ForkBase: Immutable, Tamper-evident Storage Substrate for Branchable Applications

Qian Lin[1], Kaiyuan Yang[1], Tien Tuan Anh Dinh[2], Qingchao Cai[6], Gang Chen[3], Beng Chin Ooi[1], Pingcheng Ruan[1], Sheng Wang[5], Zhongle Xie[1], Meihui Zhang[4], Olafs Vandans[7]

[1]National University of Singapore [2]Singapore University of Technology and Design [3]Zhejiang University
[4]Beijing Institute of Technology [5]Alibaba Group [6]Hudson River Trading Singapore [7]Synspective Inc.

[1]{linqian, yangky, ooibc, ruanpc, zhongle}@comp.nus.edu.sg, [2]dinhtta@sutd.edu.sg, [3]cg@zju.edu.cn,
[4]meihui_zhang@bit.edu.cn, [5]sh.wang@alibaba-inc.com, [6]qcai@hudson-trading.com, [7]to@olafs.eu

***Abstract*—Data collaboration activities typically require systematic or protocol-based coordination to be scalable. Git, an effective enabler for collaborative coding, has been attested for its success in countless projects around the world. Hence, applying the Git philosophy to general data collaboration beyond coding is motivating. We call it *Git for data*. However, the original Git design handles data at the file granule, which is considered too coarse-grained for many database applications. We argue that Git for data should be co-designed with database systems. To this end, we developed ForkBase to make Git for data practical. ForkBase is a distributed, immutable storage system designed for data version management and data collaborative operation. In this demonstration, we show how ForkBase can greatly facilitate collaborative data management and how its novel data deduplication technique can improve storage efficiency for archiving massive data versions.**

I. Introduction

Data analytics and machine learning activities generally target at insights from data and further exploit them to enhance applications. Processing on the same specific dataset usually involves multiple disciplines that run analytics or data engineering independently. And a collaboration is established with respect to the multi-entity efforts towards common goals. Efficient data collaboration comes from coordination and storage support. Most of the existing solutions for collaborative data coordination are built within the application. Such ad-hoc approach not only wastes development effort that is hardly reusable across different applications, but also misses the opportunity to optimize the underlying storage. Therefore, collaboration-oriented data management must rely on systematic or protocol-based coordination for scaling.

Git has been one of the most productive solutions in the practice of collaborative code management. This has been attested by its widespread adoption in countless projects around the world. In the context of database systems, many data collaboration applications have similar collaborative coordination requirements that Git could potentially fulfill. Hence, bringing the Git semantics into database management would benefit many kinds of branchable data applications. We call such integration *Git for data*. Table I briefly compares state-of-the-art Git for data systems.

Towards practical Git for data, we developed ForkBase [1] which is collaboration-centric by design and facilitates data forking, data version management and access control for multi-tenant data analytics. ForkBase provides Git-like operations, but focuses on data, its security, immutability and provenance. Like Git, data in ForkBase is multi-versioned, and each version uniquely identifies the data content and its history.

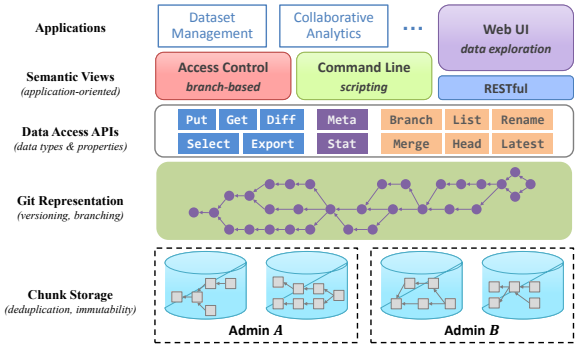

Fig. 1. Overview of the ForkBase architecture.

II. System Overview

ForkBase is a distributed storage system designed for data version management and data collaborative operation. It internally implements Git-compatible data version control and branch management based on Merkle directed acyclic graph (DAG), which empowers tamper evidence and efficient tracking of data provenance. Moreover, ForkBase is endowed with a novel content-based data deduplication technology that can remarkably reduce data redundancy between different data versions in the physical storage as well as efficiently support fast differential queries between data versions.

