Analysis of Indexing Structures for Immutable Data

Cong Yue
National University of Singapore
yuecong@comp.nus.edu.sg

Zhongle Xie
National University of Singapore
zhongle@comp.nus.edu.sg

Meihui Zhang
Beijing Institute of Technology
meihuizhang@bit.edu.cn

Gang Chen
Zhejiang University
cg@zju.edu.cn

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

Sheng Wang
Alibaba Group
sh.wang@alibaba-inc.com

Xiaokui Xiao
National University of Singapore
xkxiao@nus.edu.sg

ABSTRACT

In emerging applications such as blockchains and collaborative data analytics, there are strong demands for data immutability, multi-version accesses, and tamper-evident controls. To provide efficient support for lookup and merge operations, three new index structures for immutable data, namely Merkle Patricia Trie (MPT), Merkle Bucket Tree (MBT), and Pattern-Oriented-Split Tree (POS-Tree), have been proposed. Although these structures have been adopted in real applications, there is no systematic evaluation of their pros and cons in the literature, making it difficult for practitioners to choose the right index structure for their applications.

To alleviate the above problem, we present a comprehensive analysis of the existing index structures for immutable data, and evaluate both their asymptotic and empirical performance. Specifically, we show that MPT, MBT, and POS-Tree are all instances of a recently proposed framework, dubbed *Structurally Invariant and Reusable Indexes (SIRI)*. We propose to evaluate the SIRI instances on their index performance and deduplication capability. We establish the worst-case guarantees of each index, and experimentally evaluate all indexes in a wide variety of settings. Based on our theoretical and empirical analysis, we conclude that POS-Tree is a favorable choice for indexing immutable data.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389773

CCS CONCEPTS

• **Information systems → Database performance evaluation**; *Deduplication*; *Version management*; B-trees.

KEYWORDS

indexing, immutable data, deduplication, versioning

ACM Reference Format:
Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. 2020. Analysis of Indexing Structures for Immutable Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3318464.3389773

1 INTRODUCTION

Accurate history of data is required for auditing and tracking purposes in numerous practice settings. In addition, data in the cloud is often vulnerable to malicious tampering. To support data lineage verification and mitigate malicious data manipulation, data immutability is essential for applications, such as banking transactions and emerging decentralized applications (e.g., blockchain, digital banking, and collaborative analytics). From the data management perspective, data immutability leads to two major challenges.

First, it is challenging to cope with the ever-increasing volume of data caused by immutability. An example is the sharing and storage of the data for healthcare analytics. Data scientists and clinicians often make relevant copies of current and historical data in the process of data analysis, cleansing, and curation. Such replicated copies could consume an enormous amount of space and network resources. To illustrate, let us consider a dataset that has 100,000 records initially, and it receives 1,000 record updates in each modification. Figure 1 shows the space and time required to handle the increasing number of versions[1]. Observe that (i) the space and time overheads are significant if all versions are stored separately,

[1] Run with Intel(R) Xeon(R) E5-1620 v3 CPU and 1 Gigabit Ethernet card.

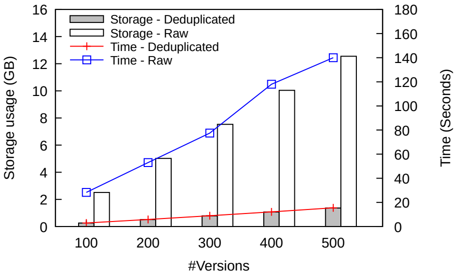


Figure 1: Data storage and transmission time improved by deduplication

and (ii) such overheads could be considerably reduced if we can *deduplicate* the records in different versions.

The second challenge is that in case a piece of data is tampered with (e.g., malicious manipulation of crypto-currency wallets or unauthorized modifications of patients' lab test data), we have to detect it promptly. To address this challenge, the system needs to incorporate tamper-resistant techniques to support the authentication and recovery of data, to ensure data immutability. Towards this end, a typical approach is to adopt cryptographic methods for tamper mitigation, which, however, considerably complicates the system design.

Most existing data management solutions tackle the above two challenges separately, using independent orthogonal methods. In particular, they typically (i) ensure tamper evidence using cryptographic fingerprints and hash links [17], and (ii) achieve deduplication with delta encoding [11, 13]. Such decoupled design incurs unnecessary overheads that could severely degrade the system performance. For example, in state-of-the-art blockchain systems such as Ethereum [5] and Hyperledger [6], tamper evidence is externally defined and computed on top of the underlying key-value store (e.g., LevelDB [2] or RocksDB [3]), which leads to considerable processing costs. In addition, delta-encoding-based deduplication (e.g., in Decibel [13]) requires a reconstruction phase before an object can be accessed, which renders data accessing rather inefficient.

Motivated by the above issues, recent work [6, 12, 21, 22] has explored data management methods to provide native supports for both tamper evidence and deduplication features. This results in three new index structures for immutable data, namely, *Merkle Patricia Trie (MPT)* [22], *Merkle Bucket Tree (MBT)* [6], and *Pattern-Oriented-Split Tree (POS-Tree)* [21]. To the best of our knowledge, there is no systematic comparison of these three index structures in the literature, although there exists experimental study in other indexing features [24]. Hence, the characteristics of each structure are not fully understood. This renders it difficult for practitioners to choose the right index structure for their applications.

To fill the aforementioned gap, this paper presents a comprehensive analysis of MPT, MBT, and POS-Tree. Specifically, we make the following contributions:

- We show that MPT, MBT, and POS-Tree are all instances of a recently proposed framework, named Structurally Invariant and Reusable Indexes (*SIRI*) [21]. Based on this, we identify the common characteristics of them in terms of tamper evidence and deduplication.
- We propose a benchmarking scheme to evaluate *SIRI* instances based on five essential metrics: their efficiency for four index operations (i.e., lookup, update, comparison, and merge), as well as their *deduplication ratios*, which is a new metric that we formulate to quantify each index's deduplication effectiveness. We establish the worst-case guarantee of each index in terms of these five metrics.
- We experimentally evaluate all three indexes in a variety of settings. We demonstrate that they perform much better than conventional indexes in terms of the effectiveness of deduplication. Based on our experimental results, we conclude that POS-Tree is a favorable choice for indexing immutable data.

