Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Data model driven goodput optimization for execute-order-validate blockchains

Scientific Students' Association Report

Author:

Máté Debreczeni

Advisor:

Attila Klenik
dr. Imre Kocsis

2022

Contents

Kivonat

Az utóbbi időben az elosztott főkönyvi technológiák (DLT) használata széles körben elterjedt, még a vállalati világban is. A legismertebb a Hyperledger Fabric (HLF), egy jogosultságkezelt DLT-megoldás a vállalati szektor számára. Ez az ágazat nagy teljesítményű és robusztus rendszereket igényel, és a HLF általában mindkét igényt kielégíti. A nagy áteresztőképességet igénylő alkalmazások esetében azonban a HLF-nek vannak potenciális hiányosságai. A naiv adatmodellek esetében az egyik lehetséges hiányosság a többverziós konkurrencia-kezelési konfliktusok (MVCC konfliktusok) miatt visszautasított tranzakciók nagy száma. Ezek a konfliktusok a HLF optimista párhuzamossági modelljének és a HLF által népszerűsített Elővégrehajtás Sorrendezés Validáció (EOV) architektúrának a következményei. Kedvező forgatókönyvek esetén ezek a tervezési döntések nagyobb teljes áteresztőképességet eredményeznek, mint az egyszerű zárolást alkalmazó megoldások, és a hagyományos adatbázisokban évtizedek óta használatosak. A HLF elosztott jellege azonban további késleltetéssel jár az elővégrehajtási és a validálási fázis között, így nagyobb a konfliktus valószínűsége.

A tranzakciós hibák számának csökkentésére számos megoldási javaslat született, amelyek többsége a HLF protokollszintű működésének megváltoztatásával próbálja kezelni a problémát. Ebben a dolgozatban két intelligens adattárolási modell megközelítést javaslok, amelyek az adatmodell és a tranzakciókészlet alapján működnek, és amelyek a változatlan protokoll felett használhatók. Mindkét technika megváltoztatja az adatok szerializálásának és a főkönyvben való tárolásának módját, de mindegyik különböző leképezési stratégiákat használ, hogy ugyanazon információhoz nagyobb számú egyidejű hozzáférést tegyen lehetővé. A két technika közül a fejlesztők választhatnak, figyelembe véve a konkrét felhasználási eseteik tárolási követelményeit és tranzakciós profilját.

Olyan keretrendszert fejlesztettem ki, amely ezeket a technikákat használja, de a hozzáadott komplexitást egy absztrakciós réteg mögé rejti, amely megőrzi a hagyományos fejlesztői élményt. Az én implementációm intuitívabb felhasználói élményt is nyújt az okosszerződés fejlesztők számára az azonnali konzisztencia, ismertebb nevén a "Read Your Writes" emulálásával. E megoldások hatékonyságának értékeléséhez az implementációmat több benchmark alkalmazáson tesztelem, és bemutatom az eredményeimet.

Továbbá javaslok egy MVCC konfliktus mérséklő stratégiát, amely dinamikusan késlelteti a tranzakciók elővégrehajtását. A stratégia a rendelkezésre álló historikus adatok alapján képes késleltetni a nagy valószínűséggel meghiúsuló tranzakciókat, így a hozzáadott késleltetésért cserébe elkerülhető a konfliktus egy folyamatban lévő tranzakcióval. Emellett kitérek a lehetséges jövőbeli munkákra is, amelyeket a megoldásaimmal együtt lehetne használni a további teljesítménynövelés érdekében. A javasolt technikák nem zárják ki egymást. Éppen ellenkezőleg, kiegészítik egymást, és egy többrétegű MVCC konfliktus mérséklési megközelítés részeként használhatók.

Abstract

In recent times the use of Distributed Ledger Technologies (DLT) have seen widespread adoption, even in the corporate world. The most prominent is Hyperledger Fabric (HLF),which is a permissioned DLT solution, meant for the enterprise sector. This sector requires performant and robust systems and HLF generally fulfills both needs. However, when it comes to high throughput use cases, HLF has potential shortcomings. In the case of naive data models, a potential shortcoming is a large amount of rejected transactions due to the occurrence of Multiversion Concurrency Control conflicts (MVCC conflicts). These conflicts are the product of HLF's optimistic concurrency model and the Execute Order Validate (EOV) architecture it popularized in its domain. In favorable scenarios, these design choices result in higher overall throughput than simple locking solutions and have been used in conventional databases for decades.

However, the distributed nature of HLF comes with added latency between the execution and validation phase, thus the probability for a conflict is greater.Several solutions have been proposed to reduce the number of these transaction failures,most of which try to tackle the issue by changing the way HLF works at the protocol level.In this paper I propose two intelligent data storage model approaches, informed by the data model and the transaction set, that can be used on top of the unmodified protocol. Both techniques change how data is serialized and saved on the ledger, but each utilizes different mapping strategies to enable more concurrent access to the same information. The choice between these two techniques is up to the developers, who can make the decision based on their specific use cases' storage requirements and transaction profiles.

I developed a framework that uses these techniques, but hides the added complexity behind an abstraction layer, that preserves the conventional developer experience. My implementation also provides a more intuitive experience for chaincode developers, by emulating immediate consistency, better known as "Read Your Writes". To evaluate the effectiveness of these solutions, I test my implementation on multiple benchmarks and present my results.

