chainifyDB: How to get rid of your Blockchain and use your DBMS instead

Felix Schuhknecht[*]
JGU Mainz
schuhknecht@uni-mainz.de

Ankur Sharma
Saarland Informatics Campus
ankur.sharma@bigdata.uni-saarland.de

Jens Dittrich
Saarland Informatics Campus
jens.dittrich@uni-saarland.de

Divya Agrawal*
Celonis GmbH
dagrawal3@gmail.com

ABSTRACT

Today's permissioned blockchain systems come in a stand-alone fashion and require the users to integrate yet another full-fledged transaction processing system into their already complex data management landscape. This seems odd as blockchains and traditional DBMSs share considerable parts of their processing stack. Therefore, rather than replacing the established infrastructure, we advocate to "*chainify*" existing DBMSs by installing a lightweight blockchain layer on top. Unfortunately, this task is challenging: Users might have different black-box DBMSs in operation, which potentially behave differently. To deal with such setups, we introduce a new processing model called *Whatever-Voting (WV)*, pronounced [weave]. It allows us to create a highly flexible permissioned blockchain layer coined *chainifyDB* that (a) is centered around bullet-proof database technology, (b) can be easily integrated into an existing heterogeneous database landscape, (c) is able to recover deviating organizations, and (d) adds only up to 8.5% of overhead on the underlying database systems while providing an up to 6x higher throughput than Hyperledger Fabric.

1. INTRODUCTION

A blockchain can be defined as an immutable ledger for recording transactions, maintained within a distributed network of mutually untrusting peers. The peers execute a consensus protocol to validate transactions, group them into blocks, and build a hash chain over the blocks. Blockchains may execute arbitrary, programmable transaction logic in the form of smart contracts. A smart contract functions as a trusted distributed application and gains its security from the blockchain and the underlying consensus among the peers. [8]

[*]Work done while being employed at Saarland Informatics Campus.

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

Many descriptions of "blockchain-technology" read as if they describe an unexplored data management planet in a far distant galaxy. To bring things back to earth, in this paper, we will take an old database researcher's attitude: What if all of this could be realized using our good old relational database systems? With this in mind, we can translate the above foreign language into the following: *We form a network of DBMSs, where each DBMS keeps a replica of the database. When executing transactions, we ensure that all databases change in the same way. If a database deviates for whatever reason, we try to recover it. As we build upon established DBMSs, we can easily integrate our layer into an existing IT-landscape and build upon the powerful features of these systems: SQL, the relational model, and high transaction processing performance.*

Feature	DBMS	Blockchain	chainifyDB
Well-integrated in IT-landscape	✓	×	✓
Convenience and accessibility	✓	×	✓
Robustness via recovery	✓	×	✓
High performance	✓	×	✓
Sync. in untrusted environments	×	✓	✓

Table 1: chainifyDB combines the best of both worlds.

Thus, instead of reinventing the wheel, we advocate to simply combine the best of both worlds. Table 1 lists the database features we build upon as well as the blockchain features we add by *chainifying* our good old relational DBMSs.

1.1 Permissioned Blockchain Systems (PBS)

Blockchain systems are required in situations, where multiple organizations, that potentially distrust each other, independently maintain a shared state. This situation is for example present in the medical area: When multiple doctors treat a patient, they independently keep a patient file and record their treatments therein. Without a blockchain system, the following problems can arise in such a situation:

Missing data synchronization. Each of the doctors *has their own local version of the patient data*, without properly synchronizing it with the data of all other doctors in charge. Thus, differences in form of errors or missing data can appear over time.

Missing transaction synchronization. Each of the doctors *treats the patient independently*, without synchronizing his or her intended treatment with the plans of the other doctors. This renders their treatment in the best case inefficient, in the worst case harmful.

A so called *permissioned blockchain system (PBS)* seems to provide exactly what the doctors need: a way of sharing and managing data across a set of known but independent organizations. However, if this is the case, why don't we see permissioned blockchains everywhere? The following numbers show how severe the need is: 86% of treatment errors are actually caused by missing or faulty information in patient files [2]. The cost caused by double treatment and redundant information is estimated at 1950 USD per hospital patient [2]. In total, 210 billion USD per year go to unnecessary or needlessly expensive care [4] in the US.

The reason lies in that although PBSs solve these two problems, they actually introduce at the same time various new problems:

First of all, the typical PBS comes as **yet another standalone system**. As such, it is extremely hard to integrate into an existing data management landscape. E.g. our doctors have a large database of patient files already, which must be migrated to the new blockchain system. Furthermore, PBSs such as Fabric are **neither based on SQL nor do they use a relational engine** underneath. The entire system is designed from the point of view of a distributed system rather from the point of view of a database system. This is very unfortunate as like that many technologies are either solved solved unsatisfactorily or get reinvented on the way. For instance, recent work has shown that Fabric can basically be considered a distributed version of MVCC [23]. Additionally, PBSs such as Hyperledger Fabric **do not provide any recovery mechanism**. This is really unfortunate as this implies that if any node deviates, it implicitly leaves the network forever.

1.2 Contributions

As a consequence of these observations, we introduce chainifyDB, which solves *all* aforementioned problems. We make the following contributions:

1. **Whatever-Voting model (WV)**. Instead of reaching consensus on the order of transactions before the execution, our WV-model reaches consensus on the state change generated by the execution. We do so by using an pluggable voting-based algorithm [18]. This allows us to form a private blockchain network out of potentially different DBMSs. In particular, it allows us to treat these DBMSs in a completely black-box fashion, i.e. we do not have to access or modify the code of the used system in any way.

2. **Synchronized Transactions without a middleman**. chainifyDB keeps local databases, which are maintained by individual organizations, synchronized. Although these organizations potentially do not trust each other, we ensure that transaction processing is reflected equally across all databases in the network. chainifyDB achieves this without introducing a middleman.

3. **Built on established DBMS-Technology**. chainifyDB reuses the already existing execution engine and query language of the underlying DBMS. If the DBMS is a SQL-based relational system, then chainifyDB communicates via SQL as well.

4. **Robustness via recovery**. chainifyDB provides efficient mechanisms to recover from any form of state deviation. Deviation can be caused by crashes and hardware problems, but also by malicious and/or non-agreeing behavior.

5. **Extensive experimental evaluation of chainifyDB**. In comparison with the comparable state-of-the-art permissioned blockchain systems Fabric [8] and Fabric++ [23], chainifyDB achieves an up to $6x$ higher throughput. We show that chainifyDB is able to fully utilize the performance of the underlying database systems and demonstrate its recovery capabilities experimentally.

