HYBRID BLOCKCHAIN DATABASE SYSTEMS DESIGN AND PERFORMANCE

NANDIGAMA SATISH, UPENDAR NANDAGIRI, SUPRIYA MENON MANIYAL
Dept of CSE,
Priyadarshini Institute of Science and Technology for Women Khammam.

ABSTRACT
We seek to give a thorough examination of the trade-offs and performance of a few typical hybrid blockchain database systems. We implement Veritas and BlockchainDB from the ground up in order to accomplish this goal. We offer two flavours of Veritas to cater to the application situations of crash fault-tolerant (CFT) and byzantine fault-tolerant (BFT). To be more precise, we use Veritas with Tendermint to target BFT application situations and Veritas with Apache Kafka to target CFT application scenarios. We contrast these three platforms with the current BigchainDB open-source implementation. BigchainDB offers two versions: an optimised version with concurrent transaction validation and blockchain pipelining, and a default implementation that leverages Tenemint for consensus.
The results of our experimental investigation verify that BFT designs, which are unique to blockchains, perform significantly worse than CFT designs, which are generally utilised by distributed databases. However, our thorough research also reveals the range of design options that the developers had to consider and helps to clarify the trade-offs that must be made when creating a hybrid blockchain database system.

DOI Number: 10.48047/nq.2021.19.12.NQ21319 NeuroQuantology 2021;19(12):1075-1092

I. INTRODUCTION

In the last few years, a handful of systems that integrate both dis tributed databases and blockchain properties have emerged in the academic database community [22, 30, 31, 37, 42, 46]. These systems, termed *hybrid blockchain database systems*, are either adding data base features to existing blockchains to improve performance and usability or implement blockchain features in distributed databases to improve their security [35]. However, there is little comparison among these systems. In this paper, we aim to fill this gap by providing an in-depth analysis of a few representative hybrid blockchain database systems.
To achieve this goal, we first have to overcome a challenge: only one such system, namely BigchainDB [42] is open-source. Moreover, by the time of writing this paper, we did not manage to get the source code of any other hybrid blockchain database system. Hence, we undertake the tedious task of implementing Veritas [22] and BlockchainDB [30]. By doing so, we gain the flexibility of changing some parts of the systems to better compare them to other systems. For example, we can change the underlying database from a relational SQL type to a NoSQL type. Or we can change the consensus mechanism from crash fault-tolerant (CFT) to Byzantine fault-tolerant (BFT).
The original design of Veritas uses Apache Kafka [20], which is a CFT service, as the broadcasting service among server nodes. That is, when a server node needs to update its local shared database as a result of a transaction's execution, it sends this

update to all the other server nodes via the Kafka service. Moreover, the other server nodes send their agreement or disagreement regarding that update via the Kafka service. Apache Kafka uses a primary backup mechanism to achieve CFT. Hence, it has high efficiency at the cost of decreased liveness. On the other hand, most blockchain systems adopt a BFT consensus mechanism. Hence, we also implement a version of Veritas where Tendermint [8], which is a BFT consensus, is used as middleware, similar to BigchainDB [42].

BlockchainDB [30] implements a shared (and sharded) database on top of a classic blockchain. The atabase interface provides a simple key-value store API to the user, similar to Veritas and BigchainDB. The blockchain layer is flexible, being able to interact with different blockchains, such as Ethereum [9] and Hyperledger Fabric [21]. In this paper, as well as in [30], BlockchainDB uses Ethereum as its underlying blockchain.

BigchainDB introduces two optimizations. The first optimization, called blockchain pipelining, allows nodes to vote for a new block while the current block of the ledger is still undecided. Each node will just reference its last decided block when voting for a new block. By doing so, the system avoids waiting for blocks to be fully committed before proposing new blocks. Second, BigchainDB includes a flavor called BigchainDB with parallel validation, or BigchainDB (PV), where transactions are validated in parallel on multiple CPU cores. These parallel validations should increase the throughput, however, we do not observe any improvement, as we shall see in our experimental analysis.

In summary, we make the following contributions:

• We survey and qualitatively compare the existing hybrid blockchain database systems.

• We provide flexible implementations of BlockchainDB and Veritas. In particular, we implement Veritas with Apache Kafka and Tendermint as mechanisms to broadcast the transactions. While Kafka provides crash fault tolerance, Tendermint is designed to work in a Byzantine environment, closer
to what the blockchains are supposed to do.

• We analyze the performance of five systems, namely, Veritas
(Kafka), Veritas (Tendermint), BlockchainDB, BigchainDB, and BigchainDB with parallel validation. Our analysis exposes the trade-offs to be considered by the developers to achieve the best performance in their application scenario.

• Among others, we show that Veritas (Kafka) exhibits a higher performance compared to all the other systems. For example, Veritas (Kafka) exhibits close to 30,000 TPS, while Veritas (TM), BigchainDB, and BigchainDB (PV) exhibit close to 1,700, 180, and 180 TPS, respectively, on networks of four peers. On the other hand, BlockchainDB exhibits less than 100 TPS. While there is room for optimization in all the systems, the significant gap in performance between CFT and BFT consensus mechanisms will be hard to reduce.

The rest of this paper is organized as follows. In Section 2 we survey the existing hybrid blockchain database systems. In Section 3, we first describe our implementations of Veritas and BlockchainDB, followed by the existing open-source implementtation of BigchainDB. In Section 4 we conduct our performance analysis. In Section 5 we discuss the trade-offs and challenges that users and developers encounter when using a hybrid blockchain database system. Finally, we conclude the paper in Section 6.

II. BACKGROUND AND RELATED WORK

In the last few years, there have been a few works published in database conferences that integrate blockchains and databases [22,
30, 31, 37, 42, 46]. Such systems allow companies to do business with peace of mind because every database operation is tracked by a distributed ledger. However, different business scenarios have different

requirements, and as a result, many hybrid blockchain database systems have been built for different application scenarios.

In this section, we describe the existing *hybrid blockchain database systems* and we briefly mention some similar systems that can be classified as *ledger databases*.

2.1 Hybrid Blockchain Database Systems

Veritas [22] is a shared database design that integrates an underlying blockchain (ledger) to keep auditable and verifiable proofs.

