Spitz: A Verifiable Database System

Meihui Zhang[1] Zhongle Xie[2] Cong Yue[2] Ziyue Zhong[1]

[1] Beijing Institute of Technology [2] National University of Singapore

meihui_zhang@bit.edu.cn, zhongle@comp.nus.edu.sg, yuecong@comp.nus.edu.sg, ziyue_zhong@bit.edu.cn

ABSTRACT

Databases in the past have helped businesses maintain and extract insights from their data. Today, it is common for a business to involve multiple independent, distrustful parties. This trend towards decentralization introduces a new and important requirement to databases: the integrity of the data, the history, and the execution must be protected. In other words, there is a need for a new class of database systems whose integrity can be verified (or verifiable databases).

In this paper, we identify the requirements and the design challenges of verifiable databases. We observe that the main challenges come from the need to balance data immutability, tamper evidence, and performance. We first consider approaches that extend existing OLTP and OLAP systems with support for verification. We next examine a clean-slate approach, by describing a new system, Spitz, specifically designed for efficiently supporting immutable and tamper-evident transaction management. We conduct a preliminary performance study of both approaches against a baseline system, and provide insights on their performance.

PVLDB Reference Format:
Meihui Zhang, Zhongle Xie, Cong Yue, Ziyue Zhong. Spitz: A Verifiable Database System. *PVLDB*, 13(12): 3449-3460, 2020.
DOI: https://doi.org/10.14778/3415478.3415567

1. INTRODUCTION

Traditional database systems are indispensable for businesses.They excel at storing, processing, and performing analytics over business transactions. Recent digital optimization and transformation have enabled businesses to transact directly with each other, without relying on a central party. As a result, multiple parties can access a shared database. Since the parties are mutually distrustful, the underlying database must consider support for auditing, tamper evidence, and dispute resolution in its design. For instance, it must maintain a trusted data history and allow users to verify the integrity of both current and historical data.

Blockchains demonstrate one practical design for database systems with strong integrity [21]. Public blockchain systems, such as Ethereum, support secure peer-to-peer applications through smart contracts. Private blockchains, such as Hyperledger Fabric, target business settings and achieve higher performance than public ones. Blockchains have drawn interests from banks and regulators, with the prospect of offering digital currency and digital banking. Combined with recent advances in 5G, AI and IoT, blockchains are expected to speed up the transformation and further disrupt the e-commerce and financial industries.

To make a database that can be accessed by potentially malicious parties trustworthy, it must be *verifiable.* A verifiable database system protects integrity of the data, of its provenance, and of its query execution. More specifically, any tampering such as changing the data content, changing a historical record, or modifying query results, can be detected. We note that the demand for verification is on the rise due to the requirements imposed by the regulators on various business sectors, investment and banking in particular.

The first requirement in the design of a verifiable database (VDB) is data immutability, which is necessary for maintaining trusted provenance. Immutability means data is only written once and never deleted. It is not a new concept. It has been used in NoSQL systems such as HBase [4], CouchDB [1] and RethinkDB [8] to achieve more efficient concurrency, due to the fact that no synchronization of accesses is needed. It is also used in Resilient Distributed Datasets (RDDs)[9] for lineage and fault tolerance. The second requirement of a verifiable database is query verifiability. It means the query results contain integrity proofs for both the data and query execution. More specifically, a user can detect if either the data or the query execution has been tampered.

In this paper, we discuss four challenges in realizing the design of verifiable databases. The first challenge is in storage management, as data immutability requires managing the ever-increasing volume of data. Consider a typical healthcare analytic application, in which health data needs to be kept for the lifetime of a patient, and each diagnosis, lab test, prescription, etc., is appended to the patient profile. Disease and procedure coding standards evolve over time, e.g., from ICD-9-CM to ICD-10 in recent years. Such changes in classification and coding standards require updates or mapping onto the existing medical record. To ensure good data provenance, the data must be immutable and a new version of the database, i.e., a snapshot, is appended. The data volume is

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415567

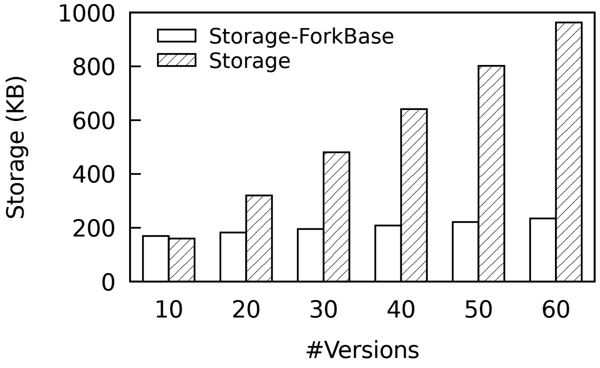


Figure 1: Data storage improved by deduplication.

increasing with time, and therefore its management needs to be efficient and reliable. Let us consider another example where an immutable database stores 10 WIKI pages of 16 KB each initially. We create a new version when updating a page, while keeping the previous versions. Figure 1 shows the space utilized with an increasing number of versions. Clearly, the space utilization increases substantially with the number of immutable versions, and the use of an efficient multi-version storage engine such as ForkBase [51] helps to reduce it. This highlights the importance of storage efficiency for a database that is forever increasing in size.

The second challenge is to provide efficient access methods for querying immutable data. While some existing database systems archive historical data and support temporal query processing, they have not been designed to support "permanent" immutability. In VDB, data is never deleted; query processing and frequent searching on older versions of the data will be prohibitively expensive if efficient storage layout and indexes are not supported. This may entail scanning of a substantial portion of the database for answering verification queries.

The third challenge is to minimize performance overhead of verification. VDB must generate integrity proofs whose cost can be significant. Blockchains, for example, have poor transaction throughput due to their protocols for guaranteeing security in the Byzantine environment. The performance gap between traditional databases and blockchains is significant due to their different design focus. As a result, a verifiable database must adopt a hybrid blockchain-database approach in order to strike a better balance between performance and security.

The fourth challenge is the need to support both OLTP and OLAP workloads, as illustrated by the emergence of HTAP database systems. The former requires serializability which is important for applications such as e-commerce. Most existing OLTP systems adopt optimistic concurrency control (OCC), instead of pessimistic concurrency control, because of its simplicity and high performance. In contrast, analytic queries in OLAP do not require the strict ordering provided by serializability. Existing OLAP systems adopt multi-version concurrency control (MVCC) to achieve data consistency with high performance. Most existing works on verifiable queries focus on OLTP workloads. While general OLAP queries can be made verifiable, for example by using fully-homomorphic encryption, they involve complex cryptographic operations and incur significant overhead [40, 47].

