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 [§], Beng Chin Ooi [#], Pingcheng Ruan [#]

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

[#]{wangsh,dinhtta,linqian,zhongle,caiqc,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 designed for blockchain and forkable applications. By integrating core application properties into the storage, *ForkBase* not only delivers high performance but also reduces development effort. The storage manages multiversion data and supports two variants of fork semantics which enable different fork worklflows. *ForkBase* is fast and space efficient, due to a novel index class that supports efficient queries as well as effective detection of duplicate content across data objects, branches and versions. We demonstrate *ForkBase*'s performance using three applications: a blockchain platform, a wiki engine and a collaborative analytics application. We conduct extensive experimental evaluation against respective state-of-the-art solutions. The results show that *ForkBase* achieves superior performance while significantly lowering the development effort.

PVLDB Reference Format:
Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, Pingcheng Ruan. ForkBase: An Efficient Storage Engine for Blockchain and Forkable Applications. *PVLDB*, 11(10): 1137-1150, 2018.
DOI: https://doi.org/10.14778/3231751.3231762

1. INTRODUCTION

Developing a new application today is made easier by the availability of many storage systems that offer different data models and operation semantics. At one extreme, key-value stores [22, 38, 8, 37] provide a simple data model and semantics, but are highly scalable. At the other extreme, relational databases [59] support more complex, relational models and strong semantics, i.e. ACID, which render them less scalable. In between are systems that make other trade-offs between data model, semantics and performance [16, 20, 15, 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 existing storage systems. First, blockchain systems, such as Bitcoin [47], Ethereum [2] and Hyperledger [5], implement a distributed ledger abstraction — a globally consistent history of changes made to some global states. Because blockchain systems operate in an untrusted 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 [26], GoogleDocs [4], and Github [3] to more recent and advanced analytics platforms like Datahub [43], allow many users to work together on the shared data. Such applications need explicit *data versioning* to track data derivation history, and *fork semantics* to let users work on independent data copies. Third, recent systems that favor availability over consistency allow concurrent write access to the data, which results in implicit forks that must be eventually resolved by upper layer applications [22, 21].

Without proper storage support, the applications have to implement the above mentioned properties on top of a generic storage such as key-value stores or file systems. One problem with this approach is the additional development cost. Another problem is that the resulting implementations may fail to generalize to other applications. But more importantly, the bespoke implementation may incur unnecessary performance overhead. For example, current blockchain platforms (e.g., Ethereum, Hyperledger) build their tamper evident data structures on top of a key-value store, (e.g., LevelDB [6] and RocksDB [9]). However, we observe that these ad-hoc implementations do not always scale well, and the current blockchain data structures are not optimized for analytical queries. As another example, collaborative applications over large, relational datasets, use file-based version control systems such as git. But they do not scale to big datasets, and only offer limited query functionality.

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 with additional features at no extra effort. Finally, the storage engine can exploit optimization that is otherwise hard to achieve at the application layer.

[*]corresponding author

Proceedings of the VLDB Endowment, Vol. 11, No. 10
Copyright 2018 VLDB Endowment 2150-8097/18/06.
DOI: https://doi.org/10.14778/3231751.3231762

In this paper we present *ForkBase*, a novel and efficient storage engine that meets the high demand in modern applications for versioning, forking and tamper evidence[1]. One challenge in building *ForkBase* is to keep the storage overhead small when maintaining a large number of data versions. Another challenge is to provide elegant and flexible semantics to many classes of applications. *ForkBase* overcomes these challenges in two novel ways. First, it defines a new class of index called *Structurally-Invariant Reusable Indexes* (*SIRI*), which facilitates both fast lookups and effective identification of duplicate content. The latter helps drastically lower the storage overhead for multiversion data. *ForkBase* implements an instance of *SIRI* called *POS-Tree*, that combines ideas from content-based slicing [45], Merkle tree [44] and B$^+$-tree [19]. *POS-Tree* directly offers tamper evidence, making *ForkBase* a natural choice for applications in untrusted environments. Second, *ForkBase* provides a generic fork semantics that affords the applications the flexibility of having both implicit and explicit forks. Fork operations are efficient, thanks to *POS-Tree*'s use of copy-on-write that eliminates unnecessary copies.

ForkBase exposes simple APIs and an extended key-value data model. It provides built-in data types that help reduce development effort and enable multiple trade-offs between query efficiency and storage overhead. *ForkBase* also scales well to many nodes, as it employs a two-layer partitioning scheme which helps distribute skewed workloads evenly across nodes.

To demonstrate the values of our design, we build 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 engine, e.g., 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 it analytics-ready.

In summary, we make the following contributions:

- We identify common properties in modern applications, i.e., versioning, forking and tamper evidence. We examine the benefits of a storage that integrates all these properties.

- We introduce a novel index class called *SIRI* that effectively removes duplicates across multiversion data. We present *POS-Tree*, an instance of *SIRI* that additionally offers tamper evidence. We propose a generic fork semantics that captures the workflows of many different applications.

- We implement *ForkBase* that efficiently supports blockchains and forkable applications. *ForkBase* derives its efficiency from the *POS-Tree* and flexibility from the generic fork semantics. With elegant interfaces and rich data types, *ForkBase* offers a powerful building block for high-level systems and applications.

- We demonstrate the usability of *ForkBase* by implementing three representative applications, namely a blockchain platform, a wiki service and a collaborative analytics application. We show via extensive experimental evaluation that *ForkBase* helps these applications outperform the respective state-of-the-arts in terms of coding complexity, storage overhead and query efficiency.

In the following, we first discuss relevant background and motivation in Section 2. We introduce the design in Section 3, followed by the interfaces and implementation in Section 4 and 5 respectively. We describe the modeling and evaluation of three applications in Section 6 and 7. We discuss related work in Section 8 before concluding in Section 9.

2. BACKGROUND AND MOTIVATIONS

In this section, we discuss several common properties underpinning many modern applications. We motivate the design of *ForkBase* by highlighting the gap between what the application requires of these properties and what existing solutions offer.

2.1 Deduplication for Multiversion Data

Data versioning is an important concept in applications that keep track of data changes. Each update to the data creates a new version, and the version history can be either linear or non-linear (i.e. consisting of forks and branches). Systems that support linear version histories include multiversion file systems [55, 60, 58] and temporal databases [11, 54, 61]. Systems that have non-linear histories include software version control such as git, svn and mercurial for files, and collaborative dataset management such as Decibel [43] and OrpheusDB [31] for relational tables. Blockchains can be also seen as versioning systems in which each block represents a version of the global states.

One major challenge in supporting data versioning is to reduce storage overhead. The most common approach used in dataset versioning systems, e.g. Decibel and OrpheusDB, is record-level *delta encoding*. In this approach, the new version stores only the records that are modified from the previous version. As a result, it is highly effective when the differences between consecutive versions are small, but requires combining multiple deltas to reconstruct the version's content. OrpheusDB optimizes for reconstructions by flattening the deltas into a full list of record references. However, this approach does not scale well for large tables. Delta encoding suffers from the problem that it creates a new copy whenever a version is modified, even if the new content is identical to that of some older versions or of versions in a different branch. In other words, delta encoding is not effective for non-consecutive versions, divergent branches or datasets. For instance, in multi-user applications like Datahub [13], duplicates across branches and datasets are common. Even in the extreme case that users upload same data as a new dataset, delta encoding offers no benefits. Another example is application implementations that have no explicit versioning, such as WordPress and Wikipedia [63], which store versions as independent records. In these cases, the storage sees no relationship between the two versions and cannot exploit delta encoding to eliminate duplicate content.

