BlockchainDB - A Shared Database on Blockchains

Muhammad El-Hindi
TU Darmstadt
muhammad.el-hindi@cs.tu-darmstadt.de

Carsten Binnig
TU Darmstadt
carsten.binnig@cs.tu-darmstadt.de

Arvind Arasu
Microsoft Research
arvinda@microsoft.com

Donald Kossmann
Microsoft Research
donaldk@microsoft.com

Ravi Ramamurthy
Microsoft Research
ravirama@microsoft.com

ABSTRACT

In this paper we present *BlockchainDB*, which leverages blockchains as a storage layer and introduces a database layer on top that extends blockchains by classical data management techniques (e.g., sharding) as well as a standardized query interface to facilitate the adoption of blockchains for data sharing use cases. We show that by introducing the additional database layer, we are able to improve the performance and scalability when using blockchains for data sharing and also massively decrease the complexity for organizations intending to use blockchains for data sharing.

PVLDB Reference Format:
Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, Ravi Ramamurthy. BlockchainDB - A Shared Database on Blockchains. *PVLDB*, 12(11): 1597-1609, 2019.
DOI: https://doi.org/10.14778/3342263.3342636

1. INTRODUCTION

Motivation: Blockchain (BC) technology emerged as the basis for crypto-currencies, like Bitcoin [20] or Ethereum [8], 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 [7, 4, 26].

An important aspect that many scenarios have in common, is that blockchains are being used to provide shared data access for parties that do not trust each other. For example, one use case is that the blockchain is used for tracking goods in a supply chain where independent parties log the location of individual goods. 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 data. Second, blockchains can be operated reliably in a decentralized manner without the need to involve a central trusted instance which often does not exist in data sharing. 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 [12] have shown that state-of-the-art blockchain systems such as Ethereum or Hyperledger, that can be used for building general applications on top, 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 like 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 a blockchain to decide on the visibility of updates.

Contribution: In this paper, we present *BlockchainDB* that tackles the before-mentioned issues. The main idea is that *BlockchainDB* leverages blockchains as the native storage layer and implements an additional database layer on top to enable access to data stored in shared tables. That way, existing blockchain systems can be used (without modification) as a tamper-proof and de-centralized storage. On top of the storage layer, *BlockchainDB* implements a database layer with the following functions:

- *Partitioning and Partial Replication:* A major performance bottleneck of blockchains today is that all peers hold a full copy of the state and still only provide (limited) sharding capabilities. In the database layer of *BlockchainDB*, we allow applications to define 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 DB layer, *BlockchainDB* additionally provides shared tables as 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 future, we want to extend the query interface to shared tables to support SQL with full transactional semantics.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342636

In addition to these functions, the database layer of *BlockchainDB* comes with an *off-chain verification procedure* in which peers can easily verify the read- and write-set of their own clients. The idea of the verification procedure is that peers can detect other potentially misbehaving peers in the *BlockchainDB* network. This is needed since not all *BlockchainDB* peers hold the full copy of the database and the storage layer of a remote peer could potentially drop puts or return a spurious value for a read operation (i.e., a value that was not persisted in the database).

By introducing a database layer on top of an existing blockchain, *BlockchainDB* is not only able to provide higher performance, but also to decrease the complexity for organizations intending 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., [14, 13, 19, 9]), to the best of our knowledge, *BlockchainDB* is the first system to provide a fully functional DB layer on top of blockchains. Our experiments show that *BlockchainDB* allows to increase the performance significantly to support many real-world applications.

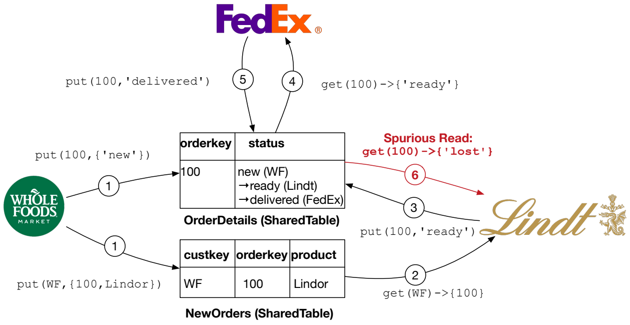
Outline: The remainder of this paper is organized as follows: First, in Section 2 we give an overview of what functionality and security guarantees *BlockchainDB* provides for data sharing. Afterwards, in Section 3 we present the *BlockchainDB* architecture and discuss the trust assumptions as well as potential attacks. Then, in Section 4 and Section 5 we discuss the details of the database layer and how blockchains are being used as the storage layer. Section 6 afterwards outlines our off-chain verification protocol. The results of our evaluations with the YCSB benchmark are then presented in Section 7. Finally, we conclude with related work in Section 8 and a summary in Section 9.

2. OVERVIEW AND GUARANTEES

As explained already in the introduction, there are many different applications where untrusted parties need to have shared access to the same database. To enable such a shared access to data, *BlockchainDB* provides so-called *shared tables*. For accessing a shared table, clients can use the put/get interface of *BlockchainDB* to access tuples in the shared table based on their primary key. In order to understand the functionality and guarantees that *BlockchainDB* provides for untrusted parties to access data via shared tables, we will introduce a short motivating scenario and use this scenario also to outline the guarantees that applications get when using *BlockchainDB* for data sharing.

Scenario: Figure 1 shows an example scenario for data sharing where three untrusted parties (WholeFoods, FedEx, and Lindt) share access to the same database. The scenario describes a typical supplier scenario where WholeFoods acts as customer, Lindt as supplier, and FedEx as shipping company. In this scenario, WholeFoods first places a new order by inserting two new entries into the shared database consisting of two shared tables ①. After the order is placed, Lindt processes the new order ②. To keep track of the order, Lindt updates the status from *new* to *ready* as shown in ③. Once the order is ready, FedEx starts its operation ④. After the order has been shipped to the customer, FedEx updates the status of the order to *delivered* as shown in ⑤.

A naïve way of implementing such a scenario would be that one of the parties is hosting the shared database; e.g., say WholeFoods hosts the shared database as a service for all their suppliers. In this setup, however, WholeFoods could easily leverage the fact that it can manipulate the shared data or return false values to the other parties about the order status, without any chance for the parties to verify that WholeFoods was in fact acting in a malicious manner. For example, WholeFoods could claim that the order was lost during transit by returning a spurious (i.e., false) order status to Lindt as shown in ⑥ to trigger that a replacement is sent (without paying for it). Even worse, WholeFoods could actually delete the order or not store it in the database in the first place. In case of a lawsuit, no evidence could thus be found that WholeFoods (or any other party) was actually acting maliciously.



orderkey	status
100	new (WF) →ready (Lindt) →delivered (FedEx)

OrderDetails (SharedTable)

custkey	orderkey	product
WF	100	Lindor

NewOrders (SharedTable)

Figure 1: A Data Sharing Scenario

Guarantees: In order avoid these problems, a blockchain network such as Ethereum or Hyperledger could be used to implement a shared database. The benefit of using a blockchain is that every party (WholeFoods, Lindt, and FedEx) keeps a full copy of the database and the majority of parties needs to agree (based on the blockchain-based consensus protocol) on every update before its effect becomes visible. As a consequence of using blockchains, we get the following important guarantees for data sharing: First, the data in each peer is stored in a tamper-proof log. That way, any data modification of the log by any potentially malicious party could be detected since the cryptographic hashes used in the blockchain would not be valid anymore. Second, all parties read only state from their local copy of the database. That way, spurious reads (that are a problem if data needs to be read form a remote party) can be avoided.

However, as discussed in the introduction, using blockchains directly for data sharing comes with significant problems (e.g., w.r.t. performance) and complexities due to missing abstractions. The idea of *BlockchainDB* is thus to provide the same security guarantees as blockchains — (1) a tamper-proof (auditable) log as well as (2) verifiable reads and writes. At the same time, *BlockchainDB* enables high performance and provides an easy-to-use query interface.

