FalconDB: Blockchain-based Collaborative Database

Yanqing Peng
University of Utah
ypeng@cs.utah.edu

Min Du
UC Berkeley
min.du@berkeley.edu

Feifei Li
University of Utah
lifeifei@cs.utah.edu

Raymond Cheng
UC Berkeley
ryscheng@cs.berkeley.edu

Dawn Song
UC Berkeley
dawnsong@berkeley.edu

ABSTRACT

Nowadays an emerging class of applications are based on collaboration over a shared database among different entities. However, the existing solutions on shared database may require trust on others,

have high hardware demand that is unaffordable for individual users, or have relatively low performance. In other words, there is a trilemma among security, compatibility and efficiency. In this paper, we present FalconDB, which enables different parties with limited hardware resources to efficiently and securely collaborate on a database. FalconDB adopts database servers with verification interfaces accessible to clients and stores the digests for query/update authentications on a blockchain. Using blockchain as a consensus platform and a distributed ledger, FalconDB is able to work without any trust on each other. Meanwhile, FalconDB requires only minimal storage cost on each client, and provides anywhere-available, real-time and concurrent access to the database. As a result, FalconDB overcomes the disadvantages of previous solutions, and enables individual users to participate in the collaboration with high efficiency, low storage cost and blockchain-level security guarantees.

ACM Reference Format:
Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. 2020. FalconDB: Blockchain-based Collaborative Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3380594

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380594

1 INTRODUCTION

The growth of the Internet has triggered tremendous opportunities for data cooperation. Many applications could benefit from a shared database among multiple parties, such as collaborative benchmarking [4], crowdsourcing [43], sharing economy [5], cooperative scientific computation [14], and collaborative machine learning[7].

In such scenarios, a key opponent is a shared database management system that allows each party to execute updates and queries to the database, while maintaining a consistent view among all participants in the network. Many projects require collaborations among a group of *individual users*. As individual participants, they may need to collaborate through their personal devices with limited storage and computation power. Furthermore, they could be easily compromised and become malicious. Therefore, it is essential to design a shared database that enables individual users to collaborate with each other without any trust.

Traditionally, individual users who have limited computational power and local storage typically utilize a centralized server to facilitate collaboration. Such a service enables anywhere-available, real-time, and concurrent access to the database. However, it requires to fully trust the central server, which is expected to *correctly* execute all requests. One immediate concern is that a malicious server can fool any client without being detected. Another issue arises from the client side: a client may manipulate the records arbitrarily for its own interest. A sophisticated mechanism must be adopted to prevent clients from issuing undesired updates.

The emergence of blockchain techniques brings practical solutions to handle Byzantine failures [26], namely, a party could behave arbitrarily. Specifically, a permissioned blockchain platform could be viewed as a distributed system, where there is a chain of blocks that keep immutable records. The blocks of records are agreed by all blockchain nodes via some consensus protocol that tolerates Byzantine failures. Such platforms include Hyperledger [2], Tendermint [8], HotStuff [47] and BigchainDB [33]. This design does not require a centralized server, and works in an untrusted environment with up to a small proportion (e.g., 1/3) of nodes

being malicious. Undesired operations like malicious updates could be denied by the system.

Although blockchain systems have attracted much attention, it is still very rare for an individual user to participate as a node, due to the high cost. First, blockchain nodes need to locally store a full copy of the blocks, which could easily be hundreds of gigabytes, potentially exhausting local disk space for individual users. Second, blockchain nodes must maintain network communication with others to promptly receive and validate new blocks, which consumes both network bandwidth and computation power. Finally, since there is no longer a service provider to facilitate queries, users have to execute queries locally with their personal hardware. Executing complex queries with limited CPU power could be unacceptably slow.

The high cost on blockchain nodes limits the compatibility of blockchain. The concept of lightweight clients is proposed to alleviate the cost on individual participants [35]. Lightweight clients communicate with full nodes to update or query the blockchain database. Hence their capabilities are constrained by the APIs exposed by the full nodes. Moreover, query integrity needs to be ensured if the full nodes are not trustworthy [45]. A baseline solution is to leverage smart contracts [22]. A user could submit a query to a smart contract. Then, all full nodes execute the query and run a Byzantine fault tolerance (BFT) consensus protocol on the query result. Once a consensus is reached, the result will be committed to blockchain as well as revealed to the client. This approach guarantees integrity with a majority of honest nodes. However, it has limited throughput, high latency, high gas consumption and introduces privacy concerns.

None of the previous solutions is able to simultaneously achieve security, compatibility and efficiency. This becomes a paramount trilemma when we need all three properties. Consider the case of charitable giving, where a large number of donors make donations to multiple non-profit organizations (NGOs). An NGO needs to record each received donation, while an individual donor may want to audit how an NGO is spending her money. In fact, such a database already exists [13], where a centralized database server decides the execution results of all updates and queries. In such a design, the database host is able to hide any possible corruption or misusage of donations, while the users have no way to verify them. Therefore, a decentralized blockchain database solution is more desirable. However, existing blockchain database designs as mentioned above [33] are not sufficient because: 1) the individual contributors hardly have enough local storage to store the whole data; and 2) acting as a lightweight client and querying full nodes with smart contracts is not efficient enough. As a result, it is of great importance to design an *efficient* system that has *low cost* on users, but still enables a *traceable and tamper-free* historical record of charitable giving [40].

Our contribution. We present FalconDB, which is a blockchain database that has low hardware requirements (e.g., storage, computation, bandwidth) on individual clients, achieves query efficiency as high as a centralized database design, and ensures security guarantees as strong as blockchain level. In FalconDB, a blockchain node could either be a server node that stores the entire database, or a client node that uses the database by sending query and update requests to the server nodes. To grant the clients with the ability to verify if an update or query is correctly executed, the database contents are stored with authenticated data structures (ADS) on server nodes. The ADS generates a small piece of digest for the current database content, and the clients can then use this digest to authenticate the results returned by the server. The servers and clients are synchronized under a decentralized blockchain network. Each server node is a full blockchain node that stores the entire database and blockchain blocks, while each client node stores only the block headers. The newest block header contains the digest of the current database content, which the clients are able to access for authentication. With this digest, FalconDB could tolerate up to 1/3 of the total blockchain nodes being malicious, and is able to proceed even all full nodes are dishonest except one, which is enabled by the authenticated database design.

Note that, many previous works on outsourced database (ODB) [21, 28, 29, 34, 36, 37, 46, 46, 49, 50] have studied the problem of query authentication, which validates the results of the database server with essential digests. Unfortunately they cannot be directly applied to the collaborative database scenario as we will show later in Section 2.3.1.

We summarize our contributions as follows.

- We propose and implement FalconDB, which to the best of our knowledge, is the first platform that enables individual users to collaborate on a database with strong security guarantee, low storage cost, as well as high efficiency.
- FalconDB leverages a blockchain platform which tolerates up to 1/3 participants being malicious, as well as employs a temporal data model which provides a transparent history log of the database, enabling clients to examine the history and revert updates made by malicious collaborators.
- FalconDB server nodes store the entire database with authenticated data structures. With this, the client nodes are only required to store block headers, and able to validate the results returned by server nodes. Therefore, each client has minimal storage cost, and FalconDB could tolerate all full nodes being malicious except one.
- Moreover, FalconDB provides an incentive model that motivates each server to take extra storage cost and respond

honestly to clients' requests, which further reduces the chance of the server being malicious.
- Finally, we conduct empirical evaluation to validate the performance of FalconDB. Our results show that FalconDB is able to achieve security guarantee that aligns with state-of-the-art blockchain protocols, high efficiency as outsourced databases, and little storage cost on clients.

2 PRELIMINARIES

2.1 Blockchain

A typical blockchain system consists of multiple nodes which do not trust each other. Together, the nodes maintain a set of shared, global states and perform blockchain transactions that modify the global states. Some nodes may exhibit Byzantine behavior, but the majority are expected to be honest. The blockchain nodes are able to agree on the blockchain transactions and their order even with some malicious nodes present in the network.

