BlockchainDB - Towards a Shared Database on Blockchains

Muhammad El-Hindi
TU Darmstadt

Martin Heyden
TU Darmstadt

Arvind Arasu
Microsoft Research

Carsten Binnig
TU Darmstadt

Donald Kossmann
Microsoft Research

Ravi Ramamurthy
Microsoft Research

ABSTRACT

In this demo we present *BlockchainDB*, which leverages blockchains as storage layer and introduces a database layer on top that extends blockchains by classical data management techniques (e.g., sharding). Further, *BlockchainDB* provides a standardized key/value-based query interface to facilitate the adoption of blockchains for data sharing use cases. With *BlockchainDB* we can thus not only improve the performance and scalability of blockchains for data sharing but also decrease the complexity for organizations intending to use blockchains for this use case.

ACM Reference Format:
Muhammad El-Hindi, Martin Heyden, Arvind Arasu, Carsten Binnig, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB - Towards a Shared Database on Blockchains . In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320237

1 INTRODUCTION

Motivation. Blockchain (BC) technology emerged as the basis for crypto-currencies, like Bitcoin [9] or Ethereum [3], allowing parties that do not trust each other to exchange funds and agree on a common view of their balances. With the advent of smart contracts, blockchain platforms are being used for many other use cases beyond crypto-currencies and include applications in domains such as governmental, healthcare and IoT scenarios [1, 2, 10].

An important aspect that many scenarios have in common is that blockchains are being used to implement shared data access of parties that do not trust each other. For example, one use case is that the blockchain is used for tracking of goods in a supply chain. What makes blockchains attractive in those scenarios are two main characteristics: First, blockchains store data in an immutable append-only ledger that contains the history of all data modifications. That way, blockchains enable auditability and traceability in order to detect potential malicious operations on the shared state. Second, blockchains can be operated reliably (tolerating byzantine failures) in a completely decentralized manner without the need to involve a central trusted instance which often does not exist in the use cases mentioned before.

However, while there is a lot of excitement around blockchains in industry, they are still not being used as a shared database (DB) in many real-world scenarios. This has different reasons: First and foremost, a major obstacle is their limited scalability and performance. Recent benchmarks [5] have shown that state-of-the-art blockchain systems such as Ethereum or Hyperledger that can be used for general applications can only achieve 10's or maximally 100's of transactions per second which is often way below the requirements of modern applications. Second, blockchains lack easy-to-use abstractions known from data management systems such as a simple query interface as well as other guarantees such as well-defined consistency levels that guarantee when/how updates become visible. Instead, blockchains often come with proprietary programming interfaces and require applications to know about the internals of the blockchain system to decide on the visibility of updates. For example, when using Ethereum, clients have to manually check if their updates have successfully been committed to the main branch by waiting a certain number of blocks.

Contributions. In this demo, we present *BlockchainDB* to tackle the before-mentioned issues. The main idea is that *BlockchainDB* leverages blockchains as storage layer. That way, existing blockchain systems can be used (without modification) as a temper-proof and de-centralized storage. On top of the storage layer, *BlockchainDB* adds a database layer that implements the following functions:

- *Partitioning and Replication*: A major performance bottleneck of blockchains today is that all peers hold a full copy of the state and still only provide (limited)

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3320237

sharding capabilities. In *BlockchainDB*, we allow applications to define in the database layer how data is replicated and partitioned across all available peers. Thus, applications can trade performance and security guarantees in a declarative manner.

- *Query Interface and Consistency*: In the database layer, *BlockchainDB* additionally provides easy-to-use abstractions including different consistency protocols (e.g., eventual and sequential consistency) as well as a simple key/value interface to read/write data without knowing the internals of a blockchain system.

In addition to these two functions, *BlockchainDB* comes with an *off-chain verification procedure*. The idea of the verification procedure is that clients can detect potential misbehaving peers in the *BlockchainDB* network. This is needed since not all *BlockchainDB* peers hold the full state and the storage layer of a remote peer could potentially drop put-operations or return spurious values for get-operations.

By introducing a database layer on top of an existing blockchain as storage layer, *BlockchainDB* is able to not only provide higher performance, but also to massively decrease the complexity for organizations, that intend to use blockchains for data sharing. While the concept of moving certain functions out of the blockchain into additional application layers has been studied previously (e.g., [4, 6–8]), to the best of our knowledge, *BlockchainDB* is the first system to provide a fully functional database layer on top of blockchains.