3. SYSTEM ARCHITECTURE

3.1 Architecture Overview

The main idea of *BlockchainDB* is that it implements a database layer on top of an existing blockchain. The database layer provides clients with a simple-to-use abstraction (called a shared table) with a put/get interface and stores all data in its storage layer that relies on blockchains as discussed before. Figure 2 shows a possible deployment of *BlockchainDB* across four different *BlockchainDB* peers

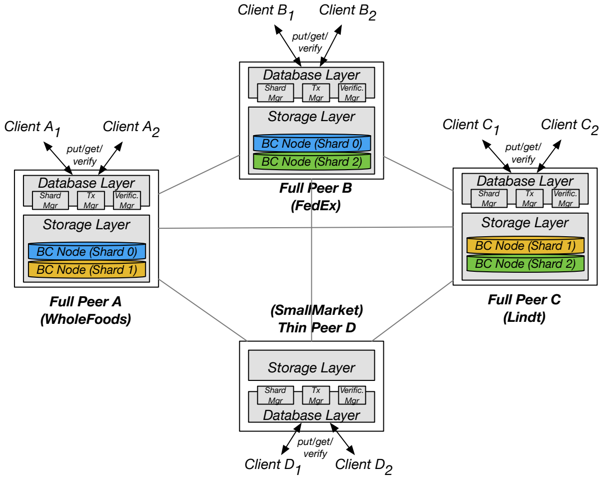


Figure 2: A typical *BlockchainDB* Network

(i.e., untrusted parties) to enable access to shared tables as discussed in the scenario before.

The key idea of *BlockchainDB* is that data is not replicated to all peers to avoid the high overhead of blockchain consensus. Instead, shared tables are partitioned (i.e., sharded), thereby each shard is implemented as a separate blockchain network. Moreover, shards are only replicated to a limited number of peers instead of replicating the data to all peers. For example, in the scenario explained in the previous section, both shared tables (*NewOrders* and *OrderDetails*) can be partitioned using the *OrderKey* as partitioning key and replicated only to a subset of the peers (WholeFoods, Lindt, and FedEx).

As a direct consequence of sharding to speed-up the performance not all peers store all data locally. When accessing the shared table, the database layer thus needs to redirect the request either to the local or a remote storage depending on the requested key. While storing data in a blockchain still gives us a tamper-proof log, the remote peer can drop a put or return a spurious value for a read. To verify all remote reads/writes an additional verification procedure is thus provided by *BlockchainDB*.

In order to participate in a *BlockchainDB* network and allow clients of a party to read/write data into a shared database, a peer in *BlockchainDB* can be either deployed as a *full peer* which hosts a database and a replica of at least one shard or as a *thin peer* which only connects to other remote peers to access data in a shard (i.e., the peer does not store a copy of a shard). Having thin peers enables parties with only limited resources to participate in a *BlockchainDB* network and access the shared tables (such as a small supermarket, called *SmallMarket* in our example).

Finally, similar to permissioned blockchains, *BlockchainDB* assumes that the parties who want to share data are previously authenticated and known to each other. However, parties do not need to trust each other (since they might have contrary goals). More details about our security assumptions will be provided in Section 3.3.

3.2 System Components

Next, we explain how clients interact with *BlockchainDB* as well as the functionality of each component.

Clients: Clients interact with a shared table via their own *BlockchainDB* peer (which they trust). Thus, instead of interacting with a blockchain network directly, in *BlockchainDB* clients interact with their peer (i.e., the database layer) through a simple `put/get` interface to read/write shared data. Furthermore, clients can use the `verify` method of the database layer to trigger an off-chain verification procedure for the online verification of the last read/write-operation of that client. This allows the client to detect potential misbehaving peers in a *BlockchainDB* network (e.g., to detect if another *BlockchainDB* peer — more precisely its storage layer — returned a spurious value for a `get`) and is needed in *BlockchainDB* since not all peers keep a full copy of the database state. So, all reads/writes that are being redirected to a remote storage layer must be additionally verified to get the same guarantees as local reads/writes. We additionally support (a deferred) offline verification procedure that is called from the database layer for batches of reads/writes from all clients. The deferred verification procedure provides a higher throughput than the online verification, since it performs the verification while new reads/writes are being executed. However, clients might work (for a limited amount of time) on an unverified database state. More details about the verification procedure(s) are discussed in Section 6.

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 shard of a table the operation should be directed to. The shard can be either stored in its local storage or remotely in another *BlockchainDB* peer depending on the partitioning scheme of the shared table. 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 shared table. Another major difference of *BlockchainDB* and a pure blockchain network is that the database layer of *BlockchainDB* implements a *Transaction Manager* that provides well-known consistency levels (i.e., eventual consistency and sequential consistency). That way, clients get a defined behavior for concurrent puts/gets without the need to know the internals of blockchains. Additionally, the database layer can re-order/batch puts/gets depending on the chosen consistency level to optimally leverage the underlying blockchain and thus further improve the performance. Details about the database layer are discussed in Section 4.

Storage Layer: The storage layer serves as a persistent, auditable storage backend of *BlockchainDB* and is based on existing blockchain systems. As depicted in Figure 3, the storage layer of *BlockchainDB* is able to parallelize data processing across different shards whereas each shard is implemented as a separate blockchain network where the data in a shard is replicated to multiple (but not necessarily all) peers using the internal blockchain consensus protocols. For example, in Figure 2 the *BlockchainDB* network uses in total three different blockchain networks (one for each shard) with a replication factor of two. Details about the storage layer are discussed in Section 4.

3.3 Trust Assumptions and Threat Model

As mentioned previously, we assume a permissioned setting in which the set of participants is known at the beginning. For simplicity, we assume the set of participants is fixed; extending our techniques to a dynamic set and incorporating more complex consortium rules is orthogonal to our

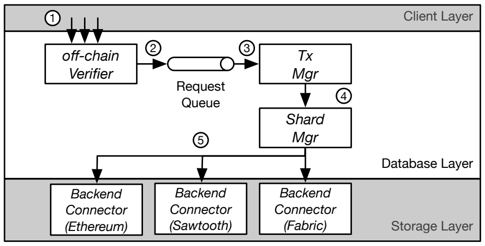


Figure 3: Database and Storage Layer

work. Further, since we use an off-the-shelf blockchain such as Ethereum for storing data, we inherit several security characteristics and guarantees of the underlying blockchain system, e.g., w.r.t. peer authentication using public/private keys, replay protection, number of tolerable malicious nodes.

Moreover, the need for an off-chain verification procedure stems from the fact that a *BlockchainDB* peer might need to read/write data from/to a remote peer; i.e., the local peer does not participate in the blockchain network for that shard. In particular, *Thin Peers* need to run the verification procedure for all read/write operations. For our threat model we thus make the following assumptions:

- Clients that connect to a *BlockchainDB* peer trust their local database and storage layer.
- This allows a *BlockchainDB* peer (i.e., the database layer) to perform verification on behalf of all locally connected clients.
- Moreover, a *BlockchainDB* peer can trust the data that is written to or read from a local shard. Those operations thus do not need to be verified.
- If the majority of the peers that keep a copy of a shard is not malicious, then a client can trust all puts/gets once verified. Thereby, the number of peers that can form a majority depends on the security assumptions of the used blockchain system. For example, some blockchains might require $\frac{2}{3}$ of the nodes to be trusted while others have different properties.

In consequence, whenever a client accesses data that is stored on or written to a remote shard on another peer, the local peer will run an additional verification protocol to verify the operation and mitigate the following attacks:(1) The drop of a `put` operation that needs to write data to a remote peer will be detected. (2) Spurious/fake data returned for any `get` operation that needs to read data from a remote peer will also be detected. Details about the off-chain verification procedure will be explained in Section 6.

4. DATABASE LAYER

In this section, we describe the `put/get` interface of *BlockchainDB* and how these operations are being executed by the database layer to implement different consistency levels on top of blockchains as a storage layer.