Furthermore, I propose an MVCC conflict mitigation strategy, that dynamically delays the endorsement of transaction proposals. Based on available historical data, it is capable of delaying transactions with a high probability for failure, thus avoiding conflict with a pending transaction, in exchange for the added latency. I also elaborate on possible future works that could be used in conjunction with my solutions, for further performance gains. The proposed techniques are not mutually exclusive, on the contrary, they complement each other and can be utilized as part of a multi-layered MVCC conflict mitigation approach.

Chapter 1

Introduction

Ever since the creation of Bitcoin [14], Distributed Ledger Technologies (DLTs) have been ever-increasing in popularity. Permissionless and permissioned DLT solutions are being used to power systems people rely on and use daily. One of the most prominent of these systems is Hyperledger Fabric (HLF) [2], which is hosted by the Linux foundation with the goal of providing a modular, robust and performant blockchain framework for the enterprise sector. HLF has since become a de facto standard in this area, and because of this, it has been applied to a large number of use cases. It generally fulfils the requirements of most of these use cases. However, because it has been applied in a wide variety of fields, some potential shortcomings have also been exposed. One of these potential shortcomings stems from an optimization effort, HLF's optimistic concurrency control model. In an optimistic concurrency control model, transactions are simulated/pre-executed, and their result is not yet final. Only after a validation step can these transactions be considered final. The transactions that are deemed invalid at this step are discarded. This is a well-known technique for increasing the number of concurrent accesses to a system and is used heavily by traditional databases. HLF was the first to popularize the use of optimistic concurrency control in the permissioned blockchain domain and gained a performance advantage [18, 15] over conventional blockchain systems like Ethereum [3]. This required complex changes and represented a paradigm shift from Order Execute (OX) to Execute Order Validate (EOV).

What makes the use of this paradigm potentially problematic in HLF is the added latency, relative to conventional databases, that comes with the distributed nature of DLTs. In some use cases, this can lead to the rejection of a large number of transactions. The invalidated transactions represent wasted computation, electricity, network resources and time. In order to mitigate the effects of Multiversion Concurrency Control conflicts (MVCC conflicts), the cause of the rejections, multiple techniques have been proposed. [4, 1, 20, 19, 24, 10, 25, 16, 11, 26] In my report, I provide a taxonomy of the current MVCC conflict mitigation literature for EOV blockchains and discuss the methods used. I categorize the contributions of others hierarchically, and I identify previously unexplored possibilities in the literature. I propose multiple novel mitigation approaches and include them in the hierarchical taxonomy. My contributions are the following:

- *Taxonomy of the current MVCC conflict mitigation literature:* Hierarchical classification of the current mitigation strategies.
- *Proposition of a new category of mitigation approaches:* A novel category of data storage techniques optimizing the storage scheme.

- *Prototype implementation for two data storage approaches* Partitioning the traditional data model along attributes:
  - *Total partitioning:* Maximizes the number of partitions for maximal MVCC mitigation effect in all cases.
  - *Affinity-based partitioning:* Partitions based on access patterns, creating a smaller number of partitions than Total partitioning for comparable results.
- *Validation of the implementations of the data storage techniques:* Multiple scenarios are benchmarked, and the effect of the implementations are compared to each other, as well as the traditional approach.
- *Proposition of a novel protocol extension based mitigation approach:* Dynamically delayable execution phase to probabilistically avoid MVCC conflicts.

The proposed data storage-based mitigation techniques can be used on top of the unmodified protocol and require no modifications to the system configuration. The techniques change how the data is saved on the ledger by partitioning the saved objects and storing them under several keys. I implemented both approaches as part of a framework that hides the added complexity of partitioning the objects behind an abstraction layer that preserves the conventional chaincode development experience. The framework also uses a cache within the context of each transaction, which enables it to support immediate consistency, better known as Read Your Writes. The proposed protocol extension is capable of delaying the endorsement of transaction proposals. Based on available historical data and the pending transactions, it is capable of identifying incoming transactions with a high probability of failure and can delay them to potentially avoid conflict with a pending transaction. These proposals are intended to be part of a Multi-Layered MVCC conflict mitigation approach, as the parallel usage of both techniques does not impede each other; they work synergistically.

The paper is structured in the following way. In chapter 2, HLF's architecture is discussed to provide a better understanding of the system architecture. Chapter 2 also contains formal definitions of MVCC conflicts and several related terms that are referenced throughout the report. In chapter 3, the current literature of MVCC conflict mitigation techniques is categorized hierarchically, and the techniques are discussed as part of the taxonomy. In chapter 4 the theory behind the proposed data storage approaches is described, and I showcase my implementations. In chapter 5, test scenarios are described, and the results of the benchmarking campaigns are presented. The implications of their outcomes will also be elaborated upon. In chapter 6, the topic of discussion is the proposed protocol extension. The proposed architecture and possible approaches to determine the probability of conflict are showcased. In chapter 7, I provide an overview of the results and discuss some future works that could be used alongside my proposed strategies.

Chapter 2

Hyperledger Fabric Architecture

Hyperledger Fabric[1] (HLF) is an open-source, permissioned, distributed ledger technology (DLT) platform, hosted by the Linux foundation. It is built to be open, performant, robust and modular, which is a difficult task and requires ingenuity and creativity. To achieve these goals, the creators of HLF had to come up with unique solutions, which resulted in a novel system architecture. The complex architecture of the system enabled it to be more performant than other similar systems in its domain (e.g. Ethereum). As the central topic of the report and an important contribution to the domain of DLTs, I find it important to discuss its architecture.

