FalconDB: Blockchain-based Collaborative Database

Yanqing Peng¹, Min Du², Feifei Li¹, Raymond Cheng², Dawn Song²

University of Utah¹       UC Berkeley²
Shared database - multiple clients

Multiple clients accessing the same database.
Example: Crowdsourcing restaurant rating

Tell me all the Chinese restaurants in New York with high rating.

The restaurant “XX kitchen” is good! 5 stars!

```
select * from Restaurants R where R.Zip = 94707 and R.Category="Chinese" and R.Avg_rat > 4;
```

```
insert into Rate values (1000, 2, 5);
```
Trusted database server solution

Database held by Yelp

Server decides:
1. Transaction order;
2. The result of a query;
3. If an update should go through;
...

Google docs  Overleaf  Github

How about a decentralized rating system without centralized server?
Shift the trust

**Blockchain-based solution**

Consensus among all participants:

1. Transaction order;
2. The result of a query;
3. If an update should go through;
   ....

Guarantees transparency and immutability.
Shift the trust

**Naive Blockchain-based solution**

Each database client stores a full copy of the database, and run consensus in a permissioned blockchain network.

- FluereeDB
- BigchainDB
- swarmDB

**High storage cost!**

Find a restaurant with an old smartphone?
Shift the trust

Light blockchain node

Individual users querying full nodes.

How to ensure integrity without trusting full nodes?
Tool: Verifiable Computation

➢ Assume the service provider and the users both have a “digest” of the data;
➢ Service provider returns results with cryptographic proof based on the digest;
➢ Users verify integrity of results using the proof based on the digest.

➤ Guarantee: if the digest is correct, then the validation process passes if and only if the query result is correct

❖ Enabling trust on full nodes:

➤ Light nodes query full nodes, and verify the results using VC.
FalconDB: architecture overview

- **Block:**
  - contains all update logs

- **Full nodes (servers):**
  - store data + update logs (block data)

- **Light nodes (clients):**
  - store digest of data (block header)
FalconDB: verifiable query execution

(1) **Client** extracts digest (D) from the newest blk header.

(2) **Client** sends query: Q

(3) **Server** reply result R + proof P

(4) **Client** verifies the result (using proof+digest)
FalconDB: updating the database

1. Client sends update: U

2. Server broadcasts the new block with update U and new digest d’

3. Blockchain nodes validate the block (in particular, check the correctness of new digest)
Challenge: VC overhead

1. Fetch newest digest
2. Send query

3. Execute query
4. Generate proof
5. Return result
6. Return proof

7. Validate the result with proof and digest

Overhead: Steps 1, 4, 6, 7 (temporal)

Step 4 could be very slow for complicated queries
- Normal query: ~10s; Big query: ~6000s
- Bottleneck of query/update!
Asynchronous proof generation

❖ Recall:
  ➢ Query execution is fast; proof generation is slow.

❖ Observation:
  ➢ With ADS, all dishonest behavior from server could be detected.

❖ Solution:
  ➢ Optimistically trust the results from server and ask for a proof later.
  ➢ Proof generation become asynchronous; won't block any process.
  ➢ Impose high penalty if the server fails to provide proof later.
  ➢ Use external smart contract as incentive model.
Incentive model

• Servers deposit to a smart contract.
• Client can challenge the server.
• Failed to provide a proof in time -> deposit is confiscated.
• Successfully provide a proof -> client pays the server a transaction fee.

<table>
<thead>
<tr>
<th>Server</th>
<th>Deposit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>100,000</td>
</tr>
<tr>
<td>Client 1</td>
<td>100</td>
</tr>
<tr>
<td>Client 2</td>
<td>100</td>
</tr>
</tbody>
</table>
FalconDB advantages

❖ Low requirements on clients
  ➢ Allow participating from any device

❖ Secure
  ➢ Result Integrity
  ➢ Transparent and Immutable updates
  ➢ Clients can’t be cheated even all full nodes are malicious

❖ High performance
  ➢ Query Performance = Traditional Server-Client
  ➢ Blockchain performance / validation performance / update performance =
    Most state-of-the-art work
    ➢ Since blockchain consensus and ADS are used as blackboxes, we can always replace
      them by the newest work
Evaluation Setup