4.1 Query Interface

As mentioned before, *BlockchainDB* provides shared tables as main abstraction. Each table has multiple columns (attributes), whereas one is the dedicated primary key that can be used to access the table. More details about the data model and table creation is discussed in Section 5. The query interface of *BlockchainDB* enables clients to execute the following three operations on shared tables:

- `get(t, k)` → `v`: This call returns all attributes of the row in table t that has the key k.
- `put(t, k, v)` → `void`: This call inserts a new row in table t with key k. All attribute values are encoded in v similar to what document stores do. In case a row with key k is already in table t, the row is updated with the new values in v. For simplicity, we assume that values for all attributes are given in v.
- `verify()` → `bool`: This call is for online verification; i.e., a client can call `verify` immediately after any `get/put` call and gets back `true` or `false` to indicate whether or not the verification was successful. For a `put`, it means whether or not the `put` was actually committed in the storage layer and for a `get` it is checked whether or not the returned value was correct or a spurious value.

Next, we explain how `get/put` methods are implemented. The verification (online/offline) is explained in Section 6.

4.2 Query Execution

In the following we explain the execution process and discuss the involved components as depicted in Figure 3.

For accessing the table, clients send put/get requests to their local *BlockchainDB* peer using the query interface discussed before. Requests arriving from the client in the database layer are first received by the off-chain verifier ① that records the unverified reads/writes (i.e., all remote reads/writes) of a client. This information is used for the online/offline verification. The verifier then forwards the request to an internal *RequestQueue* ②. Next, the *TransactionManager* (Tx.Mgr)[1] polls the requests from the queue and processes them according to a specified consistency level[2] ③. Currently, we support sequential and eventual consistency and a version of eventual consistency (called bounded staleness) that guarantees a limited staleness of the accessed data as discussed in the next section.

The different consistency levels differ in how quickly the database layer can process operations. For example, for eventual consistency a get-operation is immediately processed; 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-requests in a global order and thus, potentially blocks a get-request until an outstanding put-request has been committed to the blockchain.

Once a put/get-request 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 requests as read/write operations in parallel to the table shards. To access data in blockchains, different *BackendConnectors* can be used ⑤ that allow *BlockchainDB* to read/write data from/into the underlying blockchain network. The details about the *BackendConnectors* are discussed in Section 5.

4.3 Consistency Levels

As mentioned before, *BlockchainDB* provides well defined consistency levels on top of blockchains. In order to understand how different consistency levels can be implemented

[1] We call this component transaction manager since it translates every put/get of a client into a blockchain transaction.
[2] The consistency level can be specified for each table individually.

on top of blockchains, we first discuss how blockchains make updates visible to clients. Afterwards, we explain how sequential consistency is implemented and how eventual consistency is supported. Moreover, a version of eventual consistency that guarantees a limited staleness is introduced.

Processing Model in Blockchains: In general, blockchain networks (i.e., in our case all data that is stored in one shard) agree on a global order of writes (i.e., blockchain transactions) in all replicas in which they are appended to their log. Thus, a naïve way to implement sequential consistency in the database layer of *BlockchainDB* on top of a blockchain network would be to wait after every write (i.e. put) until the blockchain transaction is committed. However, as shown in [12] the latency until a blockchain transaction is committed can take from seconds to minutes and would severely limit the throughput of write-intensive workloads significantly. Another challenge of blockchains is that some transactions might end up in a fork (e.g., if proof-of-work is used as in Ethereum). These transactions must be re-executed which further increases latency under the blocking execution model discussed before.

Sequential Consistency: In *BlockchainDB*, we thus follow a different approach. Instead of waiting after each write (in an eager manner), we monitor all pending writes in the database layer of *BlockchainDB* to enable lazy waiting. This means, only in case a read operation comes in for a pending write (i.e., read and write share the same key), we wait for that write, and all other pending writes that have been issued before, to be committed to the blockchain. Reads can only be executed once the write is committed to the blockchain. This enables that clients not only read their own writes but defines a global order of writes (for all clients) connected to the same database layer. Moreover, since the blockchain network (which is used to implement a shard table in *BlockchainDB*) orders all writes sequentially, we get a global order of all writes and thus even clients connected to different peers in *BlockchainDB* see the same global order of write operations.

Eventual Consistency and Bounded Staleness: Providing eventual consistency on top of the sequential model of blockchains is simple. Instead of waiting for pending writes, we execute each incoming read operation of a client (i.e., a get) immediately. This, however, could lead to two issues: First, the pending-write queue might grow quickly for write-intensive workloads. Second, for reads (i.e., get operations of clients), *BlockchainDB* might return stale values (without any bound on the staleness) since the time until a blockchain transaction is committed can take up to minutes (as mentioned before). In order to mitigate these issues, a user can define a maximum staleness-factor in *BlockchainDB* (which defines the maximum number of writes in the pending-write queue). That way, applications can control staleness and latency; i.e., with a longer queue the staleness will increase but the latency of reads decreases (as we will also show in our experiments). We call this version of eventual consistency, bounded staleness which is similar to the ideas discussed in [25].

5. STORAGE LAYER

The storage layer of *BlockchainDB* is responsible for all interactions with the blockchain networks that are used to store shared tables. In the following, we first explain the creation of shared tables as well as their data model. Afterwards, we discuss the interface that is exposed by the storage layer to the database layer for executing reads/writes on shared tables. Finally, we exemplary discuss how the methods of the storage interface are implemented in so called backend connectors that allow *BlockchainDB* to access the data in a blockchain. Currently, we provide connectors for Ethereum, Hyperledger Sawtooth and Hyperleder Fabric.

5.1 Shared Tables

As a first step of a data sharing scenario, a new shared table has to be created in *BlockchainDB*. A new table in *BlockchainDB* is defined by its schema and its sharding configuration.

In *BlockchainDB*, the schema of a new table is defined using a key/value data model where the key is the primary key and the value represents the payload of a tuple. The sharding information of a new table contains values for the parameters such as the number of shards or the replication factor and the allocation. These parameters not only have an influence on the overall performance but more importantly on the trust guarantees a new table provides. For example, the number of replicas directly dictates how many malicious peers can be tolerated; e.g., most blockchain networks (such as Ethereum) tolerate if less than half of the peers are malicious.

For creating a new shared table in *BlockchainDB*, one of the clients involved in the data sharing application proposes a new table and submits the information about the table name and its schema as well as the sharding information to its local peer. The local peer then coordinates the table creation process with all other peers on behalf of the initiating client. The main steps of the process are discussed below.

The first step of the table creation process is implemented as a smart contract which takes the information about the new table including the sharding parameters as input and updates the *BlockchainDB* catalog (i.e., its metadata). The metadata of *BlockchainDB* is stored in a dedicated blockchain that is replicated to *all* peers. By storing the metadata in a dedicated blockchain network that is fully replicated and governed by a smart contract, we can not only guarantee that all peers have the same view on the metadata but also that no peer can tamper with the metadata. Fully replicating the metadata is not a performance problem since metadata is typically small and updated less frequently.

As a second step of the table creation process, and once the metadata is updated successfully by the smart contract, the peer which coordinates the table creation process signals all other peers to deploy the shards for the new table. For each new shard that should be stored on a peer, a new blockchain node is started by the peer and connected to the other blockchain nodes, thus forming a new blockchain network for the shard. For finding out which shards need to be deployed for a new table and to which other peers the shard should be connected to, each peer uses its local trusted copy of the metadata.

One important question of the table creation process is, why clients of the other peers should trust the new table. One could think that the table creation process opens up a possible attack since the client and the local peer who coordinates the table creation could be already malicious at the time of table creation and thus could decide to create a new shared table with low trust guarantees that in the extreme case has only one shard consisting of one replica (that might even be assigned to the local peer). In this case, the new table would not provide any trust guarantees to clients of other peers since the peer which created the table stores the only copy and thus could drop puts and return spurious values for get operations without the possibility for the other clients or thin peers to verify their operations.