Outline. The remainder of this paper is organized as follows: First, in Section 2 we provide an overview of the *BlockchainDB* architecture as well as the provided trust guarantees. Then, in Section 3 and Section 4 we discuss the details of *BlockchainDB*'s database and storage layer. Finally, in Section 5 we conclude with a description of our demo scenario that we intend to present to the audience.

2 OVERVIEW

In the following, we first give an overview of the architecture of *BlockchainDB* and then discuss which attacks can be mitigated by the off-chain verification of *BlockchainDB*.

2.1 System Architecture

Figure 1 shows a typical deployment of *BlockchainDB* across a network of four different *BlockchainDB* peers (i.e., untrusted parties) that access the same shared data. Similar to permissioned blockchains, in *BlockchainDB* we assume that the parties who want to share data are previously known to each other.

In order to participate in a *BlockchainDB* network and allow clients of a party to read/write data into a shared database/table, a peer in *BlockchainDB* can be either deployed as a *full peer* which hosts data and participates in a shard

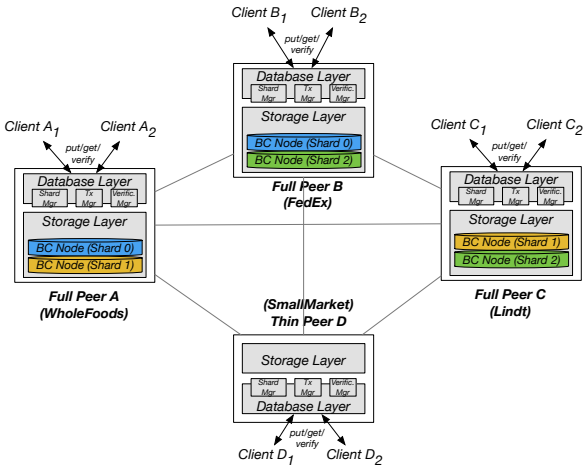

Figure 1: A typical *BlockchainDB* Network

with its storage layer or as a *thin peer* which only hosts a database layer (i.e., the peer's storage layer does not store data locally). Having thin peers enables parties with only limited resources to participate in a *BlockchainDB* network and access the shared database or table. In the following, we explain how clients interact with a *BlockchainDB* as well as the functionality of each layer.

Clients. Instead of interacting with a blockchain peer directly, in *BlockchainDB* clients interact with the database layer through a simple `put/get` interface to read from or write to the shared state. Furthermore, clients can use the `verify` method of the database layer to trigger an off-chain verification procedure for the read/write-set of that client (since the last verification call) to detect potential misbehaving peers in a *BlockchainDB* network (e.g., to detect if another *BlockchainDB* peer — more precisely its storage layer — returns a spurious read or drops a put). An example to intuitively explain verification is discussed in the next section.

Database Layer. The database layer in *BlockchainDB* is mainly responsible to execute the `put/get` calls from the clients. If a `put/get` call comes in, the database layer uses the *Shard Manager* to decide to which (copy of a) shard the operation should be directed to. The shard can be either stored in its local storage layer or remotely in another *BlockchainDB* peer depending on a pre-defined distribution scheme. Currently, *BlockchainDB* implements a hash-based sharding approach, in which the user defines the number of shards and their allocation to *BlockchainDB* peers when creating a new shared table. Another major difference of *BlockchainDB* and a pure blockchain network is that the database layer of *BlockchainDB* implements a *Transaction Manager* that implements well-known consistency levels (e.g., eventual consistency and sequential consistency). That way, clients not only get a defined behavior for concurrent read/writes without the need to know the internals of blockchains, but the database layer can additionally re-order/batch reads/writes depending

on the chosen consistency level to optimally leverage the underlying blockchain-based storage layer and thus further improve the performance.

Storage Layer. The storage layer serves as a persistent, auditable storage backend of *BlockchainDB* using existing blockchain systems as temper-proof storage. As depicted, the storage layer of *BlockchainDB* is able to parallelize data processing across different shards whereas each shard is implemented using a separate blockchain network which replicates its state to multiple (but not necessarily all) peers using the blockchain internal consensus protocols. For example, in Figure 1 *BlockchainDB* uses in total three different blockchain networks (one for each shard) while each shard is only replicated to two peers.

2.2 Attacks and Trust Guarantees

In *BlockchainDB*, we aim to detect two categories of attacks.