2. WHATEVER-VOTING MODEL

In summary, we "chainify" existing data management infrastructures. This creates a set of interesting challenges: As mentioned, we have to assume that *different* DBMSs come together in a single *heterogeneous* network. Although these systems all understand SQL, they might execute the very same transaction slightly differently. Thus, we cannot rely on that each organization executes every transaction in exactly the same way. Further, we have to treat the DBMSs as complete black-boxes, which cannot be adapted in any way. Connected to this, we cannot rely on that the underlying storage and execution engine provides features tailored towards our system. This is drastically different to classical PBSs. For example, Fabric requires the computation of read/write-sets as a side-product of transaction execution — something, a black-box DBMS does not provide. Additionally, we have to assume, that *any* component of the system can potentially behave in a malicious way.

To address these challenges, we argue that the consensus-phase must be pushed towards the end of the processing pipeline. Instead of reaching consensus only on which transaction to execute, we advocate to reach consensus on the actual state changes performed by the transactions. By this, we are able to detect any deviating behavior that might have happened during execution. In fact, this enables the organizations to implement *whatever* they want there. The result is a new, highly flexible processing model we call the *Whatever-Voting model* (*WV*). It has the following two phases:

(1) **W**hatever-phase. Each organization does whatever it deems necessary to pass the V-phase later on. As a side-product, each organization produces a digest of its behavior of the W-phase.

(2) **V**oting-phase. We perform a voting round on the digests of the W-phase to check whether agreement can be reached on them with respect to an agreement policy. Only if agreement is reached, the state changes are committed to a ledger by the individual organizations. If an organization is non-agreeing, it can try to recover. If no agreement is reached at all, all organizations can try to recover.

This WV model allows us to design and implement our highly flexible permissioned blockchain system: *chainifyDB*. It supports *different* transaction processing systems across organizations to build a *heterogeneous* blockchain network.

Note that WV is also more powerful than classical 2PC or 3PC-style protocols in the sense that they still assume deterministic behavior of the organizations without looking back at the performed state changes. In WV, we simply do not care about whether organizations claim to be *prepared* for a transaction and what they claim to *commit*. In contrast, we solely look at the *outcome*. In summary, *we measure the outcome rather than the promises*.
Figure 1 presents a visualization of the entire model. Let

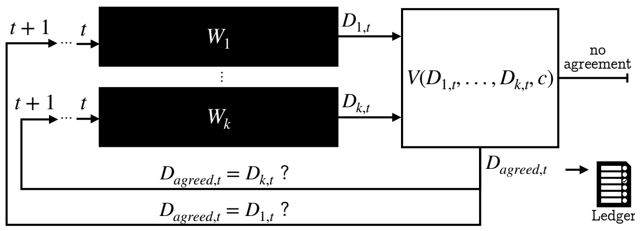

$t+1$
t
W_1
W_k
$D_{1,t}$
$D_{k,t}$
$V(D_{1,t}, \dots, D_{k,t}, c)$
no agreement
$D_{agreed,t}$
Ledger
$D_{agreed,t} = D_{k,t}$?
$D_{agreed,t} = D_{1,t}$?

Figure 1: Round-based processing of the WV model.

$O_1, \dots, O_k$ be k independent organizations in the network. They process one transaction per round. Currently, we are in round t.

By default, our model treats the W-phase as a blackbox and makes no assumptions on its internal behavior. The only requirement is that as a side-product of executing the W-phase W_l of organization O_l in round t, a *digest* $D_{l,t}$ must be produced. The idea of this digest is to externally represent the impact of the changes that happened in the W-phase. For example, the digest could be a cryptographic hash-code of all updates that were performed. The k digests $D_{1,t}$ to $D_{k,t}$ are now passed to the V-phase to check whether an agreement can be reached on them. Our V-phase supports the usage of arbitrary vote-based mechanisms or consensus mechanisms, such as Paxos oder PBFT, depending on the required guarantees. To showcase our system, we use a lightweight vote-based algorithm [18]. We believe this allows for a fair comparison with Fabric, which relies on a trusted ordering service.

Additionally to the digests, an *agreement policy* P is passed to the V-phase. It specifies, *which* organizations are expected to agree on their digests. If the digests are equal for at least one of the combinations $\{C_1, C_2, ...\}$, then agreement is reached. For example, $P = \{\{1,2\},\{1,3\}\}$ specifies that either the organizations O_1 and O_2 or organizations O_1 and O_3 must have the same digest to reach agreement. The logic of the V-phase is shown in Algorithm 1. It essentially tests the passed combinations one by one and tries to reach agreement on one combination.

If no agreement is reached, then no one is allowed to start the next round. In such a case, all organizations try to recover. If agreement on a digest is reached, it is returned as $D_{agreed,t}$. Now, each organization O_l, whose digests equals the agreement digest ($D_{l,t} = D_{agreed,t}$) is allowed to proceed to the next round $t+1$. If the locally computed digest differs from the agreement digest, then the organization must not proceed but can try to recover, as described in the next Section.

Algorithm 1 V-phase

```
 1: procedure V(Digest $D_1$,...,Digest $D_k$, Agreement Policy $P = \{C_1, C_2, ...\}$)
 2:     for each Combination $C$ in $P$ do
 3:         boolean agreement = true
 4:         Digest $D_{cons} = D_{C.p_1}$
 5:         for Digest $D = D_{C.p_2}$ to $D_{C.p_l}$ do
 6:             if $D_{cons} \neq F$ then
 7:                 agreement = false
 8:                 break
 9:         if agreement then return $D_{agreed}$
10:     return no agreement reached
```

2.1 Whatever Recovery

As described, our model allows us to treat the W-phase as a complete blackbox. This is possible as the proceeding is only determined by whether an agreement is reached on the produced digests of the W-phase and not by its internal behavior. However, if we have no information about the internal behavior of the W-phase available, then we have very limited options in recovering from the situation that the system as a whole or an individual organization cannot proceed to the next round. To tackle this problem, we allow the W-phase to optionally expose information about its internal behavior to enable a more sophisticated recovery.

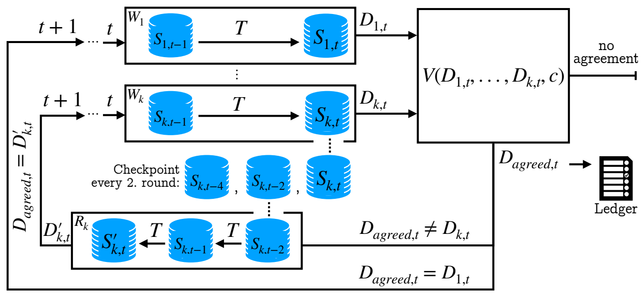


Figure 2: Checkpoint-based recovery of organization O_k. The state is first restored from the most recent checkpoint $S_{k,t-2}$ earlier than t and then replayed till round t, resulting in $S'_{k,t}$. If the new digest $D'_{l,t}$ now matches $D_{agreed,t}$, recovery was successful.