Thus as a last step of the table creation process, all *BlockchainDB* peers have to confirm that they agree with the table (i.e., in particular the sharding information) proposed by the coordinating peer. The key for the confirmation step is that *BlockchainDB* provides trusted and replicated metadata across peers. That way, all other peers can check whether they agree in a trusted manner with the sharding information before using that table for data sharing. Once the trust guarantees for the new table are confirmed by all peers in *BlockchainDB*, they update the metadata (i.e., by incrementing a confirmation counter). Only once all peers confirmed the new table, it can be used by clients of any peer for actual data sharing by executing put/gets on it.

A last point we want to mention is that *BlockchainDB* assumes a permissioned setup where only authenticated peers can participate in a network. The peers can work together while they do not necessarily need to trust each other.

5.2 Storage Interface

As shown in Figure 3, the database layer uses so called backend connectors in the storage layer to access data in a shared table (i.e., a blockchain network). The idea of the backend connector is to provide a stable interface to the database layer to access the data independent of which blockchain is being used as backend. The main methods of the storage interface are:

- `read(s, k)` → `v`: This method allows the database layer to read a value v (i.e., the tuple) for a given shard s (which is just a global unique identifier in *BlockchainDB*) and a key k.
- `write-async(s, k, v)` → `tx-id`: This method allows the database layer to write a value v (representing a tuple) with a key k into shard s. Important is that the write is an asynchronous operation and just returns an identifier `tx-id` of the blockchain transaction that was created for that write.
- `check-tx-status(s, tx-id))` → `TX-STATUS`:
  In order to check if a `write-async` operation has been successfully committed, this operation can be called. This method takes a shard identifier s and a transaction identifier `tx-id`, and returns the status of the blockchain transaction in that shard. The status can either be `COMMITED` if the write was successful, `ABORTED` if the write failed (e.g., due to failed validation in the blockchain), or `PENDING` if the transaction is submitted but not yet added to a valid block in the blockchain.
- `get-writeset(s, e))` → `ws`: Returns all writes that were executed on shard s in epoch e. This method is used for offline verification, which verifies the workload of all clients connected to one *BlockchainDB* peer in epochs as discussed in Section 6.

The first three methods of the storage interface are called by the database layer in order to implement different consistency levels. For example, under sequential consistency, the `write-async` method is called when a `put(t, k, v)` (for a table t, key k, and value v) from a client is processed by the database layer. The returned transaction identifier `tx-id` is put together with the table t and key k into a pending-write queue in the database layer. If a `get(t, k, v)` operation for the same key is coming in from a client after the `put`-call, the database layer has to check the pending-write queue, and if a pending write is found, it needs to call `check-tx-status(tx-id)` to see whether the transaction committed or not. If not, the database layer has to block until the transaction status changes to `COMMITED`.

5.3 Backend Connector

The methods discussed before need to be implemented by each backend connector for different blockchain systems. In the following, we discuss the implementation of those methods for Ethereum as an example.

It is important to note that the backend connector stores the connection information for each shard identifier and uses a native blockchain client (such as *geth* for Ethereum) to access a blockchain network which stores the data of a shard. The connection information contains the list of IP-addresses and ports of all *BlockchainDB* peers which host a copy of the shard (i.e., the peers which participate in the blockchain network that store the data of the shard). For executing operations, the local IP-address is used if it exists in the connection information (which means that a shard copy is stored on the local peer). Otherwise, one of the remote IP-addresses is selected in a random manner to load balance the execution across different peers.

For accessing shared data in a blockchain network, *BlockchainDB* needs to install a minimal smart contract definition which provides a simple *read/write* interface for each shard. These smart contracts are then called by the connector to implement the interface methods presented before. In the following, we show an extract of the smart contract code installed for an Ethereum network.

```
contract KVContract {
  // state variables and constructor omitted

  // read method in contract
  function read-blockchain(bytes memory key)
  public view returns(bytes memory value){
    return data[key];
  }

  // write method in contract
  function write-blockchain(bytes memory key, bytes memory val)
  public returns(bool success){
    data[key] = val; return true;
  }
}
```

The first method of the backend connector is the `write-async` method. This method takes the incoming tuple (i.e., a key/value pair) as well as the shard identifier s. Afterwards, the connection information is looked up for this shard and the key as well as the value is converted into a byte representation before sending it to the blockchain network for processing (or more precisely to the smart contract of the blockchain as discussed above). The byte-data is then sent to the `write-blockchain` method of the `KVContract` using *geth* as client for Ethereum. We found that representing the data in a byte-format before storing it in a blockchain network not only leads to decreased storage cost, but also allows for more efficient processing of the transaction on the blockchain. The unique transaction identifier `tx-id` returned by *geth* client is returned to the database layer as a

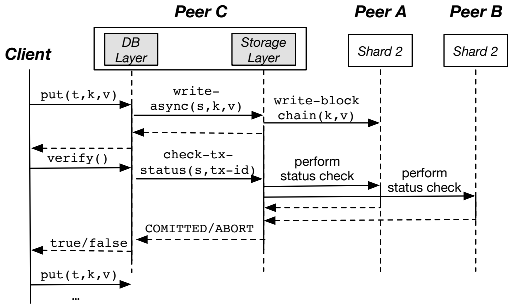


Figure 4: Online Verification

return value. This identifier can be used to check the status of the transaction from the database layer.

The second method implemented in a backend connector is the `read` method. This method takes a shard s and a key k as input. For the execution, the connection information is looked up for this shard and the backend controller then converts the key into a byte representation and sends the data to the `read-blockchain` method of the `KVContract`. Different from the `write-async` method, the backend connector does not use a blockchain transaction to execute the `read-blockchain` contract but it uses a call (which is a read-only operation in Ethereum). The benefit of this method is that it does not require the heavy-weight processing of a blockchain transaction and usually only needs a few milliseconds to be executed. The result of the call is the byte representation of the value (i.e., tuple). Before sending the value back to the database layer, the byte representation is converted into the original data type of the table.

Finally, the third method implemented in a backend connector is the `check-tx-status` method. This method takes a shard s and a transaction identifier `tx-id` as input and returns the transaction status to the database layer. To check the status of a transaction, Ethereum provides different options: First, the storage layer can regularly poll the blockchain for the latest status of the transaction using *geth*. Second, the storage layer can subscribe to events and be notified when, e.g., a new block has been created. When notified, the storage layer can query the blockchain for details of the new block and determine the status of the transaction. However, in order to detect failed/rejected transactions, the storage layer still has to poll the blockchain regularly for the transaction status.

6. OFF-CHAIN VERIFICATION

In this section we describe the details of our off-chain verification protocol. The main goal of the verification procedures is to prevent (1) dropped puts and (2) spurious reads if a peer needs to read/write data from/to a remote shard.

6.1 Online Verification

Overview: In online verification every operation issued by a client is subsequently verified if the client calls the `verify` operation as shown in Figure 4. While all blockchain systems internally make use of Merkle-Trees and similar structures to store data in a verifiable way, only few blockchain systems expose an interface to clients that allows them to retrieve data along with a verifiable proof. Hence, in order to verify the result of an operation, a *BlockchainDB* peer needs to contact the majority of the blockchain network to verify that the retrieved result is valid. In the following, we explain how `get`- and `put`-operations can be verified.

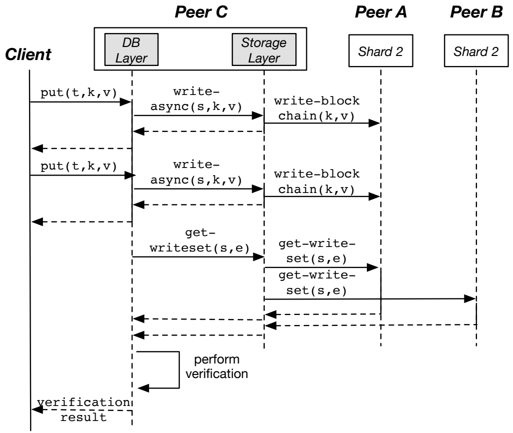


Figure 5: Offline Verification

Verification Procedure: In order to verify the value v returned by a `get`-operation, we extend the storage interface to return the block number from which the value was read for the given shard. Afterwards, we read that block from the majority of peers which hold a copy of the shard and then verify if the value v for key k in those blocks matches the returned value. If the returned value does not match with that of the majority, a manipulated read was detected.

