The Easiest Way of Turning your Relational Database into a Blockchain — and the Cost of Doing So

Felix Schuhknecht[1], Simon Jörz[2]

Abstract: Blockchain systems essentially consist of two levels: The network level has the responsibility of distributing an ordered stream of transactions to all nodes of the network in exactly the same way, even in the presence of a certain amount of malicious parties (byzantine fault tolerance). On the node level, each node then receives this ordered stream of transactions and executes it within some sort of transaction processing system, typically to alter some kind of state. This clear separation into two levels as well as drastically different application requirements have led to the materialization of the network level in form of so-called blockchain frameworks. While providing all the "blockchain features", these frameworks leave the node level backend flexible or even left to be implemented depending on the specific needs of the application. In the following paper, we present how to integrate a highly versatile transaction processing system, namely a relational DBMS, into such a blockchain framework to power a large variety of use-cases. As framework, we use the popular Tendermint Core, now part of the Ignite/Cosmos eco-system, which can run both public and permissioned networks and combine it with relational DBMSs as the backend. This results in a "relational blockchain", which is able to run deterministic SQL on a fully replicated relational database. Apart from presenting the integration and its pitfalls, we will carefully evaluate the performance implications of such combinations, in particular, the throughput and latency overhead caused by the blockchain layer on top of the DBMS. As a result, we give recommendations on how to run such a systems combination efficiently in practice.

Keywords: Blockchain; Relational Databases; Distributed Query Processing; Tendermint

1 Introduction

In recent years, blockchain systems gained interest in various contexts, as they provide distributed transaction processing in potentially untrusted environments. Whereas the original applications mainly targeted public environments such as crypto currencies [Na09, Et22], blockchain systems have also gained interest in permissioned setups, where independent and potentially distrusting organizations, such as for instance companies trading with each other, want to perform some sort of mutual transaction processing [IB22a, IB22b, Te22a]. While the needs and environments for blockchain systems exist, a major downforce for the application of this technology has always been its hard entry level. Existing blockchain systems are often tailored towards a specific use-case or application domain and therefore are hard to apply for new application types. To deal with this challenge, one of the three following strategies is typically applied: (1) To reinvent the wheel and to engineer a new

[1] Johannes Gutenberg University Mainz, Institute of Computer Science, Staudingerweg 9, 55128 Mainz, Germany schuhknecht@uni-mainz.de
[2] sjoerz@students.uni-mainz.de

doi:10.18420/BTW2023-07

blockchain system from scratch, fitting to the specific needs. (2) To carefully adapt an existing blockchain system to the new requirements. (3) To not install a blockchain solution at all. Of course, often, consequence (3) is picked as (1) and (2) are cumbersome and therefore costly.

A step towards solving this problem is the observation that all blockchain systems essentially consist only of two major components. The first component manages the network level. It receives input transactions, orders them globally, and distributes the transaction sequence to each node of the network in exactly the same way. The challenge here lies in performing this in an untrusted environment, where a certain amount of participants might behave maliciously. To guarantee safety and liveness in such an environment, network levels implement sophisticated consensus mechanisms, secure message passing, and tamper-proof transaction logging. Despite various different implementations, the network level is rather independent from the actual application, as the semantics of the transactions are not relevant for this part. The second component manages the node level and centers around the processing of transactions within each node. Naturally, the requirements here are highly application dependent. As a consequence of these observations, blockchain *frameworks* have emerged that try to strictly separate their components by design. The prominent framework Tendermint Core [Te22b], that we will utilize in the following, even leaves the node level backend fully unimplemented. It is up to the application to provide a backend which receives and processes the transactions that are distributed by the framework to each node.

This allows us to easily tackle another typical downside of blockchain systems: an overly simplistic data model and low-level transaction logic. Many prominent systems, like the widely-used Hyperledger Fabric [An18], implement only a key-value model that is accessed via `put()/get()/delete()` calls, from a smart contract containing the transaction logic, often written in a general-purpose programming language like Go [An18]. Of course, this highly complicates the process of transaction writing. To tackle this problem, we want to support the widely-used relational model SQL, by connecting a relational DBMS as backend to the framework. Therefore, we create a "relational blockchain" with minimal effort and are especially interested in the overhead that is caused by this combination. We will investigate the latency and throughput of the relational blockchain under the drastically different synchronous, pseudo-synchronous, and asynchronous communication, each appropriate for different types of applications. Further, we will look at the scaling behavior of the system and discuss important configuration parameters. In summary, we will provide recommendations on how to use such a relational blockchain efficiently in practice.

1.1 Contributions

(1) We present how to integrate a stand-alone single-node relational DBMS into the blockchain framework Tendermint. Our current implementation supports PostgreSQL and MySQL and can easily be extended for further systems. As a result of this combination, we produce a *relational blockchain* that can execute (deterministic) SQL transactions equally across a set of potentially untrusted nodes to modify a fully replicated database.

(2) We evaluate *latency* and *throughput/end-to-end runtime* of the relational blockchain under Smallbank [Sm13] and TPC-C [TP22] transactions. We compare its performance with a standalone execution of the workloads in single-instance and distributed PostgreSQL to identify the overhead that is caused by the blockchain framework.
(3) We evaluate the impact of three different *communication methods*, namely synchronous, pseudo-synchronous, and asynchronous communication. We show that the choice of the communication method has a drastic impact on the performance of the system.
(4) We evaluate the impact of the *relational backends*, namely PostgreSQL and MySQL, under synchronous and asynchronous communication.
(5) We evaluate the *scaling capabilities* of the relational blockchain. Here, we first scale the number of virtual nodes within a physical node, which factors out network latency and resembles the Blockchain-as-a-Service (BaaS) setup. Then, we scale number of physical nodes within and across data-centers, resembling the classical distributed setup, facing network/internet latency.
(6) We provide *practical recommendations* in which situations a relational blockchain yields a good performance – and in which situations it does not. To allow and easy application of our findings, all code, results, scripts and auxiliary material of this paper is available in the repository: https://gitlab.rlp.net/fschuhkn/relational-blockchain

2 Related Work

Before presenting our relational blockchain, let us discuss other work that sits at the intersection of blockchains and database systems.

There exists other interesting work that analyzes and/or builds upon the Tendermint framework. In [Ca21, Bu22], the authors perform an interesting performance analysis of the internal behavior of the framework. In [Am18], the authors analyze correctness and fairness of the system. The findings in these works justify our use and setup of Tendermint: The framework powers hundreds of applications of the Cosmos network, where most networks are tightly coupled with only few nodes. Latency and throughput decreases gracefully with the number of nodes participating in the consensus. Tendermint has also been used before to connect DBMSs as the backend. A prominent example is BigchainDB [Bi22], which uses the document store MongoDB [Mo22] as backend. Apart from Tendermint, there exist other blockchain frameworks. The most prominent representative is clearly Hyperledger Fabric [An18], designed to power permissioned blockchain networks. The modular design is composed of interchangeable components that allow a tuning of the network to the specific needs of the application up to a certain degree. Unfortunately, the system is hardcoded against a key-value model, such that the integration of a relational backend is not possible without deep changes of the system. Another blockchain framework is ChainifyDB [Sc21b], that allows the creation of heterogeneous blockchain networks. Here, heterogeneous means that different relational systems can be used across a single network. The applied processing model still ensures correctness of transaction processing.

