ForkBase: An Efficient Storage Engine for Blockchain and Forkable Applications

Sheng Wang #, Tien Tuan Anh Dinh #, Qian Lin #, Zhongle Xie #, Meihui Zhang †, Qingchao Cai #, Gang Chen §, Wanzeng Fu #, Beng Chin Ooi #, Pingcheng Ruan #

# *National University of Singapore,* † *Beijing Institute of Technology,* § *Zhejiang University*

# {wangsh,dinhtta,linqian,zhongle,caiqc,fuwanzeng,ooibc,ruanpc}@comp.nus.edu.sg,

† meihui_zhang@bit.edu.cn, § cg@zju.edu.cn

ABSTRACT

Existing data storage systems offer a wide range of functionalities to accommodate an equally diverse range of applications. However, new classes of applications have emerged, e.g., blockchain and collaborative analytics, featuring data versioning, fork semantics, tamper-evidence or any combination thereof. They present new opportunities for storage systems to efficiently support such applications by embedding the above requirements into the storage.

In this paper, we present *ForkBase*, a storage engine specifically designed to provide efficient support for blockchain and forkable applications. By integrating the core application properties into the storage, *ForkBase* not only delivers high performance but also reduces development effort. Data in *ForkBase* is multi-versioned, and each version uniquely identifies the data content and its history. Two variants of fork semantics are supported in *ForkBase* to facilitate any collaboration workflows. A novel index structure is introduced to efficiently identify and eliminate duplicate content across data objects. Consequently, *ForkBase* is not only efficient in performance, but also in space requirement. We demonstrate the performance of *ForkBase* using three applications: a blockchain platform, a wiki engine and a collaborative analytics application. We conduct extensive experimental evaluation of these applications against respective state-of-the-art system. The results show that *ForkBase* achieves superior performance while significantly lowering the development cost.

1. INTRODUCTION

Designing a new application today is made easier by the availability of many storage systems that have different data models and operation semantics. At one extreme, key-value stores [21, 33, 9, 31] provide a simple data model and semantics, but they are highly scalable. At the other extreme, relational databases [47] support more complex, relational models and strong semantics, i.e. ACID, which render them less scalable. In between are systems that make other tradeoffs between data model, semantics and performance [15, 19, 14, 7]. Despite these many choices, we observe that there emerges a gap between modern applications' requirements and what existing storage systems have to offer.

Many classes of modern applications demand properties (or features) that are not a natural fit to current storage systems. First, blockchain systems such as Bitcoin [39], Ethereum [2] and Hyperledger [5], implement a distributed ledger abstraction — a globally consistent history of changes made to some global states. Changes are bundled into blocks, each of which represents a new version of the states. Because blockchain systems operate in an unstrusted environment, they require the ledger to be *tamper evident*, i.e. the states and their histories cannot be changed without being detected. Second, collaborative applications, ranging from traditional platforms like Dropbox [25], GoogleDocs [4], and Github [3] to more recent and advanced analytics platforms like Datahub [35] allow many users to work together on the same data. Such applications need explicit *data versioning* to track data derivation history, and *fork semantics* to let users work on independent copies of the data. Besides, cryptocurrencies, the most popular blockchain applications, also allow for temporary forks in the chains.

Without well-designed storage support for data versioning, fork semantics and tamper evidence, the applications have to either build on top of systems with partial or no support for these properties, or roll out their own implementations from scratch. Both approaches raise development costs and latency, and the former may fail to generalize and may introduce unnecessary performance overhead. One example is that current blockchain platforms (e.g., Ethereum and Hyperledger) build their data structures on top of a key-value store (e.g., LevelDB [6] or RocksDB [10]). The implementations provide tamper evidence, but we observe that they do not always scale well. More importantly, they are not suitable for efficient analytical query processing. Another example is collaborative applications over large, relational datasets, which can be implemented over file-based version control systems such as `git`. However, such implementations do not scale with large datasets, nor do they support rich query processing.

Clearly, there are benefits in unifying these properties and pushing them down into the storage layer. One direct benefit is that it reduces development efforts for applications requiring any combination of these features. Another benefit is that it helps applications generalize better by providing additional features, such as efficient historical queries, at no extra cost. Finally, the storage engine can exploit performance optimization that is hard to achieve at the application layer.

We propose a novel and efficient storage engine, called

ForkBase, that is designed to meet the high demand in modern applications for versioning, forking and tamper evidence[1]. One challenge in realizing this goal is to keep the storage and computation overhead small when maintaining a large number of data versions. Another challenge is to provide small, yet flexible and powerful APIs to various applications. Our approach follows well-proven database and system design principles, and adopts novel designs. First, a version number uniquely identifies the data content and its history, which can be used to quickly retrieve and verify integrity of the data. Second, large objects are split into *data chunks* and organized in a novel index structure, called `POS-Tree`, that combines the concepts of content-based slicing [38], Merkle tree [36] and B$^+$-tree [18]. This structure facilitates efficient identification and removal of duplicate chunks across objects, which drastically reduces storage overhead especially for incremental data. Third, general fork semantics is supported, providing the flexibility to fork data either implicitly or explicitly. The `POS-Tree` supports copy-on-write during forking to eliminate unnecessary copies. Forth, *ForkBase* offers simple APIs, together with many structured data types, which help to reduce development effort and induce a large trade-off space between query performance and storage efficiency. Finally, *ForkBase* scales well to many nodes because of a two-layer partitioning scheme which distributes data evenly, even when the workloads are skew.

ForkBase shares some similar goals with recent dataset management systems, namely Decibel [35], DEX [16] and OrpheusDB [50]. However, there are two fundamental distinctions. First, the other works target relational datasets. *ForkBase*'s data model is less structured and therefore more flexible. Second, the other works are designed mainly for collaborative applications, thus they focus on explicit data versioning and fork semantics. *ForkBase* additionally supports tamper evidence and general fork semantics, making it useful for blockchain and other forkable applications.

To demonstrate the values of our design, we implement three representative applications on top of *ForkBase*, namely a blockchain platform, a wiki service, and a collaborative analytics application. We observe that only hundreds of lines of code are required to port major components of these applications onto our system. The applications benefit much from the features offered by the storage, especially the fork semantics for collaborations and tamper evidence for blockchain. Moreover, as richer semantics are captured in the storage layer, it is feasible to provide efficient query processing. In particular, *ForkBase* enables fast provenance tracking for blockchain without scanning the whole chain, rendering the blockchain analytics-ready.

In summary, we make the following contributions:

- We identify common properties in modern applications, i.e., versioning, fork semantics and tamper evidence, that are not well addressed and supported by existing storage systems. We examine the benefits of a storage that integrates all of these properties.

- We design and implement *ForkBase* with the above properties. It supports generic fork semantics and exposes simple yet elegant APIs. We propose an index structure for managing large objects, called `POS-Tree`, which is tamper-evident and reduces storage overhead for multiple versions of an object.

- We demonstrate the usability and efficiency of *ForkBase* by implementing three representative and complex applications, namely a blockchain platform, a wiki service and a collaborative analytics application. We evaluate the performance of *ForkBase* and the three applications against their respective state-of-the-art implementations. We show that *ForkBase* improves these applications in terms of coding complexity, storage overhead and query efficiency.

In the following, we first motivate the design of *ForkBase* in Section 2. We then present its data model and APIs in Section 3, and the detailed design and implementation in Section 4. We describe the implementation and evaluation of three applications in Section 5 and 6 respectively, and conclude in Section 7.

2. BACKGROUND AND MOTIVATIONS

In this section, we discuss three properties underpinning many modern applications and related systems that are related to *ForkBase*.

2.1 Data Versioning

Data versioning is an important concept in applications that keep track of data evolution history, in which any update made on the data results in a new copy (or version). The history can be either linear or non-linear. Systems supporting linear data versioning include multi-versioned file systems [43, 48, 46] and temporal databases (e.g., relational [11, 42, 49], graph [32] and array [44]). Non-linear data versioning systems can support file-based, unstructured data such as version control systems (e.g., `git`, `svn` and `mercurial`), or more structured data such as Decibel [35], DEX [16] and OrpheusDB [50]. Blockchain is another example of data versioning systems, in which each block represents a version of the global states.

One challenge in the support of data versioning is to reduce storage consumption. The most common approach, called *delta-based deduplication*, is to store only the differences (or deltas) across data versions, and reconstruct the content from a chain of deltas. Decibel proposes several physical data layouts for storing deltas, while OrpheusDB bolts on a relational database in order to take advantage of the latter's query functionalities. The trade-off between storage and reconstruction cost has been studied in [13]. Another approach to storage reduction is *content-based deduplication*. File systems [41] and `git`, for examples, adopt this approach to eliminate coarse-grained duplicates. In particular, data is split into files or chunks, each of which is uniquely identified by its content. The systems then detect and eliminate all duplicates. We note that both deduplciation techniques can be combined. For example, `git` employs content-based technique at the file level, and delta-based technique for linked versions during the `repack` process.