`Put`-Operations are verified differently, since they are executed as transactions on the blockchain as discussed before. Consequently, they are mined as part of blocks. Similar to get operations, `put`-operations can be verified by querying the majority of the blockchain for the latest transaction status. If the transaction is not recorded as committed on the majority of peers, a dropped write was detected. We do not need to check the validity of the block content, since transactions are signed by clients. Thus, a manipulation of the transaction content by a remote peer is not possible.

6.2 Offline Verification

Overview: While online verification guarantees the validity of an operation right away, it has several inefficiencies and drawbacks. First, online verification is a blocking action that prevents any other operation to be executed by a client. Second, since transactions are grouped into blocks and thus mined in batches by blockchains, system throughput can be improved by verifying transactions in batches.

The basic idea of offline verification is shown in Figure 5. Instead of calling the `verify`-method after every `put`- or `get`-call, offline verification defers verification. Deferring verification allows us to batch multiple `put`- and `get`-calls together and verify multiple operations at once instead of separately. In the following, we describe the procedure for offline verification and its main parameters.

Verification Procedure: The offline verification procedure is executed per shard in batches (called epochs). An epoch of a shard in *BlockchainDB* is defined by a fixed number of writes (called epoch-size $|e|$) that can be executed in one epoch in one shard. Once the maximum number of writes is executed in one epoch, the epoch of the shard is closed.

The main idea of offline verification is that all operations that are executed (by one peer) in one epoch are verified

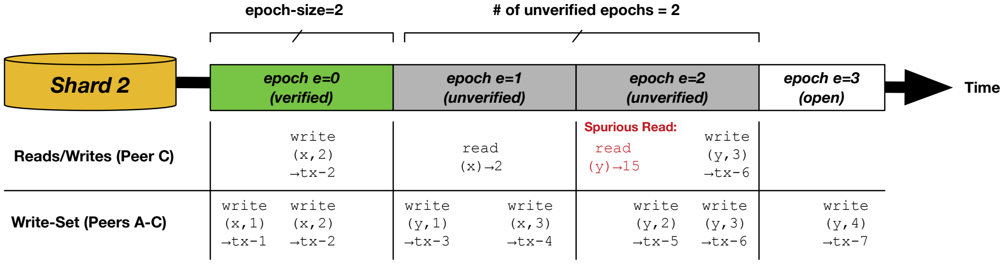


Figure 6: Epoch-based Offline Verification

together in a batch. To do so, we extend the smart contracts shown in the previous section to keep a global counter for the epoch. Moreover, all writes are added in a separate variable that keeps a separate list per epoch. The epoch counter is increased automatically by the `write-blockchain` method in the smart contract every $|e|$-th write call.

One additional parameter of offline verification is that an epoch does not need to be immediately verified by a peer once it is full (i.e., once it is closed). Instead, a peer can have a maximal number of closed but not-yet verified epochs per shard (called number of unverified epochs Δe). In case that the number of unverified epochs for a peer is growing larger than Δe, it calls the `halt(s)` operation implemented as a method in the smart contract on the storage layer which blocks all other peers from further writing into the shard s (i.e., no more new epochs are created) until the peer caught up with verifying the shard. Once the number of unverified epochs is smaller than Δe, the peer calls `continue(s)` implemented as a smart contract on shard s.

One could argue that the `halt(s)` method opens up a backdoor for malicious peers to block other peers. However, since the blockchain will keep track of those calls other peers can detect this behavior (as part of a potential audit). Further, both parameters ($|e|$ and Δe) are important as they allow to optimally tune the verification to the workload characteristics of a given application as discussed later.

In Figure 6, we show an example for offline verification. The example shows the write-sets of all peers over different epochs of one shard (Shard 2), as well as the read/write-set of peer C that should be verified. All epochs have an epoch size of $|e| = 2$ and we have four epochs in total. The first epoch $e = 0$ has been already successfully verified, while epoch $e = 1$ and $e = 2$ are closed but not yet verified (i.e., $\Delta e = 2$). The current epoch $e = 3$ is still open since only one write was executed so far.

For verifying the next epoch in a shard, the database layer of a peer runs a verifier thread that runs in continuous intervals and keeps track of the last verified epoch for that peer (e.g., $e = 0$ in the example). The verifier thread calls the `get-writeset` method for a shard using the last unverified (and closed) epoch as a parameter. In our example, peer C calls `get-writeset(s=2, e=1)` since $e = 1$ is the oldest not-yet verified epoch. The call returns the write-set (`write(y,1)`→ `tx-3`, (`write(x,3)`→ `tx-4`). In order to make sure we have the correct write-set, peer C needs to read it from the majority of peers (not shown in the example), which store that shard. For verifying its own read-/write-set, peer C compares its own read- and write-set of the same epoch ((`read(x)` → 2) against the write-set of epoch $e = 1$ and all previous epochs; i.e., the write-set of epoch $e = 0$ in our example ((`write(x,1)`→ `tx-1`, (`write(x,2)`→ `tx-2`)).

The verifier thread then checks, if the read-calls in the read-set of peer C match the value of the last committed write-operation in the global read-/write-set (to avoid spurious reads) and if all write-calls are found in the global read-/write-set (to avoid dropped writes). In order to enable an efficient offline verification, a system peer caches (parts) of the write-sets that it reads for verification in the past. What exactly needs to be cached depends on the isolation level. Under sequential consistency, only the last write to a key needs to be cached while under eventual consistency all pending writes for a key plus the latest committed write to the same key are cached. Older committed writes can be evicted from the cache.

Finally, if the checks fail, the database layer sets a flag for that shard to indicate that it is in a corrupted state. The application on top then has to decide how to react. One idea is that the client calls the operation `halt(s)` which means that all further operations (from all peers) on the shard are blocked since the database is in a corrupted state and the tamper-proof log of the shard must be audited to see what went wrong. A way to restore the shard to a non-corrupted state is to reset the shard to the last verified epoch. Restoring the database from a blockchain is possible since the blockchain keeps all writes. Discussing possible consequences and other variants for reactions is beyond the scope of this paper though.

Discussion: In the following we first discuss how to set the parameters $|e|$ as well as Δe for offline verification and then discuss what influence the parameters have on the performance that *BlockchainDB* can provide.

For setting $|e|$, we found out that the epoch size should be set at least to the number of transactions that fit into a block of the underlying blockchain (which allows to verify all transaction of a block in one epoch). This information can be retrieved from many blockchain systems and thus be used to configure $|e|$. Setting $|e|$ to a smaller value typically decreases the throughput since new generated blocks can not be completely filled with transactions.

Setting the second parameter, Δe, has a different effect. If $\Delta e = 0$, a peer will not accept any new write-operation once an epoch is closed (i.e., all peers must verify the last epoch first before new writes are accepted). Hence, Δe can be used to overlap the actual writes and verification. Moreover, Δe can be also used for mitigating issues resulting from skew between different peers. For example, a straggling peer of one party could block a faster peer of another party just because it needs more time for executing the offline verification procedure. This is an issue in blockchains where multiple parties that come with different hardware characteristics participate in the network. Thus setting Δe

Table 1: Parameters of Evaluation

Parameter	*Description*
shardCount	Number of shards in a table.
repFactor	Number of replicas that are stored for each shard, each replica is stored on a separate (full) peer.
consistency	Consistency level configured for a shared table.
numPeers	Total number of (full) peers that participate in the *BlockchainDB* network.
numClients	Total number of clients sending put/get operations.
workload	The ratio and distribution of put/get operations.
opsCount	Total number of operations issued in total.

to higher values helps to mitigate the skeweness of different peers in the system.