2.1 Network architecture

HLF was created to enable mutually beneficial cooperation between organizations that do not trust each other. This is heavily reflected in its architecture, and the way network participants are organized. This is discussed in detail in the paragraph 2.1. In paragraph 2.1, the components of the network and how they are used are explained, as it is necessary to understand how they enabled HLF to become so widely used.

Network participants At a high level, a HLF network consists of Organizations. Each organization has clients, peers, Ordering Service Nodes (OSNs) and a membership service provider (MSP). All network participants are provided identities by the MSP. MSP is just an abstraction; it can be implemented in many ways. For instance, each organization can provide certificates (conventional x509 certificates) to their peers and use the default implementation of a certificate authority (CA), Fabric-CA.

The OSNs are separate from the other peers and are unaware of the application state. There are multiple implementations available for the ordering service. Kafka, Raft[2] and Solo implementations can be used. However, as of version 2.0, Solo and Kafka are deprecated. Because of this, only Raft is discussed here. The Raft-based implementation of the Ordering Service is a Crash-fault tolerant (CFT) service. Raft is a consensus algorithm that enables multiple nodes to work together and tolerate the failure of nodes.

Peers can be divided into two groups, Endorsing peers and Validating peers. However, endorsing peers are a subset of Validating peers, so the Validating qualifier is redundant. Each network participant has a clearly defined role:

[1] https://hyperledger-fabric.readthedocs.io/
[2] https://raft.github.io/raft.pdf

- *Ordering Service Nodes:* provide the global and deterministic ordering of transactions for the endorsing peers,
- *Membership Service Providers:* provide identification and membership services for network participants,
- *Peers:* validate the simulated transactions based on the data that is read from and written to the ledger by the transactions,
- *Endorsing Peers:* execute the transaction logic and communicate the result to the client and the OSN.

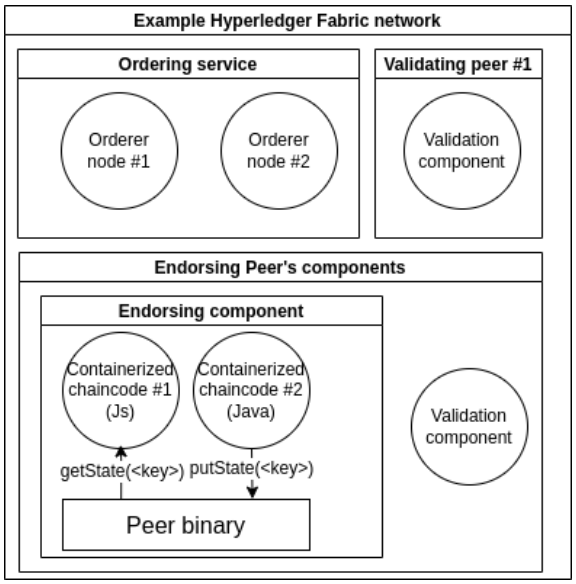

Figure 2.1: Simplified view of an example HLF network

Network components The network participants can agree to establish private subnets of communication, called *channels*. Each channel is a shared instance of a ledger between the participating members. Channels serve as a way to isolate data and the procession of the data. The data store of HLF is a key-value database, where the *keys are versioned* to enable multiple concurrent views. A key's version is equal to the *commit height* of the transaction it was updated in. Thus a written key's version in a transaction that was committed in the 800th block and was the 21st transaction in the block is :

```
{"version_block":800, "version_tx": 21}
```

The most recent version of the keys is stored in the *World state*. The data procession can be done by *chaincodes*, programs installed on a per-channel basis that contain transaction functions. The functions are executed when an authorized client sends a *proposal input* to an endorsing peer. The proposal input is a list of strings that contains the transaction function that is to be invoked and the parameters passed to it. The result of the executed transaction functions is the generated read and write sets on each endorsing peer. The *read-sets* contain the keys and their versions that the transaction retrieved from the world state, while the *write-sets* contain the key-value pairs that are to be written to the ledger. The chaincodes are HLF's version of smart contracts that run in containers and can be written in multiple general-purpose programming languages. The languages available for chaincode development are Golang, Javascript and Java. As mentioned earlier, the chaincodes run in containers, which enables HLF to support multiple development languages.

The containers use the API provided by the SDK to read, query and modify the ledger state. The transaction functions can be called by clients, who are identified channel participants. Their identity is verified and provided by the MSP, and it can be used to control access to the transaction functions. The transactions are not necessarily executed on every peer, as explained in the previous section 2.1. However, so-called *endorsement policies* are defined when a chaincode is installed, which specify rules about how many endorsements a proposal has to collect from each organization. This is made possible by the use of identities in HLF.

2.2 Execute Order Validate

Shortcomings of previous systems To understand the advantages of the Execute Order Validate model, we must first understand the shortcomings of Order Execute. In the Order Execute paradigm, every transaction is ordered via the consensus model, and then the transactions are propagated to the peers (nodes), where they are executed sequentially. At high loads, the sequential execution becomes a limiting factor which lowers the system's throughput considerably. This design is also vulnerable to Denial of Service (DoS) attacks, as malicious transactions could delay the execution of all following transactions indefinitely. Public DLTs solve this problem by introducing a cost for all computations via a cryptocurrency. Permissioned systems like HLF usually do not have cryptocurrencies because malicious actors can be dealt with in other ways thanks to the identification of participants.