ForkBase applies content-based deduplication at the chunk level. Compared to similar deduplication technique in file systems which uses large chunk sizes and treats the file content as unstructured data, *ForkBase* uses smaller chunks and splits the data based on its structure. For instance, a list object containing multiple elements is only split at

[1]This is the second version of UStore [22], which has evolved significantly from the initial design and system.

element boundaries, thus avoiding the need to reconstruct an element from multiple chunks. Taking the structure of data object into account makes updates and dedpulciations more efficient. Noms [8] applies chunk-level deduplication similar to *ForkBase*. However, it targets single storage instance with fast synchronization, whereas *ForkBase* applies deduplication over multiple storage instances to optimize for large-volume data accesses and modifications. Compared to the delta-based technique used in Decibel to remove duplicates within a dataset, *ForkBase* achieves better storage reduction because it can also eliminate cross-dataset duplicates generated by uncoordinated teams[2]. Furthermore, *ForkBase* offers richer branch management (discussed below) to support more diverse collaborative workflows.

2.2 Fork Semantics

Fork semantics elegantly captures non-linearity of the data evolution history. It consists of two core operations: fork and conflict resolution. A fork operation creates a new data *branch*, which evolves independently and its local modifications are isolated from other branches'. Data forks isolate conflicted updates which can then be merged via the conflict resolution operation. Applications exploiting this semantics can be divided into two categories: one that invokes *on-demand* (or explicit) forks and the other that relies on *on-conflict* (or implicit) forks.

On-demand forks are found in applications that have explicit demand for isolated (or private) branches. Source code version control systems like `git` allow users to fork a new branch for their own development and only merge changes to the main codebase after they are well tested. Similarly, collaborative analytics applications such as Datahub [12] allow branching off from an original dataset before applying transformation to the data, e.g., data cleansing, correction and integration. On-conflict forks are used in applications that implicitly fork a state upon concurrent modifications of the same data. Transactional systems with weak consistency, e.g. TARDiS [20], fork the database state during the concurrent execution of conflicting transactions, and delay (user-defined) conflict resolution. In blockchain applications, for instance Bitcoin [39] and Ethereum [2], forks arise when multiple blocks are appended at the same time to an old block. They are resolved by taking the longest chain or by more complex mechanisms like GHOST [45].

ForkBase is thus motivated to be the first system to natively support both implicit and explicit forks. To facilitate application development, it also provides a number of built-in conflict resolution operations.

2.3 Tamper Evidence

Security conscious applications demand protection against malicious modifications, not only from external attackers but also from malicious insiders. One example is outsourced services like storage [29] or file system [34], which provide mechanisms to detect tampering (forking) of the update logs. Another example is blockchain platforms [39, 31, 2], which require tamper evidence for the ledger. The blockchain combines the tamper-evident ledger with a distributed consensus protocol to ensure that the global states are immutable across multiple nodes. We note that there is an increasing demand for performing analytics on blockchain data [37, 1, 30], which existing blockchain storage engines were not designed for. Specifically, current systems implement the ledger on top of a key-value storage. Further, they focus on tamper evidence and do not efficiently support querying the states' history.

[2]Like any other content-based techniques, the deduplication is less effective than delta-based techniques when the deltas are much smaller than the chunks.

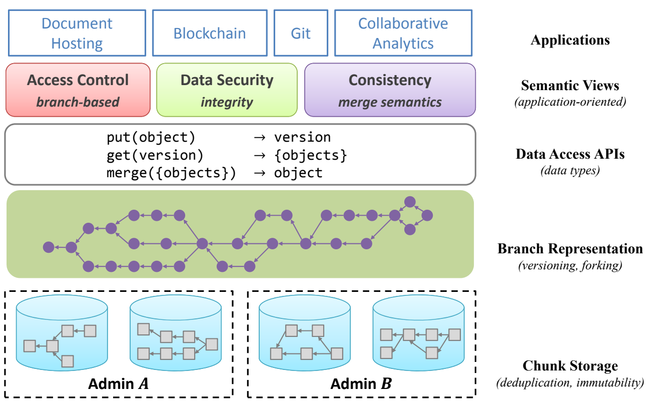


Figure 1: The *ForkBase* stack offers advanced features to various classes of modern applications.

ForkBase provides tamper evidence against malicious storage providers. Given a version number, the application can fetch the corresponding data from the storage provider and verify whether the content and its history have been changed. All data objects in *ForkBase* are tamper-evident, and hence can be leveraged to build better data models for blockchain. In particular, the blockchain's key data structures implemented on top of *ForkBase* are now easy to maintain without incurring any performance overhead. Furthermore, the richer structured information captured in *ForkBase* makes the blockchain analytics-ready.

2.4 Design Overview

Figure 1 shows the *ForkBase*'s stack, illustrating how the storage unifies the common properties and adds values to modern applications. At the bottom layer, data is chunked and deduplicated. At the representation layer, versions and branches are organized in such a way that enables tamper evidence and efficient tracking of the version history. The next layer exposes APIs that combine general fork semantics and structured data types. Other features such as access control and customized merge functions, can be added to the view layer to further enrich the top-layer applications.

3. DATA MODEL AND APIS

In this section, we describe the data model and basic operations, and show an example on how to leverage core features from provided APIs.

3.1 FObject

ForkBase extends the basic key-value data model: each data object in *ForkBase* is identified by a *key*, and contains a *value* of a specific *type*. For each key, it is possible to retrieve not only its latest value, but also its evolution history. Similar to other data versioning systems, *ForkBase* organizes versions in a directed acyclic graph (DAG) called

```
struct FObject {
  enum type;  // object type
  byte[] key;  // object key
  byte[] data;  // object value
  int depth;  // distance to the first version
  vector<uid> bases;  // versions it derives from
  byte[] context;  // reserved for application
}
```

Figure 2: The FObject structure.

object derivation graph. Each node in the graph is a structure called `FObject`, and it is associated with a unique identifier `uid`. Links between `FObjects` represent their derivation relationships.

The structure of a `FObject` is shown in Figure 2. The `context` field is reserved for application metadata, for examples commit message in git, or nonces value for blockchain proof-of-work [26]. Access to a `FObject` is via the `Put` and `Get` APIs listed in Table 1. In particular:

- `Put(key, <branch>, value)` - write a new value to the specified branch. When `branch` is absent, write to the *default branch*.
- `Get(key, <branch>)` - read the latest value from the specified branch. When `branch` is absent, read from the *default branch*.

It can be seen that the *ForkBase*'s data model is compliant to the basic key-value model when only the default branch is used.

3.2 Tamper-Evident Version

Each `FObject` is associated with a `uid` representing the data version. An important property of the `uid` is that it is tamper-evident. The `uid` uniquely identifies both the object value and its derivation history. Two `FObjects` are considered logically equivalent, i.e. having the same `uid`, only when they have the same value and derivation history. Suppose the application is given v_l as the latest version of an object, let $V = \langle v_1, v_2, .., v_l \rangle$ be the derivation history. The storage cannot prove to the application that a version $v' \notin V$ is part of the object history. In other words, the storage cannot tamper with the object value and its history.

ForkBase realizes this property by linking versions via a cryptographic hash chain. In particular, each `FObject` stores the hashes of the previous versions it derives from in the `bases` field. Two important operations on versions are supported, namely `Diff` and `LCA`. The former returns the differences between two `FObjects` of the same types (they could be of different keys). The latter returns the least common ancestor of two `FObjects` with the same key.

3.3 Fork and Merge

The latest version of a branch is called the branch *head*. A branch is only modifiable at the head. However, to change a historical version, a new branch can be created (forked out) at that version to make it modifiable. There are no restrictions on the number of branches per key. *ForkBase* generalizes fork operations by providing two fork semantics: *fork on demand* (FoD) and *fork on conflict* (FoC). Table 1 details the semantics for the basic operations supported in *ForkBase*.

Table 1: *ForkBase* APIs.

	Method	FoD	FoC	Ref
`Get`	`Get(key,branch)`	✓		M1
	`Get(key,uid)`	✓	✓	M2
`Put`	`Put(key,branch,value)`	✓		M3
	`Put(key,base_uid,value)`		✓	M4
`Merge`	`Merge(key,tgt_brh,ref_brh)`	✓		M5
	`Merge(key,tgt_brh,ref_uid)`	✓		M6
	`Merge(key,ref_uid1,...)`		✓	M7
`View`	`ListKeys()`	✓	✓	M8
	`ListTaggedBranches(key)`	✓		M9
	`ListUntaggedBranches(key)`		✓	M10
`Fork`	`Fork(key,ref_brh,new_brh)`	✓		M11
	`Fork(key,ref_uid,new_brh)`	✓		M12
	`Rename(key,tgt_brh,new_brh)`	✓		M13
	`Remove(key,tgt_brh)`	✓		M14
`Track`	`Track(key,branch,dist_rng)`	✓		M15
	`Track(key,uid,dist_rng)`		✓	M16
	`LCA(key,uid1,uid2)`	✓	✓	M17

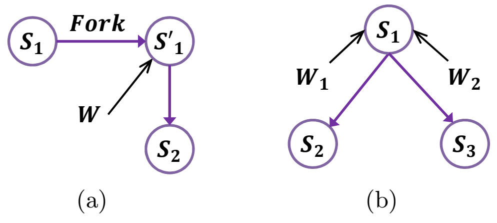


Figure 3: Generic fork semantics supported for both (a) fork on demand and (b) fork on conflict.