More detailed analysis and experimental results are presented in the technical report [25].

2 RELATED WORK

2.1 Immutability and Tamper Evidence

Immutable data are becoming common in a highly regulated industry and emerging applications. For example, blockchains [5, 6, 9, 17] maintain immutable ledgers, which keep all historical versions of the system status. LineageChain [19] enables fine-grained, secure and efficient data provenance on blockchain and needs immutability for verification. Similarly, collaborative applications [1, 8] maintain the whole evolutionary history of datasets and the analytic results derived, so as to enable provenance-related functionalities, such as tracking, branching, and rollback.

In addition, applications such as digital banking [4] and blockchains [5, 6, 17] demand the system should maintain the accurate history of data, protect their data from malicious tampering, and trigger alerts when malicious tampering occur. To serve such purposes, verifiable databases (i.e., Concerto [7], QLDB [20]) and blockchain services (i.e., Microsoft Azure Blockchain [15]) often use cryptographic hash functions (e.g., SHA) and Merkle trees [14] to verify the data integrity. *SIRI*, with the built-in support for tamper evidence, is a good candidate for the above systems.

A Merkle tree is a tree of hashes, where the leaf nodes are the cryptographic hashes calculated from blocks of data while the non-leaf nodes are the hashes of their immediate children. The root hash is also called the "digest" of the data. To verify a record, it requires a "proof" of data, which contains the nodes on the path to the root. The new root hash is recalculated recursively and equality is checked with the previously saved digest.

2.2 Data-Level Deduplication

Deduplication approaches have been proposed to reduce the overhead of storage consumption when maintaining multi-versioned data. For example, Decibel [13] uses *delta encoding*, whereby the system only stores the differences, called delta, between the new version and the previous version of data. Consequently, it is effective to manage data versions when the deltas are small, despite the extra cost incurred during data retrieval for reconstructing the specified version of data. However, it is ineffective in removing duplicates among non-consecutive versions or different branches of the data. Though some algorithms choose a more precedent version that has the smallest differences as the parent to improve the efficiency of the deduplication, it involves additional complexity to reconstruct a version.

To enable the removal of duplicates among any data versions, *chunk-based deduplication* can be applied. Unlike delta encoding, this approach works across independent objects. It is widely used in file systems [18, 23], and is a core principle of git. In this approach, files are divided into *chunks*, each of which is given a unique identifier calculated from algorithms like collision-resistant hashing. *chunks* with the same identifier can be eliminated. Chunk-based deduplication is highly effective in removing duplicates for large files that are rarely modified. In case an update leads to a change of all subsequent chunks, i.e., the boundary-shifting problem [10], content-defined chunking [16] can be leveraged to avoid expensive re-chunking.

3 STRUCTURALLY INVARIANT AND REUSABLE INDEXES

Structurally Invariant and Reusable Indexes (*SIRI*) are a new family of indexes recently proposed [21] to efficiently support tamper evidence and effective deduplication.

3.1 Background and Notations

In addition to basic lookup and update operations, the ultimate goal of *SIRI* is to provide native data versioning, deduplication and tamper evidence features. Consequently, data pages in *SIRI* must not only support efficient deduplication (to tackle the amount of replication arising from versioning) but also cryptographic hashing (to facilitate tamper evidence).

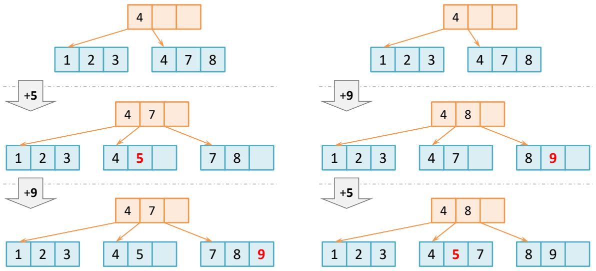


Figure 2: Two B$^+$-trees containing the same entries but with different internal structures [21]

To better elaborate the *SIRI* candidates, we use the following notations in the remaining of this paper. The indexing dataset is denoted as $\mathcal{D} = \{D_0, D_1, ..., D_n\}$ where D_i represents its i-th version. $\mathcal{I}$ is employed to represent *SIRI* structures, and I stands for one of its instances. The key set stored in I is set as $R(I) = \{r_1, r_2, ..., r_n\}$, where r_i denotes the i-th key. $P(I) = \{p_1, p_2, ..., p_n\}$ stands for the internal node set of I, where p_i represents the i-th node.

3.2 Formal Definition

We provide a formal and precise definition of *SIRI* adapted from [21] as follows.

Definition 3.1. An index class $\mathcal{I}$ belongs to *SIRI* if it has the following properties:

(1) *Structurally Invariant.* If I and I' are two instances of $\mathcal{I}$, then $P(I) = P(I') \iff R(I) = R(I')$.
(2) *Recursively Identical.* If I and I' are two instances of $\mathcal{I}$ and $R(I) = R(I') + r$, where $r \notin R(I')$, then $|P(I) \cap P(I')| \gg |P(I) - P(I')|$.
(3) *Universally Reusable.* For any instance I of $\mathcal{I}$, there always exists node $p \in P(I)$ and another instance I' such that $|P(I')| > |P(I)|)$ and $p \in P(I')$.

Definition 3.1 states that *SIRI* must possess three properties. The first property, *Structurally Invariant*, ensures that the order of update operations does not affect the internal structure of the index, while the second property, *Recursively Identical*, guarantees the efficiency when constructing a large instance from small ones. The third property, *Universally Reusable*, secures that the nodes of the index could be shared among different instances. In practice, these properties can be exploited to make *SIRI* time- and space-efficient.