The blockchain network maintains a chain of blocks. In a minimized blockchain system, each block has a block content C and a header $H = (M, V)$ which consists of metadata M and verification data V. The metadata includes the block height $height$, the hash value of previous block $lastBlockHash$, and the hash value of the block content $dataHash$.

The first block (with height 1) in the blockchain is called the genesis block, which is hardcoded in the protocol. A non-genesis block (H, C) is valid only if there exists a *valid* previous block (H', C') and all the following conditions are satisfied:
- $\mathsf{hash}(H') = H.lastBlockHash$.
- $H.height = H'.height + 1$.
- $\mathsf{hash}(C) = H.dataHash$.
- $\mathsf{validate}(V) = 1$ for a function validate defined by the blockchain protocol.

In a meaningful blockchain system, the verification data V and the validate function should ensure that the blocks can't be arbitrarily generated, which helps the blockchain remain stable. Sometimes there are multiple valid blocks at the same height, which is called a *fork*. Most blockchain protocol will choose a branch as the main chain and make sure that all nodes will work on the same main chain, so that the global state is consistent among these nodes.

A blockchain system can be categorized as either permissionless or permissioned. In the former, any node can join and leave the system, while in the latter, the access to blockchain is restricted to a group of members only. In a permissioned blockchain network, every node is authenticated and its identity is known to the other nodes. Since FalconDB is a collaborative database, we assume that the identities of all collaborators are known, and we will focus on permissioned blockchain in this paper.

2.2 Smart Contract

A smart contract is a computer program that executes in a secure environment, e.g., blockchain system, which has direct control over digital assets. Like other programs, a smart contract has variables and functions, and users can interact with the functions in the smart contract to examine or change the values of its variables. For blockchain systems where the security is ensured by BFT consensus, the inputs, outputs and states affected by the smart contract execution are agreed on by every node. Smart contracts typically define the logic of money transfer, which will happen immediately if the defined conditions are met, guaranteed by the secure environment. Existing public smart contract platforms are represented by Ethereum [44], which is an open-source platform widely used in real world applications.

2.3 Authenticated Data Structures

Authenticated data structures (ADS) are commonly used in outsourced databases, where users upload their databases to a cloud server and access the database remotely. ADSs empower database clients with the ability to verify the results of queries and updates to the remote database, which could prevent the server from being malicious. We define the abstract notion of an ADS as below.

Given a database D, an ADS is defined over a class of queries $\mathcal{Q}$ and updates $\mathcal{U}$ on D that it supports. It includes five key functions:
- Function Sum takes as input the data D, and outputs a digest δ.
- Function Qry takes as input the data D and a query $q \in \mathcal{Q}$. It returns a result $R = q(D)$ with proof π.
- Function $\mathsf{VerifyQry}$ takes as input a digest δ, a query $q \in \mathcal{Q}$, result R, and proof π. This function should guarantee that if $\delta = \mathsf{Sum}(D)$, it will return 1 if and only if $(R, \pi) = \mathsf{Qry}(D, q)$. This guarantees the correctness of the query result R as long as the client has the correct digest δ of the data D.
- Functions UpdS and UpdC are interactive algorithms run by the server and client. UpdS takes as input the data D and an update $u \in \mathcal{U}$, while UpdC takes as input the same update u and the data digest δ which satisfies $\delta = \mathsf{Sum}(D)$. After a successful interaction, UpdS outputs the new database content $D' = u(D)$ and UpdC outputs a digest δ' which satisfies $\delta' = \mathsf{Sum}(D')$. This guarantees that the client will get the right digest of the data after the update without storing the data locally, as long as the it has the correct digest before the update.

For an ADS to be practical, the sizes of δ and π as well as the data exchange between UpdS and UpdC should be much smaller than the size of D. We say an ADS is secure if for any probabilistic polynomial time adversary A, the probability

that the adversary breaks the equalities in VerifyQry and UpdC is negligible.

With the help of these functions, database users can safely upload their data to the server node while only keep a small local storage for the digest δ. The user will be able to verify if the server has correctly executed functions Qry and UpdS, through running functions VerifyQry and UpdC over the returned results and proofs.

2.3.1 Limitation. ADSs have been proved to be powerful in ODB where there is only one client in the setting. With ADS, the client can ensure the soundness, completeness and freshness of query results returned by the server, without storing the whole data locally. However, when applying ADSs to collaborative databases where there are multiple clients to update the data, there are two major challenges that remain to be solved:

- The security guarantees of ADS are based on the assumption that the user holds the correct, most recent digest of the data. However, when there are multiple clients to update the remote data, the digest stored at a client's local machine is likely to be outdated. It's challenging for a client to fetch the newest digest after the data has been updated by others, considering that the network has latency, the updates can have conflicts, and the digest received might be maliciously generated.
- The collaborators might be compromised and become malicious. A malicious client may issue undesired updates to the data, e.g., randomly insert and delete records. When this happens, unfortunately, a vanilla ADS can neither recover the data nor identify the attacker.

In this paper, we build the ADS upon a blockchain network to elegantly solve these two challenges. As a BFT consensus algorithm, blockchain can be naturally used to synchronize the digest among the clients, which solves the first challenge. Moreover, whenever there is an undesired update to the data, the properties brought by blockchain provides straightforward solution to recover the data (with immutable property) and detect the attacker (with transparent property), which solves the second challenge.

3 PROBLEM STATEMENT

As mentioned in Section 1, FalconDB consists of two types of entities: a group of servers that hold the database content, and a group of clients that can query and update the database without storing the database itself. The problem we study in this paper is how to enable a set of clients with limited hardware resources to jointly maintain a database both efficiently and securely.

3.1 Design goals

We require the following security guarantees in FalconDB:

Immutability. Any update to the database being committed on blockchain is immutable, which means it cannot be tampered with or denied/discredited under our threat model.

Transparency. The updates, insertions and deletions to the database are transparent to all. For any record that is/was in the database, users are able to get all its historical values as well as the metadata of previous updates, including the update time and the identity of the update issuer.

Data integrity. The client is able to verify that the database is updated according to the committed transactions. There is no unauthorized update to the database.

Query correctness. When a client receives a query result from a server, the client can verify the correctness of the result based on the following criteria: 1) Soundness: none of the answers in the query result have been modified and all of them satisfy the query conditions; 2) Completeness: no answers have been omitted from the result; 3) Freshness: the results are based on the most current version of database.

Furthermore, we require the system to have the following performance properties:

Highly expressive. The system can support a wide range of queries and updates.

Low cost on clients. We require that the space/computation/network overhead on clients are small, so that even devices with limited hardware resources (e.g. mobile phones) can participate in the system as database clients and enjoy all benefits of the system.

Performance. FalconDB aims to support high throughput and low latency on database transactions, despite the overhead of consensus and authentication compared with a vanilla outsourced database.

3.2 Threat model

In FalconDB, all data and schema are public. We assume either the servers or the clients could be malicious in that they may modify the data to their benefits. The database servers are incentivized (detailed in Section 4.3) to reply any query access or update request honestly in order to continuously gain profit. However, the servers may be malicious or collude with clients towards modifying the data and/or results to their benefits. FalconDB requires at least one server to be honest and return the correct results as requested. Furthermore, we assume up to 1/3 of the blockchain nodes (including both server nodes and client nodes) could be malicious (i.e., express Byzantine behavior), which is a requirement to reach consensus for BFT protocols with the presence of Byzantine nodes. Note that client nodes also participate in consensus, so the 1/3 corruption threshold still holds even if all server nodes are controlled by one company. Lastly, the authentication scheme adopted by FalconDB assumes a probabilistic polynomial-time adversary that cannot attack the hash functions.

4 FALCONDB SYSTEM DESIGN

4.1 System Overview

Figure 1 presents an overview of the FalconDB system. FalconDB consists of two types of entities:

Server nodes that hold the database and full blockchain data. The server nodes are hosted by different entities. They are responsible to answer the queries and updates from the clients, validating new blocks of blockchain, and generate proofs for query results. In return, they will be financially awarded for providing the service.

The database maintained by each server is stored in a persistent structure which preserves all historical versions. In addition, an ADS is constructed upon the database content to provide authenticated queries and updates to clients.