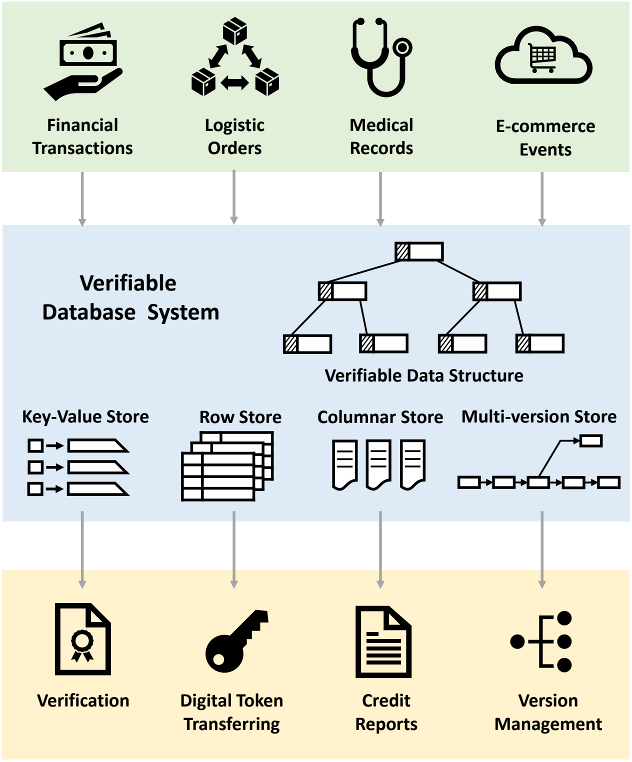


Figure 2: Verifiable database system overview.

Therefore, it is challenging to support both verifiable OLTP and OLAP queries with practical performance.

In addition to the four challenges above, we note that VDB must also aim for deployability. It is often costly to either add a new database system into an existing infrastructure, or to replace an existing database with a new one. In particular, for a business with consolidated software stacks, data conversion is necessary to move data to the new database. Furthermore, users may find the system difficult to use if the verifiable database adopts unfamiliar programming models or interface.

In this paper, we discuss two approaches to realize an efficient VDB. The first approach is to extend existing systems, and the second is to design a new system from scratch. Figure 2 shows the second approach and how it fits with existing business applications. The new system uses tamper-evident structures for verification, and efficient version management for performance.

We make the following contributions in this paper:

- We identify the requirements and design challenges of efficient verifiable databases.
- We discuss two approaches for realizing an efficient verifiable database: by extending existing systems, and a new clean-slate design called Spitz.
- We perform an experimental study on Spitz and compare it with a baseline. The results show that Spitz can achieve good performance, despite overhead from verification and additional data structures.

- We discuss various future research topics, including the integration of learning methods onto VDB and version management of the machine learning pipeline.

The rest of this paper is organized as follows. In Section 2, we present existing works and systems that are related to VDB processing. In Section 3, we discuss the research challenges and opportunities of VDB. We next describe the challenges in the approach of extending existing OLTP and OLAP system to implement VDB in Section 4. In Section 5, we present the system architecture of Spitz. We then present an experimental study in Section 6, and compare our systems against a baseline implementation based on a commercial service. We discuss the promising synergy between VDB and AI in Section 7, before concluding in Section 8.

2. VERIFIABLE DATABASES

In this section, we survey existing works and systems with verification features.

2.1 Verifiable Database

Data integrity is important in outsourced database as third-party service providers can be malicious. In particular, applications running on top of an outsourced database require the data and query results from the providers to be verifiable, that is, tampering of data and query execution can be securely detected. One way to achieve verifiability is using verifiable computation techniques. Benabbas et al. [16] present a delegation scheme on verifiable database minimizing the resources required by the clients of verifiable database. Guo et al. [25] improve the update efficiency using a long polynomial for public keys and a short polynomial for private keys. Miao et al. [36] enable efficient keyword search for VDB using enhanced vector commitment while HVDB [65] supports hierarchical verification by building a vector commitment tree. SNARKs [40] can support arbitrary computation tasks, but requiring an expensive setup phase, and incurring significant overhead. Ben-Sasson et al. [15] improve upon SNARKs by bounding the complexity of the setup phase to size of the database and the query complexity. More recently, Zhang et al. [63] propose a system called vSQL that uses an interactive protocol to support verifiable SQL queries. However, vSQL is limited to relational databases with a fixed schema.

Another way to achieve verifiability is by using authentication data structures such as Merkle trees. Li et al. [29] propose and evaluate authentication index structures combining Merkle trees and B$^+$-trees. Yang et al. [57] propose integrity-protected MR-tree for spatial data. ServeDB [54] proposes a Merkle tree index based on hierarchical cube encoding that supports efficient multi-dimensional queries. Security conscious applications enforce data integrity against malicious modifications not only from external attackers, but also from malicious insiders and cloud hosting operators. As a solution, SUNDR [30] cryptographically protects all file system contents and proposes a fork consistency protocol to detect data tampering.

More recent systems, namely VeritasDB [49] and Concerto [14], leverage trusted hardware to speed up verification. In particular, both store Merkle tree data inside SGX enclaves. Veritas stores the roots of the trees, and Concerto uses memory verification technique to avoid contention inside the enclaves.

2.2 Out-of Blockchain Database

Blockchains, which was originally designed for cryptocurrencies, is now being used as a general-purpose transactional system. Being a distributed data processing system, a blockchain system shares some similarities with a distributed database system. However, its focus is security, whereas the database's focus is performance. The design space of both systems can be viewed along four dimensions: replication, concurrency, storage and sharding. A recent work [45] provides an extensive and in-depth comparison of blockchain versus database. It shows that along the four design dimensions, different choices lead to different performance. We are seeing a trend of merging these two systems into a design that is secure, efficient, and can be readily adopted by applications such as logistic, digital banking, and digital asset management.

One step toward realizing a hybrid blockchain-database is to support rich data queries on blockchains [22, 35, 43, 10]. A simple approach is to join the network as a full node and then execute the query. However, running a full node is expensive. vChain [56] addresses this problem by embedding an aggregate and constant-size authentication data structure, constructed with multiset accumulator, in each block header. This allows users to run a light node to query with integrity guarantee. TrustDBle [23] proposes a secure and scalable OLTP engine that provides verifiable ACID-compliant transactions on shared data using trusted hardware.

BlockchainDB [22], Veritas [24], FalconDB [42], and LineageChain [44, 46] are recent systems that use blockchain as a verifiable storage and add database features on top of it. We now discuss these systems in more detail.

BlockchainDB adopts a simple key-value data model, and exposes *Put/Get/Verify* operations to clients. It consists of a database layer and a storage layer. The former controls the consistency level of requests so that clients can choose the balance of result staleness and performance. The storage layer serves as the unified interface to the underlying blockchains. It translates requests from the database layer into blockchain transactions and monitors the transaction status. When a client invokes *Verify*, a blockchain node would contact other peers to check whether the corresponding transaction is committed in the ledger. A node in BlockchainDB does not hold the complete copy of the state. Instead, the states are partitioned to multiple blockchains.

Veritas shares a similar goal and vision with BlockchainDB, but differs in three aspects. First, it targets complex data models, i.e., relational model. Second, it employs Trusted Execution Environments (TEEs) such as Intel SGX as the trustable verifiers that consume the database logs for the transaction validation. The validation results, in the form of verifiers' votes, are persisted in the blockchain. As a result, when there is a dispute, any party can resolve it by reconciling the database log with the votes on the ledger. Third, Veritas does not support partitioning, as it does not store all states on the blockchain.