3.3.1 Fork on Demand

A branch is created explicitly before any modifications. For example, in Figure 3(a) a branch with head version S_1 is forked to create a new branch. Then a new update W is applied to the new branch creating a new version S_2. S_2 is now the head of an independent branch. Branches generated in this way require user-defined names, and thus we refer to them as *tagged* branches. The `Fork` operation creates a new tagged branch by taking as input an existing branch (M11 in Table 1), or a non-head `FObject` of the existing branch (M12). (M9) lists all branch names and their head `uids`. (M1) and (M3) allow for reading and modifying the head version of a given branch. Non-head versions can be read using (M15).

3.3.2 Fork on Conflict

Branches are implicitly created from concurrent, conflicting `Put` (M4) operations in which different changes are made to the same version. For example, in Figure 3(b) two updates W_1 and W_2 are applied to the head version S_1 concurrently. This is common in decentralized environments where concurrent updates from remote users may not be immediately visible. The result is that two branches with different head versions S_2 and S_3 are created. Branches generated in this way can only be identified by their `uids`, and thus we refer to them as *untagged* branches. Conflicting branches can be checked using (M10) which returns a single head version if no conflict is found. Otherwise, all conflicting head versions are returned, with which the application can decide

```
ForkBaseConnector db;
// Put a blob to the default master branch
Blob blob {"my value"};
db.Put("my key", blob);
// Fork to a new branch
db.Fork("my key", "master", "new branch");

// Get the blob
FObject value = db.Get("my key", "new branch");
if (value.type() != Blob)
  throw TypeNotMatchError;
blob = value.Blob();

// Remove 10 bytes from beginning and append new
// Changes are buffered in client
blob.Remove(0, 10);
blob.Append("some more");
// Commit changes to that branch
db.Put("my key", "new branch", blob);
```

Figure 4: Fork and modify a Blob object.

when and how the conflicts should be resolved.

3.3.3 *Merge*

A tagged branch can be merged with another tagged branch (M5) or with a specific version (M6). In both cases, only the first branch's head is updated such that the new head contains data from both branches. A collection of untagged branches can be merged using (M7), after which the input branches are logically replaced with a new branch. When conflicts are detected during a merge, the application can resolve them in many ways (Section 4.5). To simplify the merge process, *ForkBase* provides type-specific merge functions for the built-in data types.

3.4 Data Type

ForkBase provides many built-in, structured data types. They can be categorized into two classes: *primitive* types and *chunkable* types.

Primitive types include simple types such as `String`, `Tuple` and `Integer`. They are small-size objects that are optimized for fast access. These objects are not deduplicated, since the benefits of sharing small data does not offset the extra overheads of deduplication. Apart from the basic `Get` and `Set` operations, type-specific operations are provided to modify primitive objects. Examples include `Append` and `Insert` for `String` and `Tuple` types, and `Add` and `Multiply` for numerical types.

Chunkable types are complex data structures, for examples `Blob`, `List`, `Map` and `Set`. A chunkable object is stored as a `POS-Tree` and deduplicated (Section 4.3). The chunkable object is most suitable to represent data that grows fairly large due to many updates, but each update touches only a small portion of the data. In other words, a new version has significant overlap with the previous version. The read operation returns only a handler, while the actual data is fetched gradually on demand. Iterator interfaces are provided to efficiently traverse large objects.

The rich collection of built-in types makes it easy to build high level structures, such as relational tables (Section 5). Note that different data types may have similar logical representation but different performance, for example `String` and `Blob`, or `Tuple` and `List`. The application is able to choose those types are more suitable based on their expected workloads.

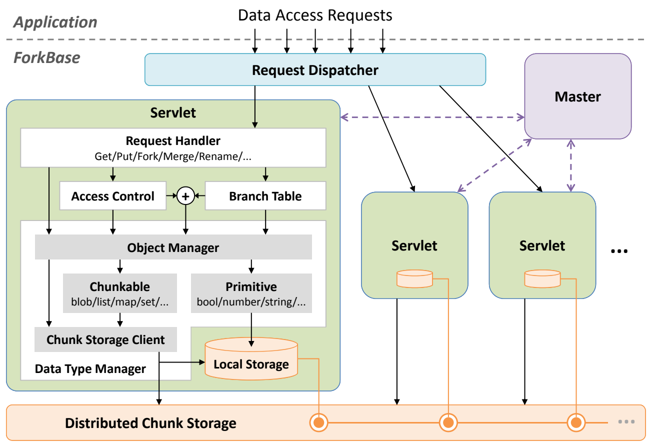


Figure 5: Architecture of a *ForkBase* cluster.

3.5 Example

In *ForkBase*, data can be manipulated at two granularities, i.e., at an individual object, and at a branch of objects. *ForkBase* exposes easy-to-use interfaces that combine both object manipulation and branch management. Figure 4 shows an example of forking and editing a `Blob` object. Since `Put` serves for both insertion and update, the *value* input to the `Put` operation could be either a whole new object or the base object that has undergone a sequence of updates. When multiple updates of the same object are batched, *ForkBase* only retains the final version.

4. DESIGN AND IMPLEMENTATION

In this section, we present the detailed design and implementation of *ForkBase*. The system can used as an embedded storage or run as a distributed service.

4.1 Architecture

Figure 5 shows the stack of a *ForkBase* cluster consisting of four main components: a *master*, a *dispatcher*, a *servlet* and a *chunk storage*. When used as an embedded storage, only one servlet and one chunk storage are instantiated. The *servlet* executes requests using three sub-modules. First, the *access controller* verifies request permission before execution. Second, the *branch table* maintains branch heads for both tagged and untagged branches. Third, the *object manager* handles object manipulations, hiding the physical data representation from the main execution logic. The *chunk storage* persists and provides access to data chunks. When deployed as a distributed service, the *master* maintains the cluster runtime information, while the *request dispatcher* receives and forwards requests to the corresponding servlets. Each servlet manages a disjoint subset of the key space, as determined by a routing policy. All chunk storage instances form a large pool of storage, which is accessible by any remote servlets. In fact, each servlet is co-located with a local chunk storage which enables fast data access and persistence.

4.2 Physical Data Representation

ForkBase objects are stored in the form of data chunks of various lengths. A small and simple object, i.e. of primitive types, contains a single chunk. A large and complex object,

Table 2: Chunk Content.

Type	Content
`Meta`	metadata for an `FObject`
`UIndex`	index entries for unsorted types (`Blob`, `List`)
`SIndex`	index entries for sorted types (`Set`, `Map`)
`Blob`	a sequence of raw bytes
`List`	a sequence of elements
`Set`	a sequence of sorted elements
`Map`	a sequence of sorted key-value pairs

i.e. of chunkable types, comprises multiple chunks organized as a `POS-Tree`.

4.2.1 *Chunk and cid*

A *chunk* is the basic unit of storage in *ForkBase*. There are multiple chunk types (Table 2), each corresponding to a chunkable data type. A chunk is uniquely identified by its `cid` which is computed from the content contained in the chunk:

$$chunk.\texttt{cid} = H(chunk.bytes)$$

where H is a cryptographic hash function (e.g., SHA-256, MD5) taking raw bytes of a chunk as input. Due to the property of cryptographic hashes, each chunk will have a unique `cid`, i.e., chunks with the same `cid` should contain identical content. *ForkBase* uses SHA-256 as the default hash function, but faster alternatives, e.g., BLAKE2, can also be used to reduce computational overhead. The chunks are stored in chunk storage and can be accessed via `cid`s.

4.2.2 *FObject and Data Types*

Recall that a `Get` request returns a `FObject`, whose layout is shown in Figure 2. A serialized chunk of a `FObject` is called a *meta* chunk. The `FObject`'s `uid` is in fact an alias for the meta chunk's `cid`. For a `FObject` of primitive type, the chunk content is embedded in the meta chunk's `data` field to facilitate fast access. For chunkable type object, its meta chunk only contains a `cid` in the `data` field, which points to an external data structure, i.e. the `POS-Tree`. As a result, updates to a chunkable object only modify the `cid` value in the `FObject` structure.