It is non-trivial to construct a *SIRI* instance from conventional structures. Take the multi-way search tree as an example. Such a structure is *Recursively Identical* since only

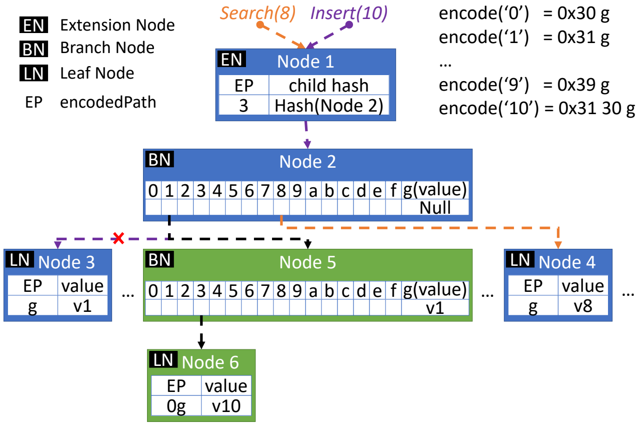


Figure 3: Merkle Patricia Trie (MPT)

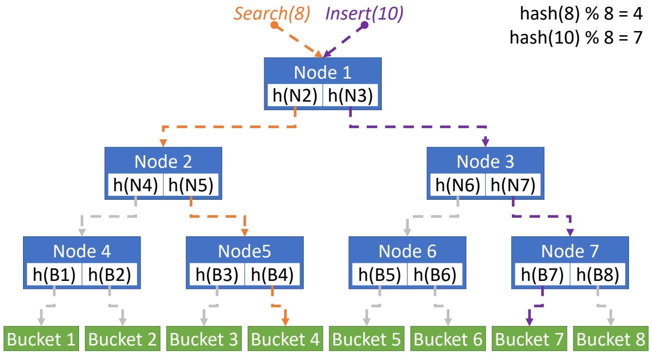


Figure 4: Merkle Bucket Tree (MBT)

a small part of nodes is changed in the new version of the instance when an update operation is performed. Further, the usage of copy-on-write implementation naturally enables node sharing among versions and branches. Hence, it can be *Globally Reusable* when applying this technique. However, it may not be *Structurally Invariant*. Take B$^+$-tree as an example, Figure 2 illustrates that identical sets of items may lead to variant structures. Meanwhile, hash tables are not *Recursively Identical* when they require periodical reconstructions as the entire structure may be updated and none of the nodes can be reused.

Surprisingly, tries, or radix trees, can meet all the three properties with copy-on-write implementation. Firstly, they are *Structurally Invariant* since the position of the node only depends on the sequence of the stored key bytes and consequently, the same set of keys always leads to the same tree structure. Secondly, being a multi-way search tree, they can be *Recursively Identical* and *Globally Reusable* as mentioned above. However, they may end up in higher tree heights, leading to poor performance caused by increasing traversal cost, as shown in Section 4.

3.3 SIRI Representatives

In this section, we elaborate on the three representatives of *SIRI*, namely MPT, MBT, and POS-Tree.

3.3.1 Merkle Patricia Trie.

Merkle Patricia Trie (MPT) is a radix tree with cryptographic authentication. Similar to the traditional radix tree, the key is split into sequential characters, namely nibbles. There are four types of nodes in MPT, namely *branch*, *leaf*, *extension* and *null*. The structures of those nodes are illustrated in Figure 3: (1) *branch* node consists of a 16-element array and a value. Each element, called "branch", of the array is indexing a corresponding child node and stores a nibble. (2) *leaf* node contains a byte string, i.e., a compressed path called "encodedPath", and a value. (3) *extension* node also contains encodedPath and a pointer to the next node. (4) *null* node includes an empty string indicating that the node contains nothing. Similar to Merkle Tree, the whole MPT can be rolled up to a single cryptographic hash for tamper evidence. The most well-known usage of this data structure is in Ethereum [5], one of the largest blockchain systems in the world.

Lookup. The lookup procedure for key "8" is illustrated in Figure 3. The key is first encoded as "0x38 g". Then, each character of the encoded key is used to match with the encodedPath in an extension node, or to select the path in a branch node, from left to right. For this example, the first character "3" matches the root node's encodedPath, therefore, it navigates to its child, Node 2. Then it takes the branch "8" since "8" equals to the second character in the encoded key. Finally, the traversal reaches the leaf node and ends with the value "v8" output.

Insert. To insert data in MPT, the index first locates the position of the given key as in the lookup operation. Once it reaches a *null* node, a leaf node containing the remaining part of the encoded key and the value is created. For example, in Figure 3, when we insert key "1" ("0x31 g"), if branch "1" in Node 2 is empty, a new node ("g", v1) is created and pointed by branch "1". In case there is a partial match at extension node, a new branch node at diverging byte is created, appended with original and new child. The insertion of key "10" in the figure can illustrate this procedure, where the path is diverged at Node 3. Hence, Node 3 is replaced by Node 5 with a newly created Node 6 attached.

3.3.2 Merkle Bucket Tree.

Merkle Bucket Tree (MBT) is a Merkle tree built on top of a hash table as shown in Figure 4. The bottom most level of MBT is a set of buckets and the cardinality of the bucket set is called *capacity*. Data entries are hashed to these buckets, and the entries within each bucket are arranged in sorted order.

The internal nodes are formed by the cryptographic hashes computed from their intermediate children. The number of children an internal node has is called *fanout*. In MBT, *capacity* and *fanout* are pre-defined and cannot be changed in its life cycle.

Lookup. To perform an MBT index lookup, we first calculate the hash of the target key and obtain the index of the bucket where the data resides. Due to the copy-on-write restrictions, we are unable to fetch the bucket directly and hence we then use the bucket number to calculate the traversal path from the root node to the leaf node. The calculation is generally a trivial reverse simulation of the complete multi-way search tree search algorithm. For example in Figure 4, key "8" falls into Bucket 4 after the hashing, and we accordingly get all node index on the path starting from the leaf. Finally, we follow the path to reach the bucket. The records in the bucket are scanned using binary search to find the target key after the retrieval of the bucket node.

Insert. The insert operation of MBT undergoes similar procedures. It first performs a lookup to check the existence of the target key. For example, the inserting key "10" falls into Bucket 7. Then Bucket 7 is fetched following the lookup process, and the key is inserted to Bucket 7 in ascending order. Finally, the hashes of the bucket and the nodes are recalculated recursively.

