FalconDB: Blockchain-based Collaborative Database

Siddharth Singh (CS19B072)

Guide: **Dr. John Augustine**

Department of Computer Science and Engineering
Indian Institute of Technology, Madras

BTech Project (Jan-May 2023)

Presentation Outline

About FalconDB

- 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.
- FalconDB enables different parties with limited hardware resources to efficiently and securely collaborate on a database thus solving this trilemma among security, compatibility and efficiency.
- 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.

Blockchain

- A 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:
  - $\text{hash}(H) = H.lastBlockHash$
  - $H.height = H.height + 1$
  - $\text{hash}(C) = H.dataHash$
  - $\text{validate}(V) = 1$ for a function validate defined by the blockchain protocol
- 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.

Authenticated data structures (ADS)

- 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 Q and updates 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. It returns a result $R = q(D)$ with proof π.
  - Function VerifyQry takes as input a digest δ, a query $q \in Q$, result R, and proof π. This function should guarantee that if $\delta = \text{Sum}(D)$, it will return 1 if and only if $(R, \pi) = \text{Qry}(D, q)$.
    - 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 $u \in U$, while UpdC takes as input the same update u and the data digest δ which satisfies $\delta = \text{Sum}(D)$.
    - After a successful interaction, UpdS outputs the new database content $D1 = u(D)$ and UpdC outputs a digest $\delta 1$ which satisfies $\delta 1 = \text{Sum}(D1)$.
    - 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.

Limitations

- ADSs have been proved to be powerful in ODB where there is only one client in the setting. 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.
  - 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.

System Overview

- 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.
  - **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.

- For each block, the block content C contains exactly one transaction that includes one or more reads/writes to the database.
- 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. The digests can be used for clients to verify if their requests have been correctly executed.

General workflow for a query

- 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 later.
- We also have a privilege function agreed by all nodes in the network. This function can also be simply implemented by any smart contract.
- 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.

Read 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 (with its signature on it) to the client.
- After that, the client can choose to 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.

Update query

- A client can issue an update 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.

Blockchain Construction

- 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.
    - $lastBlockHash = \text{hash}(H_{height} - 1)$, the hash value of previous block.
    - $\phi = \text{hash}(C)$, the hash value of the block content.
    - δ, a digest of the database version when the block is committed.
    - $\text{hash}(RW)$, 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:
    - $s_0 = \text{sign}(sk(e_0), M)$, the signature of e_0.
    - $\{e_1, \ldots, e_k\}$, the blockchain nodes that validate the block.
    - $\{s_1, \ldots, s_k\}$ where $s_i = \text{sign}(sk(e_i), M)$, k signatures of the validators.

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.**
  - 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.

Issue with proof generation mechanism and its solution

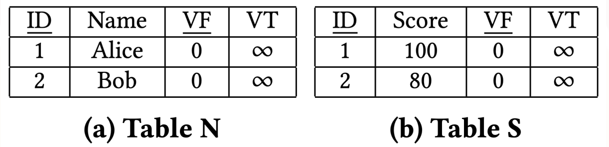
- Even the state-of-the-art verification computing technology can still take more than one hour to generate a proof for a complex query.
- This becomes a bottleneck of the system pipeline and makes the system unfavorable for databases.
- However, with the help of blockchain and smart contract, this process is made asynchronous with the rest of the pipeline.

Data Model

- All participants in the network jointly maintain a blockchain with headers $H = \{H_0, \ldots, H_{height}\}$ 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 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 $D = \{D_0, \ldots, D_{height}\}$,where $D_i = Upd(D_{i-1}, TX_i)$ denotes the database content after executing the updates in the i-th block.

- 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 resp.
- 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.
- For an insertion, the record is inserted with VF being the current height of blockchain, h, and VT being ∞. Similarly, 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 $VT = \infty$).