4.3 Pattern-Oriented-Split Tree

Large structured objects are not usually accessed in their entirety. Instead, they require fine-grained access, such as element look-up, range query and update. These access patterns require index structures, e.g., B^+-tree, to be efficient. However, existing index structures are not suitable in our context that has many versions and the versions can be merged. For example, B^+-trees and variants that support branches [28], use capacity-based splitting strategies, and their structures are determined by the values being indexed and by the insertion order. For example, inserting value A followed by B may result in a different B$^+$-tree to inserting B followed by A. There are two consequences when maintaining many versions. First, it is difficult to share (i.e., deduplicate) both index and leaf nodes even when two trees contain the same elements. Second, it is costly to find the differences between two versions and merge them, because of the structural differences. One simple solution is to have fixed-size nodes, which eliminates the effect from insertion order. However, such an approach introduces another issue,

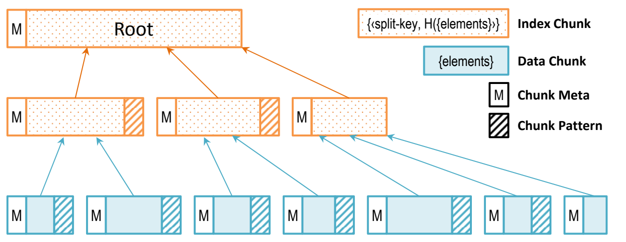


Figure 6: Pattern-Oriented-Splitting Tree (POS-tree) resembling a B$^+$-tree and Merkle tree.

called boundary-shifting problem [27], when an insertion occurs in the middle of the structure.

To solve above issues, we propose a novel index structure, called *Pattern-Oriented-Split Tree* (`POS-Tree`), which has the following properties:

- It is fast to look up and update elements;
- It is fast to find differences and merge two trees;
- It is effective in deduplication;
- It provides tamper evidence.

This structure is inspired by content-based slicing [38], and resembles a combination of a B$^+$-tree and a Merkle tree [36]. In `POS-Tree`, the node (i.e., chunk) boundary is defined as patterns detected from the object content. Specifically, to construct a node, we scan from the beginning until a predefined pattern occurs, and then create a new node to hold the scanned content. Because the leaf nodes and internal nodes have different degrees of randomness, we define different patterns for them. In following, we first describe the basic tree structure, and then discuss how it is constructed.

4.3.1 *Tree Structure*

Figure 6 illustrates the structure of a `POS-Tree`. Each node is stored as a chunk. Index nodes are stored as `UIndex` or `SIndex` chunks, whereas leaf node chunks are of the object types, such as `Blob`, `List` or `Map` chunks, as listed in Table 2. Similar to B$^+$-trees, an index node contains a number of entries for its child nodes. Each entry consists of a child node's `cid` and the corresponding split key (for `SIndex` or element count for `UIndex`). To look up a specific key (or a position for `UIndex`), we adopt the same strategy as in B$^+$-trees, i.e. follow the path guided by the split keys. Therefore, accessing a chunkable object is efficient because only the relevant nodes are fetched instead of the entire tree. `POS-Tree` is a Merkle tree in the sense that the child nodes' `cid`s are cryptographic hashes of their content. Hence, two objects with identical data will have the same `POS-Tree`. In addition, comparing two trees can be done efficiently by recursively comparing the `cid`s.

4.3.2 *Splitting a Leaf Node*

Here we describe the pattern used to split a leaf node. Given a k-byte sequence $(b_1, .., b_k)$, let P be a function taking k bytes as input and returning a pseudo-random integer of at least q bits. The pattern is said to occur if and only if:

$$P(b_1...b_k) \text{ BITWISE\_AND } (2^q - 1) = 0$$

In other words, the pattern occurs when the function P returns 0 for the q least significant bits. The complexity of

Algorithm 1: `POS-Tree` Construction

```
Input: a list of data elements $data$
Output: cid of constructed POS-Tree's root
PatternDetector $hash$;
List<Element> $elements$, $new\_entries$;
Chunk $chunk$;
cid $last\_commit$;
$new\_entries$ = $data$;
/* use pattern function P for leaf nodes      */
$hash$ = new P();
do
    move all elements in $new\_entries$ to $elements$;
    for each $e$ in $elements$ do
        $chunk$.append($e$);
        feed $e$ into $hash$ to detect pattern;
        if detected pattern or exceeded max length then
            $last\_commit$ = $chunk$.commit();
            add index entry of $chunk$ into $new\_entries$;
    /* last chunk may not have pattern        */
    if $chunk$ is not empty then
        $last\_commit$ = $chunk$.commit();
        add index entry of $chunk$ into $new\_entries$;
    /* use pattern function P' for index nodes
       */
    $hash$ = new P'();
    /* loop until root is found               */
while $new\_entries.size() > 1$;
return $last\_commit$;
```

P is at least $O(k)$. We use a special class of hash function, called rolling hash, that supports fast computation over sequence windows (e.g., Rabin-Karp, cyclic polynomial, and moving sum). In particular, we implement P as the *cyclic polynomial* [17] rolling hash function, which is of the form:

$$P(b_1...b_k) = s^{k-1}(h(b_1)) \oplus s^{k-2}(h(b_2)) \oplus ... \oplus s^0(h(b_k))$$

where $\oplus$ is exclusive-or operator, and h maps a byte to an integer in $[0, 2^q)$. s is a function that shifts its input by 1 bit to the left, and then pushes the q-th bit back to the lowest position. This function can be computed recursively as follows:

$$P(b_1...b_k) = s(P(b_0...b_{k-1})) \oplus s^k(h(b_0)) \oplus s^0(h(b_k))$$

Each time, it removes the oldest byte from the active set and adds the new one. As a result, the computation cost amortizes with many sequence windows of k bytes.

Initially, the entire object content is treated as one byte sequence. The pattern detection process scans this sequence from the beginning. When a pattern occurs, a leaf node is created with the recently scanned data. If a pattern occurs in the middle of an element (e.g., a key-value pair in `Map`), the chunk boundary is extended to cover the whole element, so that no elements are stored in more than one chunks. In this way, each leaf node (except for the last node) ends with a pattern, as shown in Figure 6.

4.3.3 *Splitting an Index Node*

The rolling hash function used for splitting leaf nodes has good randomness which keeps the structure balanced against skewed application data. However, we observe that it is costly: it accounts for 20% of the cost for building `POS-Tree`s. Thus, for index nodes, we use a more efficient function P' that exploits the randomness of the leaf nodes' `cid`s. In particular, P' directly takes the `cid` and determines that a pattern occurs when:

$$\texttt{cid} \text{ BITWISE\_AND } (2^r - 1) = 0$$

When a pattern is detected, all scanned index entries are stored in a new index node. This process is repeated for upper layers until reaching the root node. Algorithm 1 demonstrates the bottom-up construction of a new `POS-Tree`.

When updating an existing `POS-Tree`, only affected nodes are reconstructed, which results in a copy-on-write strategy. A node splits when a new pattern is found in between. When an existing boundary pattern changes, the next node needs to be merged. However, no subsequent chunks are involved during the reconstruction, because the boundary pattern of the last merged chunk is preserved. Since P' limits the pattern inside a single index entry, it reduces the chance that existing patterns are changed (compared to using P where k is larger than the `cid` length).

The expected node size (i.e., chunk size) can be configured by setting the values of q and r. By default, *ForkBase* applies a pre-defined chunk size (e.g., 4 KB) for all nodes, but it is beneficial to configure type-specific chunk sizes. For example, `Blob` chunks storing multimedia data can have large sizes, whereas `Index` chunks may need smaller sizes since they only contain tiny index entries. To ensure that a node will not grow infinitely large, an additional constraint is enforced: the chunk size cannot be α times bigger than the average size; otherwise it is forcefully chunked. Therefore, the probability of forced chunking is equal to $(1/e)^\alpha$, which can be set very low (e.g 0.0335% when $\alpha = 8$).

We note that `POS-Tree` is not designed for cases in which the object content is a sequence of repeated items (or bytes). Since there is no pattern in the leaf nodes, all nodes have the maximum chunk size. Consequently, an insertion in the middle leads to boundary shift, thus incurring overhead for re-splitting the node. Nevertheless, `POS-Tree` remains deduplicatable, because the leaf nodes are identical and can be deduplicated.

4.4 Chunk Storage

The *Chunk storage* manages data chunks and exposes a key-value interface. The key is a `cid`, while the value is that chunk of raw bytes. With tamper evidence at the chunk level, the chunk storage can verify `Get-Chunk` and `Put-Chunk` requests on demand. When `Put-Chunk` request contains an existing `cid`, the storage can respond immediately. This is thanks to the deduplication mechanism such that the same chunk from previous request can be reused. Since chunks are immutable, a log-structured layout is used for persistence, which also facilitates fast retrieval of consecutively generated chunks in a `POS-Tree`. To improve data durability and fault tolerance, chunks can be replicated over multiple nodes (or chunk storage instances). Such replication does not significantly affect the deduplication; there are only k copies of any chunk in the storage. Furthermore, replicas help reduce the latency of data access, e.g., by placing a replica on the servlet that frequently accesses its data.

4.5 Branch Management

For each data key there is a *branch table* that holds all its branches' heads, i.e., the latest `cid`s of the branches.