For understanding the performance impacts of these two parameters, it is important to note that only the verification procedure itself is batched while the actual operations (i.e., the puts/gets of clients) are executed without batching as described in Section 4. To that end, offline verification does not increase the latency of individual put/get-operations but it can have a negative impact on the overall throughput if $|e|$ and Δe are set to a too low value. However, when setting $|e|$ and Δe to a too high value clients will operate for a longer period of time on an *unverified* state of a shared table and the time before a problem (i.e., a dropped put-operation or a spurious get-operation) can be detected increases. Thus, tuning those parameters is extremely important.

In Section 7.5, we show how to set these parameters in an optimal manner for a given workload and setup of peers.

7. EXPERIMENTAL EVALUATION

In order to evaluate the different characteristics of *BlockchainDB*, we executed multiple experiments with the YCSB Benchmark [10] that provides workloads with different read-/write characteristics. The main goal of the experiments is to study the effects of the different techniques implemented in *BlockchainDB* on the performance, and also to show that *BlockchainDB* allows applications with its configuration parameters to trade performance over trust.

7.1 Setup and Workloads

For the evaluation, we implemented a *BlockchainDB* prototype in *Java 8*. All experiments used an Ethereum backend (with Geth/v1.8.23-stable). Furthermore, we ran all experiments in Azure on virtual machines with 16 vcpus and 32 GB memory, running Ubuntu 16.04 LTS. To configure the blockchain network for a new shard in *BlockchainDB*, we used similar genesis parameters as reported in [12].

In every experiment, we varied different system parameters, which are briefly described in Table 1. We will explain the parameters values, we have used for each experiment separately. Furthermore, in our experiments, we first concentrate on the performance characteristics of *BlockchainDB*. To isolate the effect of verification, we turned verification off in these experiments. In the second set of experiments, we then studied the overhead of verification as well as the different verification strategies in detail.

7.2 Exp. 1: Scalability with Peers

In this experiment we evaluate the performance of *BlockchainDB* when the size of the network is increased (i.e., more and more peers join a *BlockchainDB* network to share data). In classical blockchains the performance of the system heavily degrades, since more peers increase the storage and consensus overhead of the network as shown in[12]. We also made a similar observation in our experiments shown in Figure 7 as we discuss below.

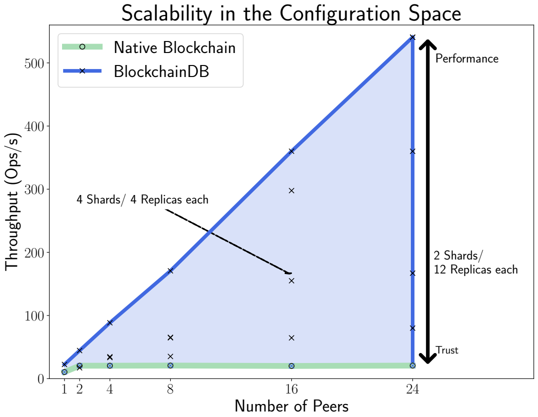


Figure 7: Scalability of *BlockchainDB*

In order to make a fair comparison between different configurations of *BlockchainDB*, we use a fixed number of n peers (i.e., a fixed amount compute and storage resources) and vary the sharding configuration of one table. For example, for a setup with a fixed number of 16 peers (shown as 16 on the x-axis of Figure 7), we evaluated the different configurations shown as individual points above the x-tic 16. The configurations used for 16 peers are: (1 shard with 16 replicas), (2 shards, each 8 replicas), (4 shards, each 4 replicas), ..., (16 shards, each 1 replica). We repeated the experiment for other setups with a lower/higher number of peers (ranging from $1 - 24$). The setups (i.e., number of peers used) are shown as different x-tics/x-labels in Figure 7 and all throughput resulting from using different sharding configurations for one setup are shown as points on the vertical line above the corresponding x-tic.

For running the experiment, we created a shared table with the number of shards and distributed them to the different peers as given in the configurations and inserted into each shard initially $4,000$ tuples. This results in the fact that in total the same number of tuples has to be stored in *BlockchainDB* for the different configurations used for a given number of peers. For example, for a fixed number of 16 peers, $64,000$ tuples are being stored in total in the table for any sharding configuration — when using 16 shards/1 replica on the one extreme but also when using 1 shard/16 replica the other extreme.

In Figure 7, we show the resulting throughput of *BlockchainDB* for the different setups each using a fixed number of peers (x-axis) and for each setup using different configurations (as indicated by the different points along the y-axis) as discussed before. The number of clients used in this experiment is equal to the number of peers (i.e. for every new peer a new client is added; i.e., *numPeers=numClients*). Furthermore, we use *consistency= sequential* and set *opsCount=1000 · numClients* using *workload= 100% write/ 0% read*; i.e., every client sends a total of $1k$ put-operations to the database and waits until these put operations have been committed (e.g. by sending one `get` at the end to force that the writes are committed under sequential consistency). As explained above, the replication factor was set to $repFactor = \frac{numPeers}{shardCount}$.

Figure 7 shows the results of the experiment. Since *BlockchainDB* allows applications to apply different partitioning

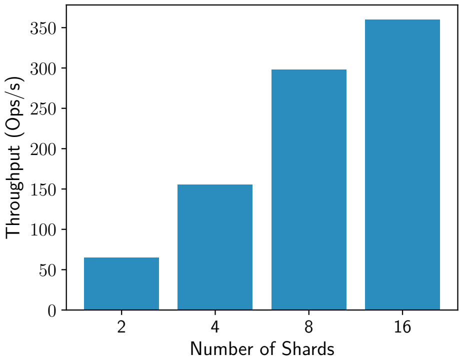


Figure 8: Effect of Sharding on Throughput

and replication strategies for shared tables, we can better trade performance and trust characteristics depending on the applications requirements. The two extremes *high trust* and *high performance* are highlighted by the green and blue line, respectively. For the green line, *BlockchainDB* only uses one shard to store the data and each newly added peer stores yet another replica of this shard and participate in its corresponding blockchain. Hence, this configuration represents a classical blockchain configuration and shows similarly worse scalability. In the other extreme, *high performance*, every peer that is added also adds a new shard to the network. This configuration is comparable to a classical distributed database, in which the parallelism and throughput of the system is increased with every new node. Yet, since only one replica exists per shard, the application does not get any trust guarantees.

An interesting aspect that is also shown in Figure 7 is that *BlockchainDB* can provide other configurations "in the middle" that provide a trade-off between trust and performance (shown as the area shaded in light-blue). For example, when using a configuration with 4 shards and 4 replicas each for 16 peers (shown as one point in the Figure 7) we can provide a 7× speedup over the full-replicated baseline and still provide some trust guarantees.

7.3 Exp. 2: Effect of Sharding

In this experiment, we show the effects of sharding where we fixed the number of peers in the network to 16 and varied the number of shards per table (with a fixed replication factor of 4 per shard). Different from the experiment before, we want to show that sharding provides a speed-up even for a setup with fixed trust guarantees.

For running the experiment, we created a shared table and increased the number of shards from 1 to 16 where we filled each shard initially with 4,000 tuples per shard (i.e., 64,000 tuples are in the table in total for 16 shards) to simulate a setup of *BlockchainDB* with fixed trust-guarantees. Furthermore, we used a constant replication factor of 4 (as mentioned before). The number of clients in this experiment is fixed to *numClients=16*. We used the same workload as in the previous experiment (*workload= 100% write/ 0% read*) but each client sends 2000 operations (i.e. *opsCount=32,000*) using *consistency=sequential*.

Figure 8 shows the result of this experiment. As we can see, the throughput of *BlockchainDB* increases linearly with the number of shards. This is because the degree of overall parallelism is increased.

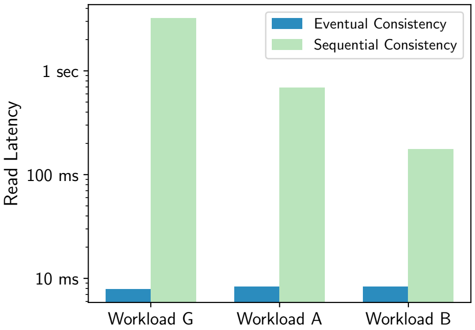


Figure 9: Effect of Consistency on Latency

7.4 Exp. 3: Effect of Consistency Levels

In this experiment, we show the effect of using different consistency levels for clients. For this experiment, we used the parameters shown in table 2.