❖ Baseline:
  ➢ “Naïve Blockchain”: each node maintains the full database; sync updates with blockchain
  ➢ “Smart Contract”: each full node maintains the database; light nodes query by submitting queries to the blockchain

❖ FalconDB setting:
  ➢ 5 servers, 27 clients

❖ Database workload:
  ➢ a single table with n rows and m columns
Evaluation

- Space cost on Servers and Clients
- FalconDB shifts the high storage cost from local clients to server only.
Evaluation

- Query and Update performance
- FalconDB has best query performance
- Updates are slower but acceptable
Summary

FalconDB:

➢ A blockchain based collaborative database.
➢ Clients store a little piece of data, and connect to servers to issue query/update.
➢ Clients are able to verify query/update results with authenticated data structures (ADS).
➢ High performance, high security guarantee.

THANK YOU!
Backup Slides
Baseline Solution #1: Smart Contract

Blk #1 -> Blk #2 -> Blk #3

Execute query Q

Result=0

Query: Q
Result: 0

Result=1 (Lie)
Baseline Solution #1: Smart Contract

Drawbacks...

Execute query Q

Limited TPS

Result=0

Privacy concern

Consensus overhead

Gas consumption
Baseline Solution #2: Verifiable Computation

❖ Verification Computation (VC)
  ➢ Service provider returns results with cryptographic proof;
  ➢ Users verify integrity of results using the proof.

❖ Enabling trust on full nodes:
  ➢ Light nodes query full nodes, and verify the results using VC.
VC as a blackbox

• Summary
  • Data $D \rightarrow$ ADS $S \ast$ Digest $\delta$

Input: array data

Output: merkle tree over the array

Example: array data with Merkle tree as VC data structure
VC as a blackbox

- Query (Server side)
  - Data $D \times \text{Query } Q \rightarrow \text{Result } R \times \text{Proof } \pi$
- Verify Query (Client side)
  - Digest $\delta \times \text{Query } Q \times \text{Result } R \times \text{Proof } \pi \rightarrow \{0, 1\}$

Send: $\text{Array}[1] = \text{?}$

Check: $h(h(h11|h(B))|h22) = \text{digest}$?

Return: $\text{Array}[1] = B$

Return: Proof = $(h11, h22)$

Example: array data with Merkle tree
Example: array data with Merkle tree

VC as a blackbox

• Update (Server side)
  • Data \( D \) * Update \( U \) -> Updated data \( D' \) * Digest of updated data \( \delta' \) * Proof \( \pi \)

• Verify Update (Client side)
  • Old digest \( \delta \) * New digest \( \delta' \) * Update \( U \) * Proof \( \pi \) -> \( \{0, 1\} \)

Send: Array[1]:=B'

Check: h(h(h11|h12)|h22)=digest?
Check: h31'= h(h(h11|h(B))|h22)?
Update: digest=h31'

Calculate: h12' = h(B);
  h21'=h(h11|h12');
  h31'=h(h21'|h22)
Return: Proof = (h11, h12, h22, h31')
## Comparison between different approaches

<table>
<thead>
<tr>
<th></th>
<th>Client storage</th>
<th>Client set-up</th>
<th>Query execution side</th>
<th>Query Throughput</th>
<th>Query Latency</th>
<th>Trust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>No</td>
<td>No</td>
<td>Server</td>
<td>High</td>
<td>Low</td>
<td>Server</td>
</tr>
<tr>
<td>Blockchain</td>
<td>High</td>
<td>Slow</td>
<td>Client</td>
<td>Low</td>
<td>High</td>
<td><strong>No</strong></td>
</tr>
<tr>
<td>Smart contract</td>
<td>Low</td>
<td>Quick</td>
<td>Server</td>
<td>Low</td>
<td>High</td>
<td>No</td>
</tr>
<tr>
<td>Blockchain +VC</td>
<td>Low 😊</td>
<td>Quick 😊</td>
<td>Server 😊</td>
<td>High 😊</td>
<td>Depends on proof generation</td>
<td>No 😊</td>
</tr>
</tbody>
</table>

**Challenge**
Other tech details

- Blockchain scalability
  - Algorand

- Easy history retrieval
  - Temporal database model

- Supporting DB transactions
  - Optimistic Concurrency Control