The interaction between the database and the blockchain is done through verifiers. A verifier takes the transaction logs from the database, makes a correctness-checking decision, and sends it to the other verifiers. The final decision agreed by all the verifiers is

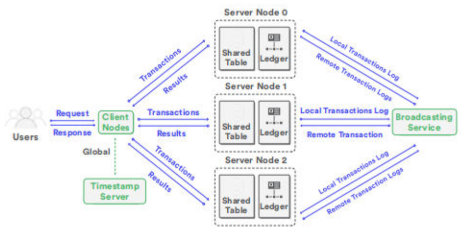


Figure 1: Veritas (Kafka) with Apache Kafka

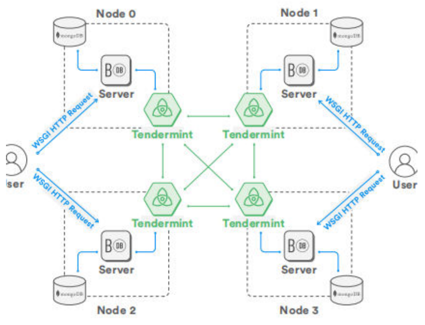


Figure 2: BigchainDB

recorded onto the blockchain. Hence, the correctness of the data can be verified based on the history stored in the blockchain.

An alternative design of Veritas [22], which is selected by us to be implemented in this paper, uses a shared verifiable table as its key storage. In this design, each node has a whole copy of the shared table and tamper-proof logs in the form of a distributed ledger, as shown in Figure 1. The ledger stores the update (write) logs of the shared table. Each node sends its local logs and receives remote logs via a broadcasting service. The verifiable table design of Veritas uses timestamp-based concurrency control. The timestamp of a transaction is used as the transaction log's sequence number, and each node has a watermark of the committed log sequence. When a transaction request is sent to a node, it first executes the transaction locally and buffers the result in memory. Then, it sends the transaction log to the other nodes via the broadcasting service.

The node flushes the buffer of transactions and updates the watermark of committed logs as soon as it receives approval from all the other nodes.

BigchainDB [42] uses MongoDB [3] as its storage engine. That is, each node maintains its local MongoDB database, as shown in Figure 2. MongoDB is used due to its support for assets, which is the main data abstraction in BigchainDB. Tendermint [8] is used for consensus among the nodes in BigchainDB. Tendermint is a BFT consensus protocol and it guarantees that when one node is

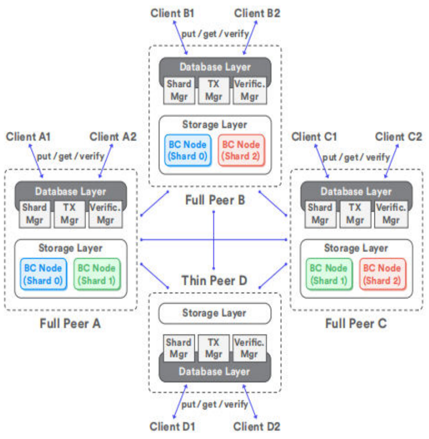


Figure 3: BlockchainDB

controlled by a malicious hacker, the MongoDB databases in the other nodes will not be affected. When a node receives an update request from a user, it first generates the results locally and makes a transaction proposal to be sent to the other nodes via Tendermint.

The node commits the buffered result and responds to the user client as soon as most of the nodes in BigchainDB reach a consensus on this transaction.

BlockchainDB [30] adopts a design that builds a shared data ase on top of a blockchain. It is different from the other systems because it partitions the database into a few shards, as shown in Figure 3, such that the overall storage overhead is reduced. While some storage is saved, this design induces a higher latency since a data request may need to do an additional lookup to locate the corresponding shard. In BlockchainDB, each peer integrates a shard manager to locate the shard where a specific key is located. In terms of verification, it provides both synchronous (online) and asynchronous (offline) verification which is done in batches.

FalconDB [46] is a system that provides auditability and verifiability by requiring both the server nodes and the clients to keep a digest of the data. The server nodes of FalconDB keep the shared database and a blockchain to record the update logs of the shared database. Client nodes only hold the block headers of the blockchain kept by the server nodes . Using these headers, the clients are able to verify the correctness of the data obtained from the server nodes.

These client nodes act as intermediaries between the users and the actual database. ChainifyDB [37] proposes a new transaction processing model called Whatever-Ledger Consensus (WLC). Unlike other processing models, WLC makes no assumptions about the behavior of the local database. The main principle of WLC is to seek consensus on the effect of transactions, rather than the order of the transactions.

When a ChainifyDB server receives a transaction from a client, it asks for help from the agreement server to validate the transaction and then sends a transaction proposal to the ordering server. The ordering server batches the proposals into a block with FIFO order and distributes the block via Kafka [20]. When the transaction is approved by the consensus server, it will finally be executed in-order in the execution server's underlying database.

At a high level, Blockchain Relational Database [31] is very similar to Veritas [22] . However, in Blockchain Relational Database [31] (BRD), the consensus is used to order blocks of transactions, and not to serialize transactions within a single block. The transactions in a block of BRD are executed concurrently with Serializable Snapshot Isolation (SSI) on each node and they are validated and committed serially. PostgreSQL [5], which supports Serializable Snapshot Isolation, is used as the underlying storage engine in BRD.

The transactions are executed independently on all the "untrusted" databases, but then they are committed in the same serializable
order via the ordering service.

2.2 Ledger Databases

Different from hybrid blockchain database systems, ledger databases [6, 12, 26, 44, 48] are centralized, as the ledger is kept by a single organization. In this paper, we briefly describe some of the existing ledger databases. However, we are not evaluating and analyzing their performance.

Amazon Quantum Ledger Database [6] (QLDB) contains an immutable journal that documents every data change in a precise and sequential manner. The journal is made up of append-only blocks that are arranged in a hash chain. This means that data can only be appended to the journal and cannot be overwritten or deleted. The entire journal is designed as a Merkle Tree, allowing users to trace and check the integrity of data changes.

Immudb [12] is a lightweight, high-speed immutable database with built-in cryptographic proof and verification. Immudb is written in pure Go, with BadgerDB [17] as its storage engine. Badger is a fast key-value database implemented in pure Go, which uses an LSM tree structure. Immudb guarantees immutability by using a Merkle Tree structure internally where data is hashed with SHA- 256. Moreover, immudb builds a consistency checker to check the correctness of data periodically.

Spitz [48] is a ledger database that supports a tamper-evident and immutable journal of transactions. It uses Forkbase [43] as its underlying storage, which provides a git-like multi-version control for data. Spitz provides a cell store for storing data and an append only ledger to store the journal of transactions. Moreover, it builds a Merkle Tree based on the ledger to provide verifiability.