The design of MBT undoubtedly takes the advantages of Merkle tree and the hash table. On the one hand, MBT offers tamper evidence with a low update cost since only the set of nodes lying on the lookup path needs to be recalculated. On the other hand, the data entries can be evenly distributed due to the nature of the hash buckets in the bottom level.

3.3.3 *Pattern-Oriented-Split Tree.*

Pattern-Oriented-Split Tree (POS-Tree) is a probabilistically balanced search tree proposed in [21]. The structure can be treated as a customized Merkle tree built upon pattern-aware partitions of the dataset, as shown in Figure 5. The bottom most data layer is an ordered sequence of data records. The records are partitioned into blocks using a sliding-window approach and such blocks form the leaf nodes. That is, for a byte sequence within a fixed-sized window, starting from the first byte of the data, a Rabin fingerprint is computed to match a certain boundary pattern. An example pattern can be the last 8 bits of Rabin fingerprint equaling to "1". The window shifts forward to repeat the process until it finds a match, where the node boundary is set to create the leaf node. The internal layers are formed by a sequence of split keys and cryptographic hashes of the nodes in the lower layer. Since the contents in the internal layers already contain hash values, we directly use them to match the boundary pattern instead of repeatedly computing the hash within a sliding window. Such strategy improves the performance of POS-Tree by reducing the number of hash computations, while preserving the randomness of chunking.

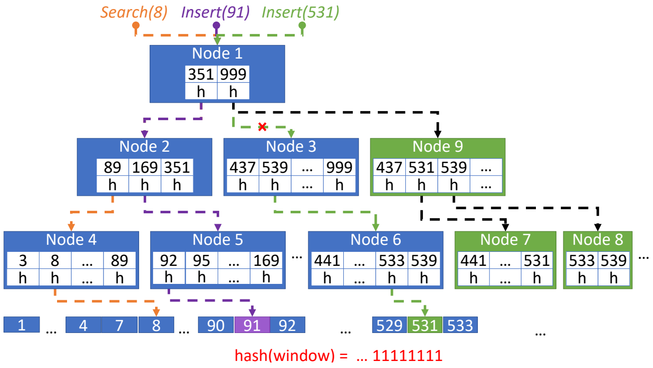


Figure 5: Pattern-Oriented-Splitting Tree (POS-Tree)

Lookup. The lookup procedure of POS-Tree is similar to B$^+$-tree. Starting from the root node, it performs binary search to locate the child node containing the target key. When it reaches the leaf node, a binary search is performed to find the exact key. As the example shown in Figure 5, the key "8" is fetched through the orange path. It goes through Node 2, which has a key range of (-∞, 351], and Node 4, which has a key range of (-∞, 89].

Insert. To perform an insert operation, POS-Tree first finds the position of the inserting key and then inserts it into the corresponding leaf node. Next, it starts the boundary detection from the first byte of the leaf node, and stops when detecting an existing boundary or reaching the last byte of the layer. For example, when insert key "91" into the tree shown in Figure 5, a boundary detection is performed from Node 5. It ends upon reaching the existing boundary of Node 5. Another instance in the figure is the insertion of key "531". A new boundary is found at element 531, and the traverse stops when finding the existing boundary of Node 6. Therefore, Node 6 splits into Node 7 and Node 8, and the new split keys are propagated to the parent node.

The pattern-aware partitioning of POS-Tree enhances the deduplication capabilities, and making the structure of the tree depending only on the data held. Such *Structurally Invariant* property supports efficient *diff* and *merge*. Moreover, the B$^+$-tree-like node structure enables efficient indexing by comparing the split keys to navigate the paths.

3.4 Theoretical Comparison

In this section, we provide a theoretical comparison of the three *SIRI* representatives discussed previously. The details on the computation and analysis can be found in [25]. We first list the theoretical bounds for common operations, i.e. lookup, update, diff and merge. In addition, we define

Table 1: Operation bound

	POS-Tree	MBT	MPT
Lookup	$O(\log_m N)$	$O(\log_m B + \log_2 \frac{N}{B})$	$O(L)$
Update	$O(\log_m N)$	$O(\log_m B + \frac{N}{B})$	$O(L)$
Diff	$O(\delta \log_m N)$	$O(\delta(\log_m B + \frac{N}{B}))$	$O(\delta L)$
Merge	$O(\delta \log_m N)$	$O(\delta(\log_m B + \frac{N}{B}))$	$O(\delta L)$

deduplication ratio as a metric for the measurement of the efficiency of deduplication provided by *SIRI*.

3.4.1 Operation Bound. A summary of the operation bound for each structure is outlined in Table 1, where N is the total number of data held in the index, L is the maximum length of the key, $\bar{L}$ is the average length of the key, B represents the capacity of MBT, m denotes the fanout of MBT and POS-Tree, and δ is used to denote the different records between two versions of the index.

In the worst case, MPT has higher tree height than a balanced search tree, i.e., L > $O(\log_m N)$, and therefore performs worse than POS-Tree. For MBT, the traverse cost $\log_m B$ is lower than other structures when in assumption of $B < N$ while the node scanning time $\log_2 \frac{N}{B}$ and creation time $\frac{N}{B}$ are dominating when $N >> B$. We can conclude from the table that POS-Tree is efficient in general cases, while MBT is a good choice when the dataset maintains a proper $\frac{N}{B}$ ratio.

3.4.2 Deduplication Ratio. Persistent (or immutable) data structures demand a large amount of space for maintaining all historical versions. To alleviate space consumption pressure, the feasibility of detecting and removing duplicated data portions plays a critical role. To clearly quantify the effectiveness of such properties, we define a measurement called *deduplication ratio* as follows.

