechanisms to satisfy such user expected semantics.

Although the current design of PEDaLS allows it to maintain the same space/time complexity of the node-copy method, this design also makes the retention of the latest K versions of the underlying data structure impossible in the face of scarcity of space. This is due to the simple fact that even the Kth version might require information that are present at the beginning of the DataLog or a LinkLog, e.g., an element inserted during the first version is still present during the Kth version. In the future, we can conduct studies to explore a new design that would potentially have to make a trade-off between space/time complexity and the possibility of retaining a specified number of the latest versions.

One possible way to extend the current work is to employ Artificial Intelligence (AI)/ Machine Learning (ML) to discover patterns in the data [68] stored in the underlying data structures. PEDaLS stores historical data and provides programmatic approach to retrieve versioned data. This can be convenient to perform time-series analysis and explore different trends in data.

Similar to trend analysis, PEDaLS can be used in conjunction with AI/ML to predict potential configuration options for autonomic systems [69]. In fact, AI/ML can be used to analyze a typical workload under consideration and to predict parameters of PEDaLS itself, such as the size of a LinkLog (cf. Section 6.5).

As PEDaLS maintains an efficient space/time complexity and stores historical data, it is a good candidate for a storage layer for real-time anomaly detection [68]. While the previous versions of data can help in exploring underlying trends, the efficiency in storing and accessing data can meet the fast response requirements of real-time systems. On a

similar note, AI/ML can be used for efficient adaptive resource scheduling [70] among replicas of PEDaLS. For example, if a resource-constrained replica tends to receive a higher volume of write requests at a particular time of day, it can decide to proactively delegate some work to nearby replica(s).

In general, PEDaLS can be extended and used as an integral part of the monitor, analyze, plan, and execute (MAPE) loop of autonomic systems [68], [71] with the help of AI/ML. As PEDaLS records all updates as separate versions of the underlying data structure, it facilitates monitoring of data. The historical data efficiently preserved by PEDaLS can then be analyzed with the help of AI/ML techniques. The analysis can lead to a possible actionable plan, upon the execution of which we enter the next iteration of MAPE loop.

REFERENCES

[1] D. Lindsay, S. S. Gill, D. Smirnova, and P. Garraghan, "The evolution of distributed computing systems: From fundamental to new frontiers," *Computing*, vol. 103, no. 8, pp. 1859–1878, 2021.

[2] J. Beal, M. Viroli, D. Pianini, and F. Damiani, "Self-adaptation to device distribution in the internet of things," *ACM Trans. Auton. Adaptive Syst.*, vol. 12, no. 3, pp. 1–29, 2017.

[3] J. Park, J. Park, Y. Lee, C.-J. Kim, B. Kim, and S. Ryu, "A framework for dynamic inter-device task dispatch with eventual consistency," in *Proc. Conf. Companion 2nd Int. Conf. Art Sci. Eng. Program.*, 2018, pp. 63–68.

[4] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity Internet of Things devices," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 461–472.

[5] M. Zhang, C. Krintz, and R. Wolski, "STOIC: Serverless teleoperable hybrid cloud for machine learning applications on edge device," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops*, 2020, pp. 1–6.

[6] M. I. Naas, L. Lemarchand, P. Raipin, and J. Boukhobza, "IoT data replication and consistency management in fog computing," *J. Grid Comput.*, vol. 19, no. 3, pp. 1–25, 2021.

[7] N. Golubovic, R. Wolski, C. Krintz, and M. Mock, "Improving the accuracy of outdoor temperature prediction by IoT devices," in *Proc. IEEE Int. Congr. Internet of Things*, 2019, pp. 117–124.

[8] N. Saquib, C. Krintz, and R. Wolski, "PEDaLS: Persisting versioned data structures," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2021, pp. 179–190.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proc. Symp. Self-Stabilizing Syst.*, 2011, pp. 386–400.

[10] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, "Making data structures persistent," *J. Comput. Syst. Sci.*, vol. 38, no. 1, pp. 86–124, 1989.

[11] N. Preguiça, C. Baquero, and M. Shapiro, *Conflict-Free Replicated Data Types CRDTs*. Cham, Switzerland: Springer, 2019, pp. 491–500. [Online]. Available: https://doi.org/10.1007/978-3-319-77525-8_185

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

[13] L. Lamport et al., "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[14] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.

[15] Git. [Online]. Available: https://git-scm.com/

[16] GitHub. Accessed: Oct. 14, 2013. [Online]. Available: https://github.com/

[17] A. A. Monrat, O. Schelén, and K. Andersson, "A survey of blockchain from the perspectives of applications, challenges, and opportunities," *IEEE Access*, vol. 7, pp. 117 134–117 151, 2019.