Table 2: Parameters for Exp. 3

parameter	*value*
shardCount	2
repFactor	2
numPeers	4
numClients	4
opsCount	8000

Furthermore, in this experiment we are using different workload mixes: workload G (new mix/not in YCSB) - a write-intensive workload (*95% write/ 5% read*), workload A (same as in YCSB) - a workload with same amount of reads and writes (*50% write/ 50% read*), and workload B (same as in YCSB) - a read-intensive workload (*5% write/ 95% read*). In order to show the effect of the two main consistency levels (eventual and sequential), we measured the read-latency on a client for the different workloads.

Exp 3a: Sequential vs. Eventual Consistency. Figure 9 shows the resulting latencies for the different workloads. As we can see, the read latency under eventual consistency is not affected by the change in workload at all (which we also expected). For, sequential consistency, however, we can see that the performance increases with a higher number of read-operations. This is a direct consequence of the fact that with a lower number of writes we also have a lower frequency of blocking read-operations. For a read intensive workload, we see a decrease in latency for sequential consistency by two orders of magnitude compared to the write-intensive workload since there are only a few write-operations that could potentially block the execution of subsequent read-operations on the same key.

Exp 3b: Bounded Staleness. While sequential consistency guarantees a client to see fresh values, it has a much higher latency than eventual consistency since it forces a client to wait until pending writes for a key are committed. In contrast, eventual consistency has as significantly lower latency, but does not provide any guarantee on the staleness of retrieved values. As described in Section 4, eventual consistency with bounded staleness allows a client to trade off staleness for improved read latency.

We therefore repeated the experiment above but used bounded staleness. In this experiment, we set the size of the write-queue to different values ranging from 0 to 900.

Table 3: Parameters for Exp. 4

parameter	*value*
shardCount	2
repFactor	2
numPeers	4
numClients	4
consistency	sequential
opsCount	4000

The resulting effects on latency are shown in Figure 10. We see that the average read latency decreases as we increase the staleness. For example, while waiting until *all* pending transactions have been committed (i.e., staleness is 0) causes a maximal delay of about 35s, tolerating around 900 pending puts can improve the latency to around 20s.

In the extreme case, bounded staleness 0 gives us the same guarantees as sequential consistency. However, we see that it results in a higher latency as for sequential consistency as shown in Figure 9 since sequential consistency blocks lazily if a read for a pending write arrives.

7.5 Exp. 4: Verification Overhead

To measure the overhead of verification, we performed two experiments. First, we compared the overhead of online and offline verification. Second, we show the effects of the different parameters for offline verification.

Exp. 4a: Online vs. Offline Verification. In this experiment we evaluated the performance of the different verification strategies supported by *BlockchainDB*. Table 3 indicates the parameters used in this experiment.

For showing the overhead of verification, we report the throughput based on the time it takes to commit and verify all $4,000$ operations. As a baseline, we show a configuration without any verification (called no-verification).

As can be seen in Figure 11, *online verification* achieves the lowest throughput, since it directly waits until a put-operation (i.e., a transaction executing a put) is committed to verify the operation, which prevents other put requests from being executed. With the help of offline verification the throughput can be increased as more transactions are added into one block. As explained earlier the overall throughput depends on the two parameters of the offline verification: epoch-size $|e|$ and number of unverified epochs Δe.

In this experiment we use an optimal configuration and set $|e| = 100$ and $\Delta e = 10$ which results in a throughput of about approx. 40 put operations per second. As can be seen this value is similar to the throughput of no-verification. This is because the verification can be performed once a new block is mined for all operations in the block. This is similar to the time it takes to commit a transaction to the DB plus a small overhead for verifying the retrieved read/write-sets (which is negligible since reading the read/write-sets is fast).

Exp. 4b: Offline Verification Parameters. In the second experiment we fixed the verification strategy to *offline verification* and varied the two parameters $|e|$ and Δe.

In a first micro-benchmark, we evaluated how $|e|$ effects the throughput for a client and set $\Delta e = 0$, i.e., only one epoch can be unverified at a time. For running the micro-benchmark we use a table with a single shard to show the effects of setting $|e|$ on an isolated shard. Further, we varied the replication factor (i.e., number of peers a shard is replicated to) to study its influence on the overall throughput.

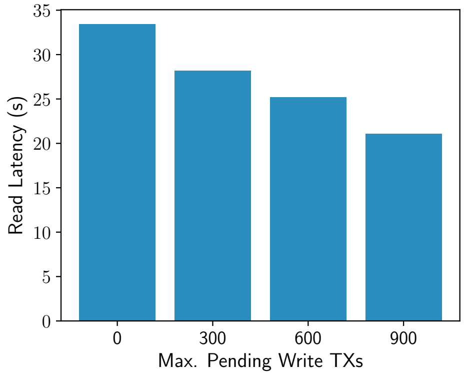


Figure 10: Effect of Staleness on Latency

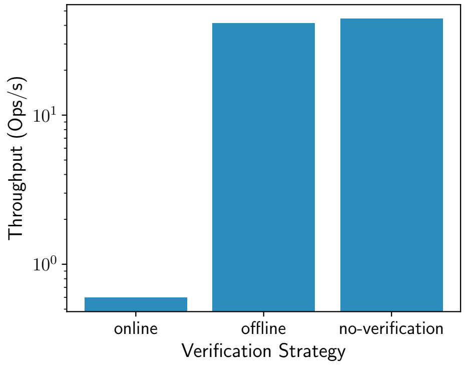


Figure 11: Online vs. Offline Verification

As we can see in Figure 12, when using a replication factor of one (1 Peer, green line) the throughput increases significantly with increasing epoch size until $|e| = 70$ is reached, which corresponds to the maximum number of transactions that fit into one block (i.e., the block size) of the underlying blockchain. When increasing $|e|$ further, the throughput increases much slower (and almost stagnates). Furthermore, if multiple peers participate in a shard the throughput is higher in total. We can see this effect in Figure 12 by the second line (2 Peers, blue line). The reason here is that the verification overhead is distributed across two peers and thus each peer only needs to verify half of the put-operations of an epoch on average. This leads to an overall higher throughput since the total elapsed time for verification is shorter.

In a second micro-benchmark we wanted to show the effect of the Δe parameter. As explained previously, the Δe parameter determines the number of unverified epochs a peer tolerates. In this experiment, we thus execute the same benchmark as before with two peers but this time one fast and one slow peer in order to analyze the effect of potential resource skew (e.g., a slower network or less computational power for one). A *skew* $= x$ means that the slower peer is only able to verify transactions with a lag of x blocks on average behind the faster peer. In order to show the sensitivity of the overall throughput on Δe, we set $|e| = 70$ (i.e., the optimal $|e|$ of the previous experiment) and vary the Δe parameter from 1 to 20.

In Figure 13, we show the throughput (including its standard deviation for 10 runs) for different skew factors where the slower peer has a lag of 4, 8, and 14 blocks on average. As we can see, the throughput improves with increasing Δe whereas for a higher skew a higher Δe is required. As our experiments show, Δe should be set according to the lag of the slower peer; e.g., the maximal throughput for a lag of

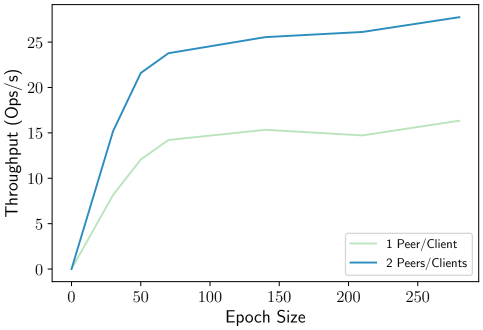


Figure 12: Offline Verification with varying $|e|$

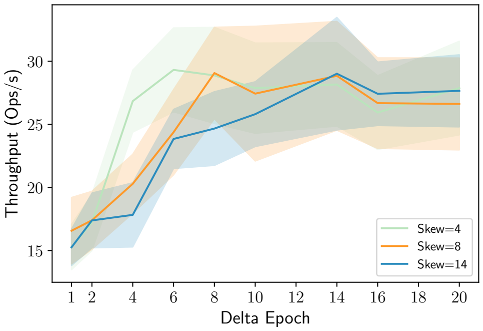


Figure 13: Offline Verification with varying Δe

14 is achieved with $\Delta e = 14$ whereas smaller lags (skew=4 and skew=8) can also tolerate a smaller Δe. In general, we see that if Δe is set to a value smaller than the lag, then the faster peer is always slowed down by the slower peer which decreases the overall throughput.