Instead of checking the ledger for log validation, FalconDB organizes database records into an authenticated data structure, such as a Merkle tree, which enables a succinct integrity proof on a database record. FalconDB employs an incentive model allowing clients to selectively challenge the transaction results from a suspicious server. If the server cannot provide proof of correctness, it will be penalized. FalconDB also supports authenticated queries on a temporal data model, so that users may access data snapshot with respect to a particular block.

LineageChain [44, 46] is a fine-grained, secure, and efficient provenance system built on top of ForkBase [51, 32] and FabricSharp[3]. It provides provenance information to smart contracts through simple interfaces to enable a new class of blockchain applications whose execution logics depend on provenance information at runtime. LineageChain captures provenance during contract execution and stores it in a Merkle tree implemented in ForkBase, and provides a novel skip list index to support efficient provenance queries.

2.3 Ledger Database

Amazon offers Quantum Ledger Database (QLDB) [11], a cloud service that provides data immutability and verifiability. QLDB consists of blocks organized in a hash chain called journal. Changes to the data, including insert, update and delete, are collected into blocks and appended to the journal. A Merkle tree is built upon the entire journal. To support efficient query, the journal is materialized to user-defined tables for the latest data and history data. QLDB aims to provide the high performance of database systems with integrity guarantees for data and historical data versions. Similarly, Oracle Blockchain Table [13] offers append-only verifiable tables by implementing a centralized ledger model. MongoDB [12] supports verifiable change history by storing document collections in a hash chain.

Datomic [2] is a distributed immutable database system designed to be ACID compliant, with datom as its database building block. It is a form of key-value store, and Datoms are collected to form an entity. It makes use of key-value stores such as Amazon DynamoDB for managing the data, and allows users to obtain a historic snapshot of the database via its APIs and query language. Immudb [6] is a recently released immutable tamper-evident open source database system. Due to the demand for VDB, we foresee active development in such kind of database systems. However, with no deletion, the database size will grow over time, and query processing efficiency and scalability could become major concerns.

3. CHALLENGES AND OPPORTUNITIES

In this section, we discuss the research challenges in implementing an efficient VDB. In particular, the requirements of immutability and verifiability have implications on storage and indexing, query verification, and concurrency control mechanisms. We discuss the methodologies from recent works that provide building blocks for VDB.

3.1 Storage and Indexing

VDB requires an immutable and tamper-evident storage engine. In particular, the storage must support integrity proof generation, and have an efficient version management mechanism.

ForkBase [51], a storage with Git-like version control and branch management, and Merkle-based directed acyclic graph (DAG) data structure, provides a good starting point. ForkBase supports collaborative analytics, and content-based data deduplication mechanism that significantly reduces data volume in the physical storage. Furthermore, it supports efficient version querying.

As in traditional databases, indexes are necessary for fast retrieval and location of records. Recent Merkle tree-based indexes, namely Merkle Patricia Trie (MPT) [53], Merkle Bucket Tree (MBT) [5], and Pattern-Oriented-Split Tree (POS-Tree) [51], support efficient queries on immutable data. [59] contains a comprehensive analysis of these indices, showing that MPT, MBT, and POS-Tree are different instances of Structurally Invariant and Reusable Indexes (SIRI) [51], and that POS-tree has better overall performance. In addition to these indices which are designed for query verifiability, other indices are needed to further speed-up data retrieval. Since versions can be modeled as temporal or historical data, indexes such as the historical R-tree[37], and rolling index B^x-tree [26] could be adapted to support the multi-dimensional and single-dimensional queries. We envision that the need for fast querying of historical data will lead to new, innovative indexes.

3.2 Verification

Query verifiability in VDB means that the user who sends the query can verify the integrity of the result, that is the data and execution have not been tampered with. We discuss here different approaches to achieve verifiability.

Client-side verification vs Server-side verification. When the data is outsourced to a third party, the users themselves must verify some proofs provided by the third party. However, verification can be expensive for the users, especially when running on low-power devices. Trusted hardware, such as Intel SGX, can help mitigate this cost for user, by supporting server-side verification. In particular, the hardware performs verification securely at the servers, by running verification inside trusted execution environments, and output only succinct proofs that can be verified cheaply by the user. However, the secure hardware has limited resources that can lead to significant performance overhead. Furthermore, existing secure hardware are vulnerable to side-channel attacks that compromise their security.

Online verification vs Deferred verification. With respect to the timing of the verification, there are two approaches: online, and deferred verification. In the former, the data must be committed after the verification succeeds, which is useful when recovery from malicious tampering is costly. In the latter, verification is done over a batch of transactions, therefore achieving higher throughput than the former.

Verification via encryption. One way to protect data integrity is by using authenticated encryption. Users can encrypt data using private key and store the ciphertexts on untrusted storage. Data tampering can be detected directly with the authentication tag. The limitation of this approach is that it restricts computation (or queries) on the ciphertexts. Encryption schemes with various support for computation on ciphertexts exist, but they have trade-off in security and computation. All of these schemes have significant performance overhead.

Verification via authentication data structure. Authentication data structures, which are based on Merkle trees, provide data integrity with low cost. In this structure, the leaf nodes contain cryptographic hashes of the data blocks, while the non-leaf nodes contain the hashes of their

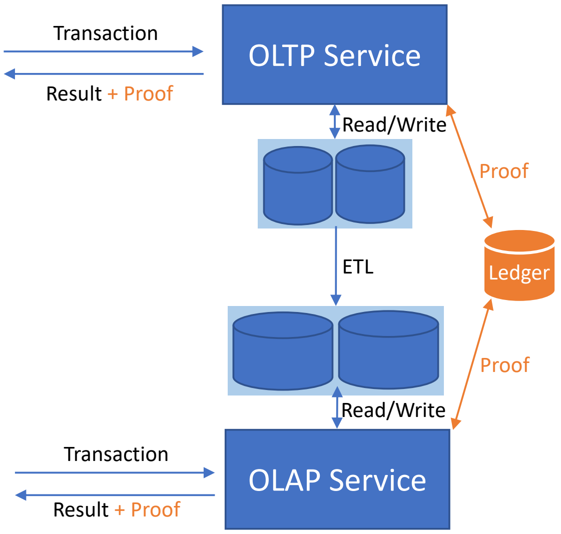


Figure 3: Non-intrusive design.

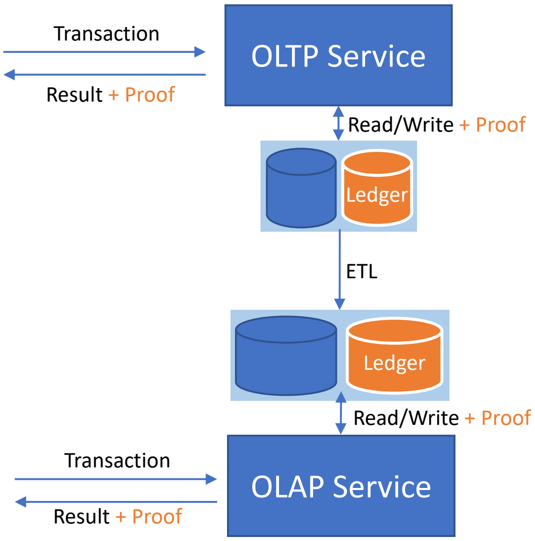


Figure 4: Intrusive design.

child nodes. The hash of the root node is called the "digest" of the data. The integrity proof consists of the hashes of the nodes from the corresponding leaf to the root of the tree. The new digest is recalculated recursively and equality is checked with the previously saved digest.