Another approach for detecting and removing duplicates is *chunk-based deduplication*. Unlike delta encoding, this approach works across independent objects. It is widely used in file systems [52, 62], and is a core feature of git. In this approach, files are divided into units called *chunk*, each of which is given a unique identifier (e.g., by applying a collision-resistant hash function over the chunk content) to detect identical chunks. Chunk-based deduplication is highly effective in removing duplicates for large files that are rarely modified. When an update leads to change existing chunks, content-based slicing [45] can be used to avoid expensive re-chunking, i.e. the boundary-shifting problem [29].

In this work, we adopt chunk-based deduplication for structured datasets, such as relational tables. Instead of deduplicating on individual records as in delta encoding, we apply deduplication on the level of data pages in primary indexes. One direct benefit is that, to

[1] *ForkBase* is the second version of UStore [24], which has evolved significantly from the initial design and implementation.

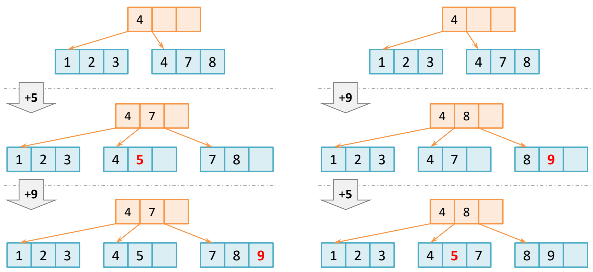


Figure 1: Two B$^+$-trees containing same entries could have different internal structures.

access a version, we no longer need to reconstruct it from records, and can directly access index and data pages for fast lookups and scans. However, deduplication could be less effective when applying to data pages in indexes, e.g. B$^+$-tree [19]. The main reason is that the content of a data page is not only determined by the items stored in the index, but also by the update history of the index. Figure 1 shows an example in which two B$^+$-trees contain identical sets of items, but have different data and index pages. As a consequence, even when the indexes contain the same data items, the probability of having identical pages remains small. Moreover, their structural differences make it more complex to compare the versions for *diff* and *merge* operations.

In *ForkBase*, we address the above issues by defining a new index class called *Structurally-Invariant Reusable Indexes* or *SIRI* (§3.1), whose properties enable effective deduplication. We then design an index structure *POS-Tree* (§3.2) belonging to this index class. This structure not only detects duplicates across independent objects[2], but also provides efficient manipulation operations, such as lookup, update, diff and merge.

2.2 Ubiquitous Forking

The concept of forks and branches captures the non-linearity of version histories. It can be found in a broad range of applications, from collaborative applications such as git and Datahub, highly available replicated systems such as Dynamo [22] and TARDiS [21], to blockchain systems such as Ethereum [2]. Two core operations in these applications are fork and merge. The former creates a new logical copy of the data, called *branch*, which can be manipulated independently such that modifications on one branch are isolated from other branches. The latter integrates content from different branches and resolves potential conflicts.

Current applications have two distinct types of fork operations, namely *on demand* (or explicit) and *on conflict* (or implicit). On-demand forks are used in applications with explicit need for isolated or private branches. One example is software version control systems, e.g. git, which allow forking a branch for development and only merging changes to the main code base (the trunk) after they are well tested. Another example is collaborative analytics applications, e.g. Datahub, which allow branching off from a relational dataset to perform data transformation tasks such as cleansing, correction and integration.

On-conflict forks are used in applications that automatically create branches on conflicting updates. Examples include distributed applications that trade consistency for better availability, latency, partition tolerance and scalability (or ALPS [42]). In particular, Dynamo [22] and Ficus [30] expose on-conflict forks to the users in form of conflicting writes. TARDiS [21] proposes a branch-and-merge semantics for weakly consistent applications. A branch containing the entire state is created whenever there is a conflicting write. By isolating changes in different branches, the application logic is greatly simplified, especially since locks and rollbacks are eliminated from the critical path. In cryptocurrency applications such as Bitcoin [47] and Ethereum [2], forks arise implicitly when multiple blocks are appended simultaneously to the ledger. The forks are then resolved by taking the longest chain or by more complex mechanisms such as GHOST [57].

ForkBase is the first storage engine with native support for both on-demand and on-conflict fork semantics (§3.3). The application decides when and how branches are created and merged, while storage optimizes branch related operations. By providing generic fork semantics, *ForkBase* helps simplify application logic and lower development cost while preserving high performance.

2.3 Blockchains

Security conscious applications demand data integrity against malicious modifications, not only from external attackers but also from malicious insiders. Examples include outsourced services like storage [34] or file system [41], which is able to detect data tampering. Blockchain systems [47, 37, 2] rely on tamper evidence to guarantee that the ledger is immutable. A common approach to achieve tamper evidence is to use cryptographic hashing [36] and Merkle tree [44]. Indeed, current blockchain systems contain bespoke implementations of Merkle-tree-like data structures on top of a simple key-value storage such as LevelDB [6] or RocksDB [9]. However, such implementations are not designed to be general and therefore difficult to be reused or ported to different blockchains.

As blockchain systems are gaining traction, there is an increasing demand for performing analytics on blockchain data [56, 1, 35]. However, current blockchain storage engines are not designed for such tasks. More specifically, blockchain data is serialized and stored as uninterpretable bytes in the storage, therefore it is impossible for the storage engine to support efficient analytics. Consequently, to perform blockchain analytics directly on the storage, the only option is to understand the data model and serialization scheme, and to reconstruct the data structures by scanning the entire storage.

ForkBase facilitates the development of blockchain systems and applications by providing multiversion, tamper evident data types and fork semantics. In fact, all data types in *ForkBase* are tamper evident. *ForkBase* helps reduce development effort, because its data types make it easy to build complex blockchain data models while abstracting away integrity issues (§6.1). More importantly, *ForkBase* makes the blockchain analytics-ready, as rich structural information is captured at the storage.

3. DESIGN

In this section, we present the design of *ForkBase*. We start by defining a new index class *SIRI* that facilitates deduplication. We then discuss a specific instance of *SIRI* called *POS-Tree* that additionally offers tamper evidence. Finally, we describe the model for generic fork semantics.

3.1 SIRI Indexes

Existing primary indexes in databases focus on improving read and write performance. They do not consider data page sharing, which makes page-level deduplication ineffective as shown in Figure 1. We propose a new class of indexes, called *Structurally-Invariant Reusable Indexes* (*SIRI*), which facilitates page sharing among different index instances.

[2] Chunk-based deduplication is less effective than delta encoding when the deltas are of much smaller size than the chunks.

Let $\mathcal{I}$ be an index structure. An instance I of $\mathcal{I}$ stores a set of records $rec(I) = \{r_1, r_2, ..., r_n\}$. The internal structure of I consists of a collection of pages (i.e. index and data pages) $page(I) = \{p_1, p_2, ..., p_m\}$. Two pages are equal if they have identical content and hence can be shared (i.e. deduplicated). $\mathcal{I}$ is called an instance of *SIRI* if it has the following properties:

1. *Structurally Invariant.* For any instance I_1, I_2 of $\mathcal{I}$:
$$rec(I_1) = rec(I_2) \iff page(I_1) = page(I_2)$$

2. *Recursively Identical.* For any instance I_1, I_2 of $\mathcal{I}$ such that $rec(I_2) = rec(I_1) + r$ for any record $r \notin I_1$:
$$|page(I_2) - page(I_1)| \ll |page(I_1) \cap page(I_2)|$$

3. *Universally Reusable.* For any instance I_1 of $\mathcal{I}$ and page $p \in page(I_1)$, there exists another instance I_2 such that:
$$(|page(I_2)| > |page(I_1)|) \wedge (p \in page(I_2))$$

The first property means that the internal structure of an index instance is uniquely determined by the set of records. By avoiding the structural variance caused by the order of modifications, all pages between two logically identical index instances can be pairwisely shared. The second property means that an index instance can be represented recursively by smaller instances with little overhead, while the third property ensures that a page can be reused by many index instances. By avoiding the structural variance caused by index cardinalities, a large index instance can reuse pages from smaller instances. As a result, instances with overlapping content can share a large portion of their sub-structures.