[18] P. Helland, "Immutability changes everything," in *Proc. Conf. Innov. Data Syst. Res.*, 2015. Accessed: Sep. 15, 2019. [Online]. Available: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf

[19] A. Fiat and H. Kaplan, "Making data structures confluently persistent," in *Proc. Symp. Discrete Algorithms*, 2001, pp. 537–546.

[20] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, no. 1, pp. 153–174, 1987.

[21] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language-based editors," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 3, pp. 449–477, 1983.

[22] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion B-tree," *VLDB J.*, vol. 5, no. 4, pp. 264–275, 1996.

[23] Amazon, "S3 Object Versioning," 2019. Accessed: Sep. 28, 2019. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html

[24] Google Cloud: Versioned Object Storage, 2018. Accessed: Sep. 12, 2018. [Online]. Available: https://cloud.google.com/storage/docs/object-versioning

[25] R. Kotla, L. Alvisi, and M. Dahlin, "SafeStore: A durable and practical storage system," in *Proc. USENIX Annu. Tech. Conf.*, 2007, pp. 129–142.

[26] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.

[27] S. Alsubaiee et al., "Storage management in AsterixDB," *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 841–852, Jun. 2014.

[28] C. Gong, S. He, Y. Gong, and Y. Lei, "On integration of appends and merges in log-structured merge trees," in *Proc. Int. Conf. Parallel Process.*, 2019, pp. 1–10.

[29] A. Twigg, A. Byde, G. Milos, T. Moreton, J. Wilkes, and T. Wilkie, "Stratified B-trees and versioned dictionaries," in *Proc. USENIX Conf. Hot Topics Storage File Syst.*, 2011, Art. no. 10.

[30] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. Davis, "CORFU: A shared log design for flash clusters," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 1–14.

[31] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Chariots: A scalable shared log for data management in multi-datacenter cloud environments," in *Proc. 18th Int. Conf. Extending Database Technol.*, 2015, pp. 13–24.

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

[33] Apache Samza, 2019. Accessed: Sep. 2019. [Online]. Available: http://samza.apache.org

[34] Apache Kafka, 2019. Accessed: Sep. 2019. [Online]. Available: http://kafka.apache.org

[35] Amazon kinesis streams service, 2020. Accessed: Apr. 15, 2020. [Online]. Available: https://docs.aws.amazon.com/kinesis/index.html

[36] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *ACM Queue*, vol. 11, no. 3, pp. 20–32, Mar. 2013.

[37] S. Burckhardt, "Principles of eventual consistency," *Found. Trends Program. Lang.*, vol. 1, no. 1/2, pp. 1–150, 2014.

[38] M. Balakrishnan et al., "Tango: Distributed data structures over a shared log," in *Proc. Symp. Operating Syst. Princ.*, 2013, pp. 325–340.

[39] Amazon S3, 2018. Accessed: Sep. 28, 2018. [Online]. Available: https://aws.amazon.com/s3/

[40] J. Kreps et al., "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1–7.

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

[42] F. Houshmand and M. Lesani, "Hamsaz: Replication coordination analysis and synthesis," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–32, 2019.

[43] X. Li, F. Houshmand, and M. Lesani, "Hampa: Solver-aided recency-aware replication," in *Proc. Int. Conf. Comput. Aided Verification*, 2020, pp. 324–349.

[44] M. Diogo, B. Cabral, and J. Bernardino, "Consistency models of NoSQL databases," *Future Internet*, vol. 11, no. 2, 2019, Art. no. 43.

[45] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating CRDTs for real-time document editing," in *Proc. 11th ACM Symp. Document Eng.*, 2011, pp. 103–112.

[46] X. Lv, F. He, W. Cai, and Y. Cheng, "A string-wise CRDT algorithm for smart and large-scale collaborative editing systems," *Adv. Eng. Informat.*, vol. 33, pp. 397–409, 2017.