Example query execution (Server side)



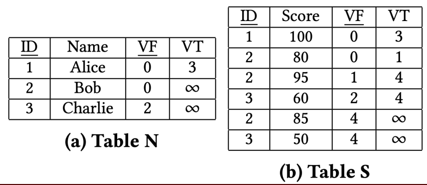
ID	Name	VF	VT
1	Alice	0	∞
2	Bob	0	∞

(a) Table N

ID	Score	VF	VT
1	100	0	∞
2	80	0	∞

(b) Table S

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



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

Queries in FalconDB

- **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.**
  - 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.
  - 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.

- **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;
```

Analysis

- **Experimental Setup.** In the evaluation, FalconDB incorporates the following components:
  1. MySQL server as the backend SQL server
  2. IntegriDB to provide ADS and support authentication
  3. Tendermint as the blockchain platform

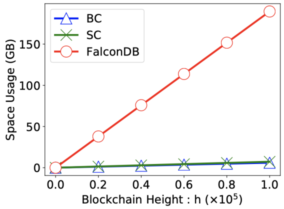
  The experiments are conducted on CloudLab, where each node is equipped with a 2.4 GHz Ten-core Intel E5-2640v4 processor and 64GB DRAM. We run our experiments on a cluster with $N_s = 5$ server and $N_c = 27$ clients. To simulate the difference between the server and the client, we manually slow down 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.

Baseline

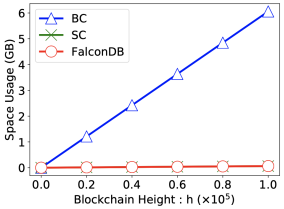
- We compare the performance of FalconDB with the two existing blockchain database solutions:
  1. a naive blockchain-based shared database where each user stores a full data copy (BC), and
  2. a smart contract based solution where a user could submit smart contracts to query full nodes and get consented 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.

Storage



(a) Server

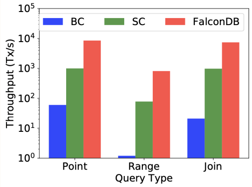


(b) Client

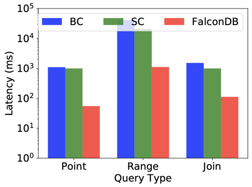
- As shown in the figure, for a table with size around 3GB, the server storage of SC and the node storage of BC are about the same as the table size, while the storage of FalconDB is around 200GB. This is a reasonable overhead compared to prior work on authenticated databases. On the other hand, each client in FalconDB or SC only needs to store less than 100MB size of blockchain for the table.

- 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.
- 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



(a) Throughput

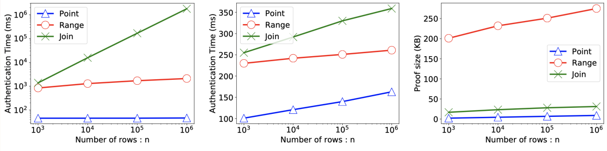


(b) Latency

- 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 106. For the second and third query types, each result contains precisely 100 rows.

- 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. On the other hand, each query in FalconDB is performed through a direct server-client communication, so the throughput is much higher.
- 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.

Query authentication latency



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

- 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. The above figure 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 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 the server side.

Update efficiency

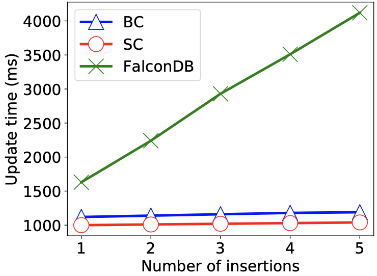


Figure 8: Update performance on an empty table with 10 columns.

- 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 [2], it is impossible to achieve sub-linear performance cost for batch updates in any powerful ADSs built upon cryptographic accumulators.

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.

References

1. Peng, Y.; Du, M.; Li, F.; Cheng, R.; Song, D. FalconDB: Blockchain-based collaborative database. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 637–652.
2. 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.

Thank You