B$^+$-trees and many other balanced search trees do not have the first property, since their structures depend on the update sequence. Similarly, indexes that require periodical reconstruction, such as hash tables and LSM-trees [50], do not have the second property. Most hash tables do not have the third property. For example, a bucket page in a small table is unlikely to be reused in a large table because the records will be placed in multiple smaller buckets. There are existing index structures that meet all properties, such as radix trees or tries. However, they are unbalanced and therefore could not bound operation costs.

3.2 Pattern-Oriented-Split Tree

We propose an instance of *SIRI* indexes called *Pattern-Oriented-Split Tree* (*POS-Tree*). Beside the *SIRI* properties above, it has three additional properties: it is a probabilistically balanced search tree; it is efficient to find differences and to merge two instances; and it is tamper evident. This structure is inspired by content-based slicing [45], and resembles a combination of a B$^+$-tree and a Merkle tree [44]. In *POS-Tree*, the node (i.e. page) boundary is defined as patterns detected from the contained entries, which avoids structural differences. Specifically, to construct a node, we scan the target entries until a pre-defined pattern occurs, and then create a new node to hold the scanned entries. Because of the distinct characteristics of leaf nodes and internal nodes, we define different patterns for them.

3.2.1 Tree Structure

Figure 2 illustrates the structure of a *POS-Tree*. Each node in the tree is stored as a page, which is the unit for deduplication. The node is terminated with a detected pattern, unless it is the last node of a certain level. Similar to a B$^+$-tree, an index node contains one entry for each child node. Each entry consists of a child node's identifier and the corresponding split key. To look up a specific key, we adopt the same strategy as in the B$^+$-tree, i.e., following the path guided by the split keys. *POS-Tree* is also a Merkle tree in the sense that the child node's identifier is the cryptographic hash value of the child (e.g., derived from SHA-1 hash function) instead of memory or file pointers. The mapping from the node identifier to storage pointer is maintained externally.

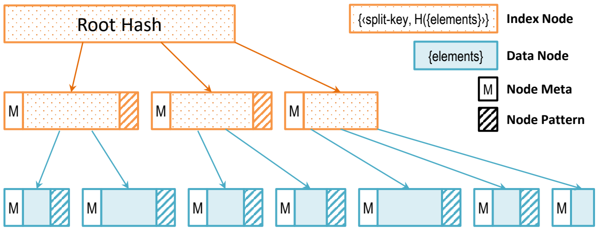


Figure 2: Pattern-Oriented-Splitting Tree (POS-tree) resembling a B$^+$-tree and Merkle tree.

3.2.2 Leaf Node Split

In order to avoid structural variance for leaf nodes, we define patterns similar to content-based slicing [45] used in file deduplication systems. These patterns help splitting the nodes into smaller ones of similar sizes. 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 occurs if and only if:
$$P(b_1...b_k) \text{ MOD } 2^q = 0$$
In other words, the pattern occurs when the function P returns 0 for the q least significant bits. This pattern can be implemented via rolling hashes (e.g. Rabin-Karp, cyclic polynomial and moving sum) which support continuous computation over sequence windows and offer satisfactory randomness. In particular, we use the *cyclic polynomial* [18] hash, 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 continuously for a sliding window:
$$P(b_1...b_k) = s(P(b_0...b_{k-1})) \oplus s^k(h(b_0)) \oplus s^0(h(b_k))$$
Each time, we remove the oldest byte and adds the latest one.

Initially, the entire list of data entries is treated as a byte sequence, and the pattern detection process scans it from the beginning. When a pattern occurs, a leaf node is created from recently scanned bytes. If a pattern occurs in the middle of an entry, the page boundary is extended to cover the whole entry, so that no entries are stored across multiple pages. In this way, each leaf node (except for the last node) ends with a pattern, as shown in Figure 2.

3.2.3 Index Node Split

The rolling hash 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-Trees*. Thus, for index nodes, we apply a simpler function Q that exploits the intrinsic randomness from cryptographic hashes used as child node ids. In particular, for a list of index entries, Q examines each child node id (i.e. a byte sequence) until a pattern occurs:
$$\texttt{id} \text{ MOD } 2^r = 0$$
When a pattern is detected, all scanned index entries are stored in a new index node.

Algorithm 1: *POS-Tree* Construction

```
Input: a list of data elements $data$
Output: id of constructed POS-Tree's root
PatternDetector $detector$;
List<Element> $elements$, $new\_entries$;
Node $node$;
id $last\_commit$;
$new\_entries$ = $data$;
/* use pattern P for leaf nodes                    */
$detector$ = new P();
do
    move all elements in $new\_entries$ to $elements$;
    for each $e$ in $elements$ do
        $node$.append($e$);
        feed $e$ into $detector$ to detect pattern;
        if pattern detected or is last element then
            $last\_commit$ = $node$.commit();
            add index entry of $node$ into $new\_entries$;
    /* use pattern Q for index nodes               */
    $detector$ = new Q();
    /* loop until root is found                    */
while $new\_entries$.size() > 1;
return $last\_commit$;
```

3.2.4 Construction and Update

Given the node splitting strategies above, a *POS-Tree* is constructed as follows. First, data records are sorted by key and treated as a sequence. Next, the pattern function P is applied to create a list of leaf nodes and respective index entries. After that, function Q is repeatedly applied on each level of index entries to construct index nodes, until the root node is reached. Algorithm 1 demonstrates this bottom-up construction. The expected node size is controlled by parameters q and r in pattern functions. To ensure that a node will not grow infinitely large, an additional constraint is enforced: the node size cannot be α times larger than the average size; otherwise it splits forcefully. The probability of force split is equal to $(1/e)^{\alpha}$, which can be very low (e.g. 0.03% when $\alpha = 8$).

To update a single entry, *POS-Tree* first seeks to the target node, applies the change, and finally propagates it to the index nodes in the path back to the root node. Copy-on-write is used to ensure that old nodes are not deleted. When modified, a node might split if a new pattern occurs, or merge with the next node if its pattern is destroyed. In any cases, at most two nodes are effected at each level. The update complexity is therefore $O(log(N))$ since the tree is probabilistically balanced. To further amortize cost from many changes, multi-update is supported, in which index nodes are updated only after all changes are applied on multiple data nodes.

POS-Tree enables a special type of update in which an entire set of records is exported, modified externally and then re-imported. The final re-import operation is efficient. In particular, *POS-Tree* rebuilds an entire new tree from the given records, but the tree shares most of its nodes with the old tree. Thanks to the *SIRI* properties, rebuilding a new tree has the same result as applying updates directly to the old tree.

3.2.5 Diff and Merge

POS-Tree supports fast *diff* operation which identifies the differences between two *POS-Tree* instances. Because two sub-trees with identical content must have the same root id, the diff operation can be performed recursively by following the sub-trees with different ids, and pruning ones with the same ids. The complexity of diff is therefore $O(D\log(N))$, where D is the number of different leaf nodes and N is the total number of data entries.