For example, we could have the very valuable information available, that the W-phase maintains and modifies a local database. Thus, the database is capturing the current *state* $S_{l,t}$ of organization O_l after round t. Having access to the state enables recovery via *checkpointing*, as visualized in Figure 2: Let us assume that agreement on the digest $D_{cons,t}$ was reached, but the digest $D_{k,t}$ of organization O_k is different. If organization O_k creates a checkpoint every second round in form of a snapshot of the state, then it can try to recover by replaying the two intermediate transactions on the most recent checkpoint before t, namely $S_{k,t-2}$. This leads to a potentially new state $S'_{k,t}$ with a corresponding new digest $D'_{k,t}$. If now $D'_{k,t} = D_{agreed,t}$, then we recovered successfully.

3. CHAINIFY DB

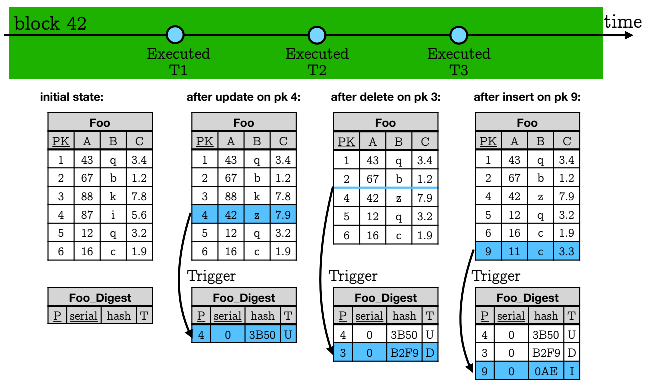
In this Section, we present chainifyDB and how it instantiates the WV model. As described in the introduction, the core idea of chainifyDB is to equip *established* infrastructures, which already consist of several database management systems, with blockchain functionality as a layer on top. The challenge is that these infrastructures can be highly heterogeneous, i.e. every participant potentially runs a *different* DBMS where each system can interpret a certain transaction differently. As a result, the digests across participants might differ. Note that per round, chainifyDB does not process only a single transaction, but a *batch* of transactions. This is a performance optimization in order to reduce the amount of overall network communication.

The W-phase of chainifyDB looks as follows: First, it takes a batch of proposed transactions and orders them in a `TransactionBlock`. Precisely, we want to establish an order that is *globally* valid, i.e. all organizations should receive the same order of transactions. The simplest way of implementing this is to host a dedicated *orderer* for this task. This (untrusted) orderer forms a `TransactionBlock` of proposed transactions, which is then distributed to all other organizations for execution. Of course, a malicious orderer could manipulate transactions or send different blocks to different organizations. However, this is not an issues in our WV-model. In the V-phase, we are able to catch all malicious or deviating behavior, that might have happened earlier in the W-phase. This is in strong contrast to the other models, which has to rely on the orderer being trustworthy or at least byzantine fault tolerant. Each transaction of the `TransactionBlock` is now executed against the local relational database. Obviously, this execution potentially updates the database. As a side-product of execution, we produce the digest $D_{l,t}$, as described in the following.

3.1 From Digests to LedgerBlocks

In chainifyDB we assume SQL-99 compliant relational DBMSs to keep the state at each organization. Thus, we can utilize SQL 99 triggers to realize a vendor-independent digest versioning mechanism, that specifically versions the data of chainifyDB in form of a *digest table*. The digest table is computed per `TransactionBlock`. We instrument every shared table in our system with a trigger mechanism to automatically populate this digest as follows: for every tuple changed by an `INSERT`, `UPDATE`, and `DELETE`-statement, we create a corresponding *digest tuple*. It captures the primary key of the changed tuple, a counter of the change, the `hash` as the digest of the values of the tuple after it was changed (for a delete: its state before removing it from the table), and the type of change that was performed. Figure 3 shows how to process a block of an update, delete, and insert transaction. Although the digest table captures all changes done to a table by the last `TransactionBlock`, it does not represent the digest yet. The actual digest is represented in form of a so called `LedgerBlock`, which consists of the following fields: **(1)** `TA_list`: The list of transactions that were part of this block. Each transaction is represented by its SQL code. **(2)** `TA_successful`: A bitlist flagging the successfully executed transactions. **(3)** `hash_digest`: A hash of the logical contents of the digest table. In our case, this is a SHA256 hash over the hash-values present in the digest table. **(4)** `hash_previous`: A hash value of the entire contents of the previous `LedgerBlock` appended to the ledger, again in form of a SHA256 hash. This backward chaining of hashes allows anyone to verify the integrity of the ledger.

This `LedgerBlock` leaves the W-phase as digest and enters the V-phase to determine whether agreement can be reached. As mentioned before, we use a lightweight voting algorithm [18] in the V-phase of chainifyDB. Other consensus mechanisms, like Paxos or PBFT, could be applied here as well.



initial state:

Foo			
PK	A	B	C
1	43	q	3.4
2	67	b	1.2
3	88	k	7.8
4	87	i	5.6
5	12	q	3.2
6	16	c	1.9

Foo_Digest			
P	serial	hash	T

after update on pk 4:

Foo			
PK	A	B	C
1	43	q	3.4
2	67	b	1.2
3	88	k	7.8
4	42	z	7.9
5	12	q	3.2
6	16	c	1.9

Foo_Digest			
P	serial	hash	T
4	0	3B50	U

after delete on pk 3:

Foo			
PK	A	B	C
1	43	q	3.4
2	67	b	1.2
4	42	z	7.9
5	12	q	3.2
6	16	c	1.9

Foo_Digest			
P	serial	hash	T
4	0	3B50	U
3	0	B2F9	D

after insert on pk 9:

Foo			
PK	A	B	C
1	43	q	3.4
2	67	b	1.2
4	42	z	7.9
5	12	q	3.2
6	16	c	1.9
9	11	c	3.3

Foo_Digest			
P	serial	hash	T
4	0	3B50	U
3	0	B2F9	D
9	0	0AE	I

Figure 3: Logical tuple-wise per block digest computation.

3.2 Logical Checkpointing and Recovery

If, after the V-phase, an organization is non-agreeing or no agreement is reached at all, a checkpointing-based recovery of the organization respectively all organizations happens.