First, a common problem in a shared data scenario is that one party might change the shared state to its advantage. For example, in a supplier-scenario the party who delivers the products might update the price of the products after an order has been placed from the other party. In *BlockchainDB* such a modification-after-the-fact could be detected since the storage layer uses blockchains to store the history of all updates in a temper-proof ledger. In consequence, any later modification of data will be transparent to all parties. Clearly, the logic to verify the validity of updates has to be implemented in the application layer on top of *BlockchainDB*, but the important fact is that the database guarantees the audibility.

Second, individual *BlockchainDB* peers do not hold the full database state and thus redirect a subset of reads/writes to the storage layer of other peers. However, these peers might be potentially malicious and drop puts or return spurious values for get-operations. To detect this type of attack, *BlockchainDB* tracks all issued operations in the blockchain and provides clients with an off-chain verification procedure that checks whether the read/write-set has been correct.

The off-chain verification procedure is implemented in the database layer of *BlockchainDB* and comes in two flavors: In online verification, a client can call a separate verify method after a put/get-operation that checks the validity of the last operation. For example, to verify a get-operation the value for a given key is retrieved from the majority of peers. However, online verification comes with significant additional overhead. We therefore additionally support (a deferred) offline verification procedure which verifies reads/writes in batches. The deferred verification procedure provides less overhead than the online verification procedure. However, clients might work (for a limited amount of time) on an unverified database state.

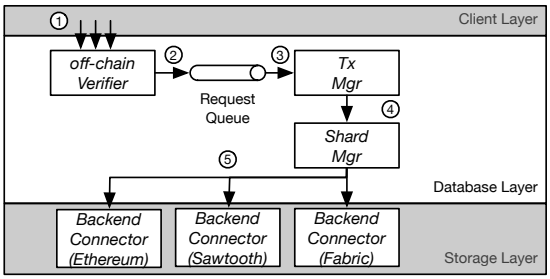

Figure 2: Database Layer

3 DATABASE LAYER

In the following we explain the database layer and briefly discuss the involved components as depicted in Figure 2.

As mentioned before, when a new shared table is created in *BlockchainDB*, the user needs to specify the number of shards and their sizes (i.e., to define how many replicas of a shard exist). The database layer of *BlockchainDB* can setup the required shards using a `ShardController` component (not shown in Figure 2) that automatically creates a new blockchain network. As shown by Dinh et al. [5], a higher number of peers in a blockchain network decreases the throughput, however, it offers higher security since the data is protected by a larger network.

For accessing a shared table, clients send requests to their local *BlockchainDB* peer using a simple key/value interface. Requests arriving from the client in the database layer are first received by the off-chain verifier ① that records the unverified reads/writes of a client for online/offline verification (as discussed before). The verifier then forwards the request to an internal *RequestQueue* ②. Next, the `TransactionManager` polls the requests from the queue and processes them according to a specified consistency level[1] ③. Currently, we support sequential and eventual consistency. The different consistency levels differ in how quickly the database layer can process operations. For example, for eventual consistency a get-operation is immediately answered, however, no guarantee is given that a potential outstanding put-operation has already been committed to the blockchain. In sequential consistency, the database layer needs to execute put/get-operations in order and thus, potentially blocks a get-operation until an outstanding put-operation has been committed to the blockchain. Once a put/get-operation is ready to be sent to the storage layer, it is forwarded to the `ShardManager` ④. The `ShardManager` service has two main purposes: First, it determines the correct shard for a given key and second, it is responsible for sending the request in parallel to the storage shards. To do this multiple `BackendConnectors` are being used ⑤ that allow *BlockchainDB* to write data into the underlying storage layer as discussed in the next section.

[1]The consistency level can be specified for each shared table.

4 STORAGE LAYER

A shared table in *BlockchainDB* has a key/value data model where key is the primary key and value can represent the payload of the table. For keys and values, the database layer provides different data types that an application can choose from when creating a new table (such as `INT`, `DECIMAL`, `STRING`). For values, *BlockchainDB* also supports complex `JSON`-based data types that are comprised of lists and nested structures.

For storing shards of a table in the storage layer, the database layer converts the key/value payload into a byte representation. We found out that representing the data in a byte-format before storing it in a blockchain backend not only leads to decreased storage cost, but also allows for more efficient processing on the blockchain. In the future, we plan to study the application of lightweight compression techniques in the database layer to further optimize writing and retrieving data from the storage layer.