Apart from frameworks, many research papers discuss the interconnection and relation of classical DBMSs and blockchain systems and how to combine both worlds. In BlockchainDB [El19b, El19a], a database layer is placed on top of a blockchain layer to combine the proper query interface of a database systems with the replication guarantees of a blockchain. In [Na19], the authors take the other route and extend a relational system, namely PostgreSQL, with a blockchain layer in order to create a blockchain network between multiple PostgreSQL instances. Unfortunately, this project requires a deep modification of PostgreSQL. Another interesting project is Veritas [Ge19]. Therein, the authors propose to extend existing DBMSs with blockchain features in a cloud environment. Apart from architectural works, many projects try to improve the performance of blockchain systems in order to converge towards the performance of traditional (distributed) DBMSs. In Fabric++ [Sh19], several optimization techniques from the database domain are transferred to Fabric in order to speed up processing. Other works try to improve blockchain performance via sharding [Da19] and various low-level optimizations in the transaction processing flow [Go19].

Note that originally, we planned to add a comparison with another comparable blockchain system to this paper to put our system into perspective. Unfortunately, there are very few systems targeting our specific setup and if they target it, they either (a) deeply modify the relational DBMS, (b) their code is not available, (c) have a very different query interface, or (d) run a different execution model providing different guarantees. Consequently, in this paper we focus on an in-depth evaluation of our relational blockchain system.

3 Setting up a Relational Blockchain

In the following section, we will discuss how to integrate a relational DBMS into the Tendermint framework, which we believe is a good template for how blockchain frameworks are reasonably engineered. On the backend side, we will focus on relational DBMSs in this work. However, the general process is applicable to non-relational transaction processing backends in a similar fashion.

3.1 The Blockchain Framework: Tendermint Core

The design goal of the blockchain framework Tendermint Core [Te22b] is to provide essentially all those components that are shared in typical blockchain environments [Di18, Sc21a], but nothing more than that. Precisely, the entire transaction processing backend is left unimplemented and must be provided by the application side. There are two requirements for the backend: (1) The same backend must be used within all nodes of the network. (2) This backend must be deterministic, i.e., it executes a block of transactions in the same way on all nodes.

The most essential components that are already provided by Tendermint Core are:
(1) A *transaction pool* which has the responsibility to receive and hold transactions that are pending for ordering and execution. All submitted input transactions first go into this

pool, where they can be rejected already, if they do not match user-specified criteria, by implementing the function `CheckTx()`. The pool itself is lazily replicated across the nodes, i.e., nodes share pending transactions with other nodes via gossip broadcasting.
(2) A *consensus mechanism* called Polka, which is a variation of the well-known PBFT [CL99] consensus. It can tolerate up to f maliciously behaving parties in a set of $3f + 1$ parties. While the mechanism is tailored towards a permissioned setup, where all participants are known at all times, it can be extended to work in a public environment as well by using a Proof-of-Stake-like approach. As this requires the integration of a currency, we run the default version of the consensus mechanism in a permissioned setup.
(3) The *ledger*, which stores the observed sequence of committed transactions at the granularity of blocks within each node in a tamper-resistant way.
(4) A *message passing system* that ensures a secure communication between individual parties of the network.

On the network level, the workflow of the system essentially looks as follows: First, a client submits a new transaction to the network. The network then stores this transaction in the transaction pool with other pending transactions. A node then picks a set of transactions from the pool and groups them into a block in an ordered way. The block then goes through multiple consensus rounds until it is either globally accepted or globally rejected. If it is rejected, another block will be proposed (potentially by another node) and consensus restarts. However, if the block is accepted, it is distributed to all nodes of the network. Each node that receives a block then appends it to its copy of the ledger and passes the block to the transaction processing backend.

Apart from transaction processing, Tendermint Core also handles the network coordination such as the integration of new nodes to an already established network. A joining node essentially downloads the ledger from another node, verifies its integrity, and executes all blocks and their transactions in the backend to reach the up-to-date state.

3.2 Communicating with the Transaction Processing Backend

The block passing between the framework and the transaction processing backend happens via a so-called *Application Blockchain Interface (ABCI)*. The interface essentially consists only of the four functions `BeginBlock()`, `DeliverTx()`, `EndBlock()`, and `Commit()`, which must be implemented by the backend and which are called by the framework. For every agreed-upon block that is distributed, the core first calls `BeginBlock()` on each node to signal the arrival of a new block to the backend. Then, for each transaction within the block, the core calls `DeliverTx()` sequentially. This function is responsible for the actual processing of the transaction. It also returns whether the execution of a transaction was successful or not. After all transactions have been delivered, the core calls `EndBlock()` to signal that the block is done. Finally, the core calls `Commit()`. This tells the backend that all changes made by the transactions of the block must become real and visible for upcoming processing, if all transactions in the block succeeded. Otherwise, `Commit()` is responsible for rolling back all changes made by all transactions of the block. To implement this ABCI and to connect a backend to Tendermint core, there are two options which we call the *server-variant* and the *builtin-variant*. In the server-variant, the backend implementing the ABCI runs as an independent socket-server and the core calls the interface via TCP. In the builtin-variant, the ABCI is implemented by the backend as a component of Tendermint core and directly compiled into it. While the server-variant offers a higher flexibility, the builtin-variant allows the core to communicate with the backend via simple function calls. In Section 4.2, we will evaluate both variants.

3.3 Integrating a Relational DBMS as Backend

Connecting a relational DBMS to the blockchain framework by implementing the ABCI is fairly natural, as both sides provide transaction semantics. However, to avoid confusion, we now have to clearly differentiate between different types of transactions and different types of commits in our system composition: We will call transactions, that are submitted to the blockchain network as *bc-transactions*. As discussed, multiple bc-transactions can be grouped in a block, which is committed as a whole by the framework. We call this a *bc-commit*. In contrast to that, we refer to transactions that are executed by the relational DBMS as *db-transactions*. The DBMS commits at the granularity of individual db-transactions, which we call *db-commit*.

Before being able to communicate with the relational DBMS from within the ABCI functions, we establish a connection to it in the bootstrapping part of Tendermint Core. To do so, we utilize the drivers `pgx` [pg22] and `go-sql-driver/mysql` [go22a] for PostgreSQL and MySQL, respectively, to open a connection to the DBMS instance. Table 1 now shows the pseudo-code implementation of `BeginBlock()`, `DeliverTx()`, and `Commit()`, where we show only the communication with the DBMS and removed any boilerplate code or error handling. As `EndBlock()` does not involve any DBMS communication, we do not show it.

```
BeginBlock() {
  // start db-transaction
  db-transaction dbTx
    = db.Begin()
  return dbTx
}
```

```
DeliverTx(db-transaction dbTx,
          bc-transaction bsTx) {
  // extract SQL statement
  // from bc-transaction
  sql stmt = DecodeTx(bsTx)
  // execute SQL statement
  // as part of db-transaction
  status s = dbTx.Execute(stmt)
  return s
}
```

```
Commit(db-transaction dbTx,
       status[] s) {
  if(s.Contains(bsTxFailed))
    dbTx.Rollback()
  else
    // perform db-commit
    // (= perform bc-commit)
    dbTx.Commit()
}
```

Tab. 1: Pseudo-code for `BeginBlock()`, `DeliverTx()`, and `Commit()`.

In our implementation, `BeginBlock()` has the sole purpose to begin a new db-transaction. The context `db` is provided by Tendermint and implements a generic interface from the Go package `sql` [Go22b] that allows the communication with relational DBMSs. Relying on a generic interface enables an easy switching between PostgreSQL and MySQL (and other relational systems). Underneath this generic interface, we again use `pgx` respectively `go-sql-driver/mysql` as a compatibility layer. It essentially translates the generic calls to their DBMS-specific counterparts. In each call to `DeliverTx()`, we receive the db-transaction in progress as well as a bc-transaction of the current block. We first decode the received bc-transaction and extract the SQL statement that is stored therein as a string. Then, we pass the SQL statement to the db-transaction context for execution. This execution returns a status (success or failure), which also contains the result of the db-transaction. We return this status to Tendermint Core. After all bc-transactions have been delivered, `Commit()` is called, which receives the open db-transaction and the execution statuses of all bc-transactions of the block. Based on the statuses, we check whether there is a bc-transaction that failed the execution. This could for instance be the case if the SQL statement contained in a bc-transaction is malformed. If a failed bc-transaction exists, we command the DBMS to rollback the db-transaction, including all changes made by bc-transactions of the block. Otherwise, we can safely db-commit the db-transaction, such that all changes of this block become visible for the processing of the next block.