The branch table comprises two data structures for tagged and untagged branches respectively. Tagged branches are maintained in a map structure called *TB-table*, in which each entry consists of a tag (i.e., branch name) and a head `cid`. Untagged branches are maintained in a set structure called *UB-table*, in which each entry is simply a head `cid`. UB-table essentially maintains all the leaf nodes in the object derivation graph.

4.5.1 Branch Update

The TB-table is updated during the `Put-Branch` operation (M3). Once the new `FObject` is constructed, its `cid` replaces the old branch head in the table. The `Fork-Branch` operation (M11) simply creates a new entry pointing to the referenced branch head. Similarly, operations (M12-M14) only modify entries in the TB-table, without creating new objects. Concurrent updates on a tagged branch are serialized by the servlet. Moreover, to further protect from overwriting others' changes by accident, additional *guarded* APIs are provided, such as:

```
PUT(key, branch, value, guard_uid)
```

which ensures that the `Put` is successful only if the current branch head is equal to `guard_vid`.

The UB-table is updated whenever a new `FObject` is created. Once the `FObject` is constructed, its `cid` is added to the UB-table, and its base `cid` is removed from the table. When the base `cid` is not found in the table, it means the base version has already been derived by others. If the new `FObject` already exists in the storage (i.e., from previous equivalent operations), the UB-table simply ignores it.

4.5.2 Conflict Resolution

A three-way merge algorithm is used for `Merge` (M5-M7) operations. To merge two branch heads v_1 and v_2, three versions (v_1, v_2 and $\texttt{LCA}(v1, v2)$), i.e., the most recent version where they start to fork) are fed into the merge function. If the merge fails, it returns a conflict list, calling for conflict resolution. This can be done at the application layer and the merged result sent back to the storage. Simple conflicts can be resolved using built-in resolution functions (such as `append`, `aggregate` and `choose-one`). *ForkBase* allows users to hook customized resolution strategies which are executed upon conflicts.

4.6 Cluster Management

When deployed as a distributed service, *ForkBase* uses a *hash-based two layer partitioning* that distributes workloads evenly among nodes in the cluster:

- **Request dispatcher to servlet:** requests received by a dispatcher are partitioned and sent to the corresponding servlet based on the request keys' hash.
- **Servlet to chunk storage:** chunks created in a servlet are partitioned based on `cid`s, and then forwarded to the corresponding chunk storage. Meta chunks are always stored locally.

Thanks to the cryptographic hash function, chunks could be evenly distributed across all nodes, even for severely skewed workloads. However, all meta chunks generated by a servlet are always stored in its local chunk storage, as they are not accessed by other servlets. By keeping the meta chunks locally, it is efficient to return a primitive object, or to track historical versions. In addition, servlets may cache the frequently accessed remote chunks. When reading `POS-Tree` nodes from a `FObject`, request dispatchers forward `Get-Chunk` request directly to the chunk storage, without going through the servlet.

4.6.1 Re-balancing `POS-Tree` Construction

Constructing the `POS-Tree` is computation-intensive, which could become a bottleneck for the servlet. Since servlets and chunk storages are decoupled when generating and persisting `POS-Tree`, an overloaded servlet can redistribute the tree construction to other servlets. First, the servlet locks the branch table of the target key, and forwards the request to another servlet. Upon receiving the `cid` of the constructed `POS-Tree`, the first servlet embeds it into the `FObject`, updates and finally unlocks the branch table. In summary, unlike updating the branch table and `FObject`, `POS-Tree` construction can be distributed.

5. FAST DEVELOPMENT OF EFFICIENT APPLICATIONS

In this section, we show how *ForkBase* can be exploited for three applications: a blockchain platform, a wiki engine and a collaborative analytics application. We describe how the storage system meets the applications' demands, reduces development efforts and offers them high performance.

5.1 Blockchain

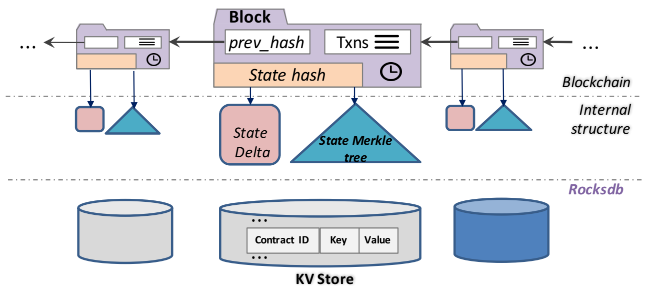
A blockchain system consists of a set of mutually distrustful nodes that together maintain a ledger data structure, which is made consistent via a distributed consensus protocol. Previous works have mainly focused on improving consensus protocols which are shown to be a major performance bottleneck [23]. The data model and storage component of the blockchain are overlooked, although there is an increasing demand for performing analytics on blockchain data [37, 1, 30]. The blockchain data consists of some global states and transactions that modify the states. They are packed into blocks linked with each other via cryptographic hash pointers, forming a chain that ends at the genesis block. In systems that support smart contracts (user-defined codes), each contract is given a key-value storage to manage its own states separately from the global states, and the contract accepts transactions that invoke computations on the states. We refer readers to [24] for a more comprehensive treatment of the blockchain design space.

Although any existing blockchain can be ported to *ForkBase*, here we focus on Hyperledger for two reasons. First, it is one of the most popular blockchains with support for Turing-complete smart contracts, making it easy to evaluate the storage component by writing contracts that stress the storage. Second, the platform targets enterprise applications whose demands for both data processing and analytics are more pronounced than public blockchain applications like crypto-currency.

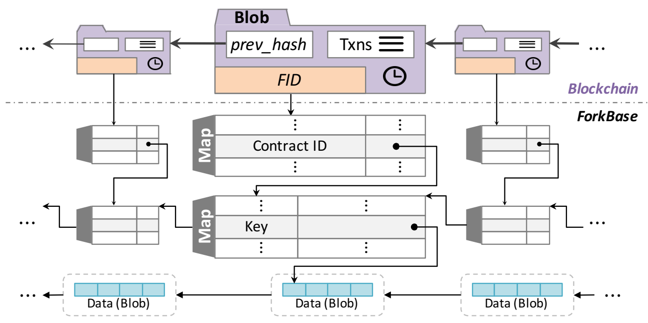
5.1.1 Data Model in Hyperledger

Figure 7(a) illustrates the main data structures in Hyperledger v0.6[3]. The states are protected by a Merkle tree: any

[3]New versions of Hyperledger, i.e. v1.0 and later, make significant changes to the data model, but they do not fit our definition of a blockchain system



(a) Hyperledger (v0.6) datastructure



(b) Hyperledger datastructure on *ForkBase*

Figure 7: Blockchain data structures.

modification results in a new Merkle tree; the old values and old Merkle tree are kept in a separate structure called *state delta*. A blockchain transaction can issue read or write operations (of key-value tuples) to the states. Only transactions that update the states are stored in the block. A read operation fetches the value directly from the storage, while write is buffered in a temporary in-memory data structure. The system batches multiple transactions, then issues a commit when reaching the desired number of transactions or when a timer fires. The commit operation first creates a new Merkle tree, then a new state delta, then a new block, and finally writes all changes to the storage.

5.1.2 *Blockchain Analytics*

One initial goal of blockchain systems is to securely maintain the states and their histories and thus, the designs are guided towards achieving tamper evidence and data versioning. As blockchain applications gain traction, the massive volume of data stored on the ledger becomes valuable for analytics. Traditional analytical data management systems, e.g., OLAP databases, achieve high performance by extensive use of indexes and query optimizations. However, Hyperledger and other blockchains fall short in this respect.

In this work we consider two representative analytical queries that can be performed on blockchain data. The first query is a *state scan*, which returns the history of a given state, i.e. how the current value comes about. The second query is a *block scan*, which returns the values of the states at a specific block. The current Hyperledger data structures are designed for fast access to the *latest* states. However, the two above queries require traversing to the previous states and involve computations with state delta. This inefficient query execution is precisely due to the lack of an index structure. We implemented both queries in Hyperledger by adding a pre-processing step that parses all the internal structures of all the blocks and constructs an in-memory index.

5.1.3 *Hyperledger on ForkBase*

Figure 7(b) illustrates how we use *ForkBase* to implement Hyperledger's data structures. The key insight here is that an `FObject` fully captures the structure of a block and its corresponding states. We replace Merkle tree and state delta with `Map` objects organized in two levels. The state hash is now replaced by the version of the first-level `Map` object. This `Map` object contains key-value tuples where the key is the smart contract ID, and the value is the version of the second-level `Map` object. This second-level `Map` contains key-value tuples where the key is the data key, and the value is the version of a `Blob` object storing the state value.