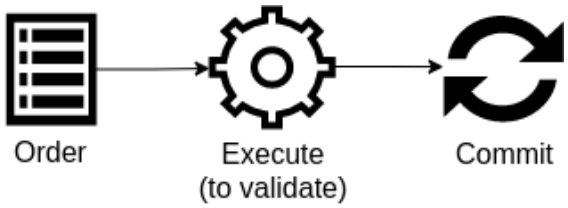

Figure 2.2: The execute-validate flow

Overview The Execute Order Validate paradigm has fundamental differences that set it apart from EV. The execution of transactions precedes the ordering step, and a third step is introduced to protect against attacks such as double spends [Figure 2.3]. The execution of transactions can be done in parallel and on any subset of the peers that take part in the endorsement (execution) process. As a result of the endorsement process, the read and write-sets are generated for each transaction. However, the endorsed transactions are not yet final; they are only simulated and need further validation before they can be committed to the ledger. Before progressing to the validation phase, a globally consistent transaction order is established by the Ordering service. Networks with a single OSN are possible. However, the use of multiple nodes is strongly encouraged in production networks, as they serve to further the resilience of the system. The ordering process is not based on the contents of the transactions, which many alternative solutions seek to alter [25, 20, 24, 19, 10]. After the transactions have been ordered, they are sent back to peers for validation, where they are evaluated sequentially in the order established by the Ordering Service. This evaluation is deterministic, and all conflicting transactions are discarded.

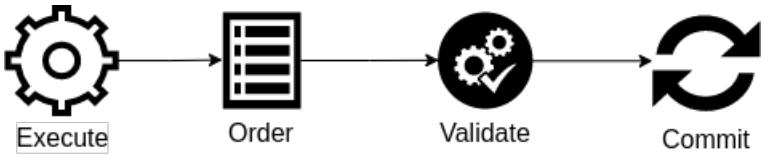

Figure 2.3: The execute order validate flow

Transaction flow The transaction flow describes each step of the transaction's lifecycle, from the moment the client initiates the transaction until it is committed on the ledger. The simplified transaction flow is depicted on Figure 2.4

Step 1. - Proposal construction The client initiates a transaction by calling a transaction function defined in one of the chaincodes installed on the channel. The transaction function can read and/or write multiple keys from the state database. The client SDK then constructs a transaction signed proposal from the input, which is sent to a number of endorsing peers. The number of endorsing peers it is sent to depends upon the endorsement policy.

Step 2. - Proposal endorsement The endorsing peers verify the signature validity and the well-formedness of the proposal and check if the client is authorized to change the ledger state. If the proposal passes all verification checks and has not already been submitted, the transaction is executed by the peers, and the proposal is endorsed. The ledger is not yet updated with the execution results.

Step 3. Assembling a transaction The client receives the proposal responses from the endorsing peers and verifies that all responses are identical. If the responses differ in any way, the transaction proposal is rejected. The endorsements are assembled into a transaction by the client, which is then sent to the OSNs for ordering and block inclusion. The assembled transaction contains the Channel ID, the read/write sets and the signatures of the endorsing peers.

Step 4. Transaction Ordering An OSN receives the endorsed transaction along with potentially many other transactions. The full contents of these transactions are not inspected, as the ordering result does not take them into account. The ordered transactions are included into blocks on a per-channel basis. The blocks are then delivered to the peers for validation.

Step 5. Transaction validation The transactions in the block are validated against the endorsement policy, as well as the current ledger state. Because of concurrent accesses, modifications could have been made to the ledger since the read and write sets were assembled. The transactions that fail any of the validations are rejected and marked as invalid. Their modifications will not be committed to the ledger.

Step 6. Transaction commit The peers add the validated blocks to each channel's ledger, after which the write sets are committed to the current state database as well. The process is completed by an event emission, which notifies the peer of the result.

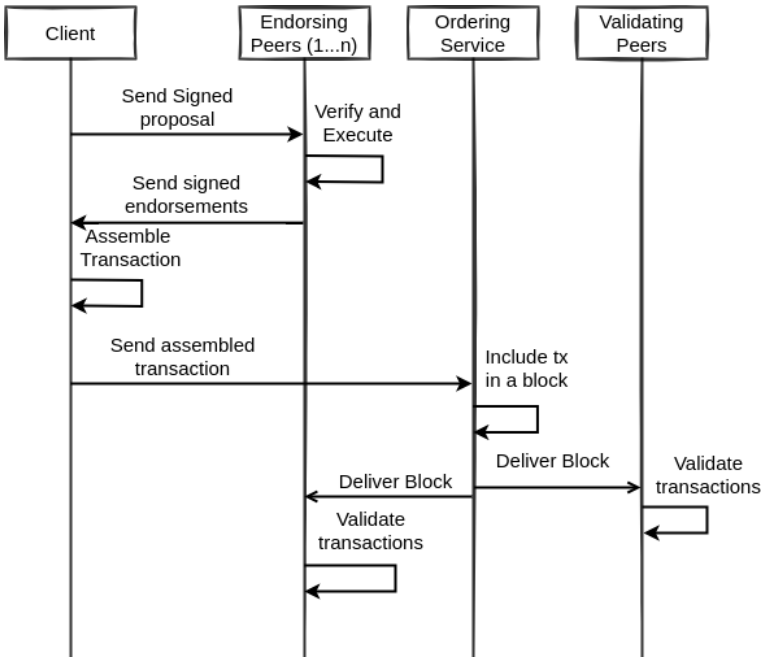

Figure 2.4: The simplified transaction flow. Notice that all peers take part in the validation process