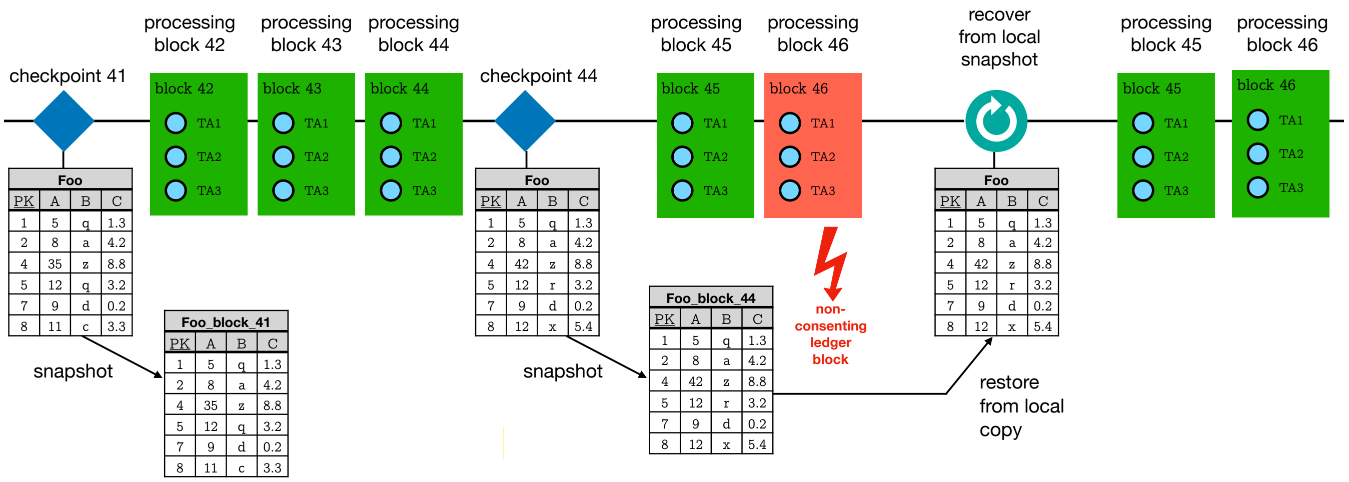
In chainifyDB, we create a checkpoint by creating a snapshot of the database after every k blocks, where k is configurable. The snapshot is created for all tables that were changed since the last consistent snapshot. Snapshots are created on the SQL-level through either a non-maintained materialized view or by a `CREATE TABLE` command. Creating such a snapshot is surprisingly fast: Snapshotting the accounts table with 1,000,000 rows, which is part of the Smallbank [5] benchmark used in our experiment evaluation, only takes 827ms in total on our machine of type 2 (see Section 4) running PostgreSQL.

Figure 4 shows an organization switching to recovery mode, as the digest of block 46 of this organization is non-agreeing. It recovers by restoring the consistent snapshot `Foo_block_44` and by replaying the transaction history, until it reaches a consistent block 46. If it would not be able to recover from this checkpoint, it would try to recover from an older available checkpoint. In general, we go back as many snapshots as are maintained by the system.

3.3 Parallel Transaction Execution

So far, we did not specify how the W-phase is actually running transactions in the underlying DBMS. Naively, we could simply execute all transactions of a block one by one in a sequential fashion. However, this strategy is highly inefficient, if the underlying system is able to execute transactions in parallel. This leads us to an alternative strategy, where we could submit all transactions of a block to the underlying (hopefully parallel) database system in one batch and let it execute them concurrently. While this strategy leverages the performance of the underlying system, it creates another problem: it is very likely that every DBMS schedules the same batch of transactions differently for parallel execution, eventually leading to non-agreement.

The strategy we apply in chainifyDB sits right between



Foo

PK	A	B	C
1	5	q	1.3
2	8	a	4.2
4	35	z	8.8
5	12	q	3.2
7	9	d	0.2
8	11	c	3.3

Foo_block_41

PK	A	B	C
1	5	q	1.3
2	8	a	4.2
4	35	z	8.8
5	12	q	3.2
7	9	d	0.2
8	11	c	3.3

Foo

PK	A	B	C
1	5	q	1.3
2	8	a	4.2
4	42	z	8.8
5	12	r	3.2
7	9	d	0.2
8	12	x	5.4

Foo_block_44

PK	A	B	C
1	5	q	1.3
2	8	a	4.2
4	42	z	8.8
5	12	r	3.2
7	9	d	0.2
8	12	x	5.4

Foo

PK	A	B	C
1	5	q	1.3
2	8	a	4.2
4	42	z	8.8
5	12	r	3.2
7	9	d	0.2
8	12	x	5.4

Figure 4: chainifyDB's recovery using checkpoints. As block 46 is non-agreeing it has to enter the recovery phase. It tries to recover using the most recent local checkpoint. If this would fail, it would try to recover from an older checkpoint.

the previously mentioned strategies and is inspired by the parallel transaction execution proposed in [24] and relates to the ideas of [15, 12, 22]. When a block of transactions is received by the execute-subphase, we first identify all existing conflict dependencies between transactions. This allows us to form mini-batches of transactions, that can be executed safely in parallel, as they have no conflicting dependencies.

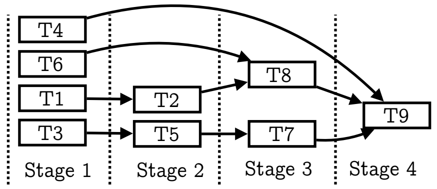


Figure 5: A topological sort of the dependency graph with $k = 9$ transactions yielding four execution stages.

Let us see in detail how it works. The process can be decomposed into three phases: (1) **Semantic Analysis**. First, for a block of transactions, we do a semantic analysis of each transaction. Effectively, this means parsing the SQL statements and extracting the read and write set of each transaction. These read and write sets are realized as intervals on the accessed columns to support capturing both point query and range query accesses. For instance, assume the following two single-statement transactions:

```
T1:UPDATE Foo SET A = 5 WHERE PK = 100;
T2:UPDATE Foo SET A = 8 WHERE PK > 42;
```

The extracted intervals for these transactions are:

```
T1: A is updated where PK is in [100,100]
T2: A is updated where PK is in [42,infinity]
```

(2) **Creating the Dependency Graph**. With the intervals at hand, we can create the dependency graph for the block. For two transactions having a read-write, write-write, or write-read conflict, we add a corresponding edge to the graph. Note that as transactions are inserted into the dependency graph in the execution order given by the block, no cycles can occur in the graph.

Let us extend the example from our Semantic Analysis Phase and let us assume, that T1 has been added to the dependency graph already. By inspecting T2 we can determine that PK[42, inf] overlaps with PK[100,100] of T1. As T2 is an update transaction, there is a conflict between T2 and T1 and add a dependency edge from T1 to T2. Figure 5 presents an example dependency graph for 9 transactions.
(3) **Executing the Dependency Graph**. Finally, we can execute the transactions in parallel. To do so, we perform topological sorting, i.e. we traverse the execution stages of the graph, that are implicitly given by the dependencies. Our example graph in Figure 5 has four stages in total. Within one stage, all transactions can be executed in parallel, as no dependencies exist between those transactions.