Client nodes that collaborate on the database. They don't store the content of database locally, but can read or write part of the database depending on their privileges. They access the database by issuing requests to any of the server nodes, and are able to authenticate the query/update results via the ADS interface. Depending on the setting, a subset of the client nodes will join the blockchain network to validate the blocks, while others passively pull the newest blocks without taking part in the blockchain consensus.

The blockchain network is collaboratively maintained by the servers and the clients. For each block, the block content C contains exactly one transaction that includes one or more reads/writes to the database (therefore in FalconDB, a blockchain transaction is equivalent to a database transaction). Meanwhile, the block header H includes the ADS digest of the database version when that block is committed, i.e., the database content after executing the corresponding transaction in the block. These digests can be used for clients to verify if their requests have been correctly executed, as described in Section 2.3.

Before running consensus, each server and each client makes a deposit to a smart contract hosted by services like Ethereum, which facilitates the incentive model as will be described in Section 4.3. Besides, we assume that there is a privilege function agreed by all nodes in the network. This function defines the kind of updates that can be made by a certain client. This privilege function is dynamic and can be adjusted. For example, if a client is compromised and become malicious, the others can revoke all its privilege to access the database. This function can be simply implemented by any smart contract, and we skip the details in this paper.

Figure 2 illustrates the general workflow for a query. A client node can send queries to any server node after transferring the query fee from its deposit to the server's account. The server will immediately execute the query and return the result to the client. After that, the client can choose to

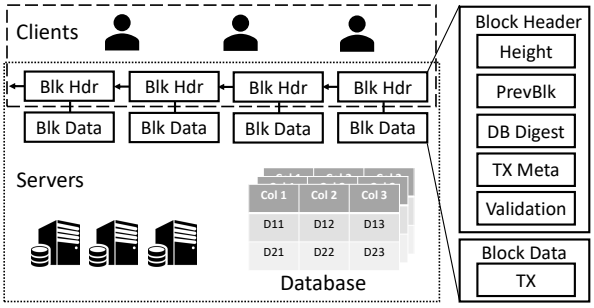


Figure 1: System overview. The clients store only the block headers, while the servers store the full blockchain and the database content. The servers and a subset of clients communicate with each other through permissioned blockchain protocol. The clients can connect to any server for updates and queries.

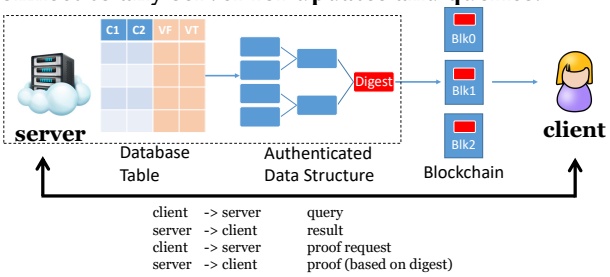


Figure 2: Simplified query workflow of FalconDB. Our system relies on an ADS to generate a digest for the data. The digest is stored in the blockchain header and the clients can use it to authenticate query results returned from a server.

challenge the query result by sending an authentication request to the smart contract. In this case, the client pays an extra amount of money to the server, and the server has to generate an ADS proof which can be validated by the digest. Failing to provide such a proof will result in the server losing all the fees in its smart contract account, including its initial deposit and revenue earned from the clients.

Furthermore, a client can issue updates to the server using the ADS interface. After interacting with a server, the client will propose a new block to the network, including the update and the interactive log in the block content, and the identity of updater and the new digest in the block header. Blockchain nodes will check if the update is valid and the new digest is correct. The block will be committed to the blockchain after reaching consensus among all blockchain nodes through the BFT consensus protocol, and the clients will update their digest to the one in this newly-proposed block.

In order to support transactions as in traditional databases, we adopt optimistic concurrency control (OCC) to provide snapshot isolation as many modern database systems. User can start a transaction with a specification on which block

it's based on. The block containing this transaction will be accepted only if there is no conflict during the specified block and the current block.

4.2 Blockchain Construction

To setup the system, the following parameters and functions need to be agreed on by all nodes before running the blockchain protocol:

- B_0, a hardcoded genesis block.
- $\mathsf{hash}(s) \rightarrow s'$, a cryptographic hash function that converts a string to a hashed string.
- (pk_i, sk_i), a pair of public/private keys for node i. pk_i is revealed to all participants in the network, and can be used to verify identities using the following two functions.
- $\mathsf{sign}(sk, s) \rightarrow s'$, a function that takes a secret key and a string, and outputs the signed string.
- $\mathsf{VerifySig}(pk, s, s') \rightarrow \{0, 1\}$ verifies the signature. It guarantees[1] that $\mathsf{VerifySig}(pk, s, s')$=1 if and only if $\mathsf{sign}(sk, s)$=s'.
- k, the number of validators for each block.

In FalconDB, each block corresponds to a database transaction with arbitrary size. For a block $B = (H, C)$, the block content C contains the transaction, and the block header $H = (M, V)$ includes both metadata and verification data. The metadata M contains the following fields:

- $height$, the block index.
- $lastBlockHash = hash(H_{height-1})$, the hash value of previous block.
- $\phi = \mathsf{hash}(C)$, the hash value of the block content.
- δ, a digest of the database version when the block is committed.
- $hash(RW)$, the hash value of the set of reads and writes associated with the update log in the block.
- e_0, the entity by whom the update is made.

The block validation data V contains the following fields:

- s_0=$\mathsf{sign}(sk(e_0), M)$, the signature of e_0.
- $\{e_1, \ldots, e_k\}$, the blockchain nodes that validate the block.
- $\{s_1, \ldots, s_k\}$ where s_i=$\mathsf{sign}(sk(e_i), M)$, k signatures of the blockchain nodes that validate the block.

FalconDB can use any permissioned BFT consensus protocol as the blockchain layer. The validators of the blockchain protocol is made up with all server nodes and a subset of client nodes. These blockchain nodes validate new blocks according to the rules described in Sections 4.6 and 4.7.

Implementation. When the number of clients is small, we can set k to be the total number of servers and clients. In this case, all clients can participate in the blockchain consensus in order to achieve maximum security guarantee. Tendermint [8] is an ideal platform that we can leverage to provide Byzantine fault tolerance in a permissioned blockchain environment. However, when the number of clients is too large such that the consensus process could be slow, we could set a smaller k and use RandHound [42] to randomly select a subset of clients to join the consensus protocol, or use Algorand [20] as the consensus protocol instead, in order to reduce the communication overhead.

[1]By guarantee we mean that the failure probability is negligible. Same for the rest of this paper.

4.3 Incentive Model

An incentive model is essential to motivate the servers to provide services to clients, as well as impose penalty for their dishonest behaviors. FalconDB relies on smart contracts (e.g., on Ethereum) to enforce the incentive model. At the beginning, all servers and clients make deposits to a smart contract. The contract has two key components:

- Service fee contract. A client pays the server it connects to a certain amount of service fee whenever it issues a query to the server, or requests a proof for the returned results. Moreover, the servers and clients are rewarded for participating in the blockchain consensus protocol and validating new blocks.
- Authentication contract. As we described earlier, the query authentication process is decoupled from blockchain. Users can request a proof for a query result by submitting the result to the smart contract. In this case, the server has to submit a corresponding proof to the smart contract later. Before the server submits a valid proof, its account (including deposit and previous earnings) in the smart contract will be temporarily frozen. The server can still receive money from the smart contract during that time, but cannot withdraw it. Therefore, a malicious server will lose all the funds in the smart contract account. On the other hand, if the server successfully submits a valid proof, the client will have to pay the server a certain amount of authentication fee in return.

When the account is not frozen, the server can withdraw funds from the smart contract. It can withdraw the revenue earned from clients freely. It can also withdraw the initial deposit and quit the blockchain network. We will discuss the rationale of this incentive model in Section 5.4.