2.3 Multiversion Concurrency Control conflicts

Data in HLF is stored as key-value pairs. This data is accessed by clients, who can read and update a key and its value independently of each other, even on multiple peers. Because of the EOV architecture, the effect of transactions does not take place instantaneously. In the case of a bank account use case, this presents the well-known double spend problem. Let *T1* and *T2* be transactions that read key k and value v, where $k = balance$, $v = 5$. Thus both transactions' read-set is: $\{k = balance, v = 5\}$. If *T1* and *T2* both decrease the value of *balance* by five and are submitted at the same time, both transactions are considered valid at the endorsement phase. The resulting write-sets would be: $\{k = balance, v = 0\}$. Without any concurrency control, both transactions could be committed, and double spends would be possible.

This is prevented in HLF by attaching a version to every key, which is checked during the last phase of the transaction lifecycle. With versioned keys, the previous scenario would change in the following way. Both transactions' read-sets are identical: $\{k = balance, v_k = version1, v = 5\}$ After execution, the write sets of both transactions are the following: $\{k = balance, v_k = version2, v = 0\}$ At the endorsement phase, both transactions are still considered valid. However, at the validation phase, the version of the keys is checked, and the transaction that tries to update the value for the second time would fail.

Transaction	Read key	written key	key version	status
Tx1	(k1, v1)	k1	v2	success
Tx2	(k1, v1)	k1	v2	fail (invalidated)

Table 2.1: Example of an MVCC conflict

2.3.1 Notations

The following notations will be used in the formalization of MVCC conflicts[3].

- *Set of endorsing peers:* $\mathbf{P}$
- *Set of transaction attempts in a system:* $\mathbf{T}$
- *Set of callable operations (denoted as functions in HLF):* $\mathbf{F}$
- *Ledger-resident keys:* $\mathbf{K}$

2.3.2 Definitions

In this subsection, I formally define important terms that are often referenced throughout the report. The precise definition of these terms is necessary to avoid ambiguity throughout the rest of my report. The following terms are defined in this section:

- *Read-set*,
- *Write-set*,
- *World state*,
- *Transaction dependency*,
- *Transaction function dependency*,
- *MVCC conflict*.

Definition: Read-set Read-set R_t^p is an ordered set of unique keys and associated version numbers, generated by an endorsing peer $p \in \mathbf{P}$ during the endorsement of $t \in \mathbf{T}$.

$R_t^p = \{(k_1, v_{k_1}), ..., (k_n, v_{k_n})\}$, where $n \in \mathbb{N}, \{k_1, ..., k_n\} \in \mathbf{K}$

Definition: Write-set Write-set W_t^p is an ordered set of unique key-value pairs generated by endorsing peer $p \in \mathbf{P}$ during the endorsement of $t \in \mathbf{T}$.

$W_t = \{(k_1, V_{k_1}), ..., (k_n, V_{k_n})\}$, where $n \in \mathbb{N}, \{k_1, ..., k_n\} \in \mathbf{K}$

Definition: world state World state S^p is an ordered set of unique keys and their associated versions and values, as accepted by a given peer $p \in \mathbf{P}$ as the latest.

$S^p = \{(k_1, v_{k_1}, V_{k_1}), ..., (k_n, v_{k_n}, V_{k_n})\}$, where $n \in \mathbb{N}, \{k_1, ..., k_n\} \in \mathbf{K}$

Definition: Transaction dependency Transaction $t_i \in \mathbf{T}$ depends on transaction $t_j \in \mathbf{T}$ if the read-set of R_{t_i} and the write-set of W_{t_j} both contain the same key. By omitting the peer index for the write and read sets, I assume that a sufficient number of peers agreed on the endorsement results (transaction dependency-related questions are only meaningful over pairs of sufficiently agreed-on endorsement sets).

$$\Delta(t_i, t_j) = \begin{cases} 1 \leftarrow R_{t_i} \cap W_{t_j} \neq \emptyset \text{ and } i \neq j \\ 0 \text{ otherwise} \end{cases}$$

[3]The definitions are inspired by [4]

Definition: Transaction function dependency Transaction function $f_i \in \mathbf{F}$ is dependent on transaction function $f_j \in \mathbf{F}$ if transactions $t_i \in \mathbf{T}$ and $t_j \in \mathbf{T}$ exist (with appropriate function call parameter bindings) such that $\Delta(t_i, t_j) = 1$.

Definition: MVCC conflict During the block validation process on a peer $p \in \mathbf{P}$, a transaction $t \in \mathbf{T}$ is deemed an MVCC conflict if and only if there exists a key $k \in \mathbf{K}$ in the read-set R_t of t, whose associated version v_k does not match the version v'_k of k found in the world state S^p.