POS-Tree supports three-way merge which consists of a diff phase and a merge phase. In the first phase, two objects A and B are diffed against a common base object C, which results in Δ_A and Δ_B respectively. In the merge phase, the differences are applied to one of the two objects, i.e., Δ_A is applied to B or Δ_B is applied to A. In conventional approaches, the two phases are performed element-wise. In *POS-Tree*, both phases can be done efficiently at sub-tree level. More specifically, we do not need to reach leaf nodes during the diff phase, as the merge phase can be performed directly on the largest disjoint sub-trees that cover the differences, instead of on individual leaf nodes, as illustrated in Figure 3.

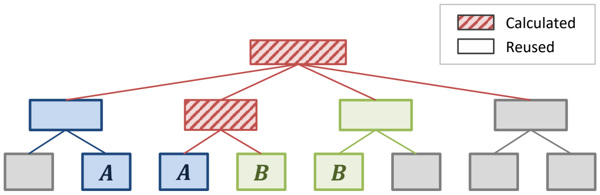


Figure 3: Three-way merge of two *POS-Trees* reuses disjointly modified sub-trees to build the merged tree.

3.2.6 Sequence POS-Tree

POS-Tree is designed for indexing records with unique keys, and therefore suitable for collection abstractions such as *Map* and *Set*. It can also be slightly modified to support sequence abstractions such as *List* and *Blob* (i.e. byte sequence). We call this variant *sequence POS-Tree*. Each index entry in this variant replaces the split key with a counter that indicates the total number of leaf-level data entries in that sub-tree. This allows for computing the path for positional accesses, e.g., read the i-th element. The diff operation is also different from the original *POS-Tree*. Finding the differences between two sequences is commonly calculated using edit distance, e.g., the *diff* tool in Linux [46]. The sequence *POS-Tree* is able to perform this operation recursively on index entries, instead of on the flattened sequence of data entries.

3.3 Generic Fork Semantics

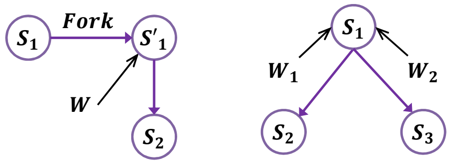
We propose a generic fork semantics that support both *fork on demand* (*FoD*) and *fork on conflict* (*FoC*). The application chooses which semantics to use, while the storage focuses on optimizing the fork related operations.

3.3.1 Fork on Demand

In this scenario, a branch is forked explicitly on the demand to create an isolated modifiable data copy. Every branch has a user-defined tag, thus we refer to it as *tagged* branch. The latest version of a branch is called the branch *head*, which is the only modifiable state. For example, in Figure 4(a) version S_1 from an existing branch is forked to a new branch. Then an update W is applied to the new branch creating a version S_2, which becomes the new branch's head. The most important operations are as follows:

- **Fork** – create an isolated modifiable branch from another branch (or a version) and attach a tag;
- **Read** – return committed data from a branch (or a version);
- **Commit** – update (or advance) a branch with new data;
- **Diff** – find the differences between branches (or versions);
- **Merge** – merge two branches and their commit histories;

Most versioning control systems such as git and Decibel follow this semantics and provide the same set of operations.



(a) (b)

Figure 4: Generic fork semantics supported for both (a) fork on demand and (b) fork on conflict.

3.3.2 Fork on Conflict

In this scenario, a branch is implicitly created from concurrent and conflicting modifications, in order to avoid blocking any operations and delay conflict resolutions. For example, in Figure 4(b) two conflicting updates W_1 and W_2 are applied to the head version S_1 concurrently. The result is that two different branches with heads S_2 and S_3 are created. Such branches can only be identified by their head versions, and thus we refer to them as *untagged* branches. The most important operations are as follows:

- **Read** – choose and read a version based on a policy which can be one of the following:
  - *any-branch*: read from any branch head;
  - *exact-version*: read the given version;
  - *version-descendant*: read from any branch head derived from the given version;
- **Commit** – update (or advance) a branch based on a policy, or create a new branch if the policy fails:
  - *exact-version*: write to the given version if it is a branch head;
  - *version-descendant*: write to a non-conflicting branch head derived from the given version;
- **ListBranches** – return all the branch heads;
- **Merge** – resolve conflicts and merge branches;

Many forkable applications can be implemented using the above operations. For example, a multi-master replicated data service can apply remote changes via Commit (*version-descendant*), specifying the last version that the remote node committed. A new branch is created if the changes conflict with local changes. Periodically, the service checks and resolves outstanding conflicts using ListBranches and Merge. Another example is cryptocurrency, in which whenever a client receives a block, it invoke Commit (*exact-version*) where the previous version is extracted from the block itself. The longest chain can be identified using ListBranches. The final example is TARDiS [21], following which we can extend our semantics to implement complex branch policies and support a wide range of consistency levels.

4. DATA MODEL AND APIS

In this section, we introduce *ForkBase*'s data model and supported operations, showing how it integrates above designs.

4.1 FNode

ForkBase adopts an extended key-value data model: each object is identified by a *key*, and contains a *value* of a specific *type*. A key may have multiple branches. Given a key we can retrieve not only the current value in each branch, but also its historical versions. Similar to other data versioning systems, *ForkBase* organizes versions in a directed acyclic graph (DAG) called *version derivation graph*. Each node in the graph is a structure called *FNode*, and it is associated with a unique identifier *uid*. Links between *FNode* represent their derivation relationships. The structure of a *FNode* is shown in Figure 5. The *context* field is reserved for application metadata, e.g., commit messages in git or nonce values for blockchain proof-of-work [28].

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

Figure 5: The *FNode* structure.

4.2 Tamper Evident Version

Each *FNode* is associated with a *uid* representing its version, which can be used to retrieve the value. The *uid* uniquely identifies both the object value and its derivation history, based on the content stored in the *FNode*. Two *FNode*s are considered equivalent, i.e., having the same *uid*, when they have both the same value and derivation history. This is due to the use of *POS-Tree* – a structurally invariant Merkle tree – to store the values. In addition, the derivation history is essentially a hash chain formed by linking the *bases* fields, thus two equal *FNode*s must have the same history.

It can be seen that *uid* is tamper evident. Given a *uid*, the user can verify the content and history of the returned *FNode*. This integrity property is guaranteed under the threat model that the storage is malicious, but the users keep track of the last *uid* of every branch that has been committed. This model is similar to that in fork-consistent storage systems [41], and does not provide other stronger guarantee, such as freshness in Concerto [12]. Instead of introducing a new tamper evidence design, *ForkBase* supports this property efficiently as a direct benefit from the *POS-Tree* design.

4.3 Value Type

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

Primitive types include simple values – *String*, *Tuple* and *Integer*. They are atomic values optimized for fast access. These values are not explicitly deduplicated, since the benefits of sharing small data does not offset the extra overheads. Apart from the basic *get* and *set* operations, many type-specific operations are provided, such as *append*, *insert* for *String* and *Tuple*, and *add*, *multiply* for numerical types.

Chunkable types are complex data structures – *Blob*, *List*, *Map* and *Set*. Each chunkable value is stored as a *POS-Tree* (or a sequence *POS-Tree*) and thus deduplicated. The chunkable types are suitable for data that may grow fairly large and have many updates. Reading a chunkable value simply returns a handler, while actual data pages are fetched gradually on demand through an iterator interface. Fine-grained access methods are naturally supported by the *POS-Tree*, such as *seek*, *insert*, *update* and *remove*.

The rich collection of built-in data types makes it easy to build high level data abstractions, such as relational tables and blockchains (§6). Note that some data types could have same logical representation but different performance trade-offs, for example *String* and *Blob*, or *Tuple* and *List*. The applications can flexibly choose those types that are more suitable for their workloads.