3.3 Concurrency Control

Many outsourced or cloud databases are multi-tenant. Applications running on top of a multi-tenant database may require different ACID isolation levels. The database often has fixed the transaction isolation levels at the time of deployment, therefore applications have to implement their own levels for their needs, which increases the complexity of the system. This problem can be mitigated by using per-tenant database architecture, but this approach does not scale well.

Consider as an example an e-commerce system with customer credits. On the one hand, the purchases of the items must occur in sequence to prevent double spending or shipping out-of-stock items. In other words, the transaction schedule needs to be serializable, which can be implemented using optimistic concurrency control (OCC) or multi-version concurrency control (MVCC) with abortion on read-write conflicts. On the other hand, the analysis report or status checking on the system may not require strict isolation. Such queries are mostly processed as read-only workloads, and many of them require near real-time responses. For example, read committed isolation will be sufficient to execute query "getting all items with stock-level lower than 50". In this case, it is unnecessary to abort the query when read-write conflicts occur.

A common approach to achieving high performance for weak isolation is to fix the isolation to a weak level (e.g., read committed), and implement customized logic to handle stricter level in the applications. Such design involves locks, checking pre-images of data and sometimes reversions, therefore complicating the application logic and incurring large overhead. By providing flexible isolation levels in the underlying database, it allows for performance optimization and lets users focus more on the application logic.

4. EXTENDING OLTP/OLAP TO VDB

VDB can be implemented by adding a verifiable ledger to an existing database system. The ledger supports immutable data and verifiable queries. Here we discuss the challenges of integrating such a ledger to OLTP and OLAP systems.

There are two designs for integration, as shown in Figure 3 and Figure 4. The blue arrows, rectangles and cylinders depict a typical data processing flow, where the data is collected by OLTP and analyzed by OLAP systems.

Non-intrusive design. As shown in Figure 3, a ledger is attached without modifying the architecture of the original database systems. However, additional steps are added during transaction processing. The OLTP and OLAP systems generate integrity proofs from an independent ledger. On the one hand, this design minimizes disruption to existing systems, as it does not require changes to existing data. On the other hand, it incurs considerable performance overhead, due to the interaction with the ledger.

Intrusive design. Another design, as depicted in Figure 4, is to embed the ledger into an existing database system. This eliminates communication with an outside ledger, by generating the integrity proof inside the database. While reducing performance overhead compared to the other design, it incurs significant cost in data migration. In particular, data must be moved to the new system, which may be too costly for users with large amounts of data.

Another approach is to integrate the ledger with a hybrid transactional/analytical processing (HTAP) system. A HTAP system is designed to unify efficient processing of operational and analytical workloads in the same database. In the HTAP system, no data migration from OLTP system to OLAP system is necessary. Existing OLTP systems are being converted to HTAP by exploiting in-memory processing [61] and both columnar and row storage structures. Some

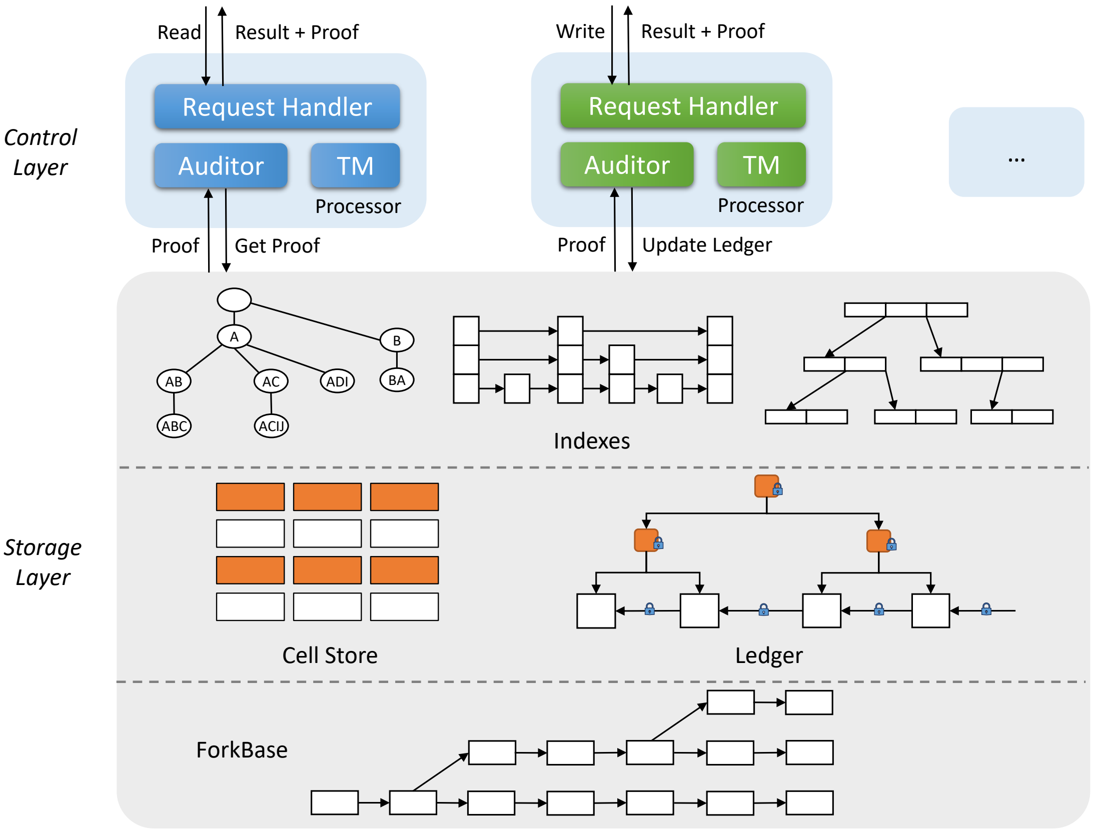


Figure 5: System architecture.

recent NewSQL systems also adopt HTAP in their design and implementation.

5. SYSTEM ARCHITECTURE

In this section, we discuss the system architecture of Spitz, a distributed database designed from scratch that supports both OLTP and OLAP workloads with verifiable ledgers. As shown in Figure 5, the system consists of two layers: the *control layer*, and the *storage layer*.

The control layer consists of multiple *processor nodes* that accept and process requests from a global message queue[1]. Each node has three main components: a request handler, an auditor, and a transaction manager (TM). The request handler accepts query requests and returns the results with the corresponding proofs. The auditor communicates with the ledger in the storage layer to keep track of data changes. The transaction manager controls the execution of the queries in the storage.

The storage layer features a distributed storage engine, namely ForkBase. Built on top of ForkBase is a virtual cell store, as opposed to row or column store in traditional databases. The system maps each cell to a universal key consisting of the column id, primary key, timestamp, and the hash of its value. There are multiple index structures built into the storage layer to support verifiable query processing.

Ledger. This structure consists of a sequence of hashed blocks. Each block tracks the modification of the records, query statements, metadata and the root node of the indexes on the entire dataset. The block and the data can be verified using the Merkle tree structure built on top of the entire ledger. Section 5.3 discussed more details regarding verification and proof generation.