Note that we use the previously described communication protocol only for *modifying* transactions. To answer read-only transactions, we implement the ABCI function `Query()` which allows us to fire read-only queries against the backend of a single node[3], effectively bypassing the costly transaction processing flow of the blockchain framework.

3.4 Synchronous vs Asynchronous Transaction Processing

To process modifying transactions in the blockchain network, the client has essentially two different modes available: (1) A *synchronous mode*, where a client-request blocks until it receives an answer from the system. (2) An *asynchronous mode*, where the request returns before receiving an answer. As we will evaluate both modes in the following, let us discuss their precise realization and behavior in the following.

We start with synchronous processing. First of all, to communicate with Tendermint Core, the client uses a Broadcast API in order to submit bc-transactions. From this API, we utilize the function `BroadcastTxCommit()`. This function basically resembles a synchronous submit that receives a bc-transaction and blocks until either it has been worked into a bc-committed block or it is rejected from the transaction pool (due to being malformed). Consequently, our test suite looks fairly simple for the synchronous case: In each iteration of the loop, a client fires a bc-transaction using `BroadcastTxCommit()` and waits for the result before proceeding with the next iteration. The asynchronous transaction processing is more complex. Here, we use the weaker API function `BroadcastTxSync()` to communicate with Tendermint Core, which already returns after the bc-transaction has been successfully added to the transaction pool. Thus, the client does not get a synchronous response on whether the transaction was committed successfully in a block or not. As we still require a reliable feedback on the success of execution, we implement a test suite as depicted in Figure 1.

[3] This can be extended to query multiple nodes to handle the risk of querying a malicious node.

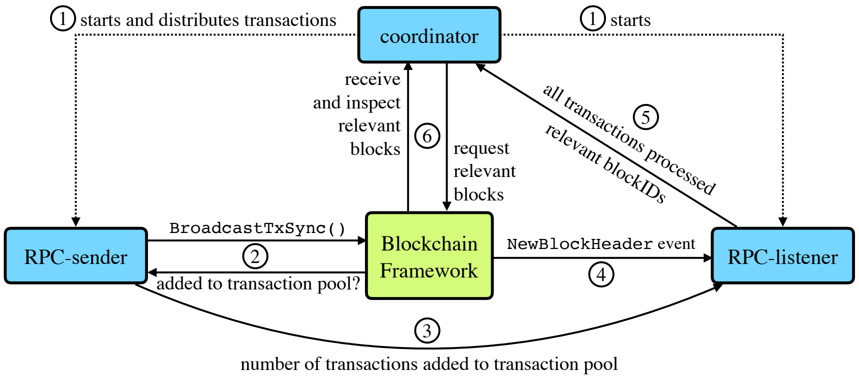

① starts and distributes transactions
coordinator
① starts
receive and inspect relevant blocks
⑥
request relevant blocks
all transactions processed
⑤
relevant blockIDs
RPC-sender
BroadcastTxSync()
②
added to transaction pool?
Blockchain Framework
NewBlockHeader event
④
RPC-listener
③
number of transactions added to transaction pool

Fig. 1: Workflow of asynchronous transactions processing.

It consists of three components: (a) The *coordinator*, responsible for orchestrating the entire run. (b) The *RPC-sender*, which broadcasts the bc-transactions to the framework. (c) The *RPC-listener*, which listens for bc-committed blocks. In ①, the main loop first starts both RPC-sender and RPC-listener. Then, in ②, the RPC-sender uses the aforementioned `BroadcastTxSync()` to push bc-transactions into the network. While doing so, the RPC-sender monitors the number of bc-transactions that made it into the transaction pool – this is the number of transactions expected to make it through the system. In ③, after submitting all bc-transactions, this number is passed to the RPC-listener. For every block that is bc-committed by the framework, in ④, the RPC-listener receives a `NewBlockHeader` event from the framework and calculates the number of already seen bc-transactions based on it. As soon as it has seen all previously entered bc-transactions, in ⑤, it informs the coordinator that all bc-transactions have now been processed and passes the blockIDs containing these transactions. In ⑥, the coordinator then requests all relevant blocks and checks whether the bc-transactions have been processed successfully. Note that the steps ② and ④ can happen interleaved, i.e., the sender can still push in new bc-transactions while the listener is already receiving headers of committed blocks.

3.5 Deterministic Execution, Error Handling, and Provided Guarantees

As the blockchain framework essentially resembles a fully replicated state machine, it requires the backend to behave deterministically, which we ensure in two steps:
(1) The repetitive calls to the ABCI function `DeliverTx()` by the framework happen sequentially. As we ensure that any communication with the relational DBMS within `DeliverTx()` happens synchronously, all bc-transactions will be executed within a db-transaction in exactly the same order within the relational DBMS of each node. Note that there exist mechanisms [Sc21b] to process transactions concurrently *and* deterministically in the backend of all nodes that could be used here. However, as we will see in the experimental evaluation, the backend-execution is dominated by Tendermint Core, and therefore, such an optimization would not lead to significant performance improvements. Consequently, we

kept the execution sequential. (2) We submit only bc-transactions that contain deterministic SQL statements, similar as done in the related work [El19b, El19a]. This requires stripping SQL from components such as random-number generators, timestamp functions, and the LIMIT-statement.

Orthogonal to that, our framework supports the (optional) computation and comparison of checksums on the state after each block commit to detect any occurred state deviation. However, this check comes with a significant overhead and is currently not practical in performance-critical production systems. Also note that a state deviation would require to be followed by a roll-back of the block that caused the deviation in order to recover. Such a recovery is currently not supported by our system, similar to the situation for other blockchain systems. Instead, a deviating node is excluded from the network. In terms of transactional safety, any communication with the relational DBMS happens through db-transactions. This also holds for read-only queries. Therefore, read-only queries are also guaranteed to see a consistent state (resembling the state after a block commit).

4 Experimental Setup

Before starting with our actual experimental evaluation and analysis, let us discuss the setup in the following. As blockchain systems are used in different setups, we will evaluate different network configurations. First, to completely factor out network latency overhead, we perform a set of experiments on a single powerful machine equipped with an Intel i9-12900K CPU (Alder Lake) running at up to 5.2GHz with 16 cores. This state-of-the-art processor is able to run a set of virtual nodes and simulates a very low latency blockchain network. The machine contains 128GB of DDR4-3200. All database files are located on a 2TB Samsung 980 Pro M2 PCIe 4.0 SSD. As operating system, a 64-bit Arch Linux is installed. Note that such a setup consisting of a single physical node running the blockchain network is not fully artificial nowadays: So called Blockchain-as-a-Service (BaaS) solutions [So22, AEE21] host all virtual nodes of the network in a single data-center, often also on the same physical node. Second, to measure the impact of a distributed setup across the internet, we also perform an additional set of experiments on a network of up to eight AWS EC2 instances (t2.small), which are distributed across the four regions Frankfurt, Ireland, London, and Paris, with up to two instances per region. Each instance has one vCPU, contains 2GB of RAM, has 16GB of gp2 volume attached (general purpose SSD), and runs Ubuntu 20.04. Additionally, in the Frankfurt data-center, we run a separate instance (t2.micro) that serves as the client and orchestrates our runs. Note that for all experiments, each client establishes a single connection that is kept alive and re-used during the benchmark run to keep the communication overhead as low as possible.