Chapter 3

MVCC conflict mitigation and avoidance

In this chapter, the current literature of MVCC conflict mitigation techniques is discussed. On Figure 3.1, a categoric tree of the current literature can be seen. I have identified the following hierarchical categories (blue-coloured branches on [Figure 3.1]):

- *MVCC conflict mitigation techniques:* the root of the categoric tree, as all categorized works contributed to the mitigation of MVCC conflicts,
- *Protocol independent techniques:* this subcategory of MVCC conflict mitigation techniques can potentially be used with all EOV blockchains,
- *Protocol optimizations:* this subcategory contains contributions that mitigate the effects of MVCC conflicts at the protocol level,
- *System configuration tuning:* the techniques in this subcategory are concerned with selecting the optimal system configuration with the specific goal of mitigating MVCC conflicts,
- *Data storage techniques:* a previously unexplored category of techniques, that mitigate the effects of MVCC conflicts by means storing the logical data in an alternate way,
- *Semantic data model techniques:* this category contains data modelling techniques that aid in mitigating the MVCC conflicts,
- *Protocol alterations:* the contributions in this category attempt to mitigate the effects of MVCC conflicts by altering the protocol of HLF, but because of the alterations, they are incompatible with a network using the unmodified protocol,
- *Protocol extensions:* the techniques in this subcategory extend the protocol of HLF in such a way that they remain compatible with a network using the unmodified protocol.

The leaves in each category have been colour-coded to help distinguish the origin of the contributions. The green leaves are my contribution, while the tan leaves feature already established literature. My contributions will be discussed in their dedicated chapters, 4, 6.

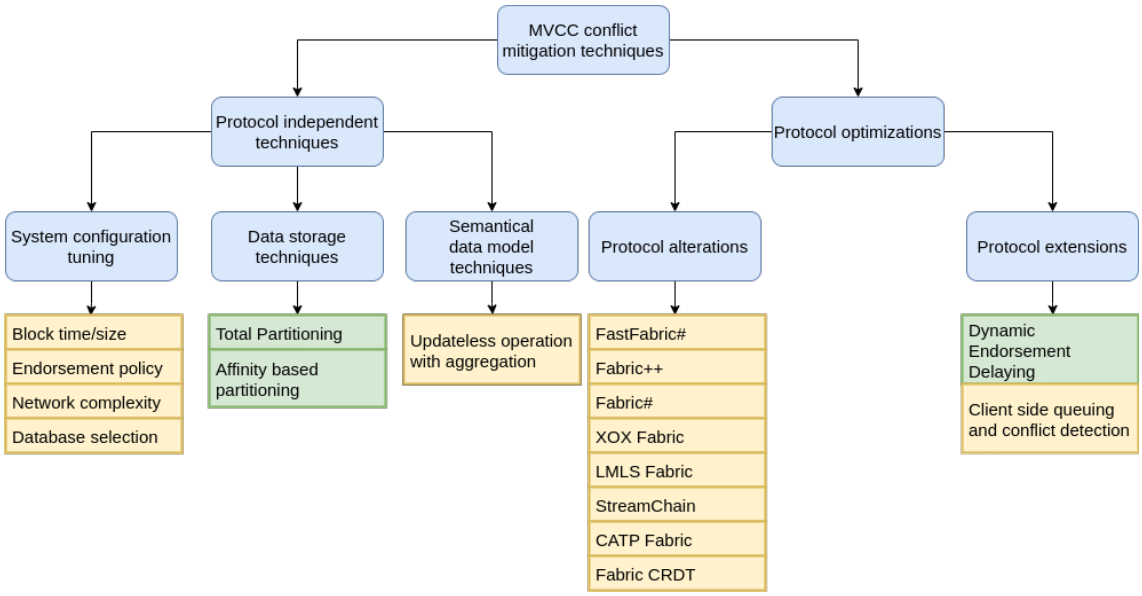

Figure 3.1: Categoric tree of the current state of the MVCC conflict mitigation literature

3.1 Protocol independent techniques

This category features optimization approaches that are independent of the used version of the protocol.[1] The techniques present in this category might be applied to any of the protocol optimized versions of HLF, and they might result in a decreased number of conflicts. The subcategory is depicted on Figure 3.2.

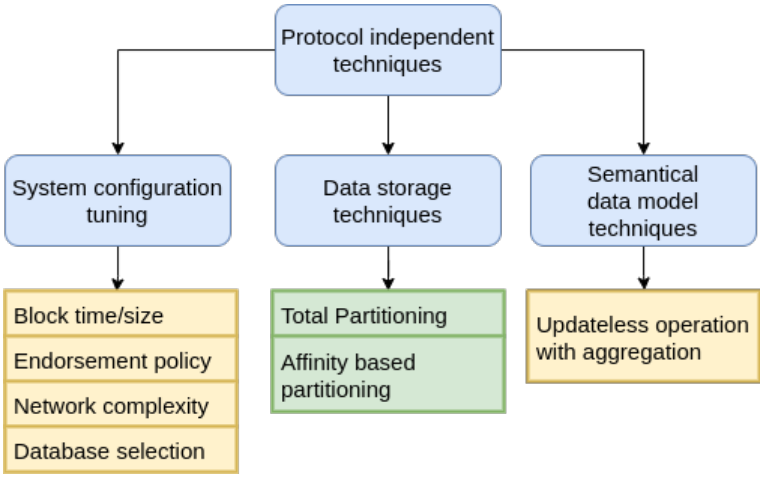

Figure 3.2: The protocol-independent subcategory of MVCC conflict mitigation techniques

3.1.1 System configuration tuning

Although there are numerous papers that are concerned with performance modelling and optimizations of HLF [22, 13, 9, 7, 23, 21], only those that are concerned with optimizing for improved transaction failures will be discussed here. At the time of writing, the only

[1] in the case of StreamChain, the block size is not applicable

paper I found is written by Chacko *et al.* [4]. In the paper, the effect of the following configuration options are investigated.