One immediate benefit of this implementation is that the code for maintaining data history and integrity becomes remarkably simple. In particular, for only 18 lines of code that uses *ForkBase*, we eliminate 1918 lines of code from the Hyperledger codebase. Another benefit is that the data is now readily usable for analytics. For state scan query, we simply follow the version number stored in the latest block to get the latest `Blob` object for the requested key. From there, we follow *base_version* to retrieve the previous values. For block scan query, we follow the version number stored on the requested block to retrieve the second-level `Map` object for this block. We then iterate through the key-value tuples and retrieve the corresponding `Blob` objects.

5.2 Wiki Engine

A wiki engine allows users to collaboratively create and edit documents (or wiki entries). Each entry contains a linear chain of versions. The wiki engine can be built on top of a multi-versioned key-value storage, in which each entry is mapped to a key and the entry's content is stored as the associated value. Such a multi-versioned storage can be directly implemented with Redis [9], for instance, using the list data type offered by Redis. More specifically, the wiki entry is of a list type, and every new version is appended to the list.

This multi-versioned key-value data model maps naturally into *ForkBase*. Reading and writing an entry are directly implemented with `Get` and `Put` operations on default branches. Because each version often changes only small parts of the data, the `Blob` type is more suitable to represent an entry. Other meta information, e.g., timestamp, can be stored directly in the `context` field. When accessing consecutive versions, *ForkBase* can leverage cached data chunks to serve out data more quickly. Another common operation in a wiki engine, namely `diff` operation between two versions, is directly and efficiently supported in *ForkBase*, thanks to the `POS-Tree` index. Finally, *ForkBase*'s two-level partitioning scheme helps alleviate skewed workloads which are common in wiki services due to hot entries.

5.3 Collaborative Analytics

It is becoming increasingly common for a group of scientists (or analysts) to work on a shared dataset, but with different analysis goals [12, 40]. For example, on a dataset of customer purchasing records, some analysts may perform customer behavioral analysis, while others use it to improve inventory management. At the same time, the

Table 3: Performance of *ForkBase* Operations.

	Throughput (ops/sec)		Avg. latency (ms)	
	1KB	20KB	1KB	20KB
Put-String	75.0K	8.3K	0.24	0.9
Put-Blob	37.5K	5.7K	0.28	1.0
Put-Map	35.8K	4.7K	0.38	1.28
Get-String	78.3K	56.9K	0.23	0.8
Get-Blob-Meta	99.7K	100.4K	0.16	0.17
Get-Blob-Full	38.4K	4.9K	0.62	2.9
Get-Map-Full	38.2K	5.0K	0.61	3.2
Track	97.8K	96.0K	0.16	0.17
Fork	113.6K	109.4K	0.17	0.17

dataset may be continually cleaned and enriched by other analysts. As the analysts simultaneously work on different versions or branches of the same dataset, there is a clear need for versioning and fork semantics. Decibel [35] and OrpheusDB [50] support these features for relational datasets, employing delta-based deduplication techniques.

ForkBase has a rich collection of built-in data types, which offer the flexibility to implement many types of structured datasets. Specifically, we implement two layouts for relational datasets: row-oriented and column-oriented. In the former, a record is stored as a `Tuple`, embedded in a `Map` keyed by its primary key. In the latter, column values are stored as a `List`, embedded in a `Map` keyed by the column name. Applications can choose the layout that best serves their queries. For instance, the column-oriented layout is more efficient for applications that perform many analytical queries.

Common operations in collaborative analytics include dataset import and export, dataset transformations, analytical queries, and version comparisons. In *ForkBase*, accessing large datasets is efficient because only relevant chunks are fetched to the client. Comparing large datasets via the `diff` operation is also efficient, thanks to the `POS-Tree` index. While other delta-based systems such as Decibel eliminates duplicates within a single dataset, *ForkBase* deduplication works across multiple datasets, therefore achieving lower storage overhead.

6. EVALUATION

We implemented *ForkBase* in about 30k lines of `C++` code. In this section, we first evaluate the performance of *ForkBase* operations. Next, we evaluate the three applications discussed in Section 5 in terms of storage consumption and query efficiency. We compare them against their respective state-of-the-art implementations.

Our experiments were conducted in an in-house cluster with 64 nodes, each of which runs Ubuntu 14.04, and is equipped with E5-1650 3.5GHz CPU, 32GB RAM, and 2TB hard disk. All nodes are connected via 1Gb Ethernet. For fair comparison against other systems, all servlets are configured with one thread for request execution and two threads for request parsing. Both leaf and index chunk sizes in the `POS-Tree` are set to 4KB.

6.1 Micro-Benchmark

We benchmark 9 *ForkBase* operations. We deployed one servlet and used multiple clients for sending requests. Table 3 lists the aggregated throughput and average latency measured at 32 clients, with varying request sizes. We observe that large requests achieve higher network throughput

Table 4: Breakdown of Put Operation (μs).

	String		Blob	
	1KB	20KB	1KB	20KB
Serialization	0.8	0.8	1.1	1.5
Deserialization	5.7	9.2	6.2	13.2
CryptoHash	8.5	56.4	9.5	80.6
RollingHash	-	-	7.5	42.2
Persistence	10.4	60.7	10.5	93.7

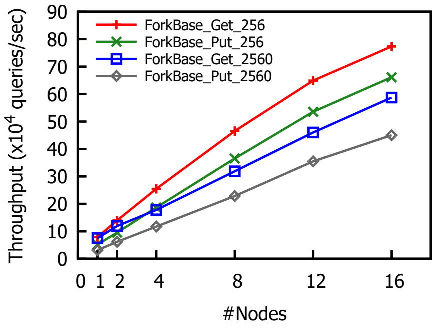


Figure 8: Scalability with multiple servlets.

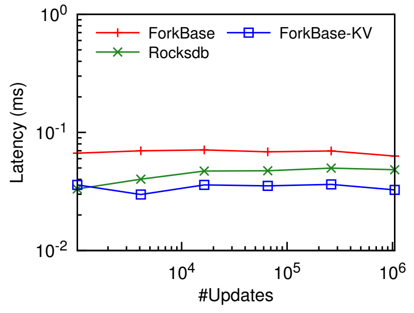
— the product of throughput and request size — because of smaller overheads in message parsing. The throughputs of primitive types are higher than those of chunkable types, due to overhead in chunking and traversing the `POS-Tree`. **`Get-X-Meta`**, **`Track`** and **`Fork`** achieve the highest throughputs, regardless of the request sizes. This is because these operations require no or very small data transfer. The average latencies of different operations do not vary much, because the latency is measured at the client side, therefore network delays have major contribution to the final latency.

Table 4 details the cost breakdown of the `Put` operation, excluding the network cost. It can be seen that the main contributor to the latency gap between primitive and chunkable types is the rolling hash computations incurred in the `POS-Tree`.

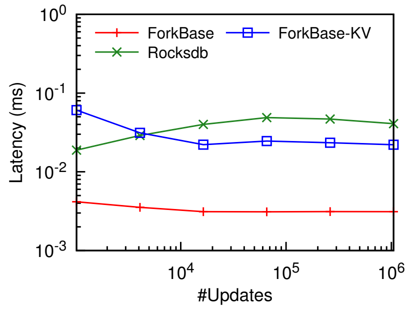
We measured *ForkBase*'s scalability by increasing the number of servlets up to 64. Figure 8 shows almost linear scalability for both `Put` and `Get` operations. The fact that *ForkBase* scales almost linearly is expected because there is no communication between the servlets.

6.2 Blockchain

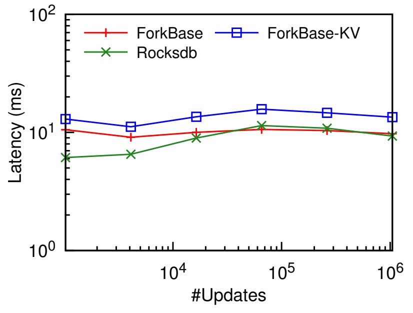
We compare *ForkBase*-backed Hyperledger [5] with the original implementation using RocksDB, and also with another implementation that uses *ForkBase* as a pure key-value storage. We refer to them as *ForkBase*, Rocksdb and *ForkBase*-KV respectively. We first evaluate how different storage engines affect normal operations of Hyperledger and the user-perceived performance. We then evaluate their efficiency on supporting analytical queries. We used Blockbench [23], a benchmarking framework for permissioned blockchains, to generate and drive workloads. Specifically, we used the smart contract implementing a key-value store. Transactions for this contract are generated based on YCSB workloads. We varied the number of keys, the number and ratio of read and write operations (r and w). Unless stated otherwise, the number of keys is the same as the number of operations. For the blockchain configuration,



(a) Read



(b) Write



(c) Commit

Figure 9: Latency of blockchain operations (b=50, r=w=0.5).

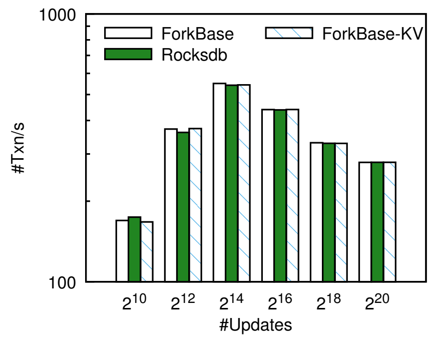


Figure 10: Client perceived throughput (b=50, r=w=0.5).

