BlockchainDB—Towards a Shared Database on Blockchains

Carsten Binnig
Data Management Lab, TU Darmstadt
Blockchains: A Shared Database?

Blockchains are not only used for crypto-currencies today. More and more applications are using blockchains as shared databases.

Main reasons why blockchains are being used for data sharing:

• Keeps history of all transactions (Even counts as evidence in court)
• No tampering after-the-fact (once data is written)
• Needs no trusted authority
Potential Use Cases

Sharing **Health Records** ([https://medicalchain.com](https://medicalchain.com))


Decentralized **Copyright Management** (e.g., [https://binded.com/](https://binded.com/) for images)

Decentralized **Domain-Name-Service** ([https://namecoin.org/](https://namecoin.org/))

...
Are existing **Blockchains** good enough to be used as a shared database?
Outline

Blockchain Background

Challenges of using Blockchains

BlockchainDB – A Shared Database on Blockchains

Summary and Next Steps
The Technology behind Blockchains
(from 10000 feet)

Blockchains peers use a tamper-proof ledger to store shared data
• Ledger is an append-only list of all tx’s (e.g., tx = transfers between accounts)
• Tx’s are appended in blocks to ledger
• Ledger is fully-replicated across peers

Consensus ensures that every peer agrees on new tx’s appended to ledger

Smart contracts are “trusted” procedures in the BC triggered by tx’s to modify data
Categories of Blockchain Networks

Public (aka permission-less)

- **Anyone** can participate in the BC network as a participant
- Uses expensive **computation-based consensus protocols** (e.g., proof of work)
- **Example:** Bitcoin, Ethereum (public)

Private (aka permissioned)

- Limited to a **small set of known participants**
- Uses less expensive **voting-based consensus protocols** (e.g., PBFT, ...)
- **Example:** Hyperledger, Ethereum (private)
Outline

Blockchain Background

Challenges of using Blockchains

BlockchainDB – A Shared Database on Blockchains

Summary and Next Steps
Challenge 1: Performance of Blockchains

Low throughput (<100’s tx/s on average) and high latency

AND bad scalability with # of peers

Not sufficient for many use-cases (e.g., Visa processes on avg. 2000 tx/s)

Challenge 2: “Zoo” of Blockchains

<table>
<thead>
<tr>
<th></th>
<th>Application</th>
<th>Smart contract execution</th>
<th>Smart contract language</th>
<th>Data model</th>
<th>Consensus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperledger (v1.0.0 [35])</td>
<td>General applications</td>
<td>Dockers</td>
<td>Golang, Java</td>
<td>Key-value</td>
<td>PBFT</td>
</tr>
<tr>
<td>IOTA [36]</td>
<td>Digital assets</td>
<td>-</td>
<td>-</td>
<td>Account-based</td>
<td>IOTA’s Tangle Consensus</td>
</tr>
</tbody>
</table>

Many different **programming and execution models**!

Unclear which one is **best for your workload**?

Hard to predict **which platforms will “survive”**!

Challenge 3: Missing Guarantees and Functions

Blockchains provide only limited guarantees for data access (e.g., no guarantees for reads -> executed by only ONE peer!!!)

Guarantees desired for shared databases

• Verifiability of execution of DB transactions (sequence of reads & writes)
• Recovery to valid checkpoints (before violation was detected)

Many other desired functions for data sharing missing in BC’s:
privacy (e.g., by encryption) of data, fine-grained authorization, ....
Outline

Blockchain Background

Challenges of using Blockchains

BlockchainDB – A Shared Database on Blockchains

Summary and Next Steps
Vision of BlockchainDB

1. **Unified API & Pluggable Backends**  
   (i.e., be the MySQL for Blockchains)

2. **Apply typical DB optimizations in Middleware**  
   (e.g., sharding, batching, ...)