- Block size: The maximum number of transactions in a block,
- Endorsement policy complexity: the number of endorsements that are required for successful transaction endorsement,
- Network complexity: the number of organizations and peers in the network,
- Database selection: whether GoLevelDB or CouchDB is used for storing ledger data.

Block size In the paper, the authors found that the effect of block size is network and workload-specific and should be experimented with on a case-by-case basis. However, as a guideline at lower transaction arrival rates, a smaller block size is recommended, as it generally results in lower latency, which is favourable for MVCC conflict mitigation. Counter-intuitively, latency is limited by a larger block size at higher transaction arrival rates, as the overhead of ordering and validating larger blocks is reduced relative to the same number of transactions in smaller blocks.

Endorsement policy complexity Endorsement policy complexity is best kept at a minimum, as the higher number of required endorsements result in higher latency values, which increase the amount of MVCC conflicts.

Network complexity Similarly to Endorsement policy complexity, it is beneficial to limit the number of peers and organizations as much as possible, as with the increased network complexity comes increased latency. The authors did not publish the impact of network complexity on MVCC conflicts, only on Endorsement policy failures. However, their results show that higher latencies lead to more MVCC conflicts. Thus network complexity should also be considered a tunable parameter when the aim is to reduce MVCC conflicts.

Database selection GoLevelDB performed better in all metrics; thus, the authors recommend choosing GoLevelDB if the rich queries offered by CouchDB are not a necessity.

3.1.2 Semantic data model techniques

The most notable paper in this subcategory is written by Alzubaidi *et al.* [1], in which the authors showcase an MVCC conflict-aware data model for an Internet of Things (IoT) chaincode that deals with high-frequency updates but low-frequency reads. Although the example the authors used was IoT-specific, the technique has a wider range of applications. It can be used in any situation where the frequency of updates is high relative to the frequency of the reads and where the updates are not dependent on the previous value of the updated key. The technique is based on essentially forbidding updates on a key and opting to write the new values to new objects on the ledger instead. The authors utilize composite keys for the objects so that they can be read with range queries in case of infrequent reads. Although the authors only showcased the technique in a case study, the approach could be automated and generalized as part of a semantic data model, which could popularize the use of the approach.

3.2 Protocol optimizations

The techniques present in this category either extend or alter the protocol of HLF. In this paper, the alteration will be used as a synonym for breaking change. This means that a peer running an instance of the altered software would not be compatible with other peers running the unaltered protocol. In contrast, the technique(s) listed as protocol extensions are compatible with a network running the unaltered version of the protocol.

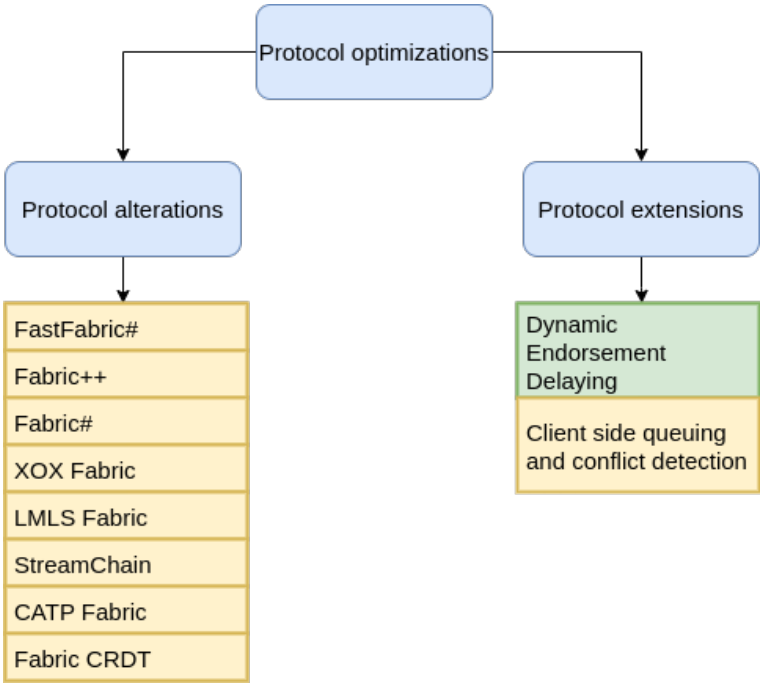

Figure 3.3: The subcategory of MVCC conflict mitigation techniques that are based on protocol optimizations

3.2.1 Protocol alterations

Fabric++ Fabric++ reorders transactions based on data dependencies and, in the process, utilizes early aborts for conflicting transactions that can not be successfully committed even with reordering. [20] The early aborts are done by greedily early aborting transactions that are part of the highest number of cycles. If all cycles are resolved, the remaining transactions can be reordered.