4.4 Data Model

All participants in the network jointly maintain a blockchain with headers $\mathcal{H} = \{H_0, \ldots, H_{height}\}$ and block contents $C = \{C_0, \ldots, C_{height}\}$. Each block content C_i contains a database transaction TX_i. Both the clients and servers need to store all blockchain headers $\mathcal{H}$. Besides, the server nodes need to store the blockchain contents C as well as a persistent version of the database from which we can recover all its historical versions $\mathcal{D} = \{D_0, \ldots, D_{height}\}$, where $D_i = \mathsf{Upd}(\mathsf{D_{i-1}}, \mathsf{TX_i})$ denotes the database content after executing the updates in the i-th block. The database content is updated upon the

requests from the clients. On the server side, the server augments the original database with temporal attributes. To be more specific, all records are paired with two extra attributes: VF (stands for 'valid from') and VT (stands for 'valid to'), which stands for the height of blocks that create and delete them respectively. At block height h, only those records with $VF \leq h$ and $VT > h$ are valid at that snapshot. The primary keys in the original tables are paired with VF as the new keys in FalconDB.

The update operations are also augmented with temporal information. Suppose we are dealing with an update at height h. For an insertion, the record is inserted with VF being the current height of blockchain, h, and VT being ∞. A deletion will be regarded as updating the VT attribute of the original record to be the current height. An update will be decomposed into two logical steps: a deletion of the original record (i.e., change VT to be the current height), followed by an insertion of the updated record (i.e., duplicate the affected record and update the attributes accordingly, then set $VF = h$ and $VT = \infty$). Note that the tables involved in the predicates of SQL queries should also be augmented by an additional condition $VT = \infty$, which guarantees that we're working on the most recent snapshot of the database.

An example is given as follows. Suppose there are two tables: N=(int ID, string Name) and S=(int ID, int Score). Both tables are extended with two additional columns VF and VT, and the keys (ID) are combined with VF respectively, which becomes the pair (ID, VF) serving as the new key. The original table content is shown in Figure 3. Now suppose the following four updates are executed in four consecutive transactions, with the indices as the block heights that contain the corresponding transaction.

Listing 1: Updates

```
(1) Update Bob's score to 95.
(2) Insert Charlie with score 60.
(3) Delete Alice.
(4) Decrease everyone's score by 10.
```

After executing these updates, Figure 4 shows the resulted database contents. As will be shown later, one can retrieve all historical versions of the database from the wrapped tables.

ID	Name	VF	VT
1	Alice	0	∞
2	Bob	0	∞

(a) Table N

ID	Score	VF	VT
1	100	0	∞
2	80	0	∞

(b) Table S

Figure 3: Example of an augmented table content.

4.5 Queries in FalconDB

4.5.1 Query Types. Empowered by the temporal model described in Section 4.4, the query types supported by FalconDB include standard queries where only the newest database version is queried, full historical queries on a particular predicate, range historical queries on all updates within a time range, and delta queries on the changes made by a database transaction.

ID	Name	VF	VT
1	Alice	0	3
2	Bob	0	∞
3	Charlie	2	∞

(a) Table N

ID	Score	VF	VT
1	100	0	3
2	80	0	1
2	95	1	4
3	60	2	4
2	85	4	∞
3	50	4	∞

(b) Table S

Figure 4: The resulting table content after executing the updates in Listing 1 on the tables in Figure 3.

Standard query. The basic type is to query FalconDB as a traditional database, where the client only cares about the query result on the newest version. In this case, the client applies a selection to results with the condition: $VT = \infty$. This guarantees that only the records that are not expired could be selected.

Full historical query. Another supported query type is historical query, which is especially important for collaborative databases since it provides a transparent history to database clients and grants them the access to understand the evolution of data records. In a full historical query where the client wants to see all historical records that satisfy a predictor. In this case, we run the query as-is, and the result set naturally contains the historical results. For example, a client might be interested in a question "Whose scores are/were above 90?". The results contain not only the people who have score above 90 currently, but also include people who used to have such a high score which is somehow lowered later on. This helps the querier to better understand the evolution of data, as well as helps participants to detect undesired updates to the database. Any malicious update attempting to temper the data, e.g., deliberately lower someone's score, will be revealed by it.

The above question is equivalent to the SQL query "SELECT * FROM S WHERE S.Score >= 90;". If we execute the query on the table shown in Figure 4, the results are:

ID	Score	VF	VT
1	100	0	3
2	95	1	4

From the results we can see that: a) there is no qualifying records in the current database, since there is no record with $VT = \infty$; and b) there were two qualifying records, but they expired at B_3 and B_4 respectively. We can further look into the updates at these two blocks to understand what was happening using delta query as described below. This will be extremely useful for reverting undesired updates. For example, if the deletion of Alice was made by a mistake, its record is still retrievable by the historical queries and a client can easily find the problem here and revert the record by inserting it as a new record.

Range historical query. Historical queries can also be issued with desired time ranges, which could be achieved by applying additional conditions on the VF and VT attributes. For example, if we want to get a snapshot of the database at block h, we could collect the records with $VF \leq h$ and $VT > h$.

Delta query. FalconDB further guarantees transparency by supporting delta queries, which help the clients to understand the influence of previous updates. Specifically, FalconDB provides an interface for clients to query the changes made by the transaction committed at any particular block, e.g., B_h. This can be achieved by issuing the following SQL query to every table T in the database:

```
SELECT * FROM T WHERE VF=h OR VT=h;
```

For example, when $h = 1$, this query will return the second and third rows in Table S (Figure 4b), which are Bob's scores before and after this update; while when $h = 4$, this query will return the last four rows of Table S, which are the scores that affected by the last update.

4.5.2 Query Authentication. The query types that FalconDB supports are restrained by the ADS being used. Following the query wrapping methods described above, each query will be wrapped with at most two more predicates by the client, before sending it to the server. For any ADS that supports range query authentication, the wrapped query will still be supported by the ADS as long as the original query (without the extra predicates) is supported. After executing the query, the server could utilize the ADS interface to provide a proof as well as the result to the user for authentication.

Query execution is typically fast on server, but proof generation is much slower and can take hours for real-world million-item queries. To address this, FalconDB decouples the proof generation process from the blockchain network. With this, the server returns the query result immediately after it is retrieved along with its signature. The client can later pay extra money to request a proof, by submitting a request to the smart contract with the signed query result received from the server. In this case, the server must submit a proof to the smart contract in order to avoid penalty as described in Section 4.3. In other words, a user pays extra money and time to request a proof for extra security. We will discuss this mechanism in detail in Section 5.4.

4.6 Updates in FalconDB

Like traditional database systems, the clients in FalconDB can use database transactions to update the database. We will first focus on update operations in this section, and discuss transaction processing later in Section 4.7.

4.6.1 Update Types. As described in Section 4.4, the update needs to be augmented with attributes to enable time travel features. Generally speaking, when an update is requested at block height h, It could be augmented according to its type:

- Insertion. When inserting a record, we add the additional attributes $VF = h$ and $VT = \infty$.
- Deletion. When deleting a record, we set $VT = h$.
- Record value change. When changing value for a record, we decompose it to an deletion of the original record followed by an insertion of a new record with the desired values. In other words, we set VT of the original record to h, and insert the new record with the desired values plus $VF = h$ and $VT = \infty$.

In the case where multiple records are involved, the augmentation process could be enforced for each affected record. If the values to be set are computation results of other fields, FalconDB adds additional queries to the transaction to compute the values, which are then used as constants to construct the requested updates. An interesting fact here is that FalconDB does not require the ADS to support deletion, but only insertion and update operations.

We still use the table in Figure 4 as an example. Consider the case where a client wants to update everyone's score to be Bob's score. It could be decomposed into three steps.

Step 1 Retrieve Bob's score:

```
SELECT S.SCORE FROM S, N
WHERE N.Name = "Bob" AND N.ID = S.ID
AND N.VT=∞ AND S.VT = ∞;
```

The result will be 85.

Step 2 Find all affected records and void them:

```
UPDATE S SET S.VT=5 WHERE S.VT = ∞;
```

The two up-to-date records in S are affected, which will be updated to $(2, 85, 4, 5)$ and $(3, 50, 4, 5)$ respectively.

Step 3 Insert the new records:

```
INSERT INTO S VALUES (2, 85, 5, ∞);
INSERT INTO S VALUES (3, 85, 5, ∞);
```

The first attribute (i.e., "2" in Step 3) is unchanged in the update, so its value comes from the original record. The second attribute (i.e., "85" in Step 3) is set by the update, so its value comes from the first step. The VF and VT values are automatically supplied by the system.