4.4 APIs

Table 1 lists the basic operations supported by *ForkBase*. A new *FNode* can be created given its value and the base *uid* from which

Table 1: *ForkBase* APIs with fork semantics.

	Method	FoD	FoC	Ref
Get	Get(key,branch)	✓		M1
	Get(key,uid)	✓	✓	M2
	Get(key,policy)		✓	M3
Put	Put(key,branch,value)	✓		M4
	Put(key,base_uid,value)		✓	M5
	Put(key,policy,value)		✓	M6
Merge	Merge(key,tgt_brh,ref_brh)	✓		M7
	Merge(key,tgt_brh,ref_uid)	✓		M8
	Merge(key,ref_uid1,...)		✓	M9
View	ListKeys()	✓	✓	M10
	ListTaggedBranches(key)	✓		M11
	ListUntaggedBranches(key)		✓	M12
Fork	Fork(key,ref_brh,new_brh)	✓		M13
	Fork(key,ref_uid,new_brh)	✓		M14
	Rename(key,tgt_brh,new_brh)	✓		M15
	Remove(key,tgt_brh)	✓		M16
Track	Track(key,branch,dist_rng)	✓		M17
	Track(key,uid,dist_rng)	✓	✓	M18
	LCA(key,uid1,uid2)	✓	✓	M19

```
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
FNode value = db.Get("my key", "new branch");
if (value.type() != Blob)
  throw TypeNotMatchError;
blob = value.Blob();
// Remove first 10 bytes and append new bytes
// 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 6: Fork and modify a *Blob* object.

it is derived (M5). Existing *FNode*s can be retrieved using their *uid*s (M2). A branch tag can be used instead of *uid*, in this case the branch head is returned for *Get* (M1) and used as the base version for *Put* (M4). When neither branch tag nor *uid* is specified, the default branch is used.

The fork related operations discussed in §3.3 can be mapped to these APIs. For FoD operations, a tagged branch can be forked from another branch (M13) or from a non-head version (M14). Commit and Read are supported by (M4) and (M1) respectively. The diff operations can be implemented by first finding the base version (M19) and then performing three-way diff (on *POS-Trees* for chunkable types). A tagged branch can merge with another branch using (M7) or with a specific version using (M8). In either case, only the active branch's head is updated such that the new head contains data from both branches. For FoC operations, Commit and Read are supported by (M6) and (M3) respectively. All the policy-based operations are actually implemented on top of version-based accesses (M2, M5), which also allows the application to implement its own policies. The branches are listed via (M12), and they can be merged in a single operation (M9). Each key in *ForkBase* has isolated space for both fork semantics, such that a branch update from one semantics will not affect the states of the other. As a result, a key can contain both tagged and untagged branches at the same time.

In summary, data in *ForkBase* can be manipulated at two granularities: at an individual object or at a branch of objects. *ForkBase* exposes easy-to-use interfaces that combine both object manipulation and branch management. Figure 6 shows an example of forking and editing a *Blob* object. Since *Put* is used for both insertion and update, its *value* field could be either an entire new value or the base object that has undergone a sequence of updates. We can commit multiple updates on the same object in a batch and *ForkBase* only retains the final version.

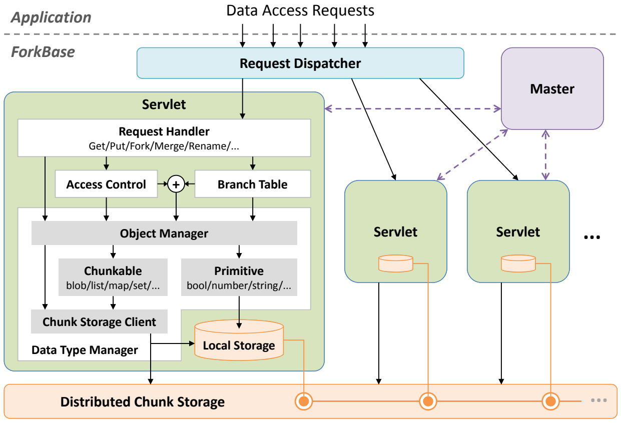


Figure 7: Architecture of a *ForkBase* cluster.

5. SYSTEM IMPLEMENTATION

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

5.1 Architecture

Figure 7 shows the architecture of a *ForkBase* cluster consisting of four main components: *master*, *dispatcher*, *servlet* and *chunk storage*. 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. A *servlet* further contains three sub-modules for executing requests: the *access controller* verifies request permission before execution; the *branch table* maintains branch heads for both tagged and untagged branches; and the *object manager* handles object manipulations, hiding the internal data representation from the main execution logic. The *chunk storage* persists and provides access to data chunks. All chunk storage instances form a large pool of shared storage, which is accessible by remote servlets. In fact, each servlet is co-located with a local chunk storage to enable fast data access and persistence. When *ForkBase* is used as an embedded storage, e.g., in blockchain nodes, only one servlet and one chunk storage are instantiated.

5.2 Internal Data Representation

Data objects are stored in the form of *data chunks*. A primitive object consists a single chunk, while a chunkable object comprises multiple chunks.

Chunk and cid. A *chunk* is the basic unit of storage in *ForkBase*, which contains a byte sequence. A chunk is uniquely identified by its *cid*, computed from the byte sequence in the chunk using a cryptographic hash function, e.g. SHA-256. Since the hash function is collision resistant, each chunk has a unique *cid*, i.e., two chunks with the same *cid* should contain identical byte sequences. The chunks are stored and deduplicated in chunk storage (§5.3) and can be retrieved via their *cid*s.

FNode and POS-tree. A *FNode* is serialized and stored as a *meta* chunk. The *uid* of the *FNode* is in fact an alias for the *meta* chunk's *cid*. A *POS-Tree* is stored in multiple chunks, one chunk per node. In particular, an index node is stored in an *index* chunk, and a leaf node in a *blob/list/map/set* chunk. The child node id stored in the index entry is the *cid* of the respective chunk.

Data Types. For a primitive object, its value is embedded in the *meta* chunk's *data* field for fast access. For a chunkable object, the *data* field contains a *cid* which indicates the root of the corresponding *POS-Tree*. Accessing a large chunkable object is efficient because only the relevant *POS-Tree* nodes are fetched on demand, as opposed to fetching the entire tree at once. By default, the expected chunk size is 4 KB for all *POS-Tree* nodes, but type-specific chunk sizes are also supported. For example, *blob* chunks storing content of large files can have large sizes, whereas *index* chunks may need smaller sizes since they only contain tiny index entries.

5.3 Chunk Storage

The chunk storage persists data chunks and supports retrieval using *cid*. It exposes a key-value interface, where the key is the *cid* of the committed chunk. The chunk storage is content addressable: it derives *cid* from content of the chunk. As a result, when a request contains an existing chunk, the storage will detect it and return immediately. The fact that the chunks are immutable is leveraged in two ways. First, the chunks are persisted in a log-structured layout which provides locality for consecutively generated chunks from a *POS-Tree*, and cached with a LRU policy. Second, each chunk is k-way replicated for better fault tolerance without introducing consistency issues, because there is no update. Delta encoding can be applied on similar chunks to further reduce space consumption [63]. We leave this enhancement for future work.

5.4 Branch Management

For each key there is a *branch table* that holds all its branches' heads. The branch table comprises two structures for tagged and untagged branches respectively.

TB-table. 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*. The *Put-Branch* operation (M4) first updates the value (in *POS-Tree*) and then creates a *FNode*, and finally replaces the old branch head with this new *FNode*'s *cid* in TB-table. The *Fork-Branch* operation (M13) simply creates a new table entry pointing to the referenced *FNode*. Therefore, fork operations are extremely lightweight in *ForkBase*. Concurrent updates on a tagged branch are serialized by the servlet. To prevent from overwriting others' changes by accident, additional *guarded* APIs are provided, which ensures that the *Put* succeeds only if the current branch head is not advanced after its last read.