FabricSharp and FastFabric# Similarly to Fabric++, FabricSharp utilizes transaction reordering based on data dependencies.[19] However, FabricSharp achieves the reordering more efficiently and works with a reduced set of constraints on transaction serializability, which prevents the abortion of transactions that read across blocks. FastFabric is a performance-optimized version of HLF, but it is not concerned with reducing MVCC conflicts [9]. However, Ruan *et al.* also implemented the improvements of FabricSharp on top of the optimizations of FastFabric.

LMLS Fabric LMLS Fabric uses a centralized database to lock accessed keys and thus early abort conflicting transactions [24]. The solution also utilizes a cache of the latest

values belonging to the keys to decrease the number of transactions that read outdated information.

XOX Fabric XOX Fabric got its name from the architectural changes it features. It utilizes a post-ordering execution phase to re-execute the failed transactions with the newly updated world state [10]. Since the transactions are re-executed with newly updated data, MVCC conflicts due to outdated reads are eliminated.

CATP Fabric CATP Fabric also utilizes transaction reordering but with a different approach. In CATP Fabric, a key-based transaction processing module is introduced after the regular protocol's ordering step [25]. This module filters transactions with outdated read-sets, prioritize read-only transactions and merges conflicting transactions' write-sets if the balance of the key is sufficient for both transactions to pass. The authors do not mention how the system deals with conflicting transactions operating on non-balance-like data.

CRDT Fabric CRDT Fabric incorporates Conflict-Free Replicated Datatypes (CRDT) into the protocol [16]. CRDTs essentially bypass MVCC checks; thus, conflicts are not possible when using them. The results of the transactions are stored in a specific data structure that merges them; thus no results are lost. CRDTs are not suitable for all applications, as the lack of MVCC validation would result in possible overspends in the case of financial balance data.

StreamChain StreamChain limits transaction latency by not using blocks and optimizing the components of HLF to deal with the overhead caused by only batching transactions before commitment [11]. To mitigate the effects of the streaming approach, StreamChain uses RAMDisks both at the Orderer, added parallel validation of transaction signatures and implemented software pipelining, among other things. Due to these changes, StreamChain achieves extremely low latency (10ms) and thus reduces the transaction failure rates [4].

3.2.2 Protocol extensions

Client side queuing and conflict detection Zhang *et al.* [26] implemented a modified client-SDK, that employs a queuing mechanism for transactions via a client-side cache. Transactions in the queue are subject to conflict analysis. Thus they can be early aborted and re-executed before they are sent to the OSNs.

Chapter 4

Partitioning Framework

Previous approaches failed to take advantage of the possible performance optimizations in a semantic data modelling approach. In general, the advantage of a data modelling-based optimization approach over previous solutions is twofold. First, it can be used without modifying the underlying protocol. Second, it can be tailored to the specific needs of the use case. In general, the downside of such approaches is that the burden of implementation and maintenance falls on the developers, which can be complex and might require a specialized workforce. I propose a novel semantic data modelling framework that can improve transaction failure rates and provide a more intuitive chaincode development experience. The framework splits the objects used in traditional Object Oriented Programming into multiple partitions, and stores each partition under a different key, thereby enabling more concurrent access.

4.1 Core idea

In practice, MVCC conflicts are usually a product of frequent updates to a key-value pair. If the value is a complex object, the frequency of the updates can potentially be reduced by storing some attributes as separate key-value pairs. Consider the following example. A factory is tracking the assembly progress of cars, and at any point in time, there are multiple robot arms working on a single car. The robots are not allowed to put some parts in before others for safety reasons. The car object is a simple checklist of parts, where the parts are the keys and the values are either "true" or "false". Since cars are complex and have many parts, this would result in a large amount of data being stored under a single key. A JSON representation of the object might look like this:

```
{"carID": "000", "engine":true, "tires":true,..., "exhaust":false}
```

This JSON object could be stored on the ledger under a single key. This would result in every transaction being dependent upon every other transaction that deals with the same car, as the the object would have to be read to verify that no part has been placed in violation of the safety code and the object would be updated with the added part in every transaction. An alternate way to store this data might be the following:

```
{"carID": "000", "engine":true},...,{"carID": "000", "exhaust":false}
```

Instead of storing every part in a single object on the ledger, every part can now be stored as a separate object under a separate key. This resolved most transaction dependencies, as the only dependencies remaining are between transactions that created the safety-critical parts that the other transactions check. The first data model could be considered the *logical data*

model, where data is grouped together based on OOP and other similar principles. The first data model is convenient to write programs with but can result in problems. The second data model is hard to write programs with. For example, in a Java chaincode, it might require the creation of many classes and would result in boilerplate code. However, the second data model is better suited for storage on the ledger, therefore, can be considered the *ledger data model*. Decoupling the two data models and programmatically translating between the two combines the benefits of both models.

As defined in the *Hot Key Theorem* by Gorenflo *et al.*, [10] if l is the average time between a transaction's execution and its state transition commitment, then the average effective throughput for all transactions operating on the same key is at most $1/l$. By dividing the data stored under key k to $n \in \mathbb{N}$ number of distinct independent parts and storing them under keys $\{k_1, ..., k_n\}$, the average effective throughput of all transactions that operated over key k becomes at most n/l.

Example Consider the data model of object A. [Figure 4.1] Object A has two attributes that can be represented as two separate objects. Representing them as such is potentially beneficial from a transaction failure perspective, as the attributes can now be accessed independently.