4.6.2 Update Authentication. After augmenting the update requests with time intervals, the server can use ADS interfaces to update the database and propose a new block to the blockchain network. In particular, an authenticated update is achieved by following steps.

(1) A client node c connects to a server and request for an update. The server first checks the privilege function to make sure that the client has the privilege to make the update. Then they use the interactive functions UpdC and UpdS to update their local states, and the client will compute a new digest δ' after the interaction. If there are multiple updates to be made, the client and server repeat this step and generate a digest for each update.

(2) The server then constructs a block by filling the fields defined in Section 4.2. In the case of multiple updates, only the digest after the final update is filled in the block. Then the server proposes it to the blockchain network along with all the interactive logs and intermediate digests in the first step.
(3) Once the new block is received, blockchain nodes perform validation by first checking if the client has the privilege to issue the update request as defined by the privilege function, and then verifying the correctness of the new digests (including the final one and the intermediate ones) by checking the corresponding interactive logs. All clients have the same local information on blockchain, so that they can replay the interactive logs and see if it follows the algorithm UpdC.
(4) The new block will be committed to blockchain if its content is validated through the BFT consensus protocol. Once committed, all other servers update their local database content to the newest version, while each client updates the local digest to the new one included in the block header.

4.7 Database Transactions in FalconDB

The four most important properties in database transactions are ACID (atomicity, consistency, isolation and durability). As described in Section 4.2, a transaction in FalconDB will always be stored entirely in a single block regardless of its size. Therefore, the AD properties are naturally guaranteed. The concept of consistency is defined as other databases including referential constraints and triggers, and can be verified by authenticated queries. In order to provide isolation property for transactions, we use a variant of optimistic concurrency control adopted from [53] to abort conflicting transactions, which ensures snapshot isolation for transactions. Note that in FalconDB, a read-only transaction is not necessary, since queries could be issued one-by-one with a specified database version for each, as described in Section 4.5. In this case, snapshot isolation is intrinsically achieved.

For each transaction that involves update requests, the user must specify which snapshot (i.e., block height) it is based on when starting the transaction. This height will be used as the *read timestamp* of OCC. When the transaction is committed, the height of the block containing this transaction will be used as the *commit timestamp*. Suppose a transaction has read timestamp i and commit timestamp j. The blockchain nodes must check if this transaction has conflict with any transaction in blocks $B_{i+1} \ldots B_{j-1}$ using OCC methods in [53]. The transaction will be aborted if there is any conflict identified within these blocks. In order to make it possible for clients to check conflicts, the server broadcast the read/write set associated with the involved blocks together with the proposed block to the network. The client validators first validate the set of reads and writes using the $hash(RW)$ set in headers, and then perform OCC validation in the same way. In addition, the blockchain protocol will also handle forks, i.e., two blocks having the same height and the two transactions having the same commit timestamp. The protocol will guarantee that the network will eventually agree on a forkless chain to avoid this situation.

5 FALCONDB ANALYSIS

In this section, we analyze FalconDB's security, performance, and space usage. We will see that FalconDB achieves all desired properties in Section 3. Note that FalconDB could utilize any blockchain platform or ADS solution that fits to the design, while the analysis may vary accordingly. In this paper, our analysis is based on two state-of-the-art work: Tendermint [8] which is a blockchain platform that has been theoretically proved to be safe and practically adopted by many systems; and IntegriDB [50], an ADS that is capable of authenticating a wide range of SQL queries efficiently.

5.1 Security analysis

Correctness of query results. The correctness of query results can be built upon the digest and the ADS proof. More specifically, with a valid digest, the soundness and completeness properties can be guaranteed by the *VerifyQry* function of the ADS as described in Section 2.3. Meanwhile, freshness is guaranteed as long as the digest held by the client is newest. This is naturally satisfied since the client can always fetch the newest digest from the newest block in the blockchain.

Data Integrity. Recall that the *UpdS* and *UpdC* interfaces provided by the ADS ensure that a client can verify whether a server has executed the requested query honestly. On the other hand, a server can only update the database upon requests from the clients, otherwise the server won't be able to generate a valid proof for any future queries since the actual digest of the database has been changed and doesn't match the digest in the blockchain.

Immutability and transparency. The update process and temporal data model provided by FalconDB ensures that all updates are permanently stored in the database, as well as verifiable via blockchain, which guarantees immutability. Moreover, all historical results are retrievable, which guarantees transparency.

Server reliability and system liveness. A major difference between FalconDB and a traditional permissioned blockchain system is, as a client that starts participating in the blockchain consensus from the beginning, it is able to identify dishonest behaviours even if all other blockchain nodes (servers and other clients) are malicious. That is because with

the initial digest being revealed to all, a client that joins at the beginning can validate each block through ADS, which implies that the first appearance of an incorrect digest will be spotted immediately. With that, the servers are not able to fool the clients. On the other hand, for clients that join later in the blockchain consensus, FalconDB can still ensure integrity as long as the portion of honest nodes (including the servers and clients) meets the BFT requirement in Section 3.2, no matter how many servers are honest. Blockchain nodes that participate in the consensus process could validate each block and ensure the correctness of the digest (or detect incorrect digest in the block and reject it) via ADS, regardless of the number of dishonest servers. The consensus being reached on the digest will help other clients to authenticate their queries based on the digest.

Once a server is detected malicious, the blockchain network can exclude that server from further participation. Furthermore, as long as there exists at least one server that is honest, the blockchain network can introduce new servers and the honest server can send all its data to the new servers. In other words, FalconDB is able to proceed even with one single server being honest.

Privacy issues. As a transparent public database, FalconDB raises privacy concerns since all data are accessible to each participant, especially the untrusted servers. If we want to ensure data confidentiality, the data and logs need to be encrypted at servers. Unfortunately, querying and updating encrypted databases still remains an open problem in academia. To make things harder, FalconDB employs an authentication scheme that allows each participant to verify if the database server has returned the correct result, which is hardly supported by current solutions of encrypted databases. That said, the current database server in FalconDB could be easily replaced with other databases that provide the ability of authentication over encrypted data in future, if there are breakthroughs in the area of encrypted database.

5.2 Performance Analysis

Authentication layer. As the state-of-the-art ADS, IntegriDB [50] is an ideal candidate for the ADS component in FalconDB. In IntegriDB, the proof generation, communication and verification time do not depend on the database size, but only on the query. Verifying a given proof takes little time on the client side. However, the proof generation time for real-world million-item queries may take hours. As described in Section 4.5.2, FalconDB addresses this challenge by decoupling the proof generation process from system pipeline so that it wouldn't become the bottleneck. On the other hand, an update in IntegriDB is fast and typically finishes within 1 second. In the case of batch updates, the time consumption grows linearly with respect to the number of updates, which is asymptotically optimal for accumulator-based ADSs [9].

Consensus layer. Modern permissioned blockchain systems can achieve satisfactory performance for consensus when the participating blockchain nodes are not too many. As our current choice, Tendermint is able to process thousands of transactions per second, with a commit latency in the order of one to two seconds, in benchmarks of 64 nodes distributed across 7 data-centers located in 5 continents accessed through commodity cloud instances. Notably, this performance is maintained even in harsh adversarial conditions, with validators crashing or broadcasting maliciously crafted votes. When the number of blockchain nodes is large, we could utilize a blockchain protocol that scales. In particular, we could adopt a random selection process, e.g., RandHound [42], to select a subset of clients to participate in consensus, which helps reduce the consensus overhead while maintaining almost the same BFT security guarantees.

Concurrency control. The OCC adopted by FalconDB assumes that it is rare for transactions to conflict with each other. However when conflicts appear very frequently, the abort rate for transactions will also be high. Note that most database systems suffer from conflicting transactions, and how to address this challenge is still an open problem.

5.3 Space analysis

Recall that each block contains all information related to a single database transaction, including the database digest, related hash values and signatures. Utilizing IntegriDB [50], the digest could be implemented in $O(1)$. All other fields together take less than one kilobyte. The clients can easily store millions of blocks which in total only take several megabytes. We will investigate in more details in the experiment section.