4.1 Benchmarks: Smallbank & TPC-C

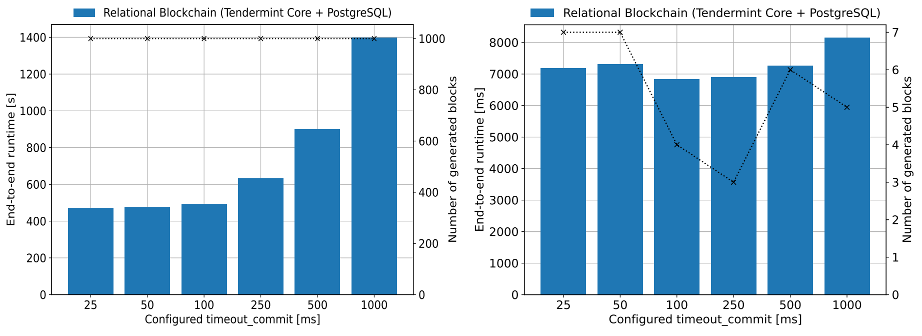
In the following evaluation, we use transactions and datasets from two established transactional benchmarks from the world of blockchains and databases, namely Smallbank [Sm13] and TPC-C [TP22]. We use transactions from these two benchmarks as they offer very different characteristics: While Smallbank contains a set of five extremely simple and short-running transactions which essentially resemble only money transfers between accounts, the three used TPC-C transactions are far more complex and long-running. In the following, we give a brief overview of the used benchmarks.

For Smallbank, the database consists of a single table with four columns, where each tuple contains a user-ID and a name having both a balance for a checking account and a savings account, initialized with random integers. We use the five modifying transactions that are specified in the original benchmark description. The transactions `TransactSavings` and `DepositChecking` each increase the respective account balance. `SendPayment` modifies two checking account balances. `WriteCheck` decreases a checking account balance. Finally, `Amalgamate` moves money from a savings account to the checking account of the same user. For each transaction, we randomly pick the account(s) as well as the amount to modify/move following a uniform distribution. For TPC-C, the database has nine tables in total and essentially represents a multi-warehouse wholesale operation. We implement the two modifying transactions `NewOrder` and `Payment` and select the parameters of each fired transaction randomly within meaningful bounds as specified by the TPC-C benchmark description. Additionally, we implement the read-only transaction `OrderStatus` to test the query-interface of the framework. We selected these three transactions as they are rather complex by accessing all nine tables and by modifying five of them, resulting in more long-running transactions than for Smallbank. The warehouses and districts are accessed by the transactions following a uniform distribution. Note that all Smallbank transactions are transmitted to the system on-the-fly. In contrast to that, the TPC-C transactions are registered in the relational DBMS as stored procedures due to their significantly higher code complexity and size. The transactions of TPC-C then simply contain a call of the corresponding stored procedure.

4.2 Framework and Backend Configuration

For Tendermint Core, we use the latest stable version 0.34 for all experiments. For PostgreSQL, we use version 14.5, for MySQL, we run version 8.0.30. Tendermint Core, PostgreSQL, and MySQL run in Docker containers and are installed from the corresponding Docker images. Tendermint Core as well as the relational DBMS are deeply configurable. To measure the "out-of-the-box" performance, we start with the default configuration and try to tune the systems as little as possible. We state and justify in the following all changes.

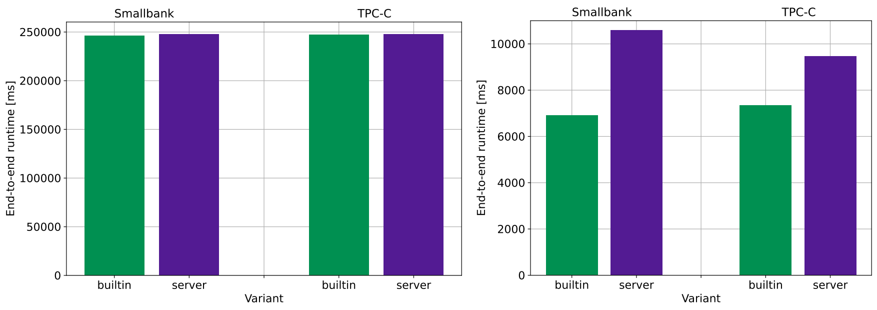
On the side of Tendermint Core, we first increase the size of the transaction pool from 5,000 to 100,000 transactions, such that the whole transaction sequence of each benchmark always fits in. Next, there are multiple timeout parameters that have an impact on both the performance and the behavior of the network. We set `timeout_broadcast_tx_commit` to a sufficiently large value (10s), such that synchronous communication never times out in our experiments. Also, we have to tune the important parameter `timeout_commit`, which determines how long the consensus mechanism does wait for additional votes, if 2/3 of the votes have been received already. To empirically identify a good value, in Figure 2, we perform an experiment where we vary `timeout_commit` from 25ms to 1000ms for both 1,000 synchronous and 10,000 asynchronous transactions of Smallbank. As a value of 100ms yields the best end-to-end runtime in both cases, we use a timeout of 100ms in all upcoming experiments. Further, we set the maximum allowed block size to 21MB such that the size is never the limiting factor for forming a block. We disable the creation of empty blocks (in case the transaction pool runs dry) as well. For PostgreSQL and MySQL, we essentially keep the configuration of the used Docker images as is.



(a) Synchronous communication. (b) Asynchronous communication.

Fig. 2: Varying the `timeout_commit` parameter from 25ms to 1000ms.

As mentioned in Section 3.2, there are two ways of connecting the backend to the framework, where the server-variant is more flexible than the builtin-variant. To identify the performance impact, we implemented both variants and evaluate them against Smallbank and TPC-C transactions. Figure 3 shows the results for both synchronous and asynchronous communication. We can see that for synchronous communication, there is hardly a difference



(a) Synchronous communication. (b) Asynchronous communication.

Fig. 3: Server-variant vs builtin-variant.

visible. The type of connecting the backend is fully overshadowed by the high cost of synchronous communication (which we will evaluate in detail in Section 5.1.1). However, for the cheaper asynchronous communication, the builtin-variant is 1.53x faster for Smallbank and 1.23x faster for TPC-C than the server-variant. This is due to the fact that in the server-variant the ABCI calls happen via TPC sockets, which are significantly more expensive than the direct function calls in the builtin-variant. Due to the higher performance, we use the builtin-variant in all following experiments.

5 Experimental Evaluation & Analysis

In the following, we perform a set of experiments to determine the performance of the relational blockchain. We are particularly interested in its overhead (Section 5.1) over the raw relational DBMS. Then, we perform a cost breakdown to see where the time actually goes (Section 5.2). Next, analyze the impact of the number of clients (Section 5.3) and the relational DBMS (Section 5.4). Then, we investigate the scaling capabilities of the network (Section 5.5). Finally, we analyze the overhead of our relational blockchain over a distributed PostgreSQL cluster (Section 5.6).

5.1 Overhead of the Blockchain Framework

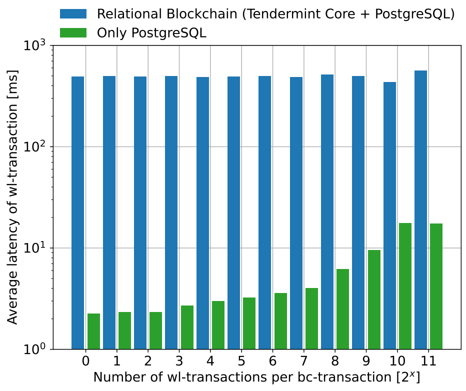
We start our experimental evaluation with the central question of how much overhead the blockchain frameworks actually adds on top of the relational DBMS. We compare the performance of our relational blockchain setup (Tendermint Core + PostgreSQL) as described previously, with the performance of the raw (single-instance) DBMS (PostgreSQL only). Note that the idea of this is not to compare our relational blockchain with PostgreSQL, but to measure the overhead of the blockchain framework over the backend.