LedgerDB [44] is a centralized database from Alibaba Group. It uses TSA time notary anchors to provide auditability. These anchors are generated by a two-way peg protocol [41]. What is different in LedgerDB compared to the previous ledger databases is that it supports not only create, update, and query methods but also purge and occult methods for verifiable data. With these methods, LedgerDB aims to meet the requirements of the real world. However, it may destroy immutability while providing strong verifiability. As for the underlying storage, LedgerDB supports file systems including HDFS [39], key-values stores such as RocksDB [2], Merkle Patricia Tree [47] and a linear-structured append-only file system called L-Stream [44] which is specially designed for LedgerDB.

Table 1: Hybrid Blockchain Database Systems

System	Storage	Consensus	User API
Veritas (Kafka) [22]	Redis	Kafka (CFT)	KV-Store
Veritas (TM)	Redis	Tendermint (BFT)	KV-Store
BigchainDB [42]	MongoDB	Tendermint (BFT)	KV-Store
BlockchainDB [30]	RocksDB (Geth)	PoW Consensus (Geth)	KV-Store
FalconDB [46]	MySQL	Tendermint (BFT)	SQL
BRD [31]	PostgreSQL	BFT-SMaRt (BFT)	SQL
ChainifyDB [37]	MySQL	Kafka (CFT)	SQL

III. SYSTEMS UNDER TEST

In this section, we describe the systems analyzed in this paper. We start with Veritas (Kafka) which uses Apache Kafka for inter-node communication. Then we describe our modification of this system into Veritas (TM) which uses Tendermint for the broadcasting service. Next, we present our implementation of BlockchainDB. We end by describing the existing implementation of BigchainDB.

3.1 Veritas (Kafka)

3.1.1 Overview. As shown in Figure 1, the key components of Veritas are the server nodes, client nodes, timestamp service, and broadcasting service. Our implementation, named

Veritas (Kafka)

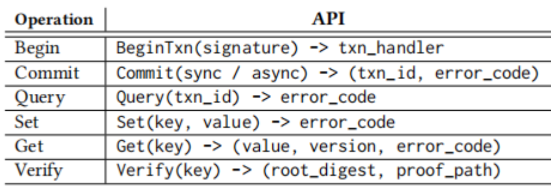
uses Apache Kafka [20], a CFT pub-sub system, as broadcasting service. Next, we describe the functionality of each component of Veritas.

Server nodes keep both the shared database and the full ledger that contains all the logs of the transactions done on the shared database. Server nodes handle query and update requests received via the client nodes. They also need to provide proofs of data integrity and accuracy as part of the verify requests.

Client nodes
act as intermediaries between users and server nodes. Upon getting a request, a client node first gets a global timestamp from the timestamp service. This timestamp represents the unique identifier of the transaction throughout its lifetime.
Timestamp service
generates global timestamps used by the client nodes. This service needs to provide unique monotonically increasing times tamps. Moreover, this service should exhibit strong availability and high performance. **Broadcasting service**
gets local transaction logs from a server node and distributes them to all the other server nodes. Similar to the timestamp service, it should exhibit strong availability and high performance.

When a transaction contains an update operation, the server node first executes the transaction locally and buffers the result in memory. Then, it sends the transaction log to the broadcasting service which distributes it to the other server nodes. When the initial server node receives the approvals from all the other server

Table 2: Veritas User API



Operation	API
Begin	BeginTxn(signature) -> txn_handler
Commit	Commit(sync / async) -> (txn_id, error_code)
Query	Query(txn_id) -> error_code
Set	Set(key, value) -> error_code
Get	Get(key) -> (value, version, error_code)
Verify	Verify(key) -> (root_digest, proof_path)

nodes, it commits the buffered result to the local database and appends the transaction log to the ledger.

3.1.2 User API. Veritas treats all user requests as transactions [22].

Each transaction contains one or more operations. The operations supported by Veritas are shown in Table 2 and explained in the following lines. *Begin* starts a transaction using the user's signature.

It returns a transaction handler to deal with the next operations of the transaction. *Commit* finalizes a transaction. It has two modes of operation. The first mode is synchronous, where the user needs to wait for the transaction commit result. The second mode is asynchronous, where the user does not need to wait for the result of the Commit operation. When the user is using the asynchronous Commit, she needs to use the Query operation to get the result of the entire transaction. *Query* checks the status of a transaction. As all the transaction logs are stored in the ledger, Veritas can easily find whether the specified transaction is committed or aborted.

Set updates the value of a specified key. The value is linked to the transaction unique identifier, which is recorded on the ledger. *Get* retrieves the value of a specified key. In our implementation of Veritas, we guarantee that the user reads the latest committed value of the key. *Verify* traces the proof path of the specified key in the Merkle Tree. Users can verify the correctness of the value by calculating the root digest of the proof path, and then compare the result with the expected root digest.

3.1.3 Implementation Details. We implemented Veritas (Kafka) in 804 lines of Go code, using Redis [10] (v6.2.1) as the underlying database for shared tables. Redis is an in-memory key-value store that exhibits very high throughput, of up to 100,000 operations per second [4]. To store the ledger, we use BadgerDB [17] which is a database engine based on LSM trees and optimized for SSD.

BadgerDB stores the keys and values separately to reduce the IO cost. Moreover, it stores pointers to the values in the LSM tree to save the compaction cost.

In the original design of Veritas [22], the ledger is stored using a Merkle Tree [29]. While it is easy to verify that something is part of a Merkle Tree, it takes more effort to prove that something is not in the tree. For this reason, we adopt a Sparse Merkle Tree [15] in our implementation. This kind of tree contains a leaf for every possible result of a cryptographic hash function, making it easy to show that certain data is not part of the tree. On the other hand, a Sparse Merkle Tree needs more storage space compared to a Merkle Tree.

The timestamp service implementation is based on Timestamp Oracle (TSO) [32] which ensures that the clock assigned to an event will not repeat. In Veritas, this timestamp service ensures that a unique and monotonically increasing timestamp is applied

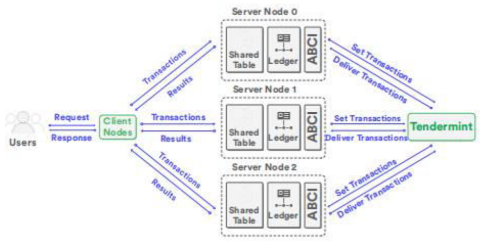


Figure 4: Veritas (TM) with Tendermint

to each transaction. Other systems, such as TiDB [25, 33], also use Timestamp Oracle. In our implementation, the TSO is crash fault-tolerant by using a persisted write-ahead log. The broadcasting service is based on Apache Kafka [20] (v2.7.0), which is a distributed messaging system that provides high through put. Kafka ensures crash fault tolerance (CFT) by persisting the messages to the disk and replicating them across nodes.