The actual parallel execution on the underlying database system is realized using k client connections to the DBMS. To execute the transactions within an execution stage in parallel, the k clients pick transactions from the stage and submit them to the underlying system. As our method is conflict free, it guarantees the generation of serializable schedules. Therefore we can basically switch off concurrency control on the underlying system. This can be done by setting the isolation level of the underlying system to the lowest support level (e.g. READ COMMITTED for PostgreSQL) or if possible completely turned off to get the best performance.

3.4 Running Example

Let us now discuss the concrete system architecture at the running example of Figure 6. chainifyDB consists of three loosely coupled components: **ChainifyServer**, **ExecutionServer**, and **CommitServer**. In **Step 1**, the client creates a `Proposal` from a SQL transaction. This `Proposal` contains the original transaction as a SQL string along with metadata information, such as the client ID and a signature of the transaction. The client then sends the `Proposal` to the ChainifyServer of O_1.

In **Step 2**, the ChainifyServer early aborts all `Proposals`,

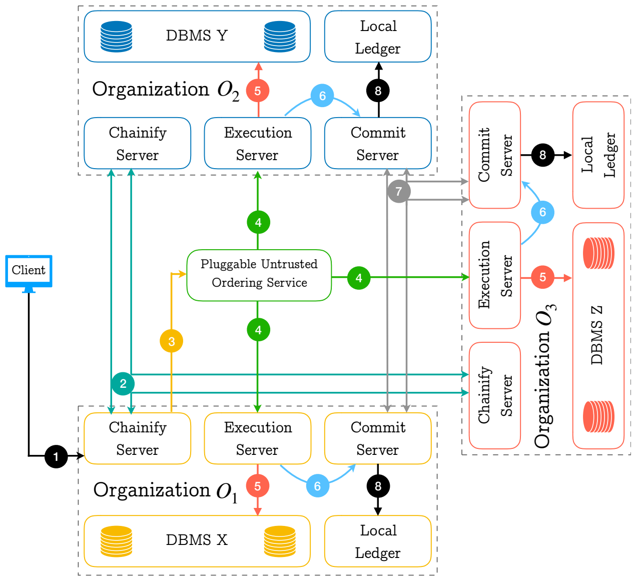


Figure 6: Architecture of a chainifyDB.

which have no chance of reaching agreement later on. To do so, the ChainifyServer of O_1 first accesses the authentication policy of the client, the public key of the client, and the agreement policy. The agreement policy P is in our example that all three organizations have to agree on the proposal. If the ChainifyServer of O_1 successfully authenticates the client and the verification of the `Proposal` is successful, it forwards the `Proposal` to O_2 and O_3 as specified in the agreement policy. Every ChainifyServer in the network now tests, whether the `Proposal` has a chance of reaching agreement later on (e.g. by checking local constraints) and prepares a signed `EarlyAbortResponse`, which contains whether the organization wants to early abort the proposal or not, as well as the `Proposal` itself. The ChainifyServers of O_2 and O_3 then send their `EarlyAbortResponse` to the ChainifyServer of O_1. In **Step 3**, as all three organizations agreed upon the `Proposal`, the ChainifyServer of O_1 creates a `ChainedTransaction` from the original `Proposal` and the three `EarlyAbortResponses`, and sends it to the pluggable and potentially untrusted OrderingService, which can be distributed as well. The OrderingService is then queuing this `ChainedTransaction`. In **Step 4**, when a certain amount of `ChainedTransactions` are queuing, the OrderingService produces a `TransactionBlock` from these `ChainedTransactions`. The dispatch service of the OrderingService then forwards the `TransactionBlock` to every ExecutionServer in the network. In **Step 5**, the ExecutionServer of each organization now computes the near-optimal execution graph and executes the `ChainedTransactions` in the `TransactionBlock` in parallel. As a side-product, the ExecutionServer computes the `LedgerBlock` as digest. In **Step 6**, the ExecutionServer forwards the `LedgerBlock` to the local CommitServer. In **Step 7**, the CommitServers perform the agreement round according to the agreement policy. This involves all three CommitServers. In **Step 8**, they reach agreement, each CommitServer generates the corresponding LedgerBlock and appends it to its ledger.

4. EXPERIMENTAL EVALUATION

To evaluate chainifyDB we use the following setup and workload:

Type 1 (small): Two quad-core Intel Xeon CPU E5-2407 running at 2.2 GHz, equipped with 48GB of DDR3 RAM.

Type 2 (large): Two hexa-core Intel Xeon CPU X5690 running at 3.47 GHz, equipped with 192GB of DDR3 RAM.

Unless stated otherwise, we use a heterogeneous network consisting of three independent organizations O_1, O_2, and O_3. As chainifyDB will be employed within a permissioned environment between a relatively small number of organizations, which know each other, we evaluate its behavior for three organizations. Organization O_1 owns two machines of type 1, where PostgreSQL 11.2 is running on one of these machines. Organization O_2 owns two machines of type 1 as well, but MySQL 8.0.18 is running on one of them. Finally, organization O_3 owns two machines of type 2, where again PostgreSQL is running on one of the machines. The individual components of chainifyDB, as described in Section 3.4, are automatically distributed across the two machines of each organization. Additionally, there is a dedicated machine of type 2 that represents the client firing transactions to chainifyDB as well as one that solely runs the ordering service.

As agreement policy, we configure two different options: In the first option *Any-2*, at least two out of our three organizations have to produce the same digest to reach agreement. In the second option *All-3*, agreement is reached only if all three organizations produce the same digest. Empirical evaluation revealed a block size of 4096 transactions to be a good fit. Of course, we also activate parallel transaction execution as described in Section 3.3.

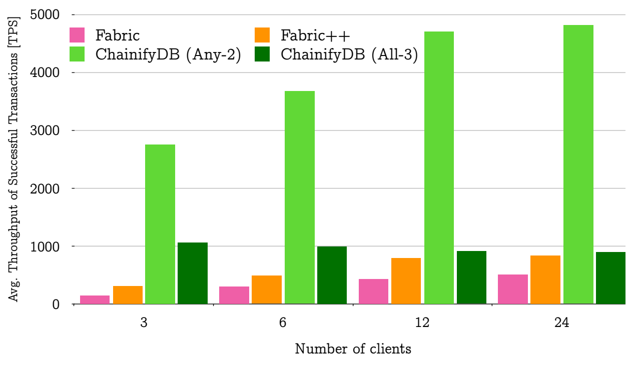
As workload we use transactions from Smallbank [5]. We create 100,000 users with a checking account and a savings account each and initialize them with random balances. The workload consists of the four transactions *TransactSavings*, *DepositChecking*, *SendPayment*, and *WriteCheck*. During a single run, we repeatedly fire these four transactions at a fire rate of 4096 transactions per client, where we uniformly pick one of the transactions in a random fashion. For each picked transaction, we determine the accounts to access based on a Zipfian distribution with a s-value of 1.1 and a v-value of 1.