UB-table. Untagged branches are maintained in a set structure called *UB-table*, in which each entry is simply a head *cid* for a conflicting branch. The *Put-Policy* (M6) and *Put-Version* (M5) operations update the UB-table accordingly. Once a new *FNode* is created, its *cid* is added to the UB-table, and its base *cid* is removed from the table. If the base *cid* is not found in the table, it means that the base version has already been derived by others, hence a new conflicting branch occurs. If the new *FNode* already exists in chunk storage (from equivalent operations), the UB-table simply ignores it.

Conflict Resolution. A three-way merge strategy is used in *Merge* (M7-M9) operations. To merge two branch heads v_1 and v_2, the *POS-Tree*s of three versions (v_1, v_2 and *LCA*$(v1, v2)$) are fed into the merge function. A conflict occurs if both branches modify a key (in *Map* and *Set*) or a position (in *List* and *Blob*). If the merge fails, it returns a conflict list, calling for conflict resolution. This can be handled at the application layer with the merged result sent back to the storage. In addition, simple conflicts can be resolved using built-in resolution functions (such as *append*, *aggregate* and *choose-one*). *ForkBase* also allows users to hook customized resolution functions which are executed upon conflicts.

5.5 Cluster Management

When *ForkBase* is deployed as a distributed service, it 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.

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 primitive objects or to track historical versions. In addition, servlets caches the frequently accessed remote chunks as they are immutable. When reading a *POS-Tree* node, request dispatchers forward *Get-Chunk* request directly to the chunk storage, bypassing the servlet.

6. APPLICATION DEVELOPMENT

In this section, we use *ForkBase* to build three applications: a blockchain platform, a wiki engine and a collaborative analytics application. We describe how it meets the applications' demands and reduces development efforts.

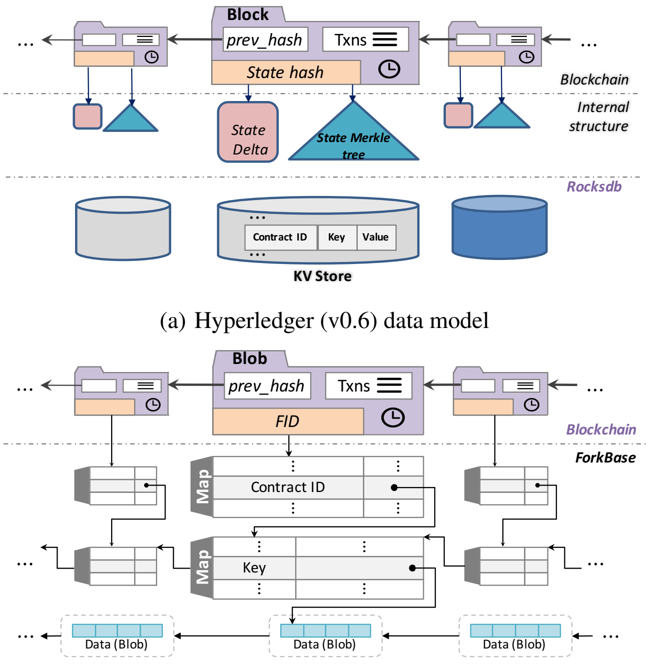
6.1 Blockchain

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. We refer readers to [25] for a more comprehensive treatment of the blockchain design space. In this paper, we focus on integrating *ForkBase* with Hyperledger for two reasons. First, Hyperledger 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 cryptocurrency.

Data Model in Hyperledger. Figure 8(a) illustrates the main data structures in Hyperledger v0.6[3]. The states are protected by a Merkle tree and any modification results in a new tree; the old values and trees 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 write is buffered in an in-memory structure. The system batches multiple transactions and then issues a commit. The commit operation first creates a new Merkle tree, followed by a new state and a new block, and finally writes all changes to the storage.

Blockchain Analytics. One initial goal of blockchain systems is to securely record the states, thus the designs are focused on tamper evidence and data versioning. As blockchain applications gain

[3] Newer versions, i.e. v1.0+, make significant changes to the data model, but they do not fit our definition of a blockchain system which requires a Byzantine fault tolerance consensus protocol.



(a) Hyperledger (v0.6) data model

(b) Hyperledger data model on *ForkBase*

Figure 8: Blockchain data models.

traction, the massive volume of data stored on the ledger present valuable opportunities for analytics [56, 1, 35]. However, traditional key-value stores which underlie current blockchain systems, are not optimized for analytics.

In this work we consider two representative analytical queries 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. 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.

Hyperledger on *ForkBase*. Figure 8(b) illustrates how we use *ForkBase* to implement Hyperledger's data structures. The key insight here is that a *FNode* 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 *uid* of the first-level *Map*. This *Map* contains key-value tuples where the key is the smart contract ID, and the value is the *uid* of the second-level *Map*. This second-level *Map* contains key-value tuples where the key is the data key, and the value is the *uid* of a *Blob* storing the state.

One immediate benefit of this model is that the code for maintaining data history and integrity becomes remarkably simple. In particular, for 18 lines of code that uses *ForkBase*, we eliminate 1900+ 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 *uid* stored in the latest block to get the latest *Blob* for the requested key. From there, we follow *base* version to retrieve the previous values. For block scan query, we follow the *uid* stored in the requested block to retrieve the second-level *Map*. We then iterate through all entries to obtain corresponding *Blob*s.

6.2 Wiki Engine

A wiki engine allows collaborative editing of documents (or wiki pages). Each entry contains a linear chain of versions. The wiki engine can be built on top of a multiversion key-value storage in which each entry maps to a wiki page. This can be directly implemented with Redis [8], for instance, using list-type values. However, in some existing implementations, e.g. Wikipedia, the versions are stored as independent records and therefore additional mechanisms are needed to capture and exploit their relationships.

Data Model in *ForkBase*. This multiversion key-value model fits naturally into *ForkBase*. Reading and writing an entry directly maps to *Get* (M1) and *Put* (M4) operations on the default branch. The *Blob* type is used instead of *String* because it helps eliminate duplicates across versions. Meta information, e.g. timestamp and author, can be stored in the *context* field. When accessing consecutive versions, *ForkBase* clients can reuse shared data chunks to serve out more quickly. Comparing two versions is efficiently supported by diff-ing two *POS-Tree*s. Finally, *ForkBase*'s two-level partitioning scheme helps scale skewed workloads from hot pages.

6.3 Collaborative Analytics

Collaborative analytics is an emerging application which allows a group of scientists (or analysts) to work on a shared dataset with different analysis goals [13, 48]. 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 dataset may be continually cleaned and enriched by other analysts. As the analysts simultaneously work on different versions, there is a clear need for versioning and on-demand fork semantics. Decibel [43] and OrpheusDB [31] are two state-of-the-art systems that support relational datasets. They provide SQL-like query interfaces, and use record-level delta encoding to eliminate duplicate records across versions.

Data Model in *ForkBase*. *ForkBase* has a rich collection of built-in data types, which offers the flexibility to construct different data models. In particular, 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 analytical queries.

Branch-related operations are readily supported by *ForkBase*'s fork-on-demand semantics. Other common operations such as data transformation, analytical queries, and version comparisons, are easy to implement as well. For example, *POS-Tree*s enable fast update to a table, as well as point queries, range queries and version diffs. While other dataset management systems require version reconstruction upon access, *ForkBase* enables direct access to any data pages in any versions. Besides, *POS-Tree*'s deduplication works across datasets, which results in significantly lower storage requirement at scale.