Definition 3.2. Suppose there is a set of index instances $S = \{I_1, I_2, ...I_k\}$, and each I_x is composed of a set of nodes P_x. The number of bytes of a node set P is denoted as $byte(P)$. The *deduplication ratio* η of S is defined as follows:

$$\eta(S) = 1 - \frac{byte(P_1 \cup P_2 \cup ... \cup P_k)}{byte(P_1) + byte(P_2) + ... + byte(P_k)},$$

The *deduplication ratio* η quantifies the effectiveness of node-level data deduplication (i.e., sharing) among related indexes. It is the ratio between the overall bytes that can be shared between different node sets and the total bytes used for all the node sets. With a high η, the storage is capable of managing massive "immutable" data versions without bearing space consumption pressure. To simplify the analysis, we assume that each instance differs its predecessor by ratio α of a continuous key range. The deduplication analysis is summarized in Table 2. Detailed analysis can be referred to in our technical report [25].

Table 2: Deduplication ratio

POS-Tree	MBT	MPT
$\eta = \frac{1}{2} - \frac{\alpha}{2}$	$\eta = \frac{1}{2} - \frac{\alpha}{2}$	$\eta \geq \frac{1}{2} - \frac{\alpha}{2}$ $(L \geq \bar{L})$ $\eta \leq \frac{1}{2} - \frac{\alpha}{2}$ $(L \leq \bar{L})$

If we compare the analysis results of the three representatives, we can conclude that MPT has the best deduplication ratio under proper query workloads and datasets (meaning $L \geq \bar{L}$). Meanwhile, POS-Tree and MBT have equal bound for the deduplication ratio in this setting.

4 EXPERIMENTAL BENCHMARKING

In this section, we evaluate the three *SIRI* representatives, namely POS-Tree, MBT and MPT with different measurements. For more experiments, such as integration with systems, please refer to [25]. Our experiments are conducted on a server with Ubuntu 14.04, which is equipped with an Intel Xeon Processor E5-1650 processor (3.5GHz) and 32GB RAM. To fairly compare the efficiency of the index structures in terms of node quantity and size, we tune the size of each index node to be approximately 1 KB. For each experiment, the reported measurements are averaged over 5 runs.

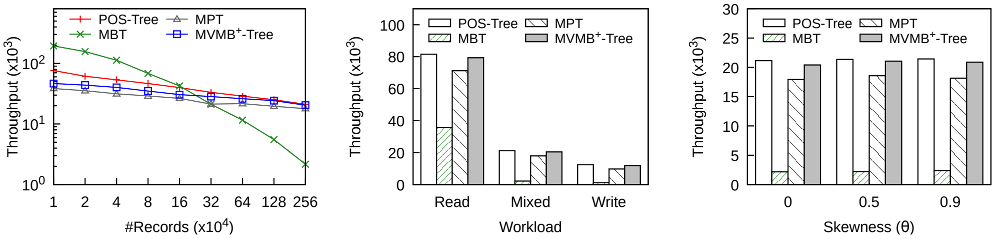
4.1 Dataset and Implementation

We use a synthesized YCSB dataset and two real world datasets, Wikipedia data dump[2] and Ethereum transaction data[3], to conduct a thorough evaluation of SIRI. The details of the setting can be found in [25].

We port the Ethereum's implementation [5] of MPT to our experiment environment, which adopts the path compaction optimization. The implementation of MBT is based on the source code provided in Hyperledger Fabric 0.6 [6]. We further make it immutable and add index lookup logic, which is missing in the original implementation. For POS-Tree, we use the implementation in Forkbase [21]. Moreover, we further apply batching techniques, taking advantage of the bottom-up build order, to reduce the number of tree traversal and hash calculations significantly. Lastly, to compare *SIRI* and non-*SIRI* structures, we implement an immutable B$^+$-tree with tamper evidence support, called Multi-Version Merkle B$^+$-tree (MVMB$^+$-Tree), as the baseline. We replace the pointers stored in index nodes with the hash of their immediate children and maintain an additional table from the hash to the actual address. For all the structures, we adopt node-level copy-on-write to achieve the data immutability.

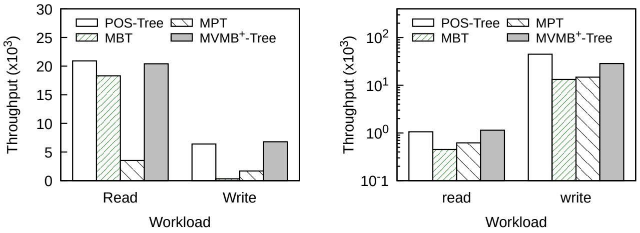
[2] https://dumps.wikimedia.org/enwiki/

[3] https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics



(a) θ = 0, write ratio = 0.5 (b) #Records = 640,000, θ = 0 (c) #Records = 640,000, write ratio = 0.5

Figure 6: Throughput on YCSB



(a) Wiki (b) Ethereum Transaction

Figure 7: Throughput on real world datasets

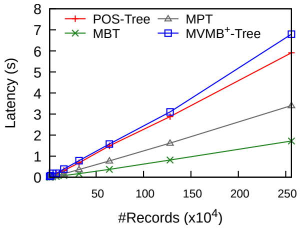


Figure 8: Diff performance

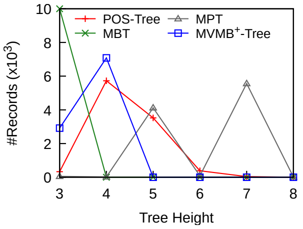
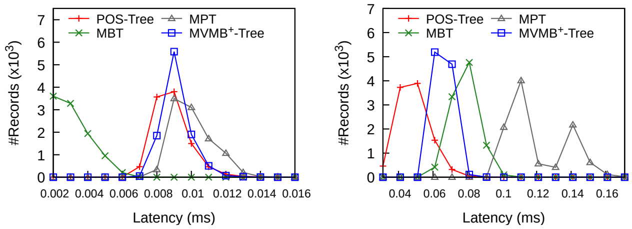
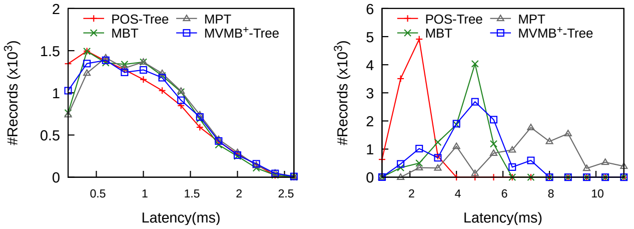


Figure 9: Tree height



(a) Read (b) Write

Figure 10: Latency on YCSB



(a) Read (b) Write

Figure 11: Latency on Ethereum transaction data

4.2 Throughput and Latency

4.2.1 *Throughput.*

First, we evaluate the throughput with the YCSB dataset. We run read, write and mixed workloads with 50% write operations under different data size and skewness settings and the results are depicted in Figure 6(a). It can be observed that the throughput of all indexes decreases as the number of data grows and complies with the operation bound formulated in Section 3.4.1. POS-Tree is 1.03x - 1.64x better than the baseline in terms of throughput while MPT is only 0.82x - 0.87x of the baseline. The throughput of MBT drops quickly from 4.17x to 0.11x of the baseline due to the dominating leaf loading and scanning process.