4.1 Throughput

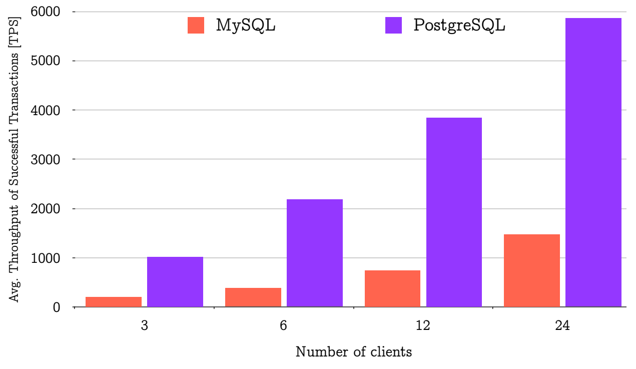
We start the experimental evaluation of ChainifyDB by inspecting the most important metric of a blockchain system: the throughput of successful transactions, that make it through the system.

Therefore, we first inspect the throughput of ChainifyDB in our heterogeneous setup under our two different agreement policies Any-2 and All-3. Additionally to ChainifyDB, we show the following two PBS baselines: (a) Vanilla Fabric [8] v1.2, probably the most prominent PBS system currently. (b) Fabric++ [23], an improved version of Fabric v1.2. Both Fabric and Fabric++ are also distributed across the same set of machines, the blocksize is 1024.

Figure 7(a) shows the results. On the x-axis, we vary the number of clients firing transactions concurrently from 3 clients to 24 clients. On the y-axis, we show the average throughput of successful transactions, excluding a ramp-up phase of the first five seconds. We can observe that ChainifyDB using the Any-2 strategy shows a significantly higher throughput than Fabric++ with up to almost 5000 trans-



(a) Throughput of ChainifyDB with *Any-2* and *All-3* policy for varying number of clients. Additionally, we evaluate Fabric and Fabric++. We use the Smallbank workload following a Zipf distribution.



(b) Throughput of standalone MySQL and PostgreSQL for varying number of clients. The same workload as in Figure 7(a) is fired using OLTP-bench. Note that OLTP-bench follows a uniform distribution.

Figure 7: Throughput of successful transactions for the heterogeneous setup as described in Section 4.

actions per second. In comparison, Fabric++ achieves only around 1000 transactions per second, although it makes considerably more assumptions than ChainifyDB: First, it assumes the ordering service to be trustworthy. Second, it assumes the execution to be deterministic and therefore does not perform any agreement round on the state.

Regarding ChainifyDB, we can also observe that there is a large performance gap between the Any-2 and the All-3 strategy. The reason for this lies in the heterogeneous setup we use. The two organizations running PostgreSQL are able to process the workload significantly faster than the single organization running MySQL. Thus, under the Any-2 strategy, the two organizations using PostgreSQL are able to lead the progress, without having to wait for the significantly slower third organization. Under the All-3 strategy, the progress is throttled by the slowest organizations running MySQL. The difference in processing speed also becomes visible, if we inspect the throughput of the standalone single-instance database systems in Figure 7(b) under the same workload. This time, we fire the transactions using OLTP-bench [11]. Note that both system are configured with a buffer size of 2GB to keep the working set in main memory. As we can see, PostgreSQL significantly outperforms MySQL under this workload independent of the number of clients.

There is one more observation we can make: By comparing Figure 7(a) and Figure 7(b) side-by-side, we can see that ChainifyDB introduces only negligible overhead over the raw database systems. In fact, for 3, 6, and 12 clients, ChainifyDB under the Any-2 policy actually produces a slightly higher throughput than raw PostgreSQL. This is due to our optimized parallel transaction execution, which exploits the batch-wise inflow of transactions, and executes the transaction at the lowest possible isolation level.

We also show in Table 2 the throughput for Smallbank, where the accounts are picked following a uniform distribution. As we can see, the throughput under a uniform distribution is even higher with up to 6144 transactions per second than under the skewed Zipf distribution, as it allows for a higher degree of parallelism during execution due to less conflicts between transactions.

Distribution	3 Clients	6 Clients	12 Clients	24 Clients
Zipf	2757 TPS	3676 TPS	4709 TPS	4812 TPS
Uniform	2279 TPS	3840 TPS	5774 TPS	6144 TPS

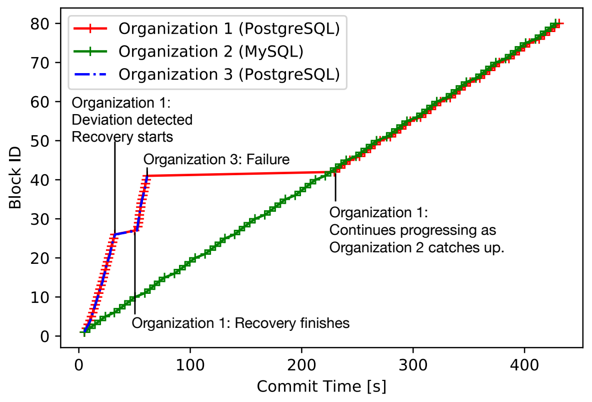
Table 2: Average throughput of successful transactions for ChainifyDB (Any-2) under Smallbank following a Zipf and a uniform distribution.

4.2 Robustness and Recovery

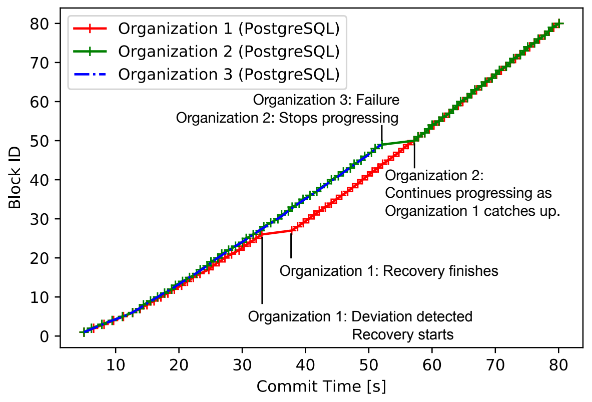
To test the recovery capabilities, we will disturb our chainifyDB network in two different ways: First, we forcefully corrupt the database of one organization and see whether chainifyDB is able to detect and recover from it. Afterwards, we bring down one organization and see whether the network is able to continue progressing.

Precisely, we have the following setup for this experiment: First, we sustain our chainifyDB network with transactions of the Smallbank workload. Then, after a certain amount of time, we manually inject an update to the table of organization O_1 and see how fast O_1 is able to recover from the deviation. Note that we do not perform this update through a chainifyDB transaction, but externally by directly modifying the table in the database. Finally, we simulate a complete failure of one organization by removing it from the network. The remaining two organizations then have to reach agreement to be able to progress under the Any-2 policy.