The overall architecture of ForkBase is illustrated in Fig. 1. At the bottom layer, data are deduplicated at the chunk level. At the representation layer, versions and branches are organized based on Merkle DAG. The API layer exposes interfaces for data manipulation, data property tracking and version/branch management. Supported data types include primitives (string, number, boolean), blob, map, set and list, as well as composite data structures built on them (e.g., relational table). The semantic view layer includes access control, command-line interface and RESTful APIs to support applications at the

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00153

TABLE I
COMPARISON WITH RELATED DATA VERSIONING SYSTEMS

System	*Data Model*	*Deduplication*	*Tamper Evidence*	*Branching*
ForkBase [1]	structured/unstructured, immutable	page level	root hash of Merkle DAG	Git-like
DataHub [2] & Decibel [3]	structured (table), mutable	table oriented	none	ad-hoc
OrpheusDB [4]	structured (table), mutable	table oriented	none	ad-hoc
MusaeusDB [5]	structured (table), mutable	table oriented	none	none
RStore [6]	unstructured, mutable	key-value	none	ad-hoc

top layer, such as a web-based UI for data exploration across versions and branches.

The aforementioned key features of ForkBase are fundamentally accredited to its novel content-addressable indexing technique. In the following, we first elaborate on the design of index, and then describe how it empowers ForkBase to achieve effective data deduplication and fast differential query between data versions, as well as tamper evidence.

A. POS-Tree

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. ForkBase addresses this issue by introducing the Pattern-Oriented-Split Tree (POS-Tree). POS-Tree implements the Structurally-Invariant Reusable Index (SIRI) [1] which facilitates page sharing among different index instances.

Definition 1. SIRI Let $\mathcal{I}$ be an index structure. An instance I of $\mathcal{I}$ stores a set of records $R(I) = \{r_1, r_2, \ldots, r_n\}$. The internal structure of I consists of a collection of pages (i.e. index and data pages) $P(I) = \{p_1, p_2, \ldots, p_m\}$. Two pages are equal if they have identical content and hence can be shared. $\mathcal{I}$ is called an instance of SIRI if it has the following properties:

(1) Structurally invariant: For any instance $I_1, I_2 \in \mathcal{I}$,

$$R(I_1) = R(I_2) \Longleftrightarrow P(I_1) = P(I_2) \tag{1}$$

(2) Recursively identical: For any instance $I_1, I_2 \in \mathcal{I}$ such that $R(I_2) = R(I_1 + r)$ for any record $r \notin I_1$,

$$|P(I_2) - P(I_1)| \ll |P(I_2) \cap P(I_1)| \tag{2}$$

(3) Universally reusable: For any instance $I_1 \in \mathcal{I}$ and page $p \in P(I_1)$, there exists another instance $I_2 \in \mathcal{I}$ such that

$$(|P(I_2)| > |P(I_1)|) \wedge (p \in P(I_2)) \tag{3}$$

Specifically, Property (1) 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. Property (2) 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, Property (3) means that 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.

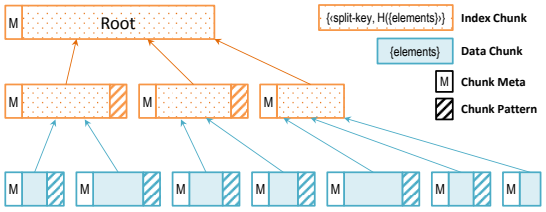

Fig. 2. Pattern-Oriented-Split Tree (POS-Tree).

POS-Tree being an instance of SIRI inherits the above properties. Moreover, POS-Tree is additionally endowed with the following three 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 resembles a combination of a B$^+$-tree and a Merkle tree [7]. 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.

Fig. 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-256 hash function) instead of memory or file pointers. The mapping from the node identifier to storage pointer is maintained externally.

In order to avoid structural variance for POS-Tree nodes, we define patterns similar to content-based slicing [8] used in file deduplication systems. These patterns help split the nodes into smaller sizes on average. Given a k-byte sequence $(b_1, b_2, \ldots, b_k)$, let Φ 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:

$$\Phi(b_1, b_2, \ldots, b_k) \text{ MOD } 2^q = 0$$

In other words, the pattern occurs when the function Φ returns 0 for the q least significant bits. This pattern can be implemented via rolling hashes which support continuous computation over sequence windows and offer satisfactory randomness.