As setup we use a fairly typical permissioned configuration: We have a single client firing the bc-transactions into a network of four virtual nodes, where each node consists of a Tendermint Core instance as well as a PostgreSQL instance, each running in its docker container. Again, we use virtual nodes here to factor out any network latency. We test two workloads: (1) The previously described TPC-C workload with 10 warehouses. (2) The Smallbank workload with 100,000 accounts. Note that to distinguish the transactions of the workload from bc-transactions/db-transaction, we will call the former *wl-transactions* in the following. As the type of communication impacts cost and usability, we perform in the following a synchronous as well as an asynchronous variant of the experiment.

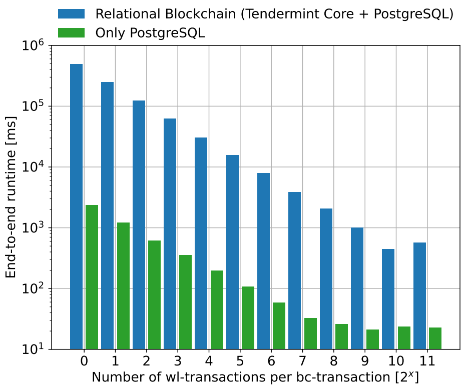
5.1.1 Synchronous and Pseudo-synchronous Communication

We start with synchronous communication. It basically resembles the typical communication with a DBMS: A client submits a transaction to the system and the call blocks until eventually, it returns the result. As we have described in Section 3.4, the blockchain framework supports such a communication style via its Broadcast-API. Additional to this fully synchronous communication, where one wl-transaction is packed into one bc-transaction, we also test a pseudo-synchronous communication style. Therein, we pack multiple wl-transactions into a single bc-transaction and fire this bc-transaction synchronously. On one hand, this results in fewer bc-transactions that have to go through the system, potentially lowering the pressure on the network and the transaction processing overhead. On the other hand, this relaxes our notion of synchronicity (hence pseudo), as the client receives a synchronous response only for a batch of wl-transactions, not for each wl-transaction individually.

To asses the overhead, we are interested in both the *latency* and the *end-to-end runtime*. In this context, latency is the time between submitting a bc-transaction and receiving a response to it. Note that we submit a new bc-transaction only after receiving a response to the previous one. The end-to-end runtime is the time between submitting the first bc-transaction and receiving the response to the last bc-transaction. Figure 4 and Figure 5 show the results for Smallbank and TPC-C, respectively, where we fire a uniform mixture of 1,000 writing wl-transactions in total. On the x-axis, we vary the number of wl-transactions per fired bc-transaction from 1 to 2,048 in logarithmic steps. As discussed, 1 resembles the synchronous case, whereas 2 to 2,048 resemble different pseudo-synchronous configurations. On the y-axis, we show in the Figures 4a and 5a the average latency of a wl-transaction over the whole transaction sequence. In Figures 4b and 5b, we show the end-to-end runtime on the y-axis. To improve readability, we use a logarithmic scale on the y-axis.



(a) Latency for a mixture of writing transactions.

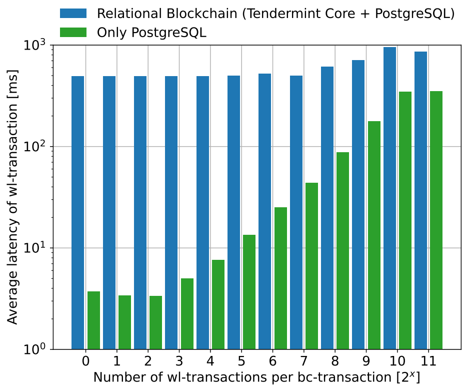


(b) Throughput for a mixture of writing transactions.

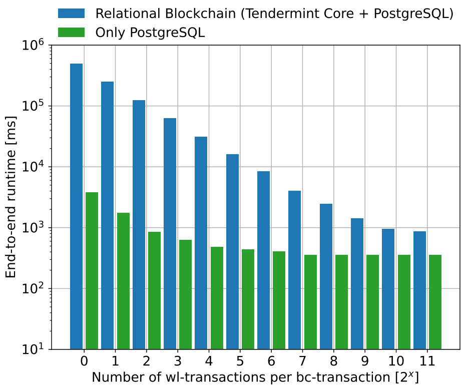
Fig. 4: Synchronous communication (Smallbank)

In the results, we can observe a significant overhead of the blockchain framework over the relational DBMS under both workloads and all synchronicity configurations. However, we can also see that the overhead depends on (a) the type of workload and (b) the number of wl-transactions packed into a single bc-transaction, i.e., the amount of required synchronicity.

Regarding (a), we can see that under Smallbank (Figure 4), the overhead of the blockchain framework over the raw relational DBMS is much more significant than under TPC-C (Figure 5). While for Smallbank, the smallest observed overhead is a still a slowdown of 32x and 25x in latency and end-to-end runtime, respectively, for TPC-C, the latency and runtime overhead decreases to only 2.5x and 2.4x in the best case. The reason for this lies in the complexity and individual runtime of the wl-transactions. As complex TPC-C transactions



(a) Latency for a mixture of writing transactions.



(b) Throughput for a mixture of writing transactions.

Fig. 5: Synchronous communication (TPC-C)

require more processing time in the relational backend than the short-running Smallbank transactions, the overhead of the framework makes a smaller fraction of the total runtime.

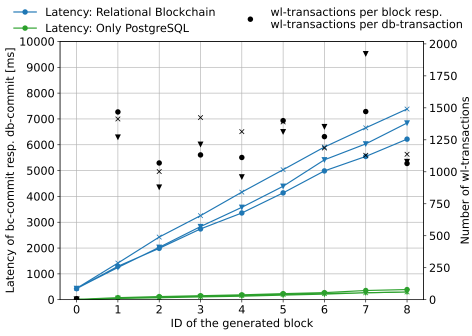
Regarding (b), we observe that packing mutliple wl-transactions into a single bc-transaction heavily impacts the performance, both for the relational blockchain and the raw relational DBMS. While for the fully synchronous case, we measure a devastating overhead of 218x (latency) and 208x (end-to-end runtime) for Smallbank and 133x (latency) and 129x (end-to-end runtime) for TPC-C, the situation gradually improves when relaxing the required synchronicity. Particularly for TPC-C, a reduction in synchronicity has a positive impact on the amount of overhead introduced by the framework, which decreases to the aforementioned acceptable 2.5x (latency) and 2.4x (end-to-end runtime). This is due to the fact that less blocks are formed for the transaction sequence, requiring less consensus rounds, decreasing the central bottleneck of the blockchain framework.

5.1.2 Asynchronous Communication

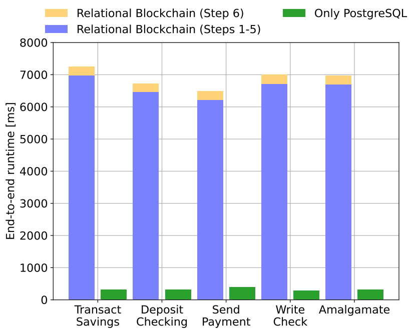
Let us now look at the asynchronous case, which resembles the typical communication style with a blockchain system: The client submits a bc-transaction and the submission returns immediately. Then, after some time, the client checks whether the transaction has been bc-committed or not (yet).