In Figure 8(a), we visualize the progress of all organizations for our heterogeneous setup. Additionally, in Figure 8(b), we test a homogeneous setup, where all three organizations run PostgreSQL. On the x-axis, we plot the time of commit for each block. On the y-axis, we plot the corresponding block IDs. Every five committed blocks, each organizations creates a local checkpoint. We can observe that the organizations O_1 and O_3, which run PostgreSQL, progress much faster than organization O_2 running MySQL. Shortly after the update has been applied to O_1, it detects the deviation in the agreement round and starts recovery from the most recent checkpoint. This also stops the progression of organization O_3, as O_3 is not able to reach agreement anymore according to the Any-2 policy: O_1 is busy with recovery and O_2 is too far behind. As soon as O_1 re-



(a) Heterogeneous Setup (2x PostgreSQL, 1x MySQL).



(b) Homogeneous Setup (3x PostgreSQL).

Figure 8: Robustness and Recovery of ChainifyDB under the Any-2 agreement policy.

covers, which takes around 17 seconds, O_3 also restarts progressing, as agreement can be reached again. Both O_1 and O_3 progress until we let O_3 fail. Now, O_1 can not progress anymore, as O_3 is not reachable and O_2 still too far behind due its slow database system running underneath. Thus, O_1 halts until O_2 has caught up. As soon as this is the case, both O_1 and O_2 continue progressing at the speed of the slower organization, namely O_2. In Figure 8(b), we retry this experiment on a homogeneous setup, where all organization run PostgreSQL. Thus, this time there is no drastically slower organization throttling the network. Again, at a certain point in time, we externally corrupt the database of organization O_1 by performing an update and O_1 starts to recover from the most recent checkpoint. In contrast to the previous experiment, this time the recovery of O_1 does not negatively influence any other organization: O_2 and O_3 can still reach agreement under the Any-2 policy and continue progressing, as none of the two is behind the other one. Recovery itself takes only around 4 seconds this time and in this case, another organization is ready to perform a agreement round right after recovery. When O_3 fails, O_2 has to halt processing for a short amount of time, as O_1 has to catch up. In summary, this experiment shows (a) that we can detect deviation and recover from it, (b) that the network can progress in the presence of failures, (c) that all organizations respect the agreement policy at all times, and (d) that recover neither penalizes the individual organizations nor the entire network.

4.3 Cost Breakdown

In Section 4.1, we have seen the end-to-end performance of chainifyDB. In the following, we want to analyze how much the individual components contribute to this. Precisely, we want to investigate: (a) The cost of all cryptographic computation, such as signing and validation, that is happening at several stages (see Section 3.4 for details). (b) The impact of parallel transaction execution on the underlying database system (see Section 3.3).
Figure 9 shows the results. We can observe that the overhead caused by cryptography is surprisingly small. Under the Any-2 policy, activating all cryptographic components decreases the throughput only by 7% for parallel execution. Under the All-3 policy, the decrease is only 8.5%. While

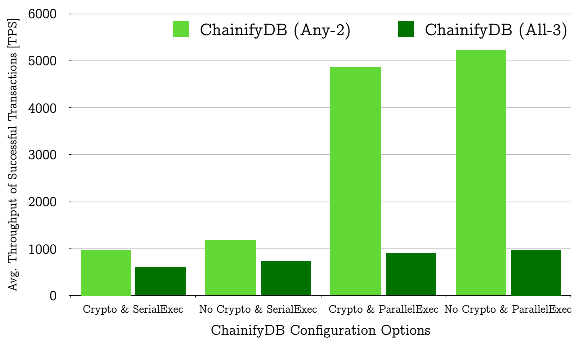


Figure 9: Cost breakdown of ChainifyDB.

our cryptographic components have a negligible negative impact, our parallel transaction execution obviously has a very positive one. With activated cryptography, parallel transaction execution improves the throughput by up to 5x.

4.4 Varying Blocksize

Finally, let us inspect the impact of the blocksize, which is an important configuration parameter in any blockchain system. We vary the blocksize from 256 transactions per block in logarithmic steps up to 4096 transactions per block and report the average throughput of successful transactions.

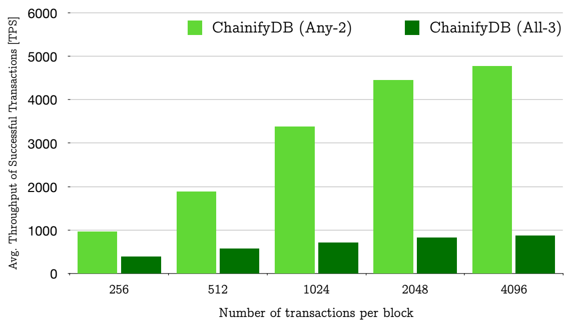


Figure 10: The effect of varying the blocksize.

Figure 10 shows the results. We can see that both under the Any-2 and All-3 policy, the throughput increases with the blocksize. This increase is mainly caused by our parallel transaction execution mechanism, which analyzes the whole block of transactions and schedules them for parallel conflict-free execution.

4.5 Varying the Number of Organizations

Adding a large number of organizations to the chainifyDB network requires a significant amount of computing resources, making the evaluation of the whole system a bit tricky. However, if we look closely, only one component in chainifyDB's transaction pipeline, namely the V-phase, involves synchronous communication between the organizations. Thus, to evaluate the scaling capabilities with the number of organizations, we simulate the computational and network-related efforts required in the V-phase from the perspective of one organization O_i. For each vote, the organization creates a signed request, asynchronously simulates the voting procedure for $k-1$ organizations by waiting for $2 \cdot l(O_i, O_j)$ seconds (round-trip latency), where $l(O_i, O_j)$ is the network latency between two organizations O_i and O_j. It then verifies the $k-1$ cryptographic signatures of the responses and finally compares the local digest with the received digest. In Figure 11, we plot the average latency (five runs) of the voting round while varying the number of organizations under three different agreement policies. We evaluate the V-phase under real-world latencies of AWS regions [1]. For each measurement, we choose a source organization located in Central-Europe AWS region. All remaining organizations were selected uniformly from a set of 15 AWS regions from US, Europe, and Asia.

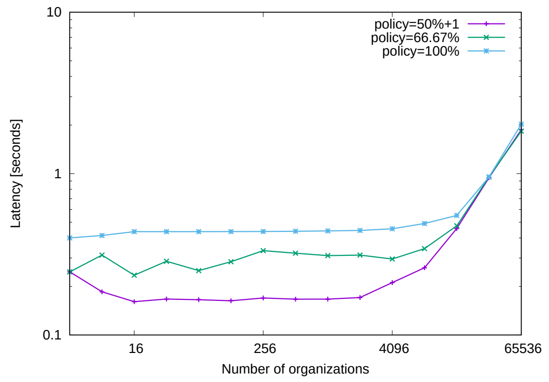


Figure 11: Latency of the V-phase from the perspective of one organization, which performs a voting round with $n-1$ other organizations. We show three different policies, where $50\%+1$, 66.67%, and 100% of the organizations have to agree.