8. RELATED WORK

Related work spans three major areas, namely, Verifiable Databases, Scalable Blockchains, and Distributed Databases. For Distributed Databases, there is a long line of work that covers relevant topics such as replication, sharding and peer-to-peer approaches. Due to space constraints, we omit a detailed discussion and refer to [21] for an overview.

Verifiable Databases. The closest work to *BlockchainDB* is work done by Allen et al. in [2]. In [2], the authors propose the idea of "Databases and tables that can be shared and verified". While their work makes use of the same abstractions of *shared tables* they differ in how they implement these abstractions. While [2] also uses blockchains to implement a voting/consensus schema they store the actual data in a traditional database on every peer and only a digest in the blockchain. In contrast, in *BlockchainDB* we store all data in blockchains, such that *BlockchainDB* not only uses the consensus and verification features provided by the blockchain but also the capability of having all data in a tamper-proof ledger that allows us to audit all changes to the database. Another major difference is that *BlockchainDB* allows applications to navigate the trade-off between trust and scalability when using blockchains as a shared database.

BlockchainDB also relates to previous work done on verifiable databases in the context of outsourcing [15, 3, 27, 29, 6, 30, 5]. This body of research addresses the question of how to securely delegate the management of data to untrusted third parties, such that the third party cannot manipulate the data or the result of queries on that data. In *BlockchainDB* peers face the same challenges when accessing data on shards stored at remote peers. However, in *BlockchainDB* we do not aim to modify the underlying blockchain and thus build our verification protocol on top.

Scalable Blockchains. The second area of related work is in the context of blockchain systems, where various proposals have been made to improve the scalability and performance of a blockchain. A good overview of the bottlenecks and approaches to scale blockchains are discussed in [11]. In the following, we discuss recent results not covered in [11].

For example, several new protocols have been developed to make use of sharding as part of the blockchain consensus protocol [17, 1, 22, 16, 28, 23] to address the scalability challenge. *BlockchainDB* differs from this line of work as it does not propose a new consensus protocol, but implements sharding on top of existing blockchains. Hence, these new blockchain systems could be also used by *BlockchainDB* as a backend and improve the overall performance of the database. A further difference is that *BlockchainDB* acts as an abstraction layer for clients, such that normal users do not need to deal with new interfaces or programming models that might be introduced by a new blockchain system.

Other proposals to overcome the scalability problem of blockchains discuss the usage of off-chain computation [14, 13, 19, 9]. While *BlockchainDB* shares the concepts of moving certain functions out of the blockchain, it does that on top of the blockchain layer and not as part of it.

Another direction of work is that systems aim to add blockchain-like functions to existing distributed and replicated databases. A prominent representative for this line of work is BigChainDB [18] which builds on MongoDB. While BigChainDB shows that it can provide a higher performance than native blockchain systems, it is being constantly under critique to not provide the same trust guarantees and fault-tolerance model as native blockchains. Some of the original shortcomings have been recently addressed in a newer version by using Tendermint to achieve Byzantine fault tolerance. Different from BigChainDB, *BlockchainDB* hence has chosen another route and instead builds directly on top of blockchain systems and their trusted execution model.

Finally, recent papers [24] also looked into blockchains with a database angle and add database techniques into the blockchain (e.g., re-ordering transactions for higher throughput). Same as before, *BlockchainDB* does all its optimizations on top of the blockchain layer and not as part of it.

9. CONCLUSION AND FUTURE WORK

In this paper, we presented *BlockchainDB*, which introduces a database layer on top of blockchains to participate in data sharing scenarios. Our experiments show that *BlockchainDB* can provide up to two-orders of magnitude higher throughput than native blockchains and allows to better scale-out with the number of peers. At the moment, *BlockchainDB* only provides a key/value interface on top of shared tables. In future, we thus want to extend the query interface to shared tables to full SQL with verifiable transactional semantics.

10. ACKNOWLEDGEMENTS

This research work has been funded by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity.

11. REFERENCES

[1] M. Al-Bassam et al. Chainspace: A sharded smart contracts platform. In *NDSS 2018*, 2018.

[2] L. Allen et al. Veritas: Shared verifiable databases and tables in the cloud. In *CIDR 2019*, 2019.

[3] A. Arasu et al. Concerto: A high concurrency key-value store with integrity. In *SIGMOD Conference 2017*, pages 251–266, 2017.

[4] A. Azaria et al. MedRec: Using Blockchain for Medical Data Access and Permission Management. In *OBD 2016*, pages 25–30, Aug. 2016.

[5] S. Bajaj and R. Sion. CorrectDB: SQL engine with practical query authentication. *PVLDB*, 6(7):529–540, 2013.

[6] M. Benedikt et al. Verifiable Properties of Database Transactions. *Information and Computation*, 147(1):57–88, Nov. 1998.

[7] C. Berger et al. On Using Blockchains for Safety-Critical Systems. In *SEsCPS 2018*, pages 30–36, May 2018.

[8] V. Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[9] R. Cheng et al. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *arXiv:1804.05141 [cs]*, Apr. 2018.

[10] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[11] K. Croman et al. On scaling decentralized blockchains - (A position paper). In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Revised Selected Papers*, pages 106–125, 2016.

[12] T. T. A. Dinh et al. BLOCKBENCH: A framework for analyzing private blockchains. In *SIGMOD Conference 2017*, pages 1085–1100, 2017.

[13] 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, pages 3–15. Springer International Publishing, 2017.

[14] J. Eberhardt et al. Off-chaining Models and Approaches to Off-chain Computations. In *SERIAL 2018*, pages 7–12. ACM Press, 2018.

[15] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *ICDE 2013*, pages 529–540. IEEE, 2013.

[16] E. Kokoris-Kogias et al. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *SP 2018*, pages 583–598, May 2018.

[17] L. Luu et al. A Secure Sharding Protocol For Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30. ACM, 2016.

[18] T. McConaghy et al. BigchainDB: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[19] C. Molina-Jimenez et al. Implementation of Smart Contracts Using Hybrid Architectures with On and OffBlockchain Components. In *SC2 2018*, pages 83–90, Nov. 2018.

[20] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

[21] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[22] I. Puddu, A. Dmitrienko, and S. Capkun. μchain: How to forget without hard forks. *IACR Cryptology ePrint Archive*, 2017:106, 2017.

[23] Z. Ren et al. Implicit Consensus: Blockchain with Unbounded Throughput. *arXiv:1705.11046 [cs]*, May 2017.

[24] A. Sharma et al. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD Conference 2019*, pages 105–122, 2019.

[25] D. Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, 2013.

[26] S. Underwood. Blockchain Beyond Bitcoin. *Commun. ACM*, 59(11):15–17, Oct. 2016.

[27] J. Wang et al. Verifiable Auditing for Outsourced Database in Cloud Computing. *IEEE Transactions on Computers*, 64(11):3293–3303, Nov. 2015.

[28] M. Zamani et al. RapidChain: Scaling Blockchain via Full Sharding. In *SIGSAC 2018*, CCS '18, pages 931–948. ACM, 2018.

[29] Y. Zhang et al. IntegriDB: Verifiable SQL for Outsourced Databases. In *SIGSAC 2015*, CCS '15, pages 1480–1491. ACM, 2015.

[30] Y. Zhang et al. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *SP 2017*, pages 863–880, May 2017.