Index. Spitz uses a B$^+$-tree for query processing. The input of the index is the requested keys, and the output is the matched data cell. This structure is efficient for both point and range queries.

Inverted Index. When processing analytical queries, the system uses an inverted index to quickly locate the rows to fetch data. Such an index uses the value recorded in each cell as index key and the universal key of the corresponding cell as value. The structure of the inverted list varies according to the type of the data stored in the cell. For instance, for numeric type, the system uses a skip list to better support range query, whereas for string type, it uses a radix tree to reduce space consumption.

5.1 Query Processing

The processor nodes handle both read, write, and mixed workload. Spitz supports both SQL and a self-defined JSON schema.

Write workload. There are four steps in handling a write workload. (1) The request handler collects a transaction from the message queue. (2) The auditor checks the write operations and updates the ledger. The ledger records

[1] Similar to other distributed systems, the coordination, as well as the resource management, is done by a master node.

the changes and returns a proof to the auditor. (3) The processor traverses the B$^+$-tree index and performs the write operations to the cell store. (4) The processor collects the results, combines them with the proof, and sends back to the user through the request handler.

Read workload. The processing of read workload follows similar steps. (1) The request handler receives a transaction from the message queue. (2) The processor collects the results by traversing corresponding inverted indexes and retrieving the cell store. (3) The processor visits the ledger via the auditor, getting the proofs of the results. The proof generation is done by the ledger using the universal keys and the internal nodes of inverted index. (4) The processor combines the results and the proofs as responses and the request handler returns them to the user.

Spitz uses a HTAP design to overcome the data movement between OLTP and OLAP systems. Similar to the intrusive design in Figure 4, it requires users to replace the underlying database systems, which might be highly tangled with their business. However, it should be highlighted that Spitz can be used as an individual ledger by solely waking up the auditor in the processor. Thus, the system can be applied into a non-intrusive design shown in Figure 3 as a short-term transition plan of integrating Spitz into the real-world business. Ultimately, users should use Spitz as a standalone and complete database system to cover and develop their business.

5.2 Concurrency Control

Concurrency control in each processor node can be implemented in the same way as in traditional database systems. However, in our design, cells are multi-versioned. Therefore, to achieve serializability guarantee, concurrency control mechanisms based on MVCC, including MVCC with 2PL [18], MVCC with timestamp ordering (T/O) [17], MVCC with OCC [31], are more suitable.

Since each processor node processes transactions independently, it is necessary to keep the data in the indexes and the virtual storage consistent across different nodes. The solution is to add distributed transactions to each node, and follow the two-phase commit (2PC) protocol to coordinate each transaction so that transactions committed by different nodes can be made serializable. The challenge in achieving serializability in distributed setting is to figure out the order of transactions in the equivalently serial schedule.

One approach to achieving serializability is to rely on a global timestamp service, like Timestamp Oracle [41], to allocate the timestamps upon a transaction starts and commits. We then order transactions based on their start timestamps. In the prepare phase of 2PC, each transaction with read/write and write/write conflict with this order will abort. However, there are two limitations. First, the timestamp allocation service can become the bottleneck. Second, the abort rate can be high in a write-intensive workload. To address the first limitation, we can adopt the hybrid logic timestamp scheme that allocates timestamps by each individual node and still has serializability guarantee [28, 50]. For the second limitation, it is possible to adopt the combination of OCC and MVCC by dynamically adjusting the transaction order to reduce abort rates [19, 34], and verifying the transactions in batch to reduce the verification cost [20]. These approaches need further investigation and evaluation.

5.3 Proof and Verification

Spitz offers timely detection of malicious data tampering by using an authentication data structure, namely the ledger shown in Figure 5. Clients can use the digest of the ledger to perform verification locally. Since changes to ledger are serializable, during the transaction processing, only the data committed before the transaction can be verified. After the processing, clients can get the data and the proof of this transaction as described in Section 5.1, along with other metadata of the authentication tree structure if applicable.

To verify the correctness of the results, clients can recalculate the digest with the received proof and compare it with the previous digest saved locally. If they match, it means the data has not been modified during the period between the verification and when the digest is generated. To improve verification throughput, we use a deferred scheme, which means the transactions are verified asynchronously in batch.

6. EXPERIMENTAL STUDY

In this section, we describe the prototype of Spitz and present its preliminary evaluation results. The full-scale implementation of Spitz is in progress and a thorough performance study will be conducted in the future.

6.1 Implementation

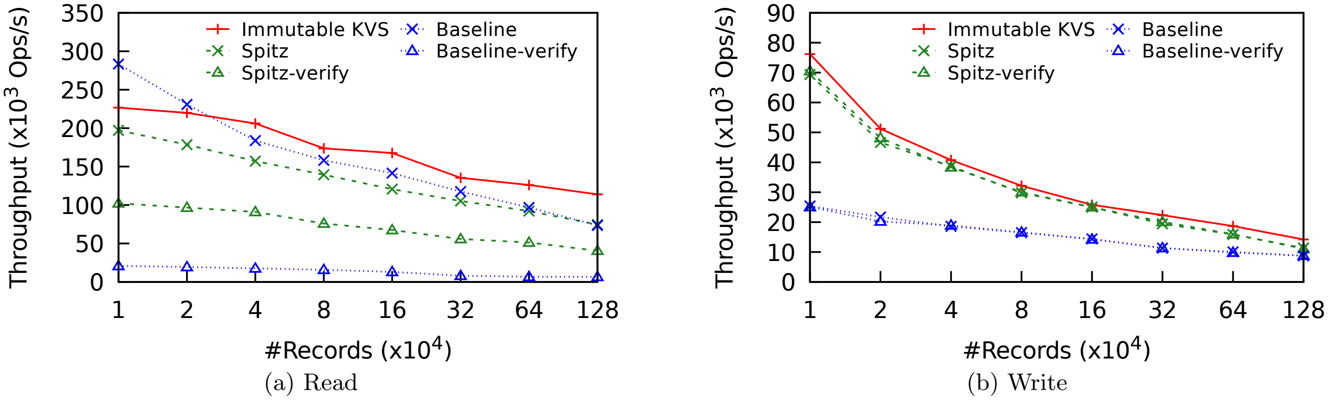
First of all, we implement a baseline system to emulate a commercial product based on the features described online and testing provided by the website. The newly inserted or modified records are collected into blocks and appended to a ledger implemented by a Merkle tree. The ledger is used for verification purposes, shadowing the nodes of a typical B$^+$-tree for query key searching. Furthermore, the appended blocks are materialized to indexed views for fast query processing. To perform a read query, users can directly fetch the data with meta information using the indexed views, which can be verified against the ledger.

For the prototype of Spitz, we modify the latest version of ForkBase and forgo irrelevant functionalities such as branch management. In particular, we implement the ledger by adopting index from Structurally Identical and Reusable Indexes (SIRI) family for both query and verification. Each block in the ledger stores a historical index instance, naturally composing a version of the ledger, and the nodes between instances can be shared, benefiting from SIRI properties.