In order to support a new blockchain backend to store shards, *BlockchainDB* requires a minimal smart contract definition which provides a simple `put/get` access to the blockchain state via a so called `BackendConnector`. Different `BackendConnectors` are currently available for Ethereum, Hyperledger Sawtooth as well as Hyperleder Fabric. In the following, we show a simplified extract of the smart contract code that is used by our Ethereum `BackendConnector`.

```
contract KVContract {
  // state variables and constructor omitted
  function get(bytes memory key)
  public view returns(bytes memory value){
    return data[key];}
  function put(bytes memory key, bytes memory value)
  public returns(bool success) {
    data[key] = value; return true;}
}
```

5 DEMONSTRATION SCENARIO

In our demo we show-case how *BlockchainDB* can support data sharing in various scenarios by optimizing the system for a given workload. To this end, our demo provides two interfaces, an *administration* and a *simulation* interface.

Administration Interface. This interface (left side in Figure 3) shows the current system status and performance (in terms of throughput/latency) for different workloads (read-heavy vs. write-heavy). For the demo, we setup *BlockchainDB* networks with up to 24 *BlockchainDB* peers. Each peer represents a participant in the network that shares data with other peers. The administration interface enables the user to change system parameters of *BlockchainDB* like the replication factor per shard (1-24), number of shards per table (1-24) or the consistency-level (eventual or sequential consistency) and see how the system performance is affected.

Simulation Interface. This interface allows the user to simulate attacks and verify that they were successfully detected

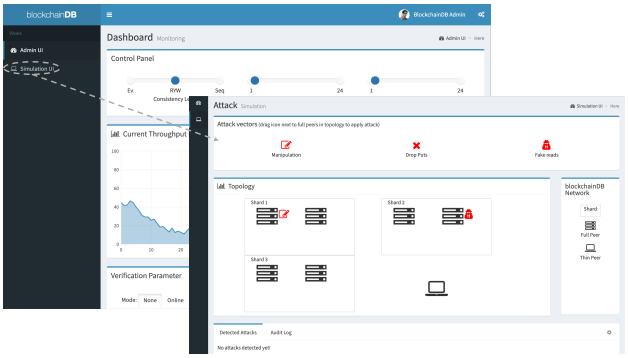

Figure 3: Admin (left) and Simulation Interface (right)

by *BlockchainDB*. While our simulation is generic, the simulation components can be mapped to any use case that benefits from data sharing, such as supply chain, health networks, etc. With the help of the simulation interface (right in Figure 3), we present three scenarios in our demo application: (1) An attacker wrote manipulated data into the shared table, which was detected by a participant in the network. An auditor wants to account which participant wrote the responsible value into the shared table. One can think of *BlockchainDB* in this context, as a black box that reliably records all writes and enables a later audition of all actions. (2) A malicious peer tries to prevent other peers from writing to the shared table by dropping their writes. Users can observe, that the attack has been detected and logged in the simulation interface. (3) A malicious peer returns spurious values (not actually stored in a shared table). As such attacks are detected, they will be also logged in the simulation interface.

REFERENCES

[1] A. Azaria et al. MedRec: Using Blockchain for Medical Data Access and Permission Management. In *OBD 2016*, Aug. 2016.
[2] C. Berger et al. On Using Blockchains for Safety-Critical Systems. In *IEEE/ACM SEsCPS 2018*, May 2018.
[3] V. Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
[4] R. Cheng et al. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *CoRR*, abs/1804.05141, 2018.
[5] T. T. A. Dinh et al. BLOCKBENCH: A framework for analyzing private blockchains. In *SIGMOD 2017*, 2017.
[6] J. Eberhardt et al. On or Off the Blockchain? Insights on Off-Chaining Computation and Data. In F. De Paoli, S. Schulte, and E. Broch Johnsen, editors, *Service-Oriented and Cloud Computing*, Lecture Notes in Computer Science. Springer International Publishing, 2017.
[7] J. Eberhardt et al. Off-chaining Models and Approaches to Off-chain Computations. In *SERIAL 2018*, Rennes, France, 2018. ACM Press.
[8] C. Molina-Jimenez et al. Implementation of Smart Contracts Using Hybrid Architectures with On and Off-Blockchain Components. In *IEEE SC2 2018*, Nov. 2018.
[9] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.
[10] S. Underwood. Blockchain Beyond Bitcoin. *Commun. ACM*, 59(11), Oct. 2016.