Figure 6(b) illustrates the throughput of read, write and mixed workload with default dataset size of 640,000. It can be seen that the throughput of all data structures decreases drastically as the ratio of write operations increases. This is natural since the cost of node creation, memory copy and cryptographic function computation increases. The absolute throughput drops over 6.6x for POS-Tree and the baseline while the same measurement drops 30x for MBT and 7.3x for MPT. Figure 6(c) shows the throughput under different data skewness. We can observe that all the structures are resistant to data skewness since there is a minor change in the results for all index structures when θ changes from 0 to 0.9. We also evaluate similar measurements on the Wiki dataset. The system first loads the entire dataset batched in

300 versions, and then executes the read and write workloads, which are uniformly selected. Figure 7(a) demonstrates the results, aligned with the metrics in the YCSB experiment.

For the experiments on Ethereum data, we simulate the way blockchain stores the transactions. For each block, we build an index on transaction hash for all transactions within that block and store the root hash of the tree in a global linked list. Versions are naturally created at a block granularity. For write operations, the system appends the new block of transactions to the global linked list while for lookup operations, it scans the linked list for the block containing the transaction, and traverses the index to obtain the value. Figure 7(b) shows the result of this experiment. It can be observed that POS-Tree outperforms other indexes in write workloads. This is because we are building indexes for each block instead of a global index. Further, instead of insert/update operations, we perform batch loading from scratch. In this case, POS-Tree's bottom-up building process is superior to the MPT's and MVMB$^+$-Tree's top-down building process, as it only traverses the tree and creates each node once. Another difference is that the throughput of read workload is lower than that of the write workload mainly due to the additional block scanning time.

4.2.2 *Latency and Path Length.*

For YCSB dataset, read-only and write-only workloads are fed into the indexes with balanced ($\theta = 0$) and highly skewed ($\theta = 0.9$) distributions. The dataset used in this test contains 160,000 keys and We run 10,000 operations and pictured the latency distribution in Figure 10. The x-axis is the range of the latency and the y-axis is the number of records fell in that latency range. It can be seen from the figure that the rankings among the indexes coincide with the previous conclusion – POS-Tree performs the best for both read and write workloads while MPT performs the worst. Meanwhile, MPT has several peak points, representing operations accessing data stored in different levels of the tree.

To take a closer observation of how the workloads affect the candidates, we further gather the traversed tree height of each operation for the write-only workload with the uniform distribution. The results are shown in Figure 9(a), where the x-axis represents the height of the lookup path and the y-axis indicates the number of operations. Most operations have to visit 4-level nodes to reach the bottom-most level of POS-Tree whilst 5- or 7-level nodes are frequently traversed for MPT. The efficiency in MBT is also verified in the figure since all requests only need 3 levels to reach the bottom of the structure in both balanced and skewed scenarios.

The result for the skewed workloads in the YCSB dataset is similar and hence omitted. The results for the Wiki dataset is also skipped here for the same reason. For the complete results, please refer to [25]. However, the experiment on Ethereum transaction data exhibits different trends as depicted in Figure 11. As can be observed, all the structures have similar read latency, caused by the dominant block scanning process.

We also run a diff workload to evaluate the performance of "diff" operations. In the experiment, each structure loads two versions of data in random order. A diff operation is performed between the two versions and the execution time is taken, as depicted in Figure 8(a). All the candidates outperform the baseline due to the structurally invariant property. Among which, MBT performs the best (4x of baseline) since the position of the nodes containing a specific data is static among all versions. The logic of diff operation is the simplest, i.e., comparing the hash of the nodes at the corresponding position. MPT performs 2x better than the baseline and 1.7x better than POS-Tree due to the simplicity that keys with the same length always lie in the same level of the tree.

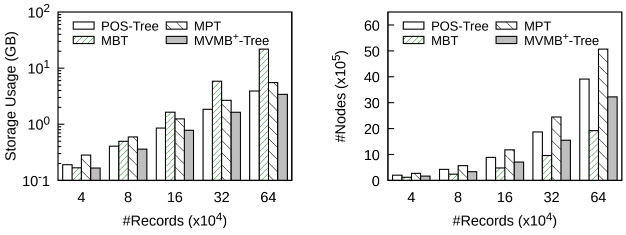
4.3 Storage

In this section, we evaluate the space consumption of the index structures under different use cases.

4.3.1 *Single Group Data Access.*

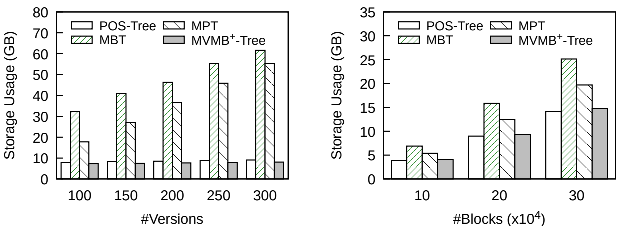
We first start with a simple case, where a dataset is accessed by multiple users. There is no sharing of data or cross-department collaborative editing in this setting. Therefore, the deduplication benefit is limited using *SIRI*. In reality, such case often happens in-house within a single group of users from the same department. Figure 12(a) shows the storage under different data sizes for the YCSB dataset. There are two main factors affecting the space efficiency, i.e., the size of the node and the height of the tree. On the one hand, larger tree height results in more node creations for write operations, which also increases the space consumption. As an example, MPT performs badly since it has the largest tree height in our experiment setting. It consumes the storage up to 1.6x higher than the baseline and up to 1.4x larger than POS-Tree. On the other hand, a large node size means that even minor changes to the node could trigger the creation of a new substantial node, which hence leads to larger space consumption. As can be seen, MBT performs the worst due to the largest node size it has in the implementation. It consumes up to 6.4x the space of that used by the baseline. POS-Tree, compared to the baseline, also has a larger node size variance due to content-defined chunking, leading to a greater number of large nodes.