Compared with clients, the servers in FalconDB have to store much larger data locally, which includes the persistent database that contains all historical information, and the block contents that contain all transactions in the past. Nevertheless, the total storage required for database servers in FalconDB is in the same order of any full node in existing naive blockchain database solutions [33]. The advantage brought by FalconDB is that it could proceed with BFT consensus even with one single server node.

5.4 Incentive model analysis

A core component of FalconDB is the incentive model. It motivates the servers to provide services to clients, and prevents them from being malicious.

A key design challenge in many security systems is the trade-off between performance and security. In particular, even the state-of-the-art verification computing technology can still take more than one hour to generate a proof for a complex query. This becomes a bottleneck of the system

pipeline and makes the system unfavorable for databases. However, with the help of blockchain and smart contract, we make this process asynchronous with the rest of the pipeline.

With a view of game theory, all servers are motivated to work honestly and lively. As discussed in Section 5.1, any malicious behavior will be noticed by clients. Once identified as malicious, the server will be excluded from the blockchain network and the related deposit will be confiscated. On the contrary, if the server stays honest, it gets paid for each request from the client. Therefore, the service providers are not only incentivized to provide honest answers to clients, but are also motivated to take actions, e.g., use replicas, to guarantee liveness and protect them from being compromised.

From the clients' view, knowing that servers are highly likely to behave honestly, as well as the fact that any malicious behavior can be detected, the clients can be more flexible on the queries. Recall that a client can choose to ask for a proof after receiving a query result from the server. Assume that the reward for the server to answer a query is r, the penalty on the server for being malicious is p, the importance value of a query answer (i.e., the profit for a server to lie on a query answer without being detected) is v, and the probability that a client requests a proof is x. The server will stay honest when the expected revenue for being honest is larger than the expected revenue for lying, which is $r > -xp + (1-x)i$, or $x > (i-r)/(p+i)$. From this inequality, we can easily check that for queries with a small importance value compared with the penalty, a rational server will behave honestly even when the client rarely request for the proof; while when the query is important, the client can choose to request for the proof with high probability in order to keep servers to stay honest. In particular, when $x = 1$, i.e., the client always ask for the proof for each query, a rational server will always stay honest disregards the values of other parameters, since the server won't be able to fool the client anyway and will be punished for sure when behave dishonestly. Note that even in the case that the client request for a proof, it could choose to trust the result before getting the proof, as long as they believe that the server is rational.

With the above analysis, we conclude that this model is *incentive compatible* (i.e., all nodes will achieve maximum profit by following the protocol honestly) as long as the reward is greater than the maintenance cost for being a blockchain server node, and the penalty is large enough to make the above inequalities hold. Therefore, the parameters can be decided by simply valuating the server maintenance cost and the values of data. However to achieve optimal performance, we should set the parameters so that the clients request proofs as infrequent as possible while still preventing the server from being malicious. Due to the space constraint, we leave a formal discussion as our future work. A detailed study on incentivizing correct computation with blockchain can be found in [25].

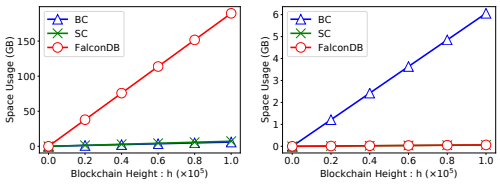
6 EXPERIMENTS

6.1 Testbed and Methodology

Experimental Setup. In our evaluation, FalconDB incorporates the following components: 1) MySQL server as the backend SQL server, 2) IntegriDB [50] to provide ADS and support authentication, and 3) Tendermint [8] as the blockchain platform. These components are selected because of the superb characteristics revealed in practice, such as compatibility and security guarantee. Nevertheless, future alternatives and more advanced solutions that provide similar functionalities could also be plugged in and used as blackboxes in FalconDB. The experiments are conducted on CloudLab [15], where each node is equipped with 2.4 GHz Ten-core Intel E5-2640v4 processor and 64GB DRAM. We run our experiments on a cluster with N_s server and N_c clients. By default, $N_s = 5$ and $N_c = 27$. To simulate the difference between the server and the client, we manually slow down the processing speed on all client nodes, by reporting the execution time with a factor $10x$, which is a reasonable ratio between the processing speed of a commercial server and a mobile phone. For simplicity, access control is ignored in the experiments.

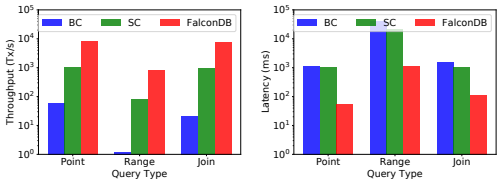
Baseline. We compare the performance of FalconDB with the two existing blockchain database solutions mentioned in Section 1: a naive blockchain based shared database where each user stores a full data copy (BC), and a smart contract based solution where a user could submit smart contracts to query full nodes and get consented results (SC). In BC, all servers and clients serve as blockchain nodes. They execute queries locally since each stores a full copy of the database. When a node wants to update the database, it will push the update command to the blockchain network. After consensus and committed on chain, all blockchain nodes execute the same update on their local copies. In SC, the clients issue queries and updates by sending commands to the blockchain network maintained by servers. Once the servers executed and reached consensus, the client can obtain the results from committed blocks. For a fair comparison, we use Tendermint as the underlying blockchain platform for all solutions.

Data set. The database construction in all experiments follows YCSB [11]: there is a single table with a column for primary key and several columns of other fields. The value of each field is a 100-byte ASCII string. In addition, the table is wrapped with two additional columns VF and VT as explained in Section 4.4. We denote the number of rows as n, and the number of columns (including VF and VT) as m. In the following we first evaluate FalconDB with different synthetic workflows, and then test it with YCSB workflows. All SQL queries are wrapped as described in Sections 4.5 and 4.6.



(a) Server (b) Client

Figure 5: Space usage on a table with $10h$ rows and 10 columns. *FalconDB shifts the high storage cost from local clients to server only.*



(a) Throughput (b) Latency

Figure 6: Query performance on a table with 10^6 rows and 10 columns.

6.2 Performance measurement

We use h to denote the number of blocks in blockchain. An *update block* contains the record of an update operation, while a *query block* includes the result of a query. Note that in SC, a block can be either a query block or an update block, while in BC and FalconDB, all blocks are update blocks.

In this set of experiments, we assume the table has n_0 rows initially, and all following updates are insertions with Δ rows. Therefore, with h update blocks the database has $n = n_0 + h\Delta$ rows. By default, $n_0 = 0$, $m = 10$ and $\Delta = 10$. We ignore transaction conflicts and focus on the performance of each operation of FalconDB in this set of experiments. Therefore, we assume that there is only one client that is making queries and updates to the network in this subsection.

Storage. In all three approaches, the servers have to store the full database content and all the blocks. In addition, in our implementation of FalconDB, the server has to maintain $O(m^2 n)$ authenticate data structures for a table with n rows and m columns. In both SC and FalconDB, the clients only need to store the blockchain headers, which is orders of magnitude smaller than the database size. Since the sizes of block headers are fixed, the space cost on clients will grow linearly with the total number of blocks. In the BC approach, however, all client nodes are equivalent to server nodes. They need to store the entire database as well as full blockchain containing all updates to the database. The storage costs on the clients are in proportional to the volume of a database with all its histories.

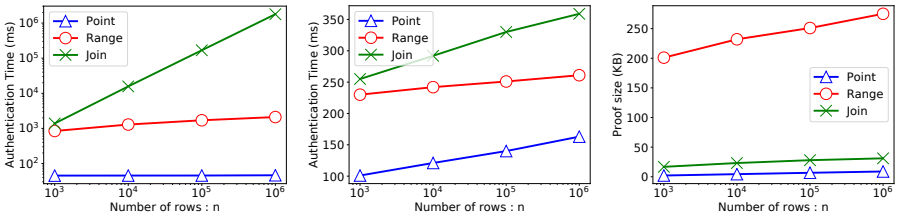
Figure 5 shows the space usage on each node for an empty table with $m = 10$ columns to be inserted by up to $h = 10^6$ times, and the records being inserted in each update is $\Delta = 10$. As shown in Figure 5a, for a table with size around 3GB, the server storage of SC and the node storage of BC are about the same as the table size, while the storage of FalconDB is around 200GB. This is a reasonable overhead compared to prior work on authenticated databases. On the other hand, each client in FalconDB or SC only needs to store less than 100MB size of blockchain for the table. This size is magnitudes less than the database size and is acceptable on most devices. The BC approach, however, requires every node to store the database and the blockchain that contains the full update history (different from the digest storage in FalconDB's blockchain). Therefore it requires about 6GB storage space, which is relatively high for many individual users. Note that this experiment only considers short strings as table contents. In a modern database where there could be many diversed data types involved, storing the whole database would take significantly more space, while the blockchain digest storage remains constant.