3.1.4 Complexity Analysis. Next, we analyze the complexity of Veritas (Kafka) in terms of the number of messages the nodes need to exchange per block of updates. We suppose there are NVeritas server nodes, KKafka nodes, and the replication factor of Kafka is R. In practice, this replication factor is set to three [1]. In Veritas (Kafka), a node sends a block of updates to the Kafka service. Besides replicating it on Rreplicas, the Kafka service sends the block to $N-1$ Veritas nodes. Each node checks and applies the updates, after which it sends an acknowledgement to Kafka. Then, Kafka sends each of the $N-1$ acknowledgements to the other $N-1$ nodes, resulting in a message complexity of (N2) for Veritas (Kafka).

This result is backed by experimental evaluation in Section 4.3. In terms of storage complexity, we note that each block is persisted in the Sparse Merkle Tree of each Veritas node, and also on RKafka nodes. Hence, the storage complexity is ($N+R$).

3.2 Veritas (TM)

3.2.1 Overview. In general, blockchains operate in a Byzantine environment [18, 35]. Hence, using a CFT broadcasting service or

consensus protocol, such as Apache Kafka or Raft, is unsuitable. That is why we implement Veritas with Tendermint [8], a BFT consensus protocol used by other systems, such as BigchainDB [42] and FalconDB [46]. Tendermint is a BFT middleware that supports a state transition machine. Similar to other BFT systems, Tendermint supports up to 1/3 malicious nodes in the network. Figure 4 depicts the architecture of **Veritas (TM)**. We use Tendermint Core as the consensus and broadcasting service and an Application BlockChain Interface (ABCI) application integrated with the Veritas node to interact with Tendermint. A *Get* request is served directly by the Veritas node. In contrast, a

Set transaction is sent to Tendermint via ABCI. When the transaction is included in a block and delivered back to the Veritas node via ABCI, it is applied to the local (Redis) database. This mechanism ensures a serial con currency control in Veritas (TM). Different from Veritas (Kafka), we do not use a timestamp oracle (TSO) service in Veritas (TM). Such a TSO service is centralized in nature, thus, incompatible with a BFT setting.

3.2.2 Implementation Details. Veritas (TM) is implemented in 517 lines of Go code, with Tendermint [8] (v0.35.0) as its consensus service, which is to be consistent with BigchainDB. The ledger module, verifiable data structure, concurrency control, and local database are the same as for Veritas (Kafka). That is, we use a Sparse Merkle Tree as the verifiable data structure and Redis (v6.2.1) as local database.

3.2.3 Complexity Analysis. The message complexity of Veritas (TM) is determined by the complexity of Tendermint, which is shown to be $\mathcal{O}(N^3)$ by a recent study [7], where N is the number of nodes in the system. We shall see in Section 4.3 that the performance of Veritas (TM) decreases drastically with the number of nodes. We attribute this to the message complexity of Tendermint. The storage complexity of Veritas (TM) is $\mathcal{O}(N)$ since each block is stored on all N nodes.

3.3 BlockchainDB

3.3.1 Overview. As shown in Figure 3, a BlockchainDB [30] node is mainly composed of a storage layer and a database layer. The storage layer uses an off-the-shelf blockchain as a shared data base which provides tamper-proof and de-centralized storage. A blockchain connector component, which is implemented for differ ent blockchain clients, handles the interactions with the blockchain network. The database layer is built on top of the storage layer with a simple read/write interface to access data from a shared table. The database layer in BlockchainDB is responsible for handling the *Set* and *Get* requests from clients. Different from all the other systems under test, BlockchainDB supports sharding. A sharding manager component defines a partition scheme based on a hash algorithm and stores the connection information for each shard.

3.3.2 User API. As shown in Table 3, BlockchainDB [30] supports three operations, namely, *Set*, *Get*, and *Verify*. The *Set* method writes a key-value pair to the given shared table and returns a correspond ing blockchain transaction id. The *Get* method retrieves the data corresponding to a key from the given shared table. The *Verify* method allows users to check the status of the given transaction id to verify if the operation was successfully committed.

3.3.3 Implementation Details. In this paper, BlockchainDB is implemented in Go, where each node runs an RPC server serving clients requests. We use a private Ethereum (geth/v1.8.23-stable) blockchain with Proof-of-Authority (PoA) consensus as the storage for experiments. A KVStore contract as defined in [30] is installed on the Ethereum network.

3.3.4 Complexity Analysis. BlockchainDB is a sharded system, hence, we analyze the communication and storage overhead for a single shard. The total overhead is the sum of the overheads of each shard. Given the Ethereum backend used by BlockchainDB, the communication complexity is (N), where N is the number of nodes in a shard. This is due to the block broadcasting phase in Ethereum. The storage complexity is (N) since the ledger is stored by each node.

Table 3: BlockchainDB User API

Operation	API
Set	`Set(table, key, value) -> void`
Get	`Get(table, key) -> value`
Verify	`Verify() -> bool`

Table 4: BigchainDB User API

Operation	HTTP Method	URL
Query	GET	`/api/v1/transactions/{transaction_id}`
Create	POST	`/api/v1/transactions?mode={mode}`
Transfer	POST	`/api/v1/transactions?mode={mode}`

Table 5: A Comparison of Features in the Systems Under Test.

	Veritas (Kafka)	Veritas (Tendermint)	BlockchainDB	BigchainDB
Fault Tolerance	CFT (Kafka)	BFT (Tendermint)	BFT (via Ethereum)	BFT (Tendermint)
Immutability	Append-only Ledger	Append-only Ledger	Ledger (Ethereum)	Ledger (Tendermint)
Verification	Sparse Merkle Tree	Sparse Merkle Tree	Merkle Patricia Trie	Merkle Tree
Concurrency	MVCC	Serial	Serial	Blockchain Pipelining
Storage	NoSQL (Redis)	NoSQL (Redis)	Ethereum (geth)	NoSQL (MongoDB)
Replication	Full Replication	Full Replication	Sharding	Full Replication

3.4 BigchainDB

3.4.1 Overview. As shown in Figure 2, a BigchainDB node consists of three key components, namely, the server, consensus component (Tendermint), and local database (MongoDB). **BigchainDB Server** provides an HTTP service that handles user requests. This service uses the Flask [23] web application framework working with Web Server Gateway Interface (WSGI) of Gunicorn [11] to expose an HTTP API. **Tendermint Consensus Component** provides a broadcasting service for the transaction blocks. It has the role of a bridge among BigchainDB nodes and it is responsible for proposing new transaction blocks and ensuring that all nodes agree on a block in a Byzantine fault-tolerant manner. After validating a transaction block, the Tendermint component sends a commit message to the local BigchainDB server to signal the commit of the transactions. **MongoDB Database Component** persists the data which can only be modified by the local BigchainDB server.