Here, ensuring a fair experimental setup between the relational blockchain and the standalone DBMS is a bit more complicated. The reason lies in the way Tendermint Core handles asynchronous transaction processing internally: As we do not have to wait for a response, we push the whole batch of wl-transactions into the network in one go. Tendermint Core then forms blocks out of pending wl-transactions and commits them one after the other. As previously described, for each block, an individual db-transaction is opened and eventually committed, each containing a sequence of applied wl-transactions. However, as Tendermint now decides by itself how many wl-transactions it packs into a single block, it is more difficult to set up a comparable run for the standalone DBMS. To solve the problem, we record the transactions that were packed in each committed block during the run of the relational blockchain. Then, to set up the run with standalone PostgreSQL, we pack the exact same transaction sequences in individual db-transactions and fire them one by one.

Figure 6 and Figure 7 show the experimental results. As asynchronous communication is less expensive than synchronous communication in total, we fire a larger sequence of 10,000 wl-transactions this time. In the Figures 6a and 7a, we show the measured *latency* of

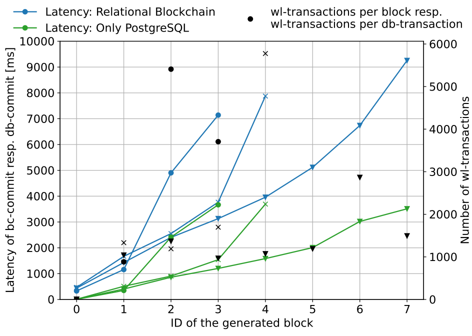


(a) Latency for a mixture of writing transactions.

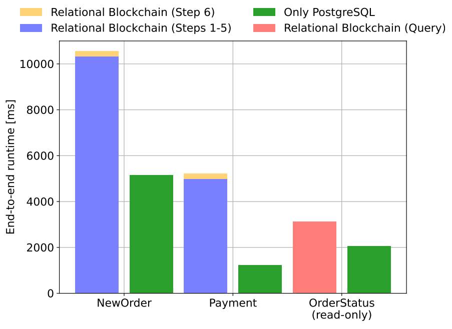


(b) End-to-end runtime.

Fig. 6: Asynchronous communication (Smallbank)



(a) Latency for a mixture of writing transactions.



(b) End-to-end runtime.

Fig. 7: Asynchronous communication (TPC-C)

each block respectively fired db-transaction. Here, we measure latency as the time between the start of the experiment (firing the first wl-transaction) until the notification about the bc-commit of the respective block (step 4 in Figure 1). We fire a uniformly selected mix of only writing transactions of the respective benchmark and plot the ID of each block that has been generated on the *x*-axis in relation to the latency of the corresponding bc-commit on the y-axis. Additionally, we plot the number of wl-transactions that were packed by the framework in each individual block (blacks dots) with respect to a second y-axis. As for individual runs, the framework might produce a different number of blocks, we plot each of the three performed runs individually.

Additionally, in the Figures 6b and 7b, we perform a set of experiments where we measure the *end-to-end runtime*. In this case, we fire 10,000 transactions of each type individually to analyze an effect of the transaction type. For the relational blockchain, we split the end-to-end runtime for the sequence of modifying transactions into the actual transaction processing time (steps 1 to 5 of Figure 1) and the time to check whether the transaction has been processed successfully (step 6 of Figure 1). For the read-only transaction `OrderStatus` of TPC-C, we show the runtime when using the query-interface of the framework.

Let us first look at the results for Smallbank in Figure 6. We can see that the difference in latency and end-to-end runtime between the relational blockchain and stand-alone PostgreSQL is significant. Processing the transactions in the framework increases the latency of the last generated block respectively db-commit by up to 24x and the end-to-end runtime by an average of 21x over all transactions. We see that the result inspection (step 6) is not responsible for the overhead, the actual transaction processing in the framework takes the majority of time. We can also see that the framework packs around $1,000$ to 2,000 wl-transactions in one block, leading to the generation of 8 blocks in total. This clearly improves the performance over the synchronous case, however, still generates significant overhead. Between the individual transaction, we observe little difference. All transactions are extremely short-running in the backend and modify at most two accounts each.

Let us now inspect the TPC-C results in Figure 7, which look quite different to the results of Smallbank. First of all, we can see that the overhead of the framework over stand-alone PostgreSQL is significantly smaller for this benchmark. This time, the framework increases the latency at most by 2.6x. The end-to-end runtime of the sequence of `NewOrder` and `Payment` transactions increases only by 2.0x and 4.3x, respectively. The reason lies in the much higher complexity of the performed transaction: If the backend requires more time to process a transaction, the overhead of processing it in the framework becomes less significant in the end-to-end runtime. For the read-only transaction, the overhead of the framework is even smaller with 1.52x, as we can bypass block forming and consensus entirely. This shows that for queries, the framework should be bypassed entirely. Overall, we can also see that the overhead under asynchronous communication is significantly smaller than in the synchronous case.

5.2 Cost Breakdown

To get a deeper insight on where the overhead originates from, we analyze the produced logfiles of Tendermint Core and isolate four individual phases: (1) The proposal phase, in which pending transactions are grouped in a block to propose. (2) The consensus phase, in which consensus on the proposed block is performed. (3) The execution phase, in which the bc-transactions of the block are executed against the backend. (4) The commit, that marks the state change. Figure 8 shows the life a block under synchronous and asynchronous

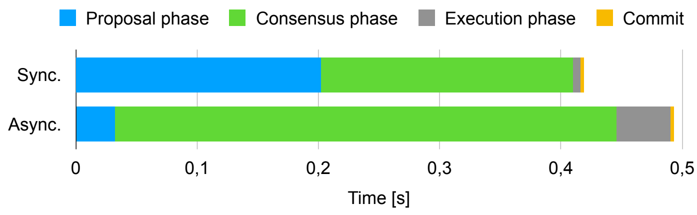
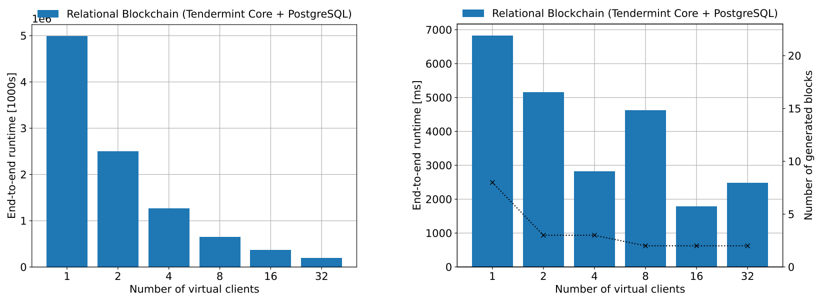


Fig. 8: Cost breakdown into individual phases.

communication. In the synchronous case, it contains one bc-transaction, whereas in the asynchronous case, 1311 bc-transactions are packed in the block. We can see clearly that the consensus phase is by far the dominating phase of the pipeline. In both cases, the actual execution is negligible in comparison. Also, we can observe that the runtime of the proposal and consensus phase varies across runs.

5.3 Impact of the Number of Clients

Let us now inspect the impact of the number of clients on the end-to-end runtime of the system. In Figure 9, we vary the number of clients firing transactions from 1 to 32 in logarithmic steps while keeping the total number of Smallbank transactions fixed to 10,000.



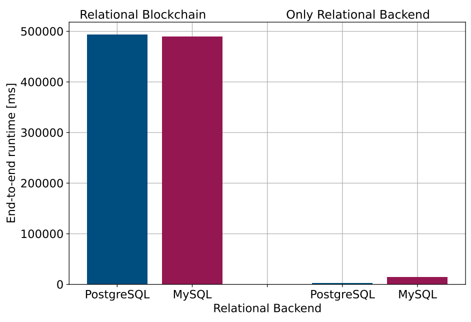
(a) Synchronous communication. (b) Asynchronous communication.

Fig. 9: Impact of the number of clients.