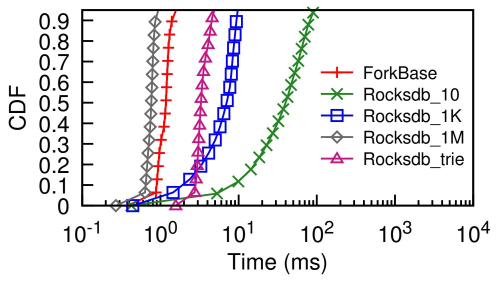


Figure 11: Commit latency distribution with different Merkle trees.

we deployed one server, varied the maximum block size b and kept the default values for the other settings.

6.2.1 Blockchain Operations

Figure 9 shows the 95th percentile latency of blockchain operations, including read, write and commit. As we can observe, both read and write operations take under $0.1ms$, two orders of magnitude faster than a commit. A read only retrieves one key at a time from the storage. *ForkBase* takes longer than the other two because multiple objects needed to be retrieved. *ForkBase*-KV is slightly better than Rocksdb, because the latter stores data in multiple levels (based on Leveldb) and requires traversing them to retrieve the key. A write in *ForkBase* simply buffers new value, whereas Rocksdb and *ForkBase*-KV need to compute temporary updates for the internal structures (Merkle tree and state delta). This explains why *ForkBase* outperforms the others in write. Though Rocksdb is designed for fast batch commits, *ForkBase* and Rocksdb still have similar latencies, as shown in Figure 9(c). Both are better than *ForkBase*-KV since using *ForkBase* as a pure key-value store introduces overhead from doing hash computation both inside and outside of the storage layer.

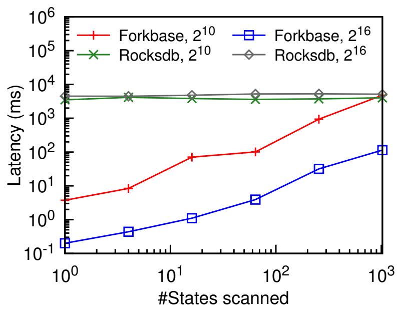
Figure 10 shows the overall throughput, measured as the total number of transactions committed to the blockchain per second. We see no differences in throughput, because the overheads in read, write and commit are relatively small compared to the total time a transaction takes to be included in the blockchain. In fact, we observe that the cost of executing a batch of transactions is much higher than that of committing the batch.

6.2.2 Merkle Trees

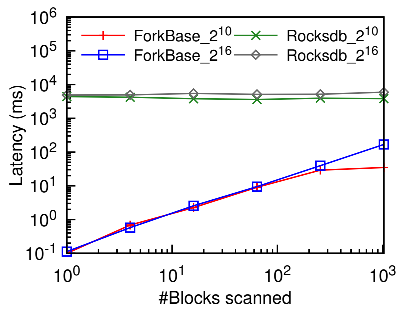
A commit operation involves updating `Map` objects in *ForkBase* or the Merkle trees in the original Hyperledger. Hyperledger provides two Merkle tree implementations. The default option is a bucket tree, in which the number of leaves is fixed and pre-determined at start-up time, and the data key's hash determines its bucket number. The other option is a trie. Figure 11 shows how different structures affect the commit latency. With bucket tree, the number of buckets (nb = 10, 1K, 1M) has significant impact on the commit latency. With fewer buckets, the latency increases and the distribution becomes less uniform. This is because with more updates, write amplification becomes more severe, which increases the cost of updating the tree. In fact, for any pre-defined number of buckets, the bucket tree is expected to fail to scale beyond workloads of a certain size. In contrast, `Map` objects in *ForkBase* scale gracefully by dynamically adjusting the tree height and bounding node sizes. The trie structure exhibits low amplification, but the latency is higher than *ForkBase* because the structure is not balanced, therefore it may require longer tree traversals during updates.

6.2.3 Analytical Queries

We populated the storage with varying numbers of keys and a large number of updates that result in a medium-size chain of 12000 blocks. Figure 12(a) compares the performance for state scan query. The x axis represents the num-

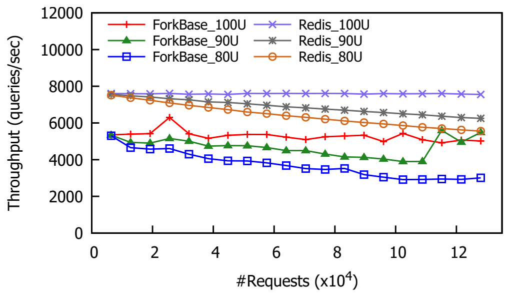


(a) State Scan

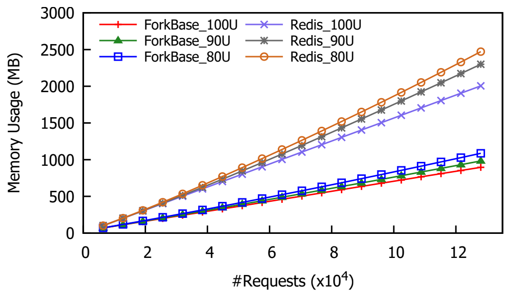


(b) Block scan

Figure 12: Scan queries. 'X, 2^y' means using storage X, and 2^y keys.



(a) Throughput



(b) Storage Consumption

Figure 13: Performance of editing wiki pages.

ber of unique keys scanned per query. For a small number of keys, the difference between *ForkBase* and Rocksdb is up to 4 orders of magnitudes. This is because the cost in Rocksdb is dominated by the pre-processing phase, which is not required in *ForkBase*. However, this cost amortizes with more keys, explaining why the performance gap gets smaller. In particular, this gap reduces to 0 when the number of unique keys being scanned is the same as the total number of keys in the storage, since scanning them requires retrieving all the data from the storage.

Figure 12(b) shows the performance for block scan query. The x axis represents the block number being scanned, where $x = 0$ is the oldest (or first) block. We see a huge difference in performance starting from 4 orders of magnitudes but decreasing with higher block numbers. The cost in *ForkBase* increases because higher blocks contain more keys to be read. This also explains why it stops increasing after a certain number of blocks, after which the gap remains at least two orders of magnitudes regardless of which block being scanned.

6.3 Wiki Engine

We compare *ForkBase* with Redis, both of which were deployed as multi-versioned wiki engines. We employed 32 clients on separate nodes to simultaneously edit 3200 pages hosted in the engine. In each request, a client loads/creates a random page whose initial size is 15 KB, edits/appends the text, and finally uploads the revised version.

6.3.1 *Edit and Read Pages*

Figure 13(a) shows the throughput of editing pages, in which `xU` indicates the ratio of in-place updates against insertions (`100U` means all updates are in-place). It is expected that Redis outperforms *ForkBase* in terms of write throughput, since the latter has to chunk the text and build the `POS-Tree`. On the other hand, the chunking overhead is paid off by the deduplication along the version history. As shown in Figure 13(b), *ForkBase* consumes 50% less storage than Redis, even though Redis uses compression during data persistence. The performance of reading wiki pages is illustrated in Figure 14. It can be seen that Redis is fast for reading the latest version. As we track more versions during a single exploration, *ForkBase* starts to outperform Redis. The reason is that the data chunks composing a `Blob` value can be cached at the clients. When reading an old version, a large number of chunks may have already been cached, resulting in smaller read latencies.

6.3.2 *Hot Pages*

We deployed a distributed wiki service in a 16-node cluster, and ran a skewed workload (zipf = 0.5). Figure 15 shows the effect of skewness to storage size distribution. With one layer partitioning on the page name (1LP), where page content is stored locally, *ForkBase* suffers from imbalance. The two layer partitioning (2LP) overcomes the problem by distributing chunks evenly among different nodes.

6.4 Collaborative Analytics

We compare *ForkBase* with OrpheusDB, a state-of-the-art dataset management system, in terms of their performance in storing and querying relational datasets. For completeness, our comparison contains update queries, but we note that OrpheusDB is not designed for efficient checkout

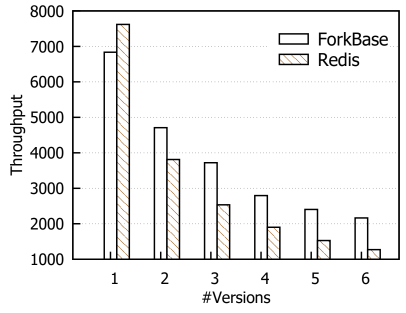


Figure 14: Throughput of read consecutive versions of a wiki page.