The exposed API of the MongoDB component is determined by the local BigchainDB server. Different MongoDB instances may provide different functions.

BigchainDB uses a concept called blockchain pipelining, which improves scalability when voting for the next blocks. In a blockchain, the transaction blocks are ordered which means that nodes cannot vote for a new block while the current block is undecided. This is because the new block needs to have a reference to a decided block. In BigchainDB, the blocks are also ordered, but server nodes are allowed to vote for a new block even if the current block is undecided. Using a voting list created at the same time with a block, expected voters are tracked while the list contains a reference to the current unsettled block. However, when voting for a block with an undecided parent block, a node has to verify that the block does not contain transactions with dependencies in the undecided block. This is a form of concurrency control adopted by BigchainDB [42]. We shall see in the next section that the validation process in BigchainDB slows down the entire system.

The transaction flow in BigchainDB can be described as follows. When a BigchainDB server receives an HTTP request with a transaction, it first performs some checks to validate the transaction.

That is, the node checks if this is not a duplicate transaction in both the transaction queue and the ledger. It also checks the inputs and outputs of the transaction. Next, it calls the *Broadcast API* provided by the local Tendermint component for the broadcasting of transactions. After receiving the commit message of the transaction from
the Tendermint component, the server updates the state of the local MongoDB instance.

3.4.2 User API. As shown in Table 4, BigchainDB supports three operations: query, create, and transfer. *Query* retrieves the details of a transaction based on its transaction id. If the transaction has beencommitted, a BigchainDB server returns the details of the transaction. Otherwise, a 404 error code is returned. *Create* has the role to
create assets for the specified user and to store them in BigchainDB. A Create transaction supports three modes, namely, async, sync, and commit. The default mode is async where a BigchanDB serverresponds before the Tendermint component validates the transaction. In sync mode, a BigchainDB server responds after the transaction block has been committed and the state in the MongoDB instance has been modified. Lastly, in commit mode, a BigchainDB server responds after checking the validation process of the block containing the transaction. Finally, *Transfer* has the role to transfer assets from one user to another. It also provides the same three modes for transaction processing as the Create API.

3.4.3 Implementation Details. In this paper, we use the open-source BigchainDB [42] (v2.2.2) with MongoDB [3] (v4.4.4) and Tendermint [8] (0.31.5). In addition to the standard system, we also evaluate BigchainDB with Parallel Validation feature

IV. PERFORMANCE ANALYSIS

In this section, we analyze the performance of the five systems described in the previous section, namely, Veritas (Kafka), Veritas (TM), BigchainDB, BigchainDB (PV), and BlockchainDB. Next, we describe the experimental setup.

4.1 Setup

All the experiments are executed on a local machine under Ubuntu 18.04 operating system (OS). The machine has 256 physical CPU cores, 8 TB of RAM, and 3TB of hard-disk (HDD) storage. Using iostattool, the IOPS of the machine is estimated at 5.35. All the server and client nodes of the systems under test are running on this machine in Docker containers on different CPU cores.

To evaluate the performance of the five systems under test, we send 100,000 transactions to each system. We send multiple transactions in parallel to a system, based on a concurrency parameter that represents the number of clients. The transactions are evenly distributed to different server nodes in the system. To compute the throughput, we record the start time of the first transaction and the completion time of all the transactions. Then, we record the number of successfully committed transactions and compute the throughput by dividing this number by the previously recorded time interval. Note that we only consider successful transactionswhen computing the throughput, but there may be failures as well.

We repeat each experiment three times and report the average. Before executing the transactions, we load 100,000 key-value records of 1,000 bytes each into each system.

We use the Yahoo! Cloud Serving Benchmark [13] (YCSB) dataset which is widely used to benchmark databases and blockchains [19, 35]. YCSB supports common database operations such as write (insert, modify, and delete), and read. While YCSB defines six workloads, in our experiments, we choose three of these six workloads. These three workloads are (i) *Workload A* which consists of 50% update operations and 50% read operations, (ii)

Workload B which consists of 5% update operations and 95% read operations, and (ii) *Workload C* which consists of 100% read operations. Moreover, we use three key distributions, namely, (i) *uniform distribution* which operates uniformly on all the keys, (ii) *zipfian distribution* which operates frequently only on a sub-set of the keys, and (iii) *latest distribution* which operates on the latest used keys. We use Workload A and uniform distribution by default, unless otherwise specified.

The benchmarking tools are implemented by us in Go using *goroutines* [14] and a *channel* [14] as a concurrent safe request queue. A channel allows goroutines to synchronize without explicit locks or condition variables. Each benchmarking client is represented by a goroutine and it gets a new request from the channel once it
completes the current request.

4.2 Effect of Number of Clients

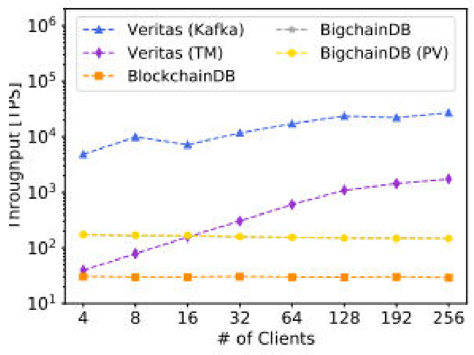
We first analyze the effect of increasing the number of client nodes to determine the saturation points of the systems. In this experiment, we set the number of server nodes in each system to four. This number is derived based on the fact that BFT systems supporting up to f faulty nodes need to have a total of at least $3f+1$ nodes. Hence, for tolerating one faulty node, we need four nodes in the system.

For consistency, we also set the number of Veritas (Kafka) server nodes to four. We use YCSB Workload A with uniform distribution and set the block size of the ledger to 100 transactions. We then increase the number of clients from 4 to 256.