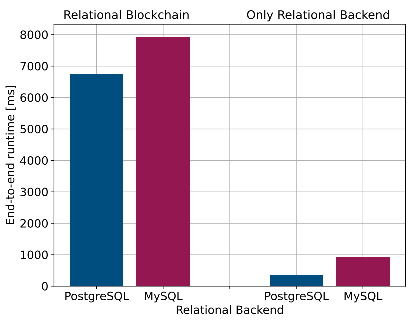
We can see that in the synchronous case, the performance drastically increases with the number of clients. This is the case as concurrently submitted transactions are now packed in the same block. For asynchronous communication, the performance improvement is naturally smaller, but also significant, showing that a single client does not saturate the system.

5.4 Impact of the Relational Backend

So far, we used PostgreSQL as the relational DBMS in the backend for all experiments. Let us now investigate whether the choice of the relational system actually matters or whether its performance is completely overshadowed, if it is part of the blockchain framework. In Figure 10, we show the end-to-end runtime of our relational blockchain under a uniform mixture of 1,000 synchronous respectively 10,000 asynchronous wl-transactions of Smallbank, where we use either PostgreSQL or MySQL as the backend in all four nodes.



(a) Synchronous communication.



(b) Asynchronous communication.

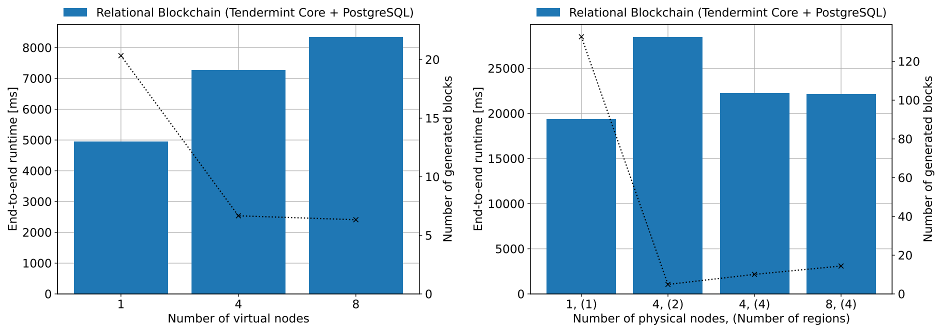
Fig. 10: Impact of the relational DBMS in the backend.

Let us first look at the raw backend performance shown on the right side of the plots. We can see that PostgreSQL is able to process the sequence of transactions significantly faster than MySQL. In the synchronous case, PostgreSQL is 6.6x faster than MySQL. In the asynchronous case, where multiple wl-transactions are packed in a single db-transaction, the speedup is still 2.6x. While the backends perform drastically different, this difference becomes less significant when embedding the backend within the relational blockchain. In the synchronous case, the backend makes no difference at all, as the runtime is dominated by block forming and consensus. Only in the asynchronous case, we see a significant difference. Therein, using PostgreSQL improves the end-to-end runtime by 1.2x over MySQL.

5.5 Impact of Scaling across Virtual Nodes and Physical Nodes

Until now, we ran all experiments using four virtual nodes running on one physical node. In the following, we will vary both the number of virtual nodes (Figure 11a) as well as the number of physical nodes (Figure 11b) to represent the network.

We start by varying the number of virtual nodes in Figure 11a. This experiment still factors out network latency. We set up a network of only one virtual node, four virtual nodes, and eight virtual nodes and report the end-to-end runtime for 10,000 modifying Smallbank transactions using asynchronous communication. Additionally, we show the number of generated blocks for the total run. Note that a network consisting of only one node can skip the consensus phase, as no other participants exist to coordinate with. We see this



(a) Virtual scaling.

(b) Physical scaling.

Fig. 11: Scaling the number of virtual and physical nodes.

setup as the baseline for the throughput that can be achieved in the system. When looking at the results in Figure 11a, we can see that the end-to-end runtime is unsurprisingly the shortest when running only one node. When using four nodes, the runtime increases by a factor of 1.47x over the single node configuration, when using eight nodes, it increases by a factor of 1.69x. This shows that an increase in the number of nodes clearly increases the overhead, however, only sublinearly. When inspecting the number of generated blocks, we can see that for one node, 20 (smaller) blocks are generated on average, whereas for four and for eight nodes, only 6 (larger) blocks are generated for the whole sequence of 10,000 transactions. This shows that the consensus that is performed for a block throttles the forming of the next block.

Let us now look at the results when scaling the number of physical nodes in Figure 11b. Here, we test one physical node in one region (Frankfurt), four physical nodes in two different regions (Frankfurt and Paris), four physical nodes in all four different regions, and eight physical nodes in all four different regions. Note that for this experiment, we repeat each run 10 times (instead of 3 times as before) to factor out variance caused by the cloud provider as much as possible. First of all, we can observe that the end-to-end runtime is overall higher than when scaling the number of virtual nodes within one physical node. This is caused by the internet latency, but also by the slower physical nodes. Again, using only one node is unsurprisingly fastest, however, using more physical nodes does not decrease the performance as heavily as for virtual scaling. Using four physical nodes within two regions shows worse performance than four physical nodes within four regions. We deduct from this that the internet traffic between the nodes is not the bottleneck here, but that the two physical nodes within the same region potentially share the same hardware resources. Going to eight physical nodes decreases the performance only marginally in comparison to four nodes. We also observed a relatively high variance between individual runs as soon execute on a distributed setup. For four physical nodes on two regions, we measured runtimes between 22s and 37s, for four physical nodes across four regions, we observed runtimes between 20s and 25s, and for eight physical nodes, we saw runtimes between 21s and 28s. This indicates that other computations happened on the same instances.

5.6 Comparison with a Distributed Relational DBMS

Let us finally investigate the overhead of our system in comparison with a fully-replicated distributed PostgreSQL cluster using Citus [Cu21] across our four EC2 nodes. We fire our typical set of Smallbank transactions against the coordinator node. This coordinator uses 2PC to synchronize all modifications with the three replicas. In comparison, we install our relational blockchain on the same nodes. Figure 12 shows the results for a varied pseudo-synchronous communication. We can see that for few wl-transactions per bc-transaction, distributed PostgreSQL performs drastically better. However, the fewer bc-transactions are formed, the more the performance of our relational blockchain approaches distributed PostgreSQL.

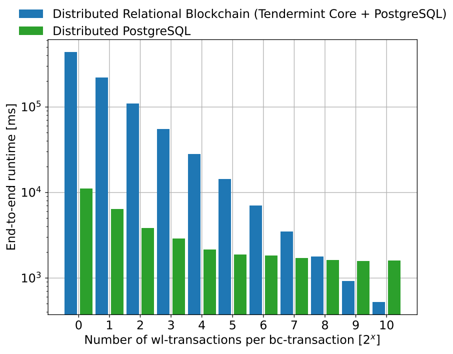


Fig. 12: Comparison of our relational blockchain with a fully-replicated distributed PostgreSQL.

6 Takeways and Conclusion

In this work, we have presented a practical and feasible way of integrating a full-fledged relational DBMSs into a blockchain framework to support the execution of deterministic SQL transactions in byzantine environments. We analyzed the performance implications of such a systems combination and identified situations where the overhead is acceptable. Also, we have seen setups where the overhead is dramatic and completely overshadows the backend performance. In Table 2, we conclude with practical recommendations on how to achieve the best performance of such a setup.

Property	**Recommendation**
Communication	If possible, chose asynchronous communication. Alternatively, chose pseudo-synchronous communication with as many wl-transactions per bc-transaction as acceptable.
Clients	Use several clients to propose bc-transactions (improvement till up to 32 clients), especially under synchronous communication.
Backend	If the workload contains complex transactions and communication is asynchronous, chose a high-performance backend (e.g. PostgreSQL over MySQL). Otherwise, the choice of backend is less important.
Transactions	Fire transactions as read-only transactions if possible to bypass the transaction processing flow of the framework.
Network	Use as few nodes as possible to keep the overhead of the consensus phase and the communication between the nodes as low as possible. Across physical nodes, the system scales better than across virtual nodes.