For comparison purpose, we also build an immutable key-value store (KVS) using ForkBase. It is the same as Spitz in terms of indexing, except that it does not maintain a ledger or provide verifiability. Therefore, by comparing the two systems, we can focus on the maintenance and verification cost of the ledger storage implemented in Spitz.

6.2 Evaluation

We evaluate the performance of the systems with read-only and write-only workloads. The number of records, which consist of different key-value pairs, vary from 10,000 to 1,280,000. The length of the key ranges from 5 to 12 bytes while the size of the value is 20 bytes. The experiments are conducted on a server with Ubuntu 14.04, which is equipped with 6 cores Intel Xeon Processor E5-1650 processor (3.5GHz) and 32GB RAM.



(a) Read

(b) Write

Figure 6: Basic operations in single-thread setup.

6.2.1 *Basic Operations*

We first evaluate the performance of read-only and write-only workloads in a single-thread setup. We vary the initial database size from 10,000 to 1,280,000 records, and execute read-only and write-only workloads on different systems.

Figure 6(a) shows the results for read-only workloads. The immutable KVS performs the best without maintaining any verifiable data structures. The baseline implementation and Spitz have comparable performance when the number of records becomes large, as the index traversal becomes a dominant factor in query processing. When the verification on the integrity of the queried results is enabled, plotted as Spitz-verify and Baseline-verify in the figure, the read performance for Spitz is approximately half of that without the verification while the baseline operations per second drops by almost two orders of magnitude. If compared directly, Spitz achieves 7x operations per second than that of the baseline. The major reason of such phenomenon is that Spitz can store the proofs of the results and the value of the target nodes in a unified index, namely the ledger implemented via SIRI. To compare, the baseline needs to visit the B$^+$-index first, and uses the resultant nodes to get the proof from the ledger. Figure 6(b) shows the results for write-only workloads. Similarly, thanks to the unified index structure, Spitz has operations per second comparable to the immutable KVS with and without verification while the performance of the baseline system is much worse because of maintaining multiple indexed views.

6.2.2 *Range Query*

In this section, we evaluate the performance of analytical workloads with range queries. Such workloads are commonly submitted by data scientists to retrieve a group of records for analysis or further aggregation. We initialize the database with 10,000 to 1,280,000 records for different runs. The selection conditions of the range query are set on the primary key and the selectivity of the query is fixed at 0.1%.

Figure 7 depicts the operations per second in all systems. As can be seen, the performance of the range query is worse than the performance of point query shown in Figure 6(a) by 25% to 90% for all cases. This is due to the additional nodes needed to be traversed and scanned when the query is processed. Meanwhile, the operations per second drops fast when the total number of records increases because the systems fetch increasing number of records with fixed selectivity for all cases.

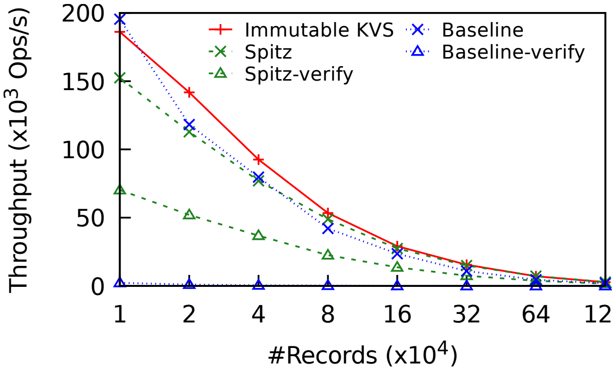
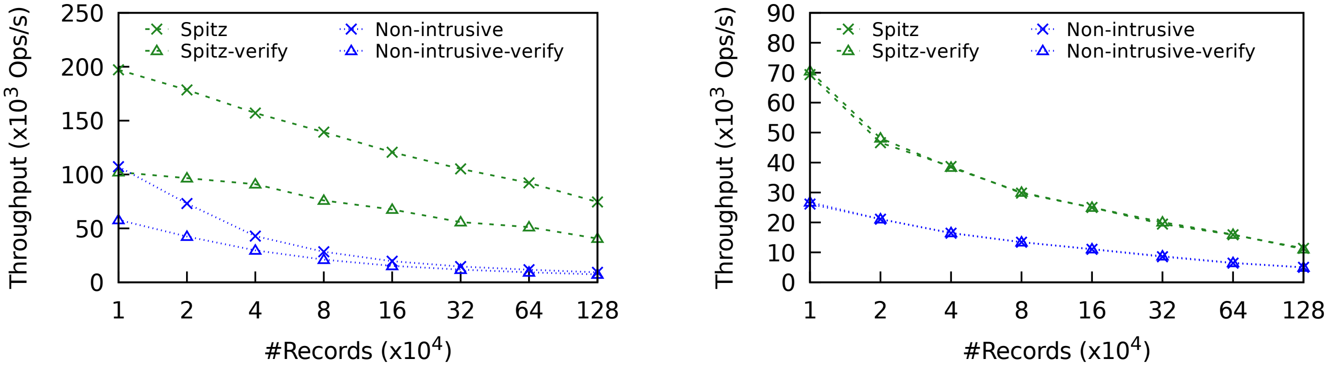


Figure 7: Range query performance.

The baseline stores the ledger and the index of the data separately, and hence, the retrieval of the proofs cannot benefit from the optimizations used in range query processing. That is, the retrieval on the proofs of resultant records, instead of being fetched in a batch by scanning keys with the given interval, must be processed by searching the digest in the ledger individually. In contrast, for Spitz, thanks to the use of the unified index structure described in Section 6.2.1, proof retrieval can leverage the traversal on the index of the data – the proofs of the resultant records are returned simultaneously when the resultant records are scanned and selected. Consequently, for queries with verification of data integrity enabled, Spitz outperforms the baseline by up to two orders of magnitude, a gap much larger than the results shown in Figure 6(a).

6.2.3 *Non-intrusive Design vs Spitz*

In this section, we evaluate the performance of a non-intrusive design of VDB and compare it with Spitz. We set up an immutable key-value store using ForkBase as the underlying system, which interacts with the ledger shown



(a) Read (b) Write

Figure 8: Non-intrusive design vs. Spitz.

in Figure 3.[2] To support data verification, we deploy Spitz on the same server as the Ledger database in the figure. In the case of read workloads, the client obtains the queried results from the underlying database and the proofs from the ledger as responses, while in the case of write workloads, the submitted data are committed in both the underlying and ledger database atomically. To verify the results, the client uses the proof from the Ledger database, calculates the digest of the returned results, and compares it with the previous digests as described in Section 5.3.

The experiment is conducted with read-only and write-only workloads and the results are shown in Figure 8. As can be seen, the non-intrusive design incurs significant overhead by maintaining two systems, i.e., the underlying database and the Ledger database. Specifically, for read-only workloads, the performance of Spitz is 6x higher than the non-intrusive design when the verification of data integrity is enabled. The huge performance gain comes from a simpler process flow: the request can be processed within a single system in Spitz, while in the non-intrusive design, it must be sent to the underlying database first to obtain the results, and passed to the Ledger database to retrieve the proofs. Obviously, the interactions between the Ledger database and the underlying database inevitably introduce additional cost on network communication, query planning, etc. For write workloads, Spitz produces 3x higher number of operations per second than the non-intrusive design.