7. 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 §6 in terms of storage consumption and query efficiency. We compare them against 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

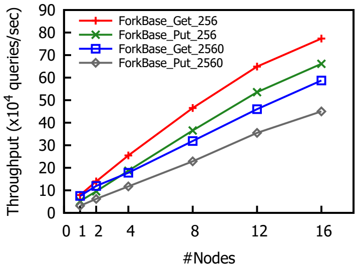


Figure 9: Scalability with multiple servlets.

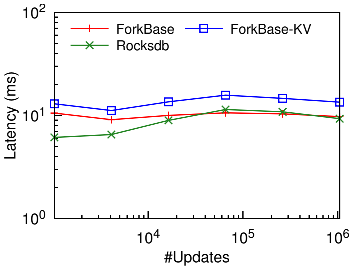


Figure 10: Latency of blockchain commits (b=50, r=w=0.5).

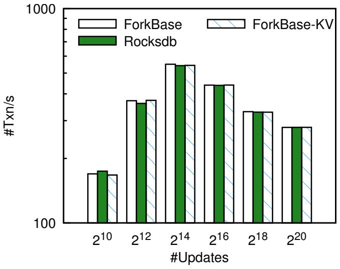


Figure 11: Client perceived throughput (b=50, r=w=0.5).

Table 2: 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

systems, all servlets in *ForkBase* are configured with one request execution thread and two request parsing threads. Both leaf and index page sizes in the *POS-Tree* are set to 4KB.

7.1 Micro-Benchmark

We deployed one *ForkBase* servlet and used multiple clients for sending requests. We benchmark nine operations and Table 2 lists the aggregated throughput and average latency over 5 million requests using 32 clients. We observe that large requests achieve higher network throughput – 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 the 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. In this case, network delays have major contribution to the final latency.

Table 3 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 9 shows the almost linear scalability for both *Put* and *Get* operations (with value size of 256 and 2560 bytes). The fact that *ForkBase* scales almost linearly is expected because there is no communication between the servlets.

7.2 Blockchain

We compare *ForkBase*-backed Hyperledger 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, which is designed to stress the storage. 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. We issued up to 1 million write operations with 1KB-size values for each test. For the blockchain configuration, we deployed one server, varied the maximum block size b and kept the default values for the other settings.

Table 3: 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

7.2.1 Blockchain Operations

Figure 10 shows the 95^{th} percentile latency of commit operations. Read and write operations are orders of magnitude faster because of memory buffers, and we omit the details here. Though Rocksdb is designed for fast batch commits, *ForkBase* and Rocksdb still have similar latencies. Both are better than *ForkBase*-KV since using *ForkBase* as a pure key-value store introduces overhead from conducting hash computation both inside and outside of the storage layer. Figure 11 shows the overall throughput, measured as the total number of transactions committed 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.

7.2.2 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 12 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

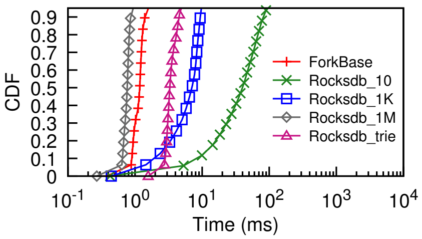
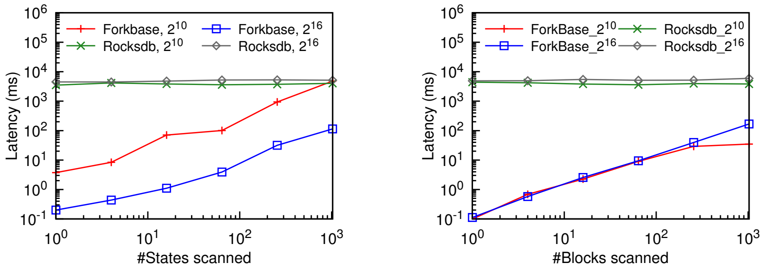


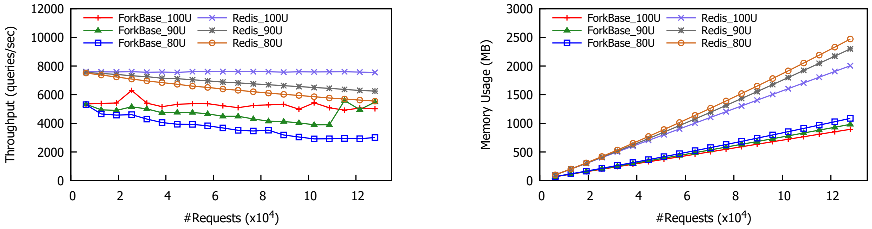
Figure 12: Commit latency distribution with different Merkle trees.



(a) State Scan

(b) Block Scan

Figure 13: Scan queries. ’X, 2^y’ means using storage X, and 2^y keys.



(a) Throughput

(b) Storage Consumption

Figure 14: Performance of editing wiki pages.

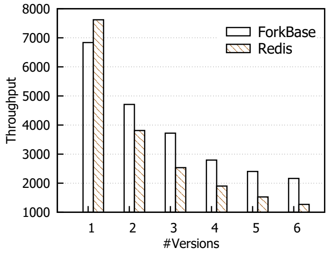


Figure 15: Throughput of read consecutive versions of a wiki page.

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.

7.2.3 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 13(a) compares the performance for state scan query. The x axis represents the number of unique keys scanned per query. For a small number of keys, the difference between *ForkBase* and Rocksdb is up to four 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 13(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. For 2^{10} unique key, the scanning cost peaks around 2500 blocks, because they contains all the keys. We note that the gap is at least two orders of magnitudes regardless of how many blocks are being scanned.

7.3 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. In all the tests, each page has approximately 40 versions in average, resulting in more than 2GB of application data. We enabled data persistence in Redis to ensure that all data are written to the disk.

7.3.1 Edit and Read Pages

Figure 14(a) shows the throughput of editing pages, in which *xU* indicates the ratio of in-place updates against insertions (e.g., *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 14(b), *ForkBase* consumes 55% less storage than Redis, thanks to the deduplication using *POS-Trees*. The performance of reading wiki pages is illustrated in Figure 15. It can be seen that Redis is fast for reading a single 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 reused from the clients. When reading an old version, a large number of chunks may have already been cached, resulting in lower read latencies and network traffics.

7.3.2 Hot Pages

We deployed a distributed wiki service in a 16-node cluster, and ran a skewed workload (zipf = 0.5). Figure 16 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.

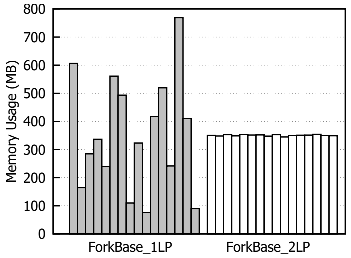
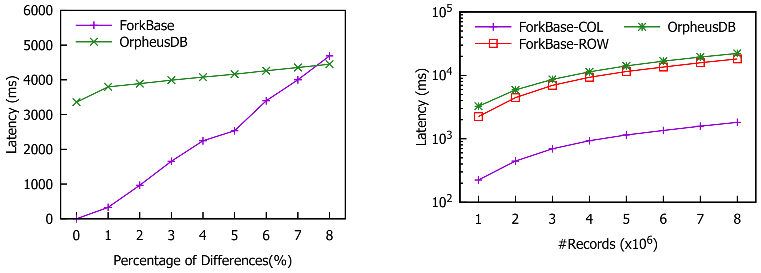


Figure 16: Storage size distribution in skewed workloads.



(a) Diff

(b) Aggregation

Figure 17: Performance of querying datasets.

7.4 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. 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 space consumption of the initial commit of the dataset is 927MB in *ForkBase* and 1167MB in OrpheusDB. We focus on queries that do not modify the dataset, because OrpheusDB is not designed for efficient commit operations.

7.4.1 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 version-to-record 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, as we need to traverse more tree nodes.

7.4.2 Analytical Queries

Figure 17(b) compares the performance of aggregation queries on the numerical fields. For *ForkBase*, both row (*ForkBase*-ROW) and column (*ForkBase*-COL) layouts were used. It can be seen that *ForkBase*-ROW and OrpheusDB have similar performance, whereas *ForkBase*-COL has 10× better performance. The gap is due to the physical layouts and the fact that *ForkBase* does not need to checkout the version and reconstruct from its deltas. More specifically, even though *ForkBase*-ROW does not explicitly reconstruct a version, the target values are scattered over all data pages, thus it incurs the same I/O cost as OrpheusDB. However, *ForkBase*-COL can efficiently locate target columns from the top-level *Map*, and then iterate over the values in the *List*s.

8. RELATED WORK

Dataset Versioning. Decibel [43] and OrpheusDB [31] are state-of-the-art systems built for relational dataset versioning, with git-like version control and SQL-like query interfaces. They use record-level delta encoding to remove duplicates between consecutive versions. The trade-off between space saving and reconstruction cost of such approaches has been studied in [14]. DEX [17] executes queries directly on delta-based storage to avoid version reconstruction. dbDedup [63] extends delta encoding to achieve global deduplication via similarity search. In contrast, *ForkBase* uses page-level deduplication which helps detect global duplicates and direct data access without checkout or reconstruction.