Query throughput and latency. In the BC solution, a database client executes the query locally. The low-performance hardware on these individual users limits the throughput and latency. In FalconDB and SC, a client issues a query to the server, and the server executes the query with powerful commercial hardware. However in SC, the query result is confirmed by committing the corresponding block to the blockchain. As a result, its performance depends on the blockchain consensus strategy, i.e., the number of blocks generated per second and the size of blocks. On the other hand, each query in FalconDB is performed through a direct server-client communication, so the throughput is much higher.

We test three types of queries in this experiment, which represent three difficulty levels in terms of computation. The first query type is a simple point query on the primary key. The second type is a multi-dimensional range query on 6 non-key fields. The third type of query is a join query on two tables. For all queries, the number of rows n in the involved tables is 10^6. For the second and third query types, each result contains precisely 100 rows.

Figure 6 shows the throughput and latency for each database solution with respect to different query types. As we can see in the figure, both the throughput and latency are magnitudes better on FalconDB, while the performance on BC and SC is much lower.

Query authentication latency. In the case that the next query depends on the previous query's verification, the client has to wait for the authentication of the previous query. Figure 7 illustrates the waiting time under this case by showing the authentication time of FalconDB with varying number of table rows, using IntegriDB as the supporting ADS. Depending on the query complexity, the authentication time varies from seconds to hours, and grows linearly according to the database size. However, regardless of query type and



(a) Proof generation time (b) Verification time (c) Proof size

Figure 7: Authentication latency on a table with n rows and 10 columns.

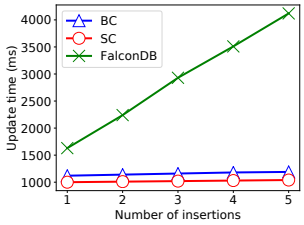


Figure 8: Update performance on an empty table with 10 columns.

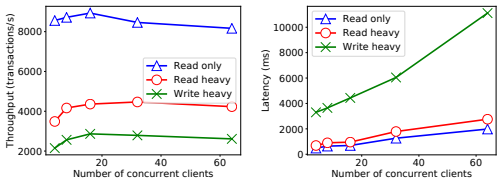
table size, the size of proof remains small while the verification on the client side can be done in one second. That means validating the query results doesn't consume much bandwidth and computation power on the client side, and can be performed on any individual. Recall that although proof generating could be slow, it is decoupled from the FalconDB main pipeline and clients still get instant results from server side, whose correctness is partially protected by the incentive model as described in Section 5.4.

Update efficiency. Figure 8 shows the execution time for updates where each update involves a varying number of record insertions into an empty table with 10 fields. It's no surprise that the update performance of BC and SC is almost independent with the number of insertions, since the performance is dominated by the consensus algorithm and the overhead on local database is negligible. A single insertion in FalconDB is comparably efficient. It takes only about one second to complete, which includes the blockchain consensus, the ADS proof generation, and the result verification on client side. For batch updates, the results of FalconDB show a linear performance cost with respect to the number of insertions. We note that as proven by [9], it is impossible to achieve sub-linear performance cost for batch updates in any powerful ADSs built upon cryptographic accumulators.

6.3 YCSB results

Finally, we test the performance of FalconDB with three YCSB workflows: read only, read heavy (comprised of 95% reads and 5% writes) and write heavy (comprised of 50% reads and 50% writes). The record selection follows Uniform distribution. We vary the number of concurrent clients to see its affect on the performance of FalconDB. As described in Section 5.2, we select at most 27 clients (32 nodes in total together with the 5 server nodes) to serve as blockchain node in order to reduce the consensus overhead. Since the authentication process is independent from the main pipeline and has been fully tested in previous experiment, in this experiment, we skip the authentication process and focus on the transactional performance on FalconDB.

In the read-only workflow, the performance of FalconDB remains stable with respect to the number of concurrent clients. This is because these transactions don't need to go



(a) Average throughput (b) Average latency

Figure 9: YCSB performance.

through blockchain consensus, and the increase in concurrent clients only increase the load on servers. In the other two workflows, the performance decreases when there are concurrent clients. The reason is the data contention from different transactions: when multiple transactions are writing to the same record simutaneously, only one of them will success and all the others are aborted. Note that this phenomenon is common in all modern databases with OCC, and we leave improving the concurrency control as future work.

Note that in reality, the server node could be busy answering queries from clients. To prevent this situation, the logical server node could be equipped with sufficient physical server nodes so that the updates and queries can be processed simultaneously. Since all replicas from a logical server node trust each other (e.g., they are all from Amazon clusters), traditional distributed system with simple failure models can be included to guarantee consistency among physical server node. However, this is out of the scope of this experiment.

7 APPLICATIONS

Besides the aforementioned use case for charitable giving, many other applications could benefit from FalconDB.

Credit score. A person's credit is influenced by many factors, e.g., interactions with banks. Credit scores are queried extensively, from housing agencies to car dealers. Currently credit bureaus collect credit information of each person from multiple organizations and acts as the "central database server" for other agencies to query. However, they have been involved in several legal and regulatory issues. If we could have a decentralized credit bureau powered by FalconDB, all relevant organizations could have update access

to the database to update a person's credit-related information. Moreover, all parties can now fully trust the credit score and history in the database because they hold a local copy of the proofs for the verifiable server in FalconDB. Moreover, if certain information are later found incorrect, it is traceable because the signatures related to an update are also recorded.

Banking. A person could have multiple bank accounts and a bank holds numerous deposit accounts. Traditionally, inter-bank money transfer may take a few days, although the involved database update only take seconds. A major reason is because the lacking of trust between banks, due to which one has to be sure that a bank actually has that amount of money it claims to. With a shared database among all banks, this process could be much simplified. Also, our design enables small banks having few data storage the ability to collaborate with large banks without worrying about trust.

Government audit. Government audit is critical in order to prevent any wrongdoings such as corruption and fraud. At present Federal Audit Clearinghouse [17] takes an important role to collect data from federal agencies and distribute to the public. It is a time-consuming and cumbersome process, and yet essential to produce an reliable and trustful report. Using FalconDB as a shared database among federal agencies and the public, each agency could directly insert records such as how they spend their funds, and the public could simply query the database to monitor the government. Note that, FalconDB's design empowers a single person with the ability to store the whole blockchain of proofs, thus a person could fully trust the query result backed by the interactive verification process between blockchain and the verifiable database server, that is coupled with each query.

8 RELATED WORKS

Verifiable outsourced database. The thriving of cloud services brings an interesting application scenario: outsourced database. There are many advantages to host data on the cloud such as flexibility, but it is challenging to ensure integrity of the outsourced data. A series of works have been proposed to address this challenge, either using authenticated data structures [6, 10, 12, 18, 21, 24, 29, 31, 34, 36–38, 46, 50–52] or zero-knowledge proofs [49]. The ADS approach is relatively efficient but can only authenticate some specified types of queries, while the zero-knowledge proof approach can support general queries but is much slower.

Blockchain platform. FalconDB leverages blockchain as the underlying platform to store essential proofs for clients to verify data server's integrity and reach consensus. Designing a blockchain protocol with high performance under large scale while ensuring blockchain's good properties has been a major research topic. Some works [27, 41] uses directed acyclic graph (DAG) instead of chain structure, which ensures that the average amount of time for each transaction is reduced since few blocks are wasted. Another direction aims to design protocols with less consensus time [1, 23, 54]. Finally, Algorand [20] and RandHound [42] achieves high scalability by randomly selecting a subset of nodes to participate in the consensus. This will result in a smaller scale in the consensus without hurting the security guarantees.