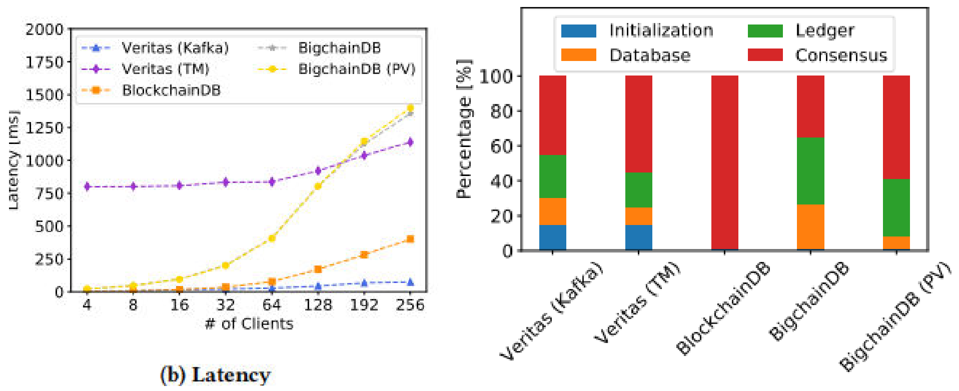
Figures 5a and 5b show the effect of increasing the number of clients on throughput and latency, respectively. We observe that the throughput of the BFT systems plateaus after using a certain number of clients, while for Veritas (Kafka) it is growing even if at a slow pace. These results are partially correlated with the latency:

we observe a sharper increase in the latency of the BFT systems when using more than 32 clients. Compared to BigchainDB, the throughput of Veritas (TM) becomes much higher starting from 32 clients. Even if both Veritas (TM) and BigchainDB use Tendermint as the underlying consensus protocol, their performance is very different. Veritas (TM) achieves its top performance of 1,742 TPS with 256 clients, while BigchainDB only achieves 175 TPS and this when using four clients. Increasing the number of clients in BigchainDB results in a sharp increase in latency while the throughput remains relatively constant.

To investigate the reasons for BigchainDB's low performance, we analyze the time spent in each major component of a hybrid blockchain database system. As presented in Section 2, each node of such a hybrid system has an underlying shared *database*, a *ledger* data structure and storage, and a *consensus* or broadcasting component. As shown in Figure 6, Veritas (Kafka), Veritas (TM), and BlockchainDB spend 45%, 55%, and 99% of their time in the con
sensus component. On the other hand, BigchainDB spends 38% of its time validating transactions, as part of the ledger component. For example, BigchainDB checks for duplicate transactions both in the transaction queue (in memory) and in the database (in MongoDB, on disk). This operation is very costly and implies sequential processing. That is why the latency increases significantly when more transactions are sent to the system at the same time (when increasing the number of clients). In BigchainDB (PV), transactions are checked in parallel but the overhead of parallel processing is around 11%. This, together with other ledger operations lead to 33% of the time spent in the ledger component. The database part in

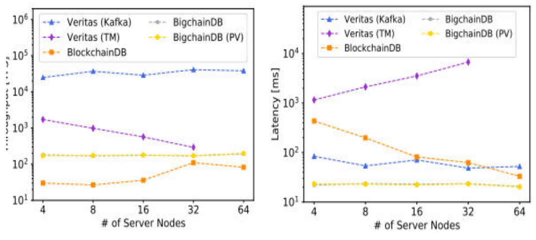


(a) Throughput



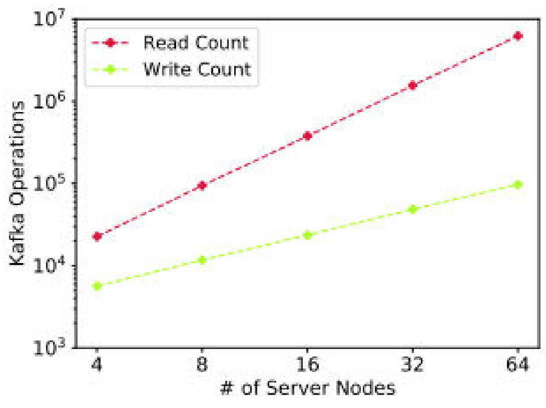
(b) Latency

Figure 5: The Effect of the Number of Clients



(a) Throughput (b) Latency

Figure 7: The Effect of the Number of Server Nodes



igure 8: The Effect of Veritas Server Count n Kafka Operations

BigchainDB (PV) is much lower compared to BigchainDB due to the bulk storage of transactions. However, the consensus accounts for 58.5% in BigchainDB (PV) compared to 35% in BigchainDB. We attribute this to the larger consensus message size in BigchainDB (PV) compared to BigchainDB.

Secondly, we observe that the performance of Veritas (Kafka) is more than 10× higher compared to the BFT systems. In particular, Veritas (Kafka) exhibits a maximum of 27,335 transactions per second (TPS) when 256 clients are used, while Veritas (TM) achieves 1,742 TPS with 256 clients. BigchainDB and BigchainDB (PV) exhibit a maximum of 175 and 174 TPS, respectively, when using four clients. On the other hand, BlockchainDB achieves only 30 TPS, but this is expected due to the use of Ethereum as the underlying data storage. Even when using the PoA consensus, the throughput of Ethereum is below 100 TPS [19]. In terms of average latency, the corresponding values are 74, 1139, 400, 23, and 23 milliseconds (ms) for Veritas (Kafka) with 256 clients, Veritas (TM) with 256 clients, BlockchainDB with 256 clients, BigchainDB and BigchainDB (PV) with four clients, respectively.

Thirdly, we observe that the throughput of Veritas (TM) is 10× higher compared to BigchainDB. As shown in Table 5, the difference between these two systems is at storage and concurrency control layers. At the storage layer, Veritas (TM) is using Redis, while BigchainDB is using MongoDB. Redis has a higher throughput than MongoDB, being an in-memory database. However, the throughputs of Veritas (TM) and BigchainDB are far lower than what Redis and MongoDB can support. To investigate if MongoDB has a significant negative impact on the performance, we replace Redis with MongoDB in Veritas (TM) and run the benchmarking again. The performance of Veritas (TM) with MongoDB is up to 22% lower compared to using Redis. In conclusion, the huge difference between Veritas (TM) and BigchainDB is due to the design and implementation of the latter, especially the complex and redundant transaction validation mechanism.

4.3 Effect of Number of Server Nodes

Next, we analyze the effect of increasing the number of server nodes on throughput and latency. We start from four nodes, the minimum number of nodes in a BFT system to support one faulty node, and increase the number of nodes up to 64. A network of 64 nodes can support 21 faulty nodes in a BFT system. Similar to the previous experiment, we use YCSB Workload A with uniform distribution and a block size of 100 transactions. We set the number of clients to 256 for Veritas (Kafka), Veritas (TM), and BlockchainDB, and to four for BigchainDB and BigchainDB (PV).