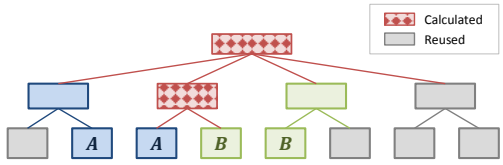

Fig. 3. Three-way merge of two POS-Trees reuses disjointly modified sub-trees to build the merged tree.

In particular, POS-Tree applies the *cyclic polynomial hash*, which is of the form:

$$\Phi(b_1 \dots b_k) = \delta(\Phi(b_0 \dots b_{k-1})) \oplus \delta^k(\Gamma(b_0)) \oplus \delta^0(\Gamma(b_k))$$

where $\oplus$ is the exclusive-or operator, and Γ maps a byte to an integer in $[0, 2^q)$. δ 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. As a consequence, for each iteration the above recursion removes the oldest byte and adds the new 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 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 node (except for the last node) ends with a pattern.

B. 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 $O(Dlog(N))$, 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 diff 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 Fig. 3.

C. Storage Efficiency

Another advantage brought by POS-Tree is its powerful data deduplication ability. As illustrated in Fig. 2, data are split into chunks, each of which is immutable after complete construction and uniquely identified by its SHA-256 hash. Chunks are materialized into the key-value based physical storage so that each distinct chunk is stored exactly once and can be shared across different data objects according to the identical content, e.g., in the example shown in Fig. 3.

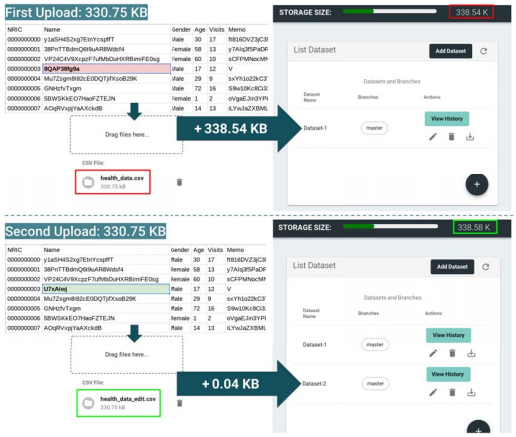

Fig. 4. Fine-grained data deduplication in ForkBase.

D. Tamper-evident Version

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 links between *FNode*s represent their derivation relationships. 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.

Each *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 following threat model: the storage is malicious, but the users keep track of the latest *uid* of every branch that has been committed. Instead of introducing a new tamper evidence design, ForkBase supports this property efficiently as a direct benefit from the POS-Tree design.

III. Demonstration

Our demonstration using the ForkBase Web UI aims to highlight some novel aspects of the ForkBase system. In particular, we focus on the capability of ForkBase to deduplicate data, perform fast differential query, branch dataset with Git-compatible semantics, and track versions with tamper evidence.

A. Data Deduplication

First, we showcase the fine-grained data deduplication feature of ForkBase. As shown in Fig. 4, two external CSV datasets with a single-word difference in terms of text content are loaded into ForkBase as two separate datasets. Loading the

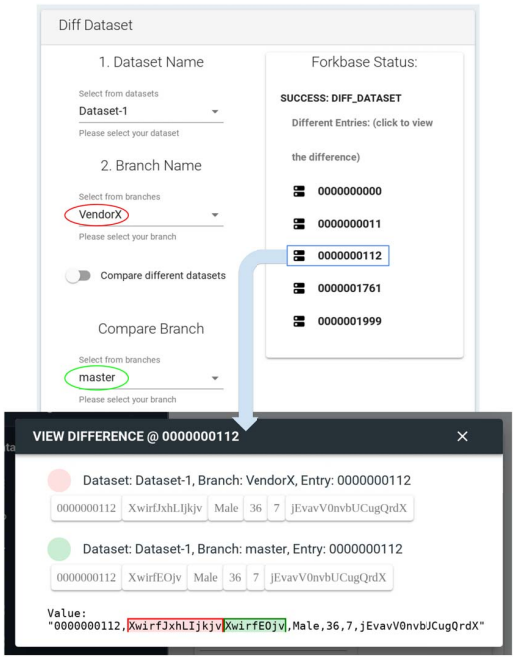

Fig. 5. An example of differential query in ForkBase.

first dataset increases 338.54 KB to the storage, but afterwards loading the second dataset only increases 0.04 KB. As the two datasets share a large portion of duplicated data, ForkBase can effectively detect such redundancy and consequently store only the marginal difference into the underlying immutable storage for the second loading. This manifests that ForkBase can significantly improve storage efficiency through its fine-grained data deduplication.