In summary, the results show that the prototype of Spitz achieves better performance than the non-intrusive design. Without doubt, its performance could further be improved with indexes and optimization strategies specifically designed for VDB.

7. AI AND VDB - A SYNERGY

We have discussed the design and implementation of a VDB that can support new database applications. This new trend in database coincides with the rapid development of artificial intelligence (AI). AI has been proven successful in a wide range of applications, could automate many tasks, and it is often better at modeling complex situations than humans.

To meet the demand for complex analytics, database systems have enabled the addition of machine (or deep) learning libraries to construct end-to-end analytics pipelines. With the evolution of dataset due to versioning, machine learning models used in the analytics pipeline may exhibit concept drift behavior, which causes the model to become less accurate over time. Therefore, iterative analytics component updates may become necessary, and the relationship between component and data versions need to be maintained for verification on the analytical results.

From system performance perspective, deep learning can be used to enhance database performance and usability, and vice versa, deep learning can benefit from the efficient data management and performance provided by databases (Apache SINGA [38] for example). Earlier works [52] have discussed the symbiotic relationship between databases and machine learning. In the following, we shall continue this discussion by making a case for the merging of VDB and AI.

7.1 AI for VDB

AI can make VDB more intelligent. Currently, VDB design is based on empirical methodologies, which might not have a good performance. As pointed out in [52], AI may help to improve VDB's performance in several aspects.

- Learning-based data structure. A number of solutions investigate how to enhance existing indexes or design new indexes for better storage and query efficiency, e.g., learned B$^+$tree [27], secondary index [55].
- Learning-based transaction management. AI could be used to predict the future transaction workload [33] and schedule the transactions such that the throughput is maximized with an acceptable abort rate [48].
- Learning-based performance tuning. To avoid manual tuning of the memory allocation or I/O control, recent works [62] apply reinforcement learning to automatically tune database configurations according to workload changes.
- Learning-based query optimization. To optimize the queries, existing works [39, 58] deploy deep neural networks such as convolutional neural networks (CNNs),

[2] ForkBase can be treated as a HTAP system here therefore we do not initialize separate OLAP and OLTP system as shown in the figure.

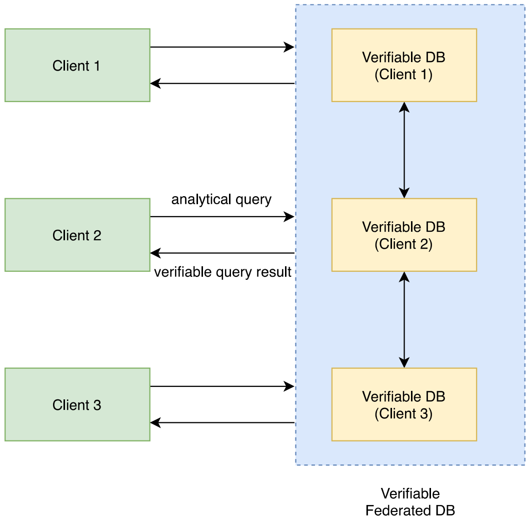


Figure 9: Verifiable federated analytical query processing.

recurrent neural networks (RNNs) and their variants to estimate the cardinality and cost.

Though many learning-based techniques for general DB have been proposed, more effort is needed to adapt them for VDB, since VDB has different data structure, storage and transaction management requirements.

7.2 VDB for AI

VDB can make AI (-based analytics) more reliable. One key feature of VDB is that it maintains historical data. Hence, a natural question might be: can VDB support analytical queries? For example, a client (e.g., a hospital) has already outsourced its database to a cloud hosting company for processing online transactions (e.g., medical records). It may then also want the VDB to process some analytical queries (even some machine learning model) for specific evaluations, such that it does not need to download the data and execute locally. However, the analytical query support in VDB is limited (e.g. [7]). More importantly, the analytics result should be verifiable, ensuring that it is computed from correct data; otherwise, it may result in a wrong decision, and lead to huge loss (e.g., could be people's health or even life in medical domain).

There are several works [64, 63] that support verifying arbitrary SQL queries over outsourced databases, but with low performance. Recent works [56, 60] on verifiable specific queries over blockchain (can be viewed as a special kind of VDB) are relatively efficient, but they only support range queries. It is necessary to investigate how to efficiently compute complex analytical queries on VDB.

Finally, it is possible to consolidate multiple clients' VDB to provide federated analytics (as illustrated in Figure 9). For example, a few hospitals want to have a more precise and comprehensive analysis of a disease. The integrity of the data and queries are important in these use cases. At the same time, each client should not be able to break the confidentiality of the other clients' data.

In summary, AI and VDB could benefit from each other: AI could improve the performance of VDB, and VDB could ensure the trustworthiness of the analytical results (or AI models).

8. CONCLUSIONS

With recent digital optimization and transformation, more and more businesses are transacting directly with each other. The current pandemic further speeds up the transformation and adoption of online business processes. This trend introduces a new important requirement to database systems: the integrity of the data, the history, and the execution must be protected. This gives rise to a new class of database systems that support the verification of the transactional integrity.

In this paper, we discuss the requirements and challenges of verifiable databases. We present approaches to extend existing systems to support verification, and an initial design and prototype of our ongoing development of a new VDB system called Spitz. We conduct an experimental study and show that Spitz is able to provide a better performance than a baseline system. As future works, we will continue to implement the system and present a comprehensive system design and a thorough performance study. We will also study and possibly introduce new indexes, concurrency control mechanisms and query processing strategies for VDB.

Acknowledgments

Meihui Zhang would like to thank the VLDB Endowment Awards Selection Committee for the 2020 VLDB Early Career Research Contribution Award, and the nominators for the nomination. After checking with the Chair of the Awards Selection Committee on the invited paper requirements, Meihui decided to report an ongoing system development, which is being built upon system components and works developed by her and collaborators. For the works that led to the award, Meihui would like to thank her mentors, colleagues, collaborators, research assistants and students for their contributions. Meihui would also like to thank her ex-dean, Prof. Heyan Huang and current dean Prof. Guoren Wang, for their support and guidance.

For this paper, Meihui, Zhongle, Cong and Ziyue would like to thank Anh Dinh, Qian Lin, Wei Lu, Beng Chin Ooi, Pingcheng Ruan, and Yuncheng Wu for their discussions, contributions and proof reading. The research of Cong Yue and Zhongle Xie are supported by Singapore Ministry of Education Academic Research Fund Tier 3 under MOEs official grant number MOE2017-T3-1-007.

9. REFERENCES

[1] Couchdb. `https://couchdb.apache.org/`.

[2] Datomic. `https://www.datomic.com/`.

[3] Fabricsharp. `https://www.comp.nus.edu.sg/~dbsystem/fabricsharp`.

[4] Hbase. `https://hbase.apache.org/`.

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

[6] immudb. `https://github.com/codenotary/immudb`.

[7] Querying your data - amazon quantum ledger database (amazon qldb). `https://docs.aws.amazon.com/qldb/latest/developerguide/working.userdata.html`.

[8] Rethinkdb. `https://rethinkdb.com/`.

[9] Spark. `https://spark.apache.org/`.

[10] Wolk SwarmDB–decentralized database service for web3. `https://laptrinhx.com/wolk-swarmdb-decentralized-database-services-for-web3-4011398543/`, 2017.