Persistent data structures. Persistent data structures, which persists all the states and thus support versioning, have been widely studied [27, 39, 53, 33]. Purely functional data structures [49] is a special class of such structures, which is mainly used in functional programming languages. *POS-Tree* can be seen as a purely functional data structure with additional *SIRI* properties, which is effective for global deduplication.

Blockchain storage. Current blockchain systems use simple key-value storage such as LevelDB [6] and RocksDB [9] as their storage backend. They implement the log-structured-merge tree [50] which consists of exponentially-sized index components that are merged periodically. They achieve superior write performance from sequential I/Os, but lacking of tamper evidence or analytics supports. *ForkBase* is designed to replace them as a more natural and efficient storage for blockchain systems.

Data Integrity. Outsourced database systems [32, 40, 51] often rely on Merkle trees to achieve data integrity, as does *ForkBase*. Recently, Concerto [12] employs trusted hardware (e.g. Intel SGX) to achieve both integrity and freshness against malicious storage providers. Although orgthogonal to *ForkBase*, Concerto's design can be integrated to enable stronger security.

9. CONCLUSIONS

In this paper, we identified three common properties in blockchain and forkable applications: data versioning, fork semantics and tamper evidence. We proposed a new index class called *SIRI* that is effective at detecting duplicate content among multiversion data. We designed *POS-Tree*, an instance of *SIRI*, that additionally offers tamper evidence. We described a generic fork semantics to support a broad range of applications. We designed and implemented *ForkBase* that integrates these ideas and is able to deliver better performance than ad-hoc, application-layer solutions. By implementing three applications on top of *ForkBase*, we demonstrated that our storage simplifies application logic, 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 and query efficiency. In summary, *ForkBase* provides a powerful building block for blockchains and the emerging forkable applications [10].

ACKNOWLEDGMENTS

This work was supported by the Singapore Ministry of Education Tier-3 Grant MOE2017-T3-1-007 (WBS No. R-252-000-A08-112). Meihui Zhang was supported by China Thousand Talents Program for Young Professionals (3070011181811). Gang Chen was supported by National Key Research and Development Program of China (2017YFB1201001). We thank the anonymous reviewers for their insightful feedback, and thank Wanzeng Fu, Ji Wang and Hao Zhang for their early contributions.

10. 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] Redis. http://redis.io.

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

[10] The Morning Paper review on ForkBase. https://blog.acolyer.org/2018/06/01/forkbase-an-efficient-storage-engine-for-blockchain-and-forkable-applications.

[11] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. *SIGMOD Record*, 15(2):96–107, 1986.

[12] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *SIGMOD*, pages 251–266, 2017.

[13] 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.

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

[15] 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*, pages 49–60, 2013.

[16] 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. *ACM TOCS*, 26(2):4, 2008.

[17] A. Chavan and A. Deshpande. Dex: Query execution in a delta-based storage system. In *SIGMOD*, pages 171–186, 2017.

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

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

[20] 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. *PVLDB*, 1(2):1277–1288, 2008.

[21] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement. Tardis: A branch-and-merge approach to weak consistency. In *SIGMOD*, pages 1615–1628, 2016.

[22] 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*, volume 41, pages 205–220, 2007.

[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*, pages 1085–1100, 2017.

[24] T. T. A. Dinh, J. Wang, S. Wang, G. Chen, W.-N. Chin, Q. Lin, B. C. Ooi, P. Ruan, K.-L. Tan, Z. Xie, and M. Zhang. UStore: A distributed storage with rich semantics. *CoRR*, abs/1702.02799, 2017.

[25] 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 (TKDE)*, pages 1366–1385, 2018.

[26] 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 Internet Measurement Conference*, pages 481–494, 2012.

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

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

[29] 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.

[30] R. G. Guy, J. S. Heidemann, W.-K. Mak, T. W. Page Jr, G. J. Popek, D. Rothmeier, et al. Implementation of the Ficus replicated file system. In *USENIX Summer*, pages 63–72, 1990.

[31] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. Parameswaran. OrpheusDB: Bolt-on versioning for relational databases. *PVLDB*, 10(10):1130–1141, 2017.

[32] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *ICDE*, pages 529–540, 2013.

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

[34] M. Kallahalla, E. Riedely, R. Swaminathan, Q. Wangz, and K. Fux. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, pages 29–42, 2003.

[35] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan. BlockSci: Design and applications of a blockchain analysis platform. *CoRR*, abs/1709.02489, 2017.

[36] J. Katz and Y. Lindell. Introduction to modern cryptography. *CRC Press*, 2014.

[37] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[38] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[39] S. Lanka and E. Mays. Fully persistent B+-trees. In *SIGMOD*, pages 426–435, 1991.

[40] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, pages 121–132, 2006.

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

[42] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *SOSP*, pages 401–416, 2011.

[43] 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.

[44] 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*, pages 369–378, 1988.

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

[46] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

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

[48] F. A. Nothaft, M. Massie, T. Danford, and et al. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*, pages 631–646, 2015.

[49] C. Okasaki. Purely functional data structures. *Cambridge University Press*, 1999.

[50] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[51] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, pages 407–418, 2005.

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

[53] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3(4), 2008.

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

[55] 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.

[56] S. Shah, A. Dockx, A. Baldet, F. Bi, C. Allchin, S. Misra, M. Huebner, B. Sherpherd, and B. Holroyd. Unlocking economic advantage with blockchain: a guide for asset managers. *Oliver Wyman and JP Morgan*, 2016.

[57] Y. Sompolinsky and A. Zohar. Secure high-rate transaction processing in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527, 2015.

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

[59] M. Stonebraker and L. A. Rowe. The design of the POSTGRES. In *SIGMOD*, pages 340–355, 1986.

[60] 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.

[61] 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.

[62] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[63] L. Xu, A. Pavlo, S. Sengupta, and G. R. Ganger. Online deduplication for databases. In *SIGMOD*, pages 1355–1368, 2017.