As we can see in Figure 11, for all three evaluated policies, the latency remains stable until the network contains 8192 organizations. This is because up to this point, the V-phase is dominated by the network communication, not the computational effort. If we increase the size of the network further, the computational effort becomes dominant, increasing the latency of the V-phase linearly with the network size. As the size of a typical permissioned blockchain network is significantly smaller than 8192 organizations, the V-phase adds negligible overhead to the transaction pipeline.

5. RELATED WORK

There are other projects that explore the intersection of databases and blockchains. In [21], the authors extend PostgreSQL with blockchain features. This results in a "blockchain relational database", which is capable of performing trusted transactions between multiple PostgreSQL instances. While this project is clearly a step in the right direction, it does not go far enough: In contrast to chainifyDB, the proposed system still comes in a stand-alone fashion and forces the users to integrate yet another new system into their infrastructure. Further, they heavily modify the internals of PostgreSQL. We intended to compare chainifyDB against this system, however, the source code was not available.

BlockchainDB [13, 14] follows exactly the opposite approach of chainifyDB: It installs a database layer *on top of* an existing blockchain, such as Ethereum [3] or Hyperledger Fabric [8]. BlockchainDB basically provides a front end to the underlying permissioned blockchain. In addition, the authors introduce a sharding mechanism which allows them to scale if a relatively large number of organizations are involved. Overall, while this concept eases the use of blockchain systems, it does not ease the integration of blockchain features in established infrastructures.

ChainSQL [20] takes the open-source blockchain system Ripple [9] and integrates relational and NoSQL databases into the storage backend. This enables it to run SQL-style respectively JSON-style transactions. However, similar to [13, 14], by integrating database systems into the heavyweight blockchain system Ripple, the transaction processing performance is limited to that of Ripple — thus overshadowing the high performance of the underlying database systems.

Another project at the intersection of databases and blockchains is Veritas [16]. This visionary paper also proposes to extend existing database systems by blockchain features; however, they focus on a cloud infrastructure. To synchronize instances, they utilize log shipping. Therefore, this solution requires the underlying database system to provide log shipping in the first place and disallows the connection of *different* database systems in one network, if their log shipping mechanisms are not compatible with each other.

BigchainDB [6] combines the blockchain framework Tendermint [7] with the document store MongoDB and therefore extends it with blockchain features. In contrast to chainifyDB, the system is shipped in a stand-alone fashion and uses the MongoDB interface.

Apart from works aiming at closing the gap between databases and blockchains from an architectural perspective, there is a considerable amount of research improving the performance of permissioned blockchains, e.g. in particular for Fabric [17], [10], and [19]. Finally, in our own recent previous work [23] we explored in depth to which degree the performance issues of Fabric can be overcome if we keep Fabric as the central component. The lessons from that work made us drop that idea and highly motivated us to stick with DBMSs as the powerhouses — and write this paper.

6. CONCLUSION

We introduced a highly flexible processing model for permissioned blockchain systems called the Whatever-Voting model, which avoids making assumptions on the precise behavior of organizations by reaching agreement on the outcome instead of the promise. To showcase the strengths of WV, we proposed chainifyDB, an implementation of a blockchain layer, which is able to chainify arbitrary DBMSs and connect them in a network. We discussed how recovery of deviating organizations can be performed. In an extensive experimental evaluation, we showed that chainifyDB does not only offer a 6x higher throughput than comparable baselines, but also introduces a robust recovery mechanism, which grant organizations the chance to participate in transaction processing again.

Acknowledgements

This research is funded by the German Federal Ministry of Education and Research (BMBF).

7. REFERENCES

[1] Aws latencies: https://medium.com/@sachinkagarwal/public-cloud-inter-region-network-latency-as-heat-maps-134e22a5ff19.

[2] CBInsights: https://cbinsights.com/research/report/blockchain-technology-healthcare-disruption/.

[3] Ethereum: https://github.com/ethereum/wiki/wiki/white-paper.

[4] https://www.scientificamerican.com/article/unnecessary-tests-and-treatment-explain-why-health-care-costs-so-much/.

[5] Smallbank: http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[6] Bigchaindb: https://www.bigchaindb.com, 2019.

[7] Tendermint: https://tendermint.com/, 2019.

[8] E. Androulaki, A. Barger, V. Bortnikov, et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. *CoRR*, abs/1801.10228, 2018.

[9] F. Armknecht, G. O. Karame, A. Mandal, F. Youssef, and E. Zenner. Ripple: Overview and outlook. In *TRUST*, volume 9229 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2015.

[10] H. Dang, T. T. A. Dinh, D. Loghin, E. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *SIGMOD Conference*, pages 123–140. ACM, 2019.

[11] D. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[12] B. Ding, L. Kot, and J. Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2):169–182, 2018.

[13] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb - a shared database on blockchains. *PVLDB*, 12(11):1597–1608, 2019.

[14] M. El-Hindi, M. Heyden, C. Binnig, R. Ramamurthy, A. Arasu, and D. Kossmann. Blockchaindb - towards a shared database on blockchains. In *SIGMOD Conference*, pages 1905–1908. ACM, 2019.

[15] J. M. Faleiro, D. Abadi, and J. M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5):613–624, 2017.

[16] J. Gehrke, L. Allen, P. Antonopoulos, A. Arasu, J. Hammer, J. Hunter, R. Kaushik, D. Kossmann, R. Ramamurthy, S. T. V. Setty, J. Szymaszek, A. van Renen, J. Lee, and R. Venkatesan. Veritas: Shared verifiable databases and tables in the cloud. In *CIDR*. www.cidrdb.org, 2019.

[17] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. Fastfabric: Scaling hyperledger fabric to 20, 000 transactions per second. In *IEEE ICBC*, pages 455–463. IEEE, 2019.

[18] S. Haber and W. S. Stornetta. How to time-stamp a digital document. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 437–455. Springer, 1990.

[19] H. Meir, A. Barger, and Y. Manevich. Increasing concurrency in hyperledger fabric. In *SYSTOR*, page 179. ACM, 2019.

[20] M. Muzammal et al. Renovating blockchain with distributed databases: An open source system. *Future Generation Comp. Syst.*, 90:105–117, 2019.

[21] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. *PVLDB*, 12(11):1539–1552, 2019.

[22] T. M. Qadah and M. Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Middleware*, pages 13–25. ACM, 2018.

[23] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD Conference*, pages 105–122. ACM, 2019.

[24] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W. Wong, and M. Zhang. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Trans. Knowl. Data Eng.*, 28(10):2635–2650, 2016.