[11] Amazon quantum ledger database (qldb). `https://aws.amazon.com/qldb/`, 2019.

[12] Implementing cryptographically verifiable change history using mongodb. `https://github.com/mongodb-labs/ledger`, 2020.

[13] Oracle blockchain tables. `https://docs.oracle.com/en/database/oracle/oracle-database/20/ftnew/oracle-blockchain-table.html`, 2020.

[14] 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 *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 251–266. ACM, 2017.

[15] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, pages 781–796. USENIX Association, 2014.

[16] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *Advances in Cryptology - CRYPTO 2011*, volume 6841, pages 111–131. Springer, 2011.

[17] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[18] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[19] C. Boksenbaum, M. Cart, J. Ferrié, and J. Pons. Certification by intervals of timestamps in distributed database systems. In *Tenth International Conference on Very Large Data Bases,VLDB*, pages 377–387. Morgan Kaufmann, 1984.

[20] B. Ding, L. Kot, and J. Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2):169–182, 2018.

[21] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 30(7):1366–1385, 2018.

[22] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb - A shared database on blockchains. *PVLDB*, 12(11):1597–1609, 2019.

[23] M. El-Hindi, S. Karrer, G. Doci, and C. Binnig. TrustDBle: Towards trustable shared databases. In *Third International Symposium on Foundations and Applications of Blockchain*, 2020.

[24] J. Gehrke, L. Allen, P. Antonopoulos, A. Arasu, J. Hammer, J. Hunter, R. Kaushik, D. Kossmann, R. Ramamurthy, S. T. V. Setty, J. Szymaszek, A. van Renen, J. Lee, and R. Venkatesan. Veritas: Shared verifiable databases and tables in the cloud. In *9th Biennial Conference on Innovative Data Systems Research, CIDR*, 2019.

[25] Z. Guo, H. Li, C. Cao, and Z. Wei. Verifiable algorithm for outsourced database with updating. *Cluster Computing*, 22:5185–5193, 2019.

[26] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b$^+$-tree based indexing of moving objects. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB*, pages 768–779. Morgan Kaufmann, 2004.

[27] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 489–504. ACM, 2018.

[28] S. Kulkarni, M. Demirbas, D. Madeppa, A. Bharadwaj, and M. Leone. Logical physical clocks and consistent snapshots in globally distributed databases. In *The 18th International Conference on Principles of Distributed Systems*, 2014.

[29] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 121–132. ACM, 2006.

[30] J. Li, M. N. Krohn, D. Mazières, and D. E. Shasha. Secure untrusted data repository (SUNDR). In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 121–136. USENIX Association, 2004.

[31] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 21–35. ACM, 2017.

[32] Q. Lin, K. Yang, T. T. A. Dinh, Q. Cai, G. Chen, B. C. Ooi, P. Ruan, S. Wang, Z. Xie, M. Zhang, and O. Vandans. Forkbase: Immutable, tamper-evident storage substrate for branchable applications. In *36th IEEE International Conference on Data Engineering, ICDE*, pages 1718–1721. IEEE, 2020.

[33] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 631–645. ACM, 2018.

[34] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. E. Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *PVLDB*, 7(5):329–340, 2014.

[35] T. McConaghy, R. Marques, A. Müller, D. D. Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. Bigchaindb: A scalable blockchain database. *Whitepaper*, 2018.

[36] M. Miao, J. Wang, S. Wen, and J. Ma. Publicly verifiable database scheme with efficient keyword search. *Information Sciences*, 475:18–28, 2019.

[37] M. A. Nascimento and J. R. O. Silva. Towards historical r-trees. In *Proceedings of the ACM Symposium on Applied Computing, SAC*, pages 235–240. ACM, 1998.

[38] B. C. Ooi, K. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. H. Tung, Y. Wang,

Z. Xie, M. Zhang, and K. Zheng. SINGA: A distributed deep learning platform. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference,MM*, pages 685–688. ACM, 2015.

[39] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. An empirical analysis of deep learning for cardinality estimation. *CoRR*, abs/1905.06425, 2019.

[40] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy, SP*, pages 238–252. IEEE Computer Society, 2013.

[41] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 251–264. USENIX Association, 2010.

[42] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song. FalconDB: Blockchain-based collaborative database. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 637–652. ACM, 2020.

[43] B. M. Platz, A. Filipowski, and K. Doubleday. Flureedb: a practical decentralized database. *Whitepaper*, 2017.

[44] P. Ruan, G. Chen, A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang. Fine-grained, secure and efficient data provenance for blockchain. *PVLDB*, 12(9):975–988, 2019.

[45] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, D. Loghin, B. C. Ooi, and M. Zhang. Blockchains and distributed databases: a twin study. *CoRR*, abs/1910.01310, 2019.

[46] P. Ruan, A. Dinh, Q. Lin, M. Zhang, G. Chen, and B. C. Ooi. Revealing every story of data in blockchain systems. *ACM SIGMOD Record*, special issue for SIGMOD Research Highlight Award, 2020.

[47] S. T. V. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the 21th USENIX Security Symposium*, pages 253–268. USENIX Association, 2012.

[48] Y. Sheng, A. Tomasic, T. Sheng, and A. Pavlo. Scheduling OLTP transactions via machine learning. *CoRR*, abs/1903.02990, 2019.

[49] R. Sinha and M. Christodorescu. Veritasdb: High throughput key-value store with integrity. *IACR Cryptology ePrint Archive*, 2018:251, 2018.

[50] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. Cockroachdb: The resilient geo-distributed SQL database. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 1493–1509. ACM, 2020.

[51] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *PVLDB*, 11(10):1137–1150, 2018.

[52] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K. Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Record*, 45(2):17–22, 2016.

[53] D. D. Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[54] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang. ServeDB: Secure, verifiable, and efficient range queries on outsourced database. In *35th IEEE International Conference on Data Engineering, ICDE*, pages 626–637. IEEE, 2019.

[55] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 1223–1240. ACM, 2019.

[56] C. Xu, C. Zhang, and J. Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 141–158. ACM, 2019.

[57] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Authenticated indexing for outsourced spatial databases. *VLDB J.*, 18(3):631–648, 2009.

[58] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, P. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *PVLDB*, 13(3):279–292, 2019.

[59] C. Yue, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, S. Wang, and X. Xiao. Analysis of indexing structures for immutable data. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 925–935. ACM, 2020.

[60] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi. Gem^2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *35th IEEE International Conference on Data Engineering, ICDE*, pages 842–853. IEEE, 2019.

[61] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 27(7):1920–1948, 2015.

[62] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the International Conference on Management of Data, SIGMOD*, pages 415–432. ACM, 2019.

[63] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE Symposium on Security and Privacy, SP*, pages 863–880. IEEE Computer Society, 2017.

[64] Y. Zhang, J. Katz, and C. Papamanthou. Integridb: Verifiable SQL for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1480–1491. ACM, 2015.

[65] Z. Zhang, X. Chen, J. Li, X. Tao, and J. Ma. HVDB: a hierarchical verifiable database scheme with scalable updates. *Journal of Ambient Intelligence and Humanized Computing*, 10(8):3045–3057, 2019.