3. **Support for verifiable DB transactions**  
   (i.e., sequences of reads/writes to BC)
First Step: BlockchainKV (Goal 1)

**BlockchainKV**: Middleware which provides a **unified put/get interface for different BC backends** (later: full transaction support on top)

![Diagram](https://via.placeholder.com/150)

Unified API

```
put(k, v)  get(k) -> v
```

Pluggable Backends

- Ethereum
- Hyperledger
- ...
Performance Optimizations in BlockchainKV

- **Sharding** of data in BC
- **Reduced # of Replicas** per shard
- **Lower Consistency Levels** -> higher performance
- **Batching of put’s** to lower the BC overhead per put
- **Caching data for get’s** but still enabling verification
- ...
BlockchainKV: Consistency

Provide different client-side consistency levels: lower cons. -> higher perf.

Read-Your-Writes:
- **Put**: submit tx to BC and add it into pending tx-queue in middleware (if tx is valid)
- **Get**: wait for pending put tx’s

Eventual consistency:
- **Put**: same as before
- **Get**: can be executed without waiting for pending put’s!

Workload: 50% reads / 50% writes (Ethereum as backend)
Blockchain has a **high per-tx overhead** (e.g., validation of tx)

Batching in BlockchainKV **merges multiple put’s into on BC tx**

**Trivial for Eventual Consistency** but more complex for **Sequential Consistency**

**Workload**: 100% writes (Ethereum as backend)
BlockchainKV: Verifiable Consistency (Goal)

Main Idea:
- Clients can verify correctness of all KV operations (put’s and get’s)
- I.e., verify that puts’ and get’s adhere to selected consistency level

Example: Eventual Consistency
- Read-set (RS) $\subseteq$ write-set (WS) of all clients (i.e., no “fake” reads)
- Liveliness (i.e., no dropped writes)
Untrusted components can be compromised (i.e., “misbehave”)

Example: Violation of Eventual Consistency

- BlockchainKV (or even a BC Peer) can “misbehave” if compromised:
  - Get’s returns “fake”-values for a key OR
  - Put’s are dropped

Client: \[ \text{put}(k, v) \quad \text{get}(k) \rightarrow v \]
BlockchainKV: Verification Procedure

BlockchainKV uses deferred verification to detect violations of consistency guarantees

Idea: Epoch-based verification for Eventual Consistency (simplified)

- Blockchain keeps updated $WS_{KV}$ of BlockchainKV (ALL put’s)
- Clients logs $RS/WS_{Clients}$ of current epoch (bypasses BlockchainKV!)
- Check at end of epoch (non-blocking)
  - $WS_{Clients} \subseteq WS_{KV}$ (no dropped writes)
  - $RS_{Clients} \subseteq WS_{KV}$ (no “fake” reads)

Deferred Verification:

- Client
  - `put(k1,v1, prn1)`
  - `recordPut(k1,v1, prn1)` sent to majority
  - `recordGet(k1,v1, prn1)` sent to majority

- KV-Store
  - `get(k1)`
  - `v1`, `prn1`

- BC Peer
  - `verifyEpoch()`
  - `tx-id`
  - `checkResult(tx-id)`

- Store data + Update $WS_{DB}$ (non-blocking)
- Update $WS_{Clients}$ (non-blocking)
- Check correctness (non-blocking)
Outline

Blockchain Background

Challenges of using Blockchains

BlockchainDB – A Shared Database on Blockchains

Summary and Next Steps
What’s next?

BlockchainKV only a first step towards a Shared Database System on Blockchains

Next Steps:
• Add further optimizations (e.g., caching) to middleware
• Add support for verifiable DB Transactions on top
• Hardware supported verifiable DB Transactions

Long term: Integration into existing DBMSs (e.g., as a “shared” column/table)?
Collaborators

Muhammad El-Hindi
Sumith Kulal
Arvind Arasu
Ravi Ramamurthi
Donald Kossmann

See also https://distributedledger.center/
Thank you!