To better analyze how the memory space is used by different pages, we further accumulate the number of nodes for all chosen indexes. The results are demonstrated in Figure 12(b) with variant dataset sizes. Typically, they follow similar trends as Figure 12(a), except that MBT generates the



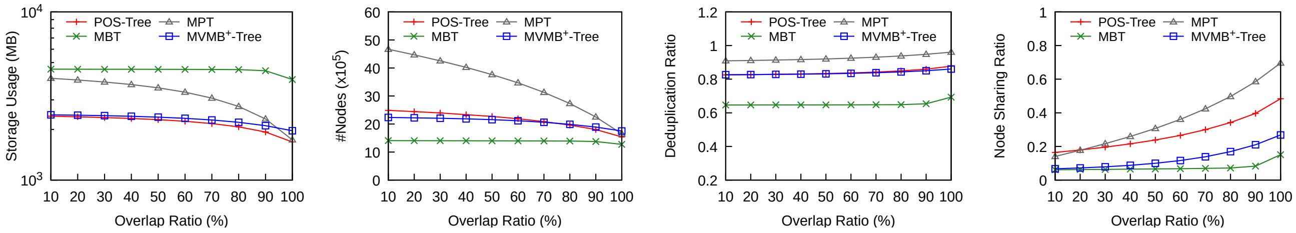
(a) Storage (b) Number of nodes

Figure 12: Performance on single group data access



(a) Wiki (b) Ethereum

Figure 13: Storage on real world datasets



(a) Storage (b) Number of nodes (c) Deduplication ratio (d) Node sharing ratio

Figure 14: Performance on diverse group collaboration

least number of nodes as the total number of nodes is fixed for the structure.

The results for the Wiki dataset and Ethereum transaction dataset are shown in Figure 13. Similar to the results of the YCSB experiment, MBT and MPT consumed more space than POS-Tree and MVMB$^+$-Tree. A difference is that MPT storage consumption increases very fast as the number of versions are loaded in Figure 13(a). This is because the key length of the Wiki dataset is much larger than that of YCSB, and the encoding method used by Ethereum further doubles the key length. For every insert/update operation, more nodes need to be re-hashed and created. Hence, the space efficiency is worse than it shows in the YCSB experiment.

4.3.2 *Diverse Group Collaboration.*

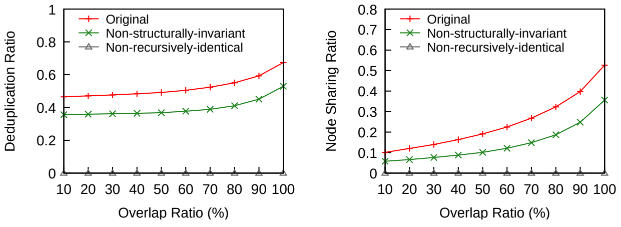
Next, we compare the storage consumption in the scenario that diverse groups of users are collaborating to work on the same dataset. This often occurs in the data cleansing process and data analysis procedure, where different parties work on different parts of the same dataset. One significant phenomenon in this case is that duplicates can be frequently found. Therefore, the deduplication advantages naturally introduced by *SIRI* is critical to improving the space efficiency. To better evaluate the deduplication capability, we define another metric called node sharing ratio as formulated as follows:

$$\eta(S) = 1 - \frac{|P_1 \cup P_2 \cup ... \cup P_k|}{|P_1| + |P_2| + ... + |P_k|},$$

where P_i is the set of nodes of an instance i. While the deduplication ratio evaluates the physical size of the storage saved, the node sharing ratio indicates how many duplicate nodes have been eliminated in the index.

The YCSB dataset is used in this experiment. We simulate 10 groups of users, each of which initializes the same dataset of 40,000 records. We generate workloads of 160,000 records with overlap ratios ranging from 10% to 100% and feed them to the candidates. Here, 10% overlap ratio means 10% of the records have the same key and value. The execution is processed with default batch size, i.e., 4,000 records.

The results of the deduplication ratio and the node sharing ratio are shown in Figure 14(c) and Figure 14(d), respectively. Both metrics of all the structures become higher when the workload overlap ratio increases since more duplicate nodes can be found due to the increasing similarity among the datasets. Benefiting from smaller node size and smaller portion of updating nodes, MPT achieves the highest deduplication ratio (up to 0.96) and node sharing ratio (up to 0.7). POS-Tree achieves a slightly better deduplication ratio than the baseline though they both have similar size of nodes and the height of the tree. The actual ratios are 0.88 and 0.86, respectively. However, POS-Tree achieves a much better node sharing ratio compared to the baseline (0.48 vs. 0.27) because of its content-addressable strategy when chunking the data. By contrast, MBT's fixed number of pages and growing leaf



(a) Deduplication ratio

(b) Node sharing ratio

Figure 15: Effect of SIRI property

nodes limit the number of duplicates, and therefore it does not perform as good as the other two *SIRI* representatives.

To be more precise, we further collect the storage usage and the number of pages created by the testing candidate and illustrate the results in Figure 14(a) and Figure 14(b). The trends in the figures match the corresponding deduplication ratio and node sharing ratio perfectly. With the increasing overlap ratio, storage reduction of POS-Tree and MPT is more obvious than the baseline, among which POS-Tree is the most space-efficient. MPT is most sensitive to overlap ratio changes due to the high node sharing ratio introduced by its structural design. Although it consumes more space for non-overlapping datasets, MPT outperforms the baseline for datasets with the overlap ratio above 90%.