As shown in Figure 7, we observe the same gap in performance among Veritas (Kafka) and the BFT systems. Note that the latency of BlockchainDB only includes the time it takes for a request to be included in the transaction pool of Ethereum. It does not include the block confirmation time. The throughput of Veritas (TM) drops

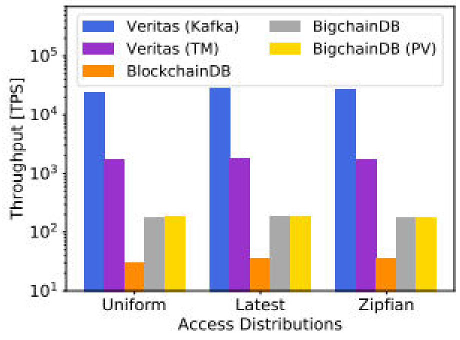
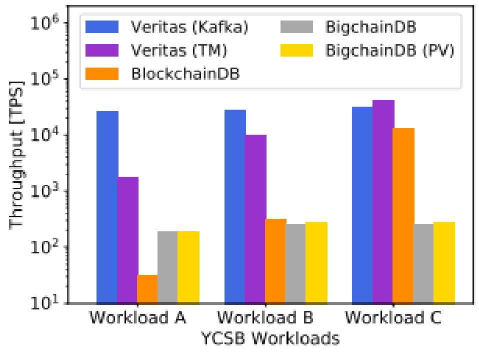


Figure 9: The Effect of Key Access Distribution on Throughput



igure 10: Performance of ifferent YCSB Workloads

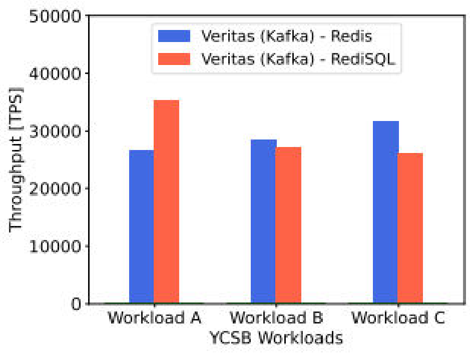


Figure 11: The Effect of the Underlying Database

from 1,722 TPS on four nodes to 292 TPS on 32 nodes. On 64 nodes, Tendermint (v0.35.0) stops working due to synchronization errors among the peers. Again, this trend can be explained by the high message complexity of Tendermint, namely, (*N*3), where *N*is the number of nodes. In BigchainDB and BigchainDB (PV), the impact of Tendermint is amortized because the system is slow in sending new transactions to the consensus component. As explained before, BigchainDB and

BigchainDB (PV) perform complex and redundant transaction validations which slow down the entire system.

In Veritas (Kafka), the throughput increases with the number of server nodes, from 25,009 TPS on four nodes to 41,262 TPS on 32 nodes. On 64 nodes, the throughput decreases to 38,283 TPS. We attribute this to the interplay between the number of server nodes and the Kafka broadcasting service. With four server nodes, there are not enough transactions exchanges to saturate Kafka which supports up to 200,000 messages per second. Also, there are too few server nodes to process the transactions as opposed to 32 nodes that can process more transactions. But when we increase the number of nodes beyond 32, Kafka starts to become a bottleneck. For example, with 64 nodes, there are close to 100,000 writes and 6,200,000 reads in the Kafka service.

In Figure 8, we plot the number of read and write operations in Kafka as the number of server nodes in Veritas (Kafka) grows, using a logarithmic scale. The number of write operations grows linearly with the number of server nodes because, for each transaction sent by a node, the other N– 1 nodes need to write (send) an acknowledgment to Kafka. On the other hand, the number of read operations grows quadratically with the number of server nodes. This is because each of those N– 1 acknowledgments are read by all the other N– 1 nodes in the system. This results in a message complexity of (N2) in Veritas (Kafka), where $N$is the number of server nodes.

4.4 Effect of Access Distributions

In this section, we evaluate the performance of the five systems with different access distributions, namely uniform, latest, and zipfian with a coefficient of 0.5. As shown in Figure 9, we do not observe significant differences among these distributions. This can be explained by the fact that there is no significant number of aborted transactions in the evaluated systems. For example, Veritas (TM) and BlockchainDB exhibit no aborted transactions due to the serial transaction commit. Even in Veritas (Kafka) with MVCC, there are up to 20 aborted transactions out of 100,000 total transactions, when using the zipfian distribution. We attribute this low number of conflicts to the fact that the version control is based on the centralized global timestamp service. This, combined with a fast Kafka delivery service, make conflicts unlikely.

In Veritas (Kafka), serialization is guaranteed by the broadcasting service. Server nodes send both new blocks and approvals of blocks concurrently, with no blocking. When a server node receives a new block, it first checks the conflicts between the transactions in that block and the current state of the underlying database. If there is no conflict, it sends an approval message with its signature via Kafka and buffers this block in memory. If there are conflicts, the server node sends an abort message via Kafka. The block is finally committed once the server node receives all the approval messages from the other server nodes. Hence, conflict checking and new block generation happen in parallel in Veritas (Kafka). But this leads to more conflicts and these conflicts are detecting during the checking phase.

4.5 Effect of YCSB Workloads

Next, we evaluate the performance of the five systems with three different YCSB workloads, namely, Workload A, Workload B, and Workload C. The number of transactions in each block of the shared log is 100, the number of server nodes is set to four, and the number of clients in each system are those that exhibit the best performance. As expected, all the systems exhibit higher performance when the number of read operations is higher. Recall that the proportion of read operations is 50%, 95%, and 100% in Workloads A, B, and C, respectively. As shown in Figure 10, the gap between Veritas (Kafka) and Veritas (TM) becomes smaller as the number of read (or *Get*) operations is higher. For example, Veritas (TM) achieves higher throughput with Workload C compared to Veritas (Kafka). This is because *Get* operations are served immediately by Veritas (TM), while Veritas (Kafka) needs to apply the MVCC mechanism that introduces a delay. For the same reason of serving *Get* operations immediately, BlockchainDB achieves an impressing throughput of 13,124 TPS with Workload C, compared to just 30 TPS with Workload A. On the other hand, BigchainDB achieves only 257 TPS with Workload C. We attribute this to the complex asset query in MongoDB used by BigchainDB to serve a *Get* operation.