Blockchain-based data sharing. The emergence of blockchain technology has led to the redesign of data sharing platforms. Some previous works build systems for data collaboration [2, 3], but the data structures they maintain are not as complex as databases. Database has also been implemented on blockchain platform [19, 33, 39], but these previous works all fall into the three approaches mentioned in Section 1 and are not able to achieve security, compatibility and efficiency simultaneously. CreDB [32] utilized trusted computing environment to achieve similar security guarantees as blockchain , but it relies on trusted hardware and is limited to simple forms of operations. Finally, vChain [45] and GEM^2 [48] aim to combine ADS with blockchain in order to support authenticated queries, but they are limited to range queries and are not able to support SQL queries.

Game theory in Blockchain. From a prospective of game theory, the incentive model of any blockchain should be *incentive compatible*, i.e., nodes will achieve maximum profit by following the protocol honestly. In many blockchains that the nodes are competing with each other for the reward, there exists attacks where a node can get more profit by performing malicious strategies. For example in Bitcoin, the nodes will have a larger chance to mine blocks by withholding selected blocks [16]. Analyzing the blockchains using game theory models is an active research topic. Interested readers could refer to a survey on game theory applications in blockchain by Liu et al. [30]. In FalconDB, the nodes don't compete for the reward. Instead, they receive maximum profit as long as they follow the protocol honestly. Therefore, our model won't explicitly suffer from existing attacks.

9 CONCLUSION

This paper presents FalconDB, a shared database that is secure, efficient, and has low hardware requirement on clients' side. It uses a blockchain platform and ADS data storage to build public outsourced database. It requires one or multiple server nodes to store the database and the chain of blocks, while the client nodes only need to store block headers. This allows FalconDB to tolerate up to 1/3 of total nodes being malicious, and all server nodes being dishonest except one.

ACKNOWLEDGMENTS

Feifei Li and Yanqing Peng are supported in part by NSF grants 1816149, 1801446, 1514520, 1718834. Min Du and Dawn Song are supported in part by NSF grant TWC-1518899 and DARPA grant N66001-15-C-4066.

REFERENCES

[1] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. In *NDSS*. The Internet Society, 2018.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

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

[4] M. Atallah, M. Bykova, J. Li, K. Frikken, and M. Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 103–114. ACM, 2004.

[5] M. J. Atallah, H. G. Elmongui, V. Deshpande, and L. B. Schwarz. Secure supply-chain protocols. In *E-Commerce, 2003. CEC 2003. IEEE International Conference on*, pages 293–302. IEEE, 2003.

[6] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 863–874. ACM, 2013.

[7] D. Billsus and M. J. Pazzani. Learning collaborative information filters. In *Icml*, volume 98, pages 46–54, 1998.

[8] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on bft consensus. *https://tendermint.com/docs/tendermint.pdf*, 2018.

[9] P. Camacho and A. Hevia. On the impossibility of batch update for cryptographic accumulators. In *International Conference on Cryptology and Information Security in Latin America*, pages 178–188. Springer, 2010.

[10] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable set operations over outsourced databases. In *International Workshop on Public Key Cryptography*, pages 113–130. Springer, 2014.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154. ACM, 2010.

[12] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of xml documents. *Journal of Computer Security*, 12(6):841–864, 2004.

[13] DONOR SEARCH. The largest searchable charitable giving database, 2018. [Online; accessed 12-October-2018].

[14] W. Du and M. J. Atallah. Privacy-preserving cooperative scientific computations. In *csfw*, page 0273. IEEE, 2001.

[15] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[16] I. Eyal and E. G. Sirer. Majority is not enough: bitcoin mining is vulnerable. *Commun. ACM*, 61(7):95–102, 2018.

[17] Federal Audit Clearinghouse (FAC). Federal Audit Clearinghouse (FAC) . https://harvester.census.gov/facweb/. [Online; accessed 25-October-2018].

[18] C. Fournet, M. Kohlweiss, G. Danezis, Z. Luo, et al. Zql: A compiler for privacy-preserving data processing. In *USENIX Security Symposium*, pages 163–178, 2013.

[19] 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 *CIDR*. www.cidrdb.org, 2019.

[20] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[21] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[22] S. Hu, C. Cai, Q. Wang, C. Wang, X. Luo, and K. Ren. Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization. In *INFOCOM*, pages 792–800. IEEE, 2018.

[23] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *IEEE Symposium on Security and Privacy*, pages 583–598. IEEE, 2018.

[24] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. Trueset: Faster verifiable set computations. In *USENIX Security Symposium*, pages 765–780, 2014.

[25] R. Kumaresan and I. Bentov. How to use bitcoin to incentivize correct computations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 30–41. ACM, 2014.

[26] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[27] C. Li, P. Li, W. Xu, F. Long, and A. C. Yao. Scaling nakamoto consensus to thousands of transactions per second. *CoRR*, abs/1805.03870, 2018.

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

[29] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries. *ACM Trans. Inf. Syst. Secur.*, 13(4):32:1–32:35, 2010.

[30] Z. Liu, N. C. Luong, W. Wang, D. Niyato, P. Wang, Y. Liang, and D. I. Kim. A survey on applications of game theory in blockchain. *CoRR*, abs/1902.10865, 2019.

[31] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[32] K. Mast, L. Chen, and E. G. Sirer. A vision for autonomous blockchains backed by secure hardware. In *SysTEX@SOSP*, pages 1:1–1:6. ACM, 2019.

[33] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[34] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *ACM SIGPLAN Notices*, volume 49, pages 411–423. ACM, 2014.

[35] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[36] D. Papadopoulos, S. Papadopoulos, and N. Triandopoulos. Taking authenticated range queries to arbitrary dimensions. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 819–830. ACM, 2014.

[37] D. Papadopoulos, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Practical authenticated pattern matching with optimal proof size. *Proceedings of the VLDB Endowment*, 8(7):750–761, 2015.

[38] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *Annual Cryptology Conference*, pages 91–110. Springer, 2011.

[39] B. M. Platz, A. F. Filipowski, and K. Doubleday. Flureedb, a practical decentralized database. *https://flur.ee/assets/pdf/flureedb_whitepaper_v1.pdf*, 2017.

[40] G. Sachs. Blockchain–putting theory into practice. *the-blockchain.com*, pages 25–32, 2016.

[41] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016:1159, 2016.

[42] E. Syta, P. Jovanovic, E. Kokoris-Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. Scalable bias-resistant distributed randomness. In *IEEE Symposium on Security and Privacy*, pages 444–460. IEEE Computer Society, 2017.

[43] A. Wiggins and K. Crowston. From conservation to crowdsourcing: A typology of citizen science. In *System Sciences (HICSS), 2011 44th Hawaii international conference on*, pages 1–10. IEEE, 2011.

[44] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[45] C. Xu, C. Zhang, and J. Xu. vChain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, Amsterdam, Netherlands, June 2019.

[46] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 5–18. ACM, 2009.

[47] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019.

[48] 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 *Proceedings of the 35th IEEE International Conference on Data Engineering*, pages 842–853, Macau SAR, China, Apr. 2019.

[49] 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*, pages 863–880. IEEE Computer Society, 2017.

[50] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for Outsourced Databases. In *ACM Conference on Computer and Communications Security*, pages 1480–1491. ACM, 2015.

[51] Y. Zhang, C. Papamanthou, and J. Katz. Alitheia: Towards practical verifiable graph processing. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 856–867. ACM, 2014.

[52] Q. Zheng, S. Xu, and G. Ateniese. Efficient query integrity for outsourced dynamic databases. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pages 71–82. ACM, 2012.

[53] T. Zhu, Z. Zhao, F. Li, W. Qian, A. Zhou, D. Xie, R. Stutsman, H. Li, and H. Hu. Solar: Towards a shared-everything database on distributed log-structured storage. In *USENIX Annual Technical Conference*, pages 795–807. USENIX Association, 2018.

[54] ZILLIQA. The zilliqa technical whitepaper. *https://docs.zilliqa.com/whitepaper.pdf*, 2017.