B. Fast Differential Query

ForkBase is able to perform fast differential query to retrieve differences among data versions and branches. To showcase such feature, we designed the ForkBase Web UI to visualize differences between datasets and their branches. For example, Fig. 5 shows the result of performing Diff operation between the master and VendorX branches of Dataset-1. Data differences are highlighted at multiple scopes, e.g., from dataset to data entry. This is akin to the Git-diff utility which helps user identify the changes of data.

C. Tamper Evidence and Validation

Along with data updates, each Put operation is stamped with a unique version that is appended to the corresponding branch of the dataset, as shown in Fig. 6. Data versions in ForkBase are generated according to the Merkle root hash of data chunks [1, 7], and encoded using the RFC 4648 Base32 alphabet [9]. Such versioning scheme enables ForkBase to provide tamper evidence against malicious storage providers. Given a version, the application can fetch the corresponding data from the storage provider (i.e., physical storage) and validate the content and its history by checking whether the Merkle root hash calculated on the spot is identical to the data

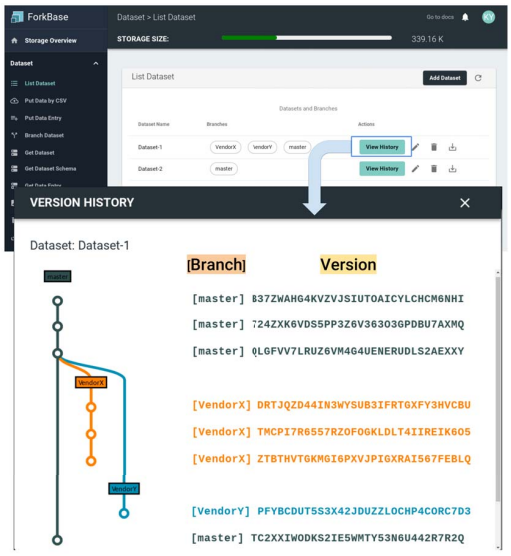

Fig. 6. Versioning for validation and tamper evidence.

version. This guarantees data stored in ForkBase is tamper-proof in spite of the underlying storage infrastructure.

IV. Conclusion

This demonstration sheds light on the basic workflow of Git-like data management for collaborative data processing. By pushing down the Git-compliant versioning and branching semantics from the application layer to the storage layer, ForkBase benefits various kinds of branchable applications built on top of it with reduced development effort. The fine-grained data deduplication feature of ForkBase favors improved storage efficiency in collaborative data activities.

Acknowledgment

This work is supported by Singapore Ministry of Education (MOE) Academic Research Fund Tier 3 under MOE's official grant number MOE2017-T3-1-007.

References

[1] S. Wang *et al.*, "ForkBase: An efficient storage engine for blockchain and forkable applications," *PVLDB*, vol. 11, no. 10, pp. 1137–1150, 2018.

[2] A. P. Bhardwaj *et al.*, "DataHub: Collaborative data science & dataset version management at scale," in *Proc. of CIDR*, 2015.

[3] M. Maddox *et al.*, "Decibel: The relational dataset branching system," *PVLDB*, vol. 9, no. 9, pp. 624–635, 2016.

[4] S. Huang *et al.*, "OrpheusDB: Bolt-on versioning for relational databases," *PVLDB*, vol. 10, no. 10, pp. 1130–1141, 2017.

[5] M. E. Schule *et al.*, "Versioning in main-memory database systems: From MusaeusDB to TardisDB," in *Proc. of SSDBM*, 2019.

[6] S. Bhattacherjee *et al.*, "RStore: A distributed multi-version document store," in *Proc. of ICDE*, 2018.

[7] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proc. of CRYPTO*, 1988.

[8] A. Muthitacharoen *et al.*, "A low-bandwidth network file system," in *Proc. of SOSP*, 2001.

[9] S. Josefsson, "The base16, base32, and base64 data encodings," https://tools.ietf.org/html/rfc4648, 2006.