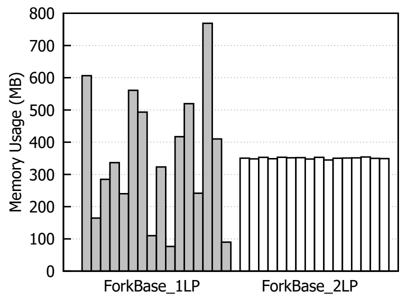
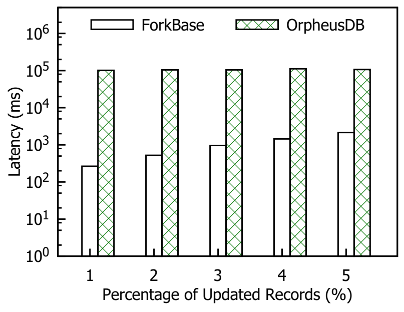
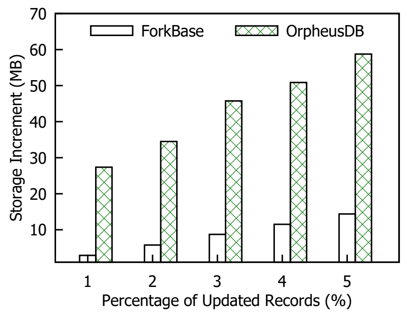


Figure 15: Storage size distribution in skewed workloads.



(a) Elapsed time



(b) Space Increment

Figure 16: Performance of dataset modifications.

and modification operations. We used a dataset containing 5 million records loaded from a csv file. Each record is around 180 bytes in length, consisting of a 12-byte primary key, two integer fields and other textual fields of variable lengths. The initial space consumption for this dataset is 927MB in *ForkBase* and 1167MB in OrpheusDB.

6.4.1 Dataset Modification

The dataset maintained by OrpheusDB is materialized as a table and can be easily manipulated through standard SQL queries. *ForkBase* uses built-in methods to implement the same table abstraction. Figure 16(a) shows the latency of dataset modification for both systems. As OrpheusDB is not designed for fast data updates, *ForkBase* outperforms it by two orders of magnitude. The performance gap is due to two factors. First, during checkout, OrpheusDB reconstructs a working copy from sub-tables, whereas *ForkBase* only returns a handler and defer fetching relevant chunks. Second, during commit, there is less data to be stored, as can be seen from Figure 16(b). OrpheusDB consumes 3× more space than *ForkBase* from newly created sub-tables. Thanks to fine-grained deduplication, *ForkBase* only needs to commit a small number of chunks.

6.4.2 Version Comparison

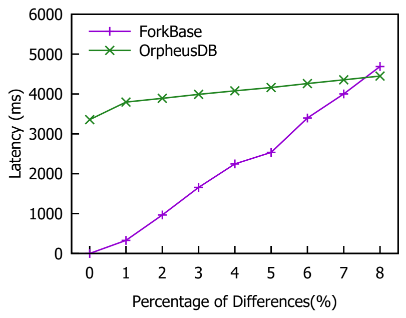
Figure 17(a) shows the cost in comparing two dataset versions with a varying degree of differences. OrpheusDB's cost is roughly consistent, because the storage maintains a vector of record-version mapping for each dataset version, and it relies on full vector comparison to find the differences. On the contrary, *ForkBase*'s cost is low for small differences, because *ForkBase* can quickly locate them using the `POS-Tree`. However, the cost increases when the differences are large, because *ForkBase* has to traverse more tree nodes.

6.4.3 Analytical Queries

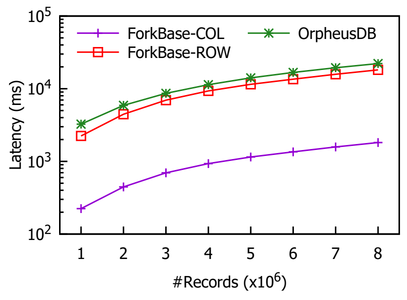
Figure 17(b) compares the performance of aggregation queries on the numerical fields. For *ForkBase*, both row and column layouts were used. It can be seen that row-oriented *ForkBase* and OrpheusDB have similar performance, whereas column-oriented *ForkBase* has 10× better performance. The gap is due to the physical layouts. More specifically, extracting fields is more expensive in row-oriented than in column-oriented layouts. This shows that *ForkBase*'s flexible data model offers opportunities to fine-tune application performance. Finally, we note that OrpheusDB supports other advanced analytics, e.g., join queries, thanks to its underlying RDBMS. Nevertheless, with additional engineering effort, it is possible to extend *ForkBase* with richer query functionalities by adding them to the view layer.

7. CONCLUSIONS

There are requirements from modern applications that have not been well addressed in existing storage systems. We identified three common properties in blockchain and forkable applications: data versioning, fork semantics and tamper evidence. We discussed the values of a unified storage engine that offers these properties off the shelf. We designed and implemented *ForkBase* that embeds the above properties and is able to deliver better performance than ad-hoc, application-layer solutions. By implementing three applications on top of *ForkBase*, we demonstrate that our storage makes it easy to express application requirements,



(a) Diff



(b) Aggregation

Figure 17: Performance of querying datasets.

thereby reducing development efforts. We showed via experimental evaluation that *ForkBase* is able to deliver better performance than state-of-the-art in terms of storage consumption, query efficiency and coding complexity. We believe that *ForkBase*'s unique properties are key enablers for building emerging applications such as blockchain and collaborative analytics.

8. REFERENCES

[1] Chainalysis - blockchain analysis. https://www.chainalysis.com.

[2] Ethereum. https://www.ethereum.org.

[3] Github. https://github.com.

[4] Googledocs. https://www.docs.google.com.

[5] Hyperledger. https://www.hyperledger.org.

[6] LevelDB. https://github.com/google/leveldb.

[7] MongoDB. http://mongodb.com.

[8] noms. https://github.com/attic-labs/noms.

[9] Redis. http://redis.io.

[10] RocksDB. http://rocksdb.org.

[11] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. In *ACM SIGMOD Record*, volume 15, pages 96–107. ACM, 1986.

[12] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. Parameswaran. Datahub: Collaborative data science & dataset version mangement at scale. In *CIDR*, 2015.

[13] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proceedings of the VLDB Endowment*, 8(12):1346–1357, 2015.

[14] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kullkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[16] A. Chavan and A. Deshpande. Dex: Query execution in a delta-based storage system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 171–186. ACM, 2017.

[17] J. D. Cohen. Recursive hashing functions for n-grams. *ACM Trans. Inf. Syst.*, 15(3):291–320, July 1997.

[18] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[19] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[20] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement. Tardis: a branch-and-merge approach to weak consistency. In *SIGMOD*, 2016.

[21] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[22] A. Dinh, J. Wang, S. Wang, G. Chen, W.-N. Chin, Q. Lin, B. C. Ooi, P. Ruan, K.-L. Tan, Z. Xie, et al. Ustore: a distributed storage with rich semantics. *arXiv preprint arXiv:1702.02799*, 2017.

[23] T. T. A. Dinh, J. Wang, G. Chen, L. Rui, K.-L. Tan, and B. C. Ooi. Blockbench: a benchmarking framework for analyzing private blockchains. In *SIGMOD*, 2017.

[24] T. T. A. Dinh, M. Zhang, B. C. Ooi, and G. Chen. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[25] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[26] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[27] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

[28] L. Jiang, B. Salzberg, D. Lomet, and M. Barrena. The bt-tree: A branched and temporal access method. 2000.

[29] M. Kallahalla, E. Riedely, R. Swaminathan, Q. Wangz, and K. Fux. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, 2003.

[30] H. Kalodner, S. Goldfeder, A. Chator, M. Mser, and A. Narayanan. Blocksci: Design and applications of a blockchain analysis platform. `https://arxiv.org/abs/1709.02489`, 2017.

[31] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE '11*, pages 195–206, 2011.

[32] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 997–1008. IEEE, 2013.

[33] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[34] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository. In *OSDI*, 2004.

[35] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. G. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *PVLDB*, 9(9):624–635, 2016.

[36] R. C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

[37] J. Morgan and O. Wyman. Unlocking economic advantage with blockchain. a guide for asset managers., 2016.

[38] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[39] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2009.

[40] F. A. Nothaft, M. Massie, T. Danford, and et al. Rethinking data-intensive science using scalable analytics systems. In *ACM SIGMOD*, 2015.

[41] J. Paulo and J. Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.

[42] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys (CSUR)*, 31(2):158–221, 1999.

[43] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: the file system that never forgets. In *HotOS*, pages 2–7, 1999.

[44] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1013–1024. IEEE, 2012.

[45] Y. Sompolinsky and A. Zohar. Secure high-rate transaction processing in bitcoin. `https://eprint.iacr.org/2013/881.pdf`.

[46] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *FAST*, 2003.

[47] M. Stonebraker. The design of the postgres storage system. In *VLDB*, 1987.

[48] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised system. In *OSDI*, 2000.

[49] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc., 1993.

[50] L. Xu, S. Huang, S. Hui, A. J. Elmore, and A. Parameswaran. Orpheusdb: A lightweight approach to relational dataset versioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1655–1658. ACM, 2017.