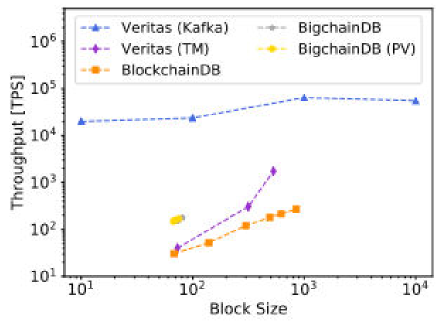


Figure 12: The Effect of Block Size on Throughput

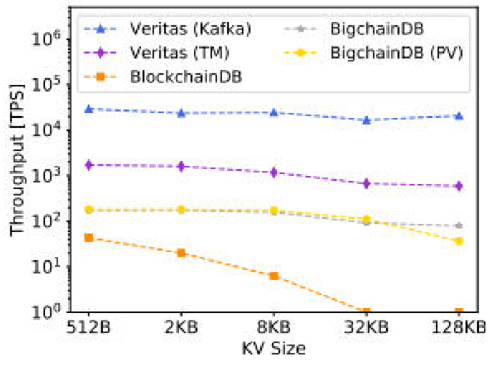


Figure 13: The Effect of Record Size on Throughput

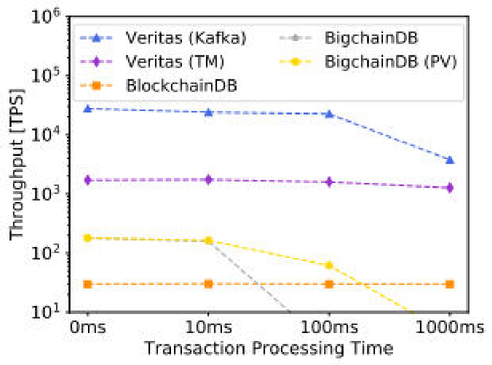


Figure 14: The Effect of Processing Time on Throughput

V. DISCUSSION

In this section, we discuss some trade-offs that users and developers need to consider when using and designing a hybrid blockchain database system. We look at these trade-offs from the perspective of performance, security, and ease of development.

CFT vs. BFT. Adopting a CFT or BFT consensus protocol depends on the environment where the system is supposed to operate. Nonetheless, it is well-known that BFT consensus is much slower than CFT consensus [16, 35]. In this paper, we show that the gap between the CFT and BFT systems throughput is 5 − 500×. We also show that the performance of both CFT and BFT systems degrades under poor real-world networking conditions. While typical blockchains employ BFT consensus protocols due to their security guarantees in a Byzantine environment, some permissioned blockchains such as Hyperledger Fabric [21] and R3 Corda [24] are employing CFT consensus protocols, while delegating the security aspects to the blockchain access/membership layer. When choosing BFT, the main focus of a developer is on optimizing the performance since the security guarantees are relatively high. Query optimization and using high-performance underlying storage are the common methods of optimizing such systems. Optimizing the performance of the BFT consensus is an important aspect that is studied in recent works [8, 28, 45]. In contrast, CFT protocols usually provide high performance and developers need to add verifiable components to provide security guarantees such as building an append-only ledger to provide verifiability.

In summary, CFT hybrid blocAkchain database systems should be used when there is a demand for high performance without strict security guarantees, such as in online transaction processing (OLTP) systems. When starting with an existing CFT system, the developer can add blockchain features, such as an append-only ledger, to improve security. BFT hybrid blockchain database systems should be used when there is a need for high security, such as for digital assets storage. When starting with an existing BFT system, the developer can add database features, such as parallel validation and query optimization, to improve the performance.

VI. CONCLUSIONS

In this article, we examine the hybrid blockchain database solutions that have been put out by the database community in recent years.

Next, using Veritas as a basis, we do an extensive performance study of five hybrid blockchain database system implementations [22],

BigchainDB [42], and BlockchainDB [30]. We may alter the system's essential elements, such the underlying database and consensus middleware, to examine how they affect performance thanks to our adaptable Veritas implementation. For instance, we find that performance decreases by more than 15× when we switch out the CFT Apache Kafka middleware with the BFT Tendermint middleware.

Nonetheless, Tendermint provides security assurances inside a Byzantine setting. However, we find that there are only minor performance changes when we use RediSQL and MongoDB in lieu of the underlying Redis database. This indicates that the consensus procedures are the cause of the systems' subpar performance, along with a breakdown of the time spent in each key component of a hybrid blockchain database system.

REFERENCES

[1] Apache Kafka, Documentation, https://archive.fo/tYpo6, 2021.

[2] Facebook Open Source, RocksDB, https://rocksdb.org/, 2021.

[3] MongoDB Inc., MongoDB, https://www.mongodb.com/, 2021.

[4] Redis Labs, Redis, https://redis.io/topics/benchmarks, 2021.

[5] The PostgreSQL Global Development Group, PostgreSQL, https://www. postgresql.org/, 2021.

[6] Amazon, Amazon Quantum Ledger Database (QLDB), https://aws.amazon.com/ qldb/, 2021.

[7] Y. Amoussou-Guenou, A. D. Pozzo, M. Potop-Butucaru, S. Tucci Piergiovanni, Dissecting Tendermint, M. F. Atig, A. A. Schwarzmann, editors, Proc. of 7th International Conference on Networked

Systems, volume 11704 of Lecture Notes in Computer Science, pages 166–182, 2019.

[8] E. Buchman, Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, PhD thesis, The University of Guelph, 2016.

[9] V. Buterin, A Next-Generation Smart Contract and Decentralized Application Platform, http://archive.fo/Sb4qa, 2013.

[10] J. L. Carlson, Redis in Action, Manning Shelter Island, 2013.

[11] B. Chesneau, Gunicorn, https://gunicorn.org/, 2021.

[12] CodeNotary, immudb, https://codenotary.io/technologies/immudb/, 2021.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking Cloud Serving Systems with YCSB, Proc. of 1st ACM Symposium on Cloud Computing, page 143–154, 2010.

[14] K. Cox-Buday, Concurrency in Go: Tools and Techniques for Developers, O'Reilly Media, Inc., 1st edition, 2017.

[15] R. Dahlberg, T. Pulls, R. Peeters, Efficient Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs, Cryptology ePrint Archive, Report 2016/683, 2016.

[16] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, B. C. Ooi, Towards Scaling Blockchain Systems via Sharding, Proc. of 2019 International Conference on Management of Data, page 123–140, 2019.

[17] Dgraph, BadgerDB, https://github.com/dgraph-io/badger, 2021.

[18] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, J. Wang, Untangling Blockchain: A Data Processing View of Blockchain Systems, IEEE Transactions on Knowledge and Data Engineering, 30(7):1366–1385, 2018.

[19] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, K.-L. Tan, BLOCKBENCH: A Framework for Analyzing Private Blockchains, Proc. of 2017 ACM International Conference on Management of Data, page 1085–1100, 2017.

[20] A. S. Foundation, Kafka, https://kafka.apache.org/, 2017.