4.4 Breakdown Analysis

In this section, we evaluate how each *SIRI* property affects the deduplication performance. We select POS-Tree as the testing object and disable the properties one by one. For each property, we first explain how each property is disabled and then provide the experimental results. We note that the *Universally Reusable* property is common for all immutable tree indexes using copy-on-write approach. Thus, it is ignored in this experiment.

- **Disabling the Structurally Invariant Property.** The pattern based partitioning is the key to guarantee the *Structurally Invariant* property. Therefore, we disable the property by forcibly splitting the entries at half of the maximum size when no pattern is found within the maximum size. Consequently, the resulting structure depends on the data insertion order. We increase the probability of not finding the pattern by increasing the bits of pattern and lowering the maximum value.
- **Disabling the Recursively Identical Property.** Originally, only the set of nodes lying in the path from the root to the leaf node is copied and modified when an update operation is performed, while the rest of the nodes are shared between the two versions in POS-Tree. We disable the *Recursively Identical* property by forcibly copying all nodes in the tree. The number of different pages between the two instances is much larger than the number of intersections, which is zero.

The deduplication result is presented in Figure 15. We can observe that both the deduplication ratio and the node sharing ratio for POS-Tree is 0 with the *Recursively Identical* property disabled, since the structure does not allow the sharing of nodes among different versions. Meanwhile, Figure 15(a) shows an up to 15% decrease in the deduplication ratio when the *Structurally Invariant* property is disabled. It is expected as the index performs the operations in different orders, resulting in different nodes and smaller number of share-able pages. Though the records stored are the same, POS-Tree cannot reuse the nodes with the *Structurally Invariant* property disabled. Similarly, Figure 15(b) shows that the node sharing ratio decreases by up to 17%, i.e., from 0.53 to 0.36, by disabling the *Structurally Invariant* property. Compared to the figures in previous sections, we can infer how this property accelerates the deduplication ratio and ultimately influences the final storage performance.

Overall, we can conclude that the *Recursively Identical* property is the fundamental property to enable indexes with deduplication and node sharing across different users and datasets. On top of this, the *Structurally Invariant* property further enhances the level of deduplication and node sharing by making structures history-independent.

5 CONCLUSIONS

Tamper evidence and deduplication are two properties increasingly demanded in emerging applications on immutable data, such as digital banking, blockchain and collaborative analytics. Recent works [6, 21, 22] have proposed three index structures equipped with these two properties. However, there have been no systematic comparisons among them. To address the problem, we conduct a comprehensive analysis of all three indexes in terms of both theoretical bounds and empirical performance. Our analysis provides insights regarding the pros and cons of each index, based on which we conclude that POS-Tree is a favorable choice for indexing immutable data.

This study is part of our ongoing work in building a ledger database based on our Forkbase engine [21]. We envisage to support data verifiability, immutability, and flexible transaction models with high performance leveraging SIRI.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2006. *GoogleDocs.* https://www.docs.google.com

[2] 2011. *LevelDB.* https://github.com/google/leveldb

[3] 2012. *RocksDB.* http://rocksdb.org

[4] 2014. *WeBank.* https://www.webank.com/en/

[5] 2015. *Ethereum.* https://www.ethereum.org

[6] 2016. *Hyperledger.* https://www.hyperledger.org

[7] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. 251–266. https://doi.org/10.1145/3035918.3064030

[8] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. 2014. DataHub: Collaborative Data Science & Dataset Version Management at Scale. *CoRR* abs/1409.0798 (2014). http://arxiv.org/abs/1409.0798

[9] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1085–1100. https://doi.org/10.1145/3035918.3064033

[10] Kave Eshghi and Hsiu Khuern Tang. 2005. A Framework for Analyzing and Improving Content-Based Chunking Algorithms.

[11] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya Parameswaran. 2017. OrpheusDB: Bolt-on Versioning for Relational Databases. *PVLDB* 10, 10 (2017), 1130–1141. https://doi.org/10.14778/3115404.3115417

[12] Qian Lin, Kaiyuan Yang, Tien Tuan Anh Dinh, Qingchao Cai, Gang Chen, Beng Chin Ooi, Pingcheng Ruan, Sheng Wang, Zhongle Xie, Meihui Zhang, et al. 2020. ForkBase: Immutable, Tamper-evident Storage Substrate for Branchable Applications. In *ICDE*.

[13] Michael Maddox, David Goehring, Aaron J. Elmore, Samuel Madden, Aditya G. Parameswaran, and Amol Deshpande. 2016. Decibel: The Relational Dataset Branching System. *PVLDB* 9, 9 (2016), 624–635. https://doi.org/10.14778/2947618.2947619

[14] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO*, Vol. 293. 369–378. https://doi.org/10.1007/3-540-48184-2_32

[15] Microsoft. 2019. *Azure Blockchain Service.* https://azure.microsoft.com/en-us/services/blockchain-service/

[16] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A Low-Bandwidth Network File System. In *SOSP*. 174–187. https://doi.org/10.1145/502034.502052

[17] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf.

[18] João Paulo and José Pereira. 2014. A survey and classification of storage deduplication systems. *Comput. Surveys* 47, 1 (2014), 11. https://doi.org/10.1145/2611778

[19] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. 2019. Fine-Grained, Secure and Efficient Data Provenance on Blockchain Systems. *PVLDB* 12, 9 (2019), 975–988. https://doi.org/10.14778/3329772.3329775

[20] Amazon Web Services. 2019. *Amazon Quantum Ledger Database.* https://aws.amazon.com/qldb/

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

[22] Daniel Davis Wood. 2014. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER.

[23] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. 2016. A comprehensive study of the past, present, and future of data deduplication. *Proc. IEEE* 104, 9 (2016), 1681–1710. https://doi.org/10.1109/JPROC.2016.2571298

[24] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A Comprehensive Performance Evaluation of Modern In-Memory Indices. In *ICDE*. 641–652. https://doi.org/10.1109/ICDE.2018.00064

[25] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. 2020. Analysis of Indexing Structures for Immutable Data. arXiv:cs.DB/2003.02090