[47] W. Yu, L. André, and C.-L. Ignat, "A CRDT supporting selective undo for collaborative text editing," in *Proc. IFIP Int. Conf. Distrib. Appl. Interoperable Syst.*, 2015, pp. 193–206.
[48] A. Charapko, A. Ailijiang, and M. Demirbas, "PigPaxos: Devouring the communication bottlenecks in distributed consensus," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 235–247.
[49] F. Nawab, D. Agrawal, and A. El Abbadi, "DPaxos: Managing data closer to users for low-latency and mobile applications," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 1221–1236.
[50] S. A. Bishop, H. I. Okagbue, P. E. Oguntunde, A. A. Opanuga, and O. Odetunmibi, "Survey dataset on analysis of queues in some selected banks in Ogun state, Nigeria," *Data Brief*, vol. 19, pp. 835–841, 2018.
[51] N. Anzum, S. F. Afroze, and A. Rahman, "Zone-based indoor localization using neural networks: A view from a real testbed," in *Proc. IEEE Int. Conf. Commun.*, 2018, pp. 1–7.
[52] W. Lin et al., "Tracking causal order in AWS lambda applications," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2018, pp. 50–60.
[53] E. D. Demaine, J. Iacono, and S. Langerman, "Retroactive data structures," *ACM Trans. Algorithms*, vol. 3, no. 2, pp. 13–es, May 2007.
[54] W.-T. Lin, F. Bakir, C. Krintz, R. Wolski, and M. Mock, "Data repair for distributed, event-based IoT applications," in *Proc. ACM Int. Conf. Distrib. Event-Based Syst.*, 2019, pp. 139–150.
[55] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: The Works of Leslie Lamport*. San Rafael, CA, USA: Morgan & Claypool, 2019, pp. 179–196.
[56] M. Kleppmann and A. R. Beresford, "A conflict-free replicated JSON datatype," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2733–2746, Oct. 2017.
[57] D. Nurmi et al., "The eucalyptus open-source cloud-computing system," in *Proc. IEEE/ACM 9th Int. Symp. Cluster Comput. Grid*, 2009, pp. 124–131.
[58] CityPulse Smart City Datasets - Datasets. Accessed: Sep. 20, 2022. [Online]. Available: http://iot.ee.surrey.ac.uk:8080/datasets.html
[59] M. Sharma, V. D. Sharma, and M. M. Bundele, "Performance analysis of RDBMS and no SQL databases: PostgreSQL, MongoDB and Neo4j," in *Proc. 3rd Int. Conf. Workshops Recent Adv. Innov. Eng.*, 2018, pp. 1–5.
[60] M.-G. Jung, S.-A. Youn, J. Bae, and Y.-L. Choi, "A study on data input and output performance comparison of MongoDB and PostgreSQL in the big data environment," in *Proc. 8th Int. Conf. Database Theory Appl.*, 2015, pp. 14–17.
[61] G. Harrison and M. Harrison, "Tuning aggregation pipelines," in *MongoDB Performance Tuning*. Berlin, Germany: Springer, 2021, pp. 155–183.
[62] P. Gupta, M. J. Carey, S. Mehrotra, and O. Yus, "SmartBench: A benchmark for data management in smart spaces," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 1807–1820, 2020.
[63] H. Homburger, M. K. Schneider, S. Hilfiker, and A. Lüscher, "Inferring behavioral states of grazing livestock from high-frequency position data alone," *PLoS One*, vol. 9, no. 12, 2014, Art. no. e114522.
[64] K. Zhao and R. Jurdak, "Understanding the spatiotemporal pattern of grazing cattle movement," *Sci. Rep.*, vol. 6, no. 1, pp. 1–8, 2016.
[65] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "CSPOT: Portable, multi-scale functions-as-a-service for IoT," in *Proc. ACM/IEEE 4th Symp. Edge Comput.*, 2019, pp. 236–249.
[66] Redis. [Online]. Available: http://redis.io
[67] N. Saquib, C. Krintz, and R. Wolski, "Ordering operations for generic replicated data types using version trees," in *Proc. 9th Workshop Princ. Pract. Consistency Distrib. Data*, 2022, pp. 39–46.
[68] S. S. Gill and A. Abraham, "AI for next generation computing: Emerging trends and future directions," *Internet of Things*, vol. 19, 2022, Art. no. 100514.
[69] S. S. Gill et al., "Transformative effects of IoT, blockchain and artificial intelligence on cloud computing: Evolution, vision, trends and open challenges," *Internet of Things*, vol. 8, 2019, Art. no. 100118.
[70] H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," *Computing*, vol. 91, no. 1, pp. 43–73, 2011.
[71] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

Nazmus Saquib is currently working toward the PhD degree with the Department of Computer Science, University of California, Santa Barbara. His current research work in IoT includes versioned data, data replication, and data provenance. Specifically, he studies similarities in concepts from these seemingly disparate aspects of data and looks at a unified model that explores all three in an IoT context.

Chandra Krintz is a professor with the Computer Science Department, UC Santa Barbara. Her research area is programming and distributed systems. Her research collaborations focus on advances that target the intersection of IoT, edge, and cloud computing, and data analytics, with applications in farming, ranching, climate change adaptation, and conservation science.

Rich Wolski received the MS and PhD degrees from UC Davis. He is the Duval Family presidential chair in energy efficiency and a professor with the Computer Science Department, UCSB. His research interests include cloud infrastructures and scientific computing, and he is the progenitor of the open source cloud infrastructure-as-a-service, Eucalyptus.