Tab. 2: Practical recommendations on how to achieve good performance.

Bibliography

[AEE21] Alshurafa, Samer Muneer; Eleyan, Derar; Eleyan, Amna: A survey paper on blockchain as a service platforms. Int. J. High Perform. Comput. Netw., 17(1):8–18, 2021.

[Am18] Amoussou-Guenou, Yackolley; Pozzo, Antonella Del; Potop-Butucaru, Maria; Tucci Piergiovanni, Sara: Correctness and Fairness of Tendermint-core Blockchains. CoRR, abs/1805.08429, 2018.

[An18] Androulaki, Elli; Barger, Artem; Bortnikov, Vita; Cachin, Christian; Christidis, Konstantinos; Caro, Angelo De; Enyeart, David; Ferris, Christopher; Laventman, Gennady; Manevich, Yacov; Muralidharan, Srinivasan; Murthy, Chet; Nguyen, Binh; Sethi, Manish; Singh, Gari; Smith, Keith; Sorniotti, Alessandro; Stathakopoulou, Chrysoula; Vukolic, Marko; Cocco, Sharon Weed; Yellick, Jason: Hyperledger fabric: a distributed operating system for permissioned blockchains. In (Oliveira, Rui; Felber, Pascal; Hu, Y. Charlie, eds): Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018. ACM, pp. 30:1–30:15, 2018.

[Bi22] BigchainDB: https://www.bigchaindb.com/, September 2022.

[Bu22] Buchman, Ethan; Guerraoui, Rachid; Komatovic, Jovan; Milosevic, Zarko; Seredinschi, Dragos-Adrian; Widder, Josef: Revisiting Tendermint: Design Tradeoffs, Accountability, and Practical Use. In: 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Supplemental Volume, Baltimore, MD, USA, June 27-30, 2022. IEEE, pp. 11–14, 2022.

[Ca21] Cason, Daniel; Fynn, Enrique; Milosevic, Nenad; Milosevic, Zarko; Buchman, Ethan; Pedone, Fernando: The design, architecture and performance of the Tendermint Blockchain Network. In: 40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021. IEEE, pp. 23–33, 2021.

[CL99] Castro, Miguel; Liskov, Barbara: Practical Byzantine Fault Tolerance. In (Seltzer, Margo I.; Leach, Paul J., eds): Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999. USENIX Association, pp. 173–186, 1999.

[Cu21] Cubukcu, Umur; Erdogan, Ozgun; Pathak, Sumedh; Sannakkayala, Sudhakar; Slot, Marco: Citus: Distributed PostgreSQL for Data-Intensive Applications. In (Li, Guoliang; Li, Zhanhuai; Idreos, Stratos; Srivastava, Divesh, eds): SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021. ACM, pp. 2490–2502, 2021.

[Da19] Dang, Hung; Dinh, Tien Tuan Anh; Loghin, Dumitrel; Chang, Ee-Chien; Lin, Qian; Ooi, Beng Chin: Towards Scaling Blockchain Systems via Sharding. In (Boncz, Peter A.; Manegold, Stefan; Ailamaki, Anastasia; Deshpande, Amol; Kraska, Tim, eds): Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. ACM, pp. 123–140, 2019.

[Di18] Dinh, Tien Tuan Anh; Liu, Rui; Zhang, Meihui; Chen, Gang; Ooi, Beng Chin; Wang, Ji: Untangling Blockchain: A Data Processing View of Blockchain Systems. IEEE Trans. Knowl. Data Eng., 30(7):1366–1385, 2018.

[El19a] El-Hindi, Muhammad; Binnig, Carsten; Arasu, Arvind; Kossmann, Donald; Ramamurthy, Ravi: BlockchainDB - A Shared Database on Blockchains. Proc. VLDB Endow., 12(11):1597–1609, 2019.

[El19b] El-Hindi, Muhammad; Heyden, Martin; Binnig, Carsten; Ramamurthy, Ravi; Arasu, Arvind; Kossmann, Donald: BlockchainDB - Towards a Shared Database on Blockchains. In (Boncz, Peter A.; Manegold, Stefan; Ailamaki, Anastasia; Deshpande, Amol; Kraska, Tim, eds): Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. ACM, pp. 1905–1908, 2019.

[Et22] Ethereum Yellow Paper, https://gavwood.com/paper.pdf, September 2022.

[Ge19] Gehrke, Johannes; Allen, Lindsay; Antonopoulos, Panagiotis; Arasu, Arvind; Hammer, Joachim; Hunter, James; Kaushik, Raghav; Kossmann, Donald; Ramamurthy, Ravi; Setty, Srinath T. V.; Szymaszek, Jakub; van Renen, Alexander; Lee, Jonathan; Venkatesan, Ramarathnam: Veritas: Shared Verifiable Databases and Tables in the Cloud. In: 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings. www.cidrdb.org, 2019.

[Go19] Gorenflo, Christian; Lee, Stephen; Golab, Lukasz; Keshav, Srinivasan: FastFabric: Scaling Hyperledger Fabric to 20, 000 Transactions per Second. In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019. IEEE, pp. 455–463, 2019.

[go22a] go-sql-driver/mysql - MySQL Driver, https://github.com/Go-SQL-Driver/MySQL/, October 2022.

[Go22b] Golang sql package, https://pkg.go.dev/database/sql, September 2022.

[IB22a] IBM Food Trust: https://www.ibm.com/de-de/blockchain/solutions/food-trust, September 2022.

[IB22b] IBM Health: https://www.ibm.com/de-de/blockchain/industries/healthcare, September 2022.

[Mo22] MongoDB: https://www.mongodb.com/, September 2022.

[Na09] Nakamoto, Satoshi: Bitcoin: A Peer-to-Peer Electronic Cash System. May 2009.

[Na19] Nathan, Senthil; Govindarajan, Chander; Saraf, Adarsh; Sethi, Manish; Jayachandran, Praveen: Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. Proc. VLDB Endow., 12(11):1539–1552, 2019.

[pg22] pgx - PostgreSQL Driver and Toolkit, https://github.com/jackc/pgx, September 2022.

[Sc21a] Schuhknecht, Felix Martin: Talking Blockchains: The Perspective of a Database Researcher. In: 37th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2021, Chania, Greece, April 19-22, 2021. IEEE, pp. 72–75, 2021.

[Sc21b] Schuhknecht, Felix Martin; Sharma, Ankur; Dittrich, Jens; Agrawal, Divya: chainifyDB: How to get rid of your Blockchain and use your DBMS instead. In: 11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings. www.cidrdb.org, 2021.

[Sh19] Sharma, Ankur; Schuhknecht, Felix Martin; Agrawal, Divya; Dittrich, Jens: Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. In (Boncz, Peter A.; Manegold, Stefan; Ailamaki, Anastasia; Deshpande, Amol; Kraska, Tim, eds): Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. ACM, pp. 105–122, 2019.

[Sm13] Smallbank Benchmark, `https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/`, May 2013.

[So22] Song, Jie; Zhang, Pengyi; Alkubati, Mohammed; Bao, Yubin; Yu, Ge: Research advances on blockchain-as-a-service: architectures, applications and challenges. Digit. Commun. Networks, 8(4):466–475, 2022.

[Te22a] Telekom Blockchain Applications: https://dmexco.com/stories/how-deutsche-telekom-uses-blockchain/, September 2022.

[Te22b] Tendermint Core, https://tendermint.com/core/, September 2022.

[TP22] TPC-C Benchmark, https://www.tpc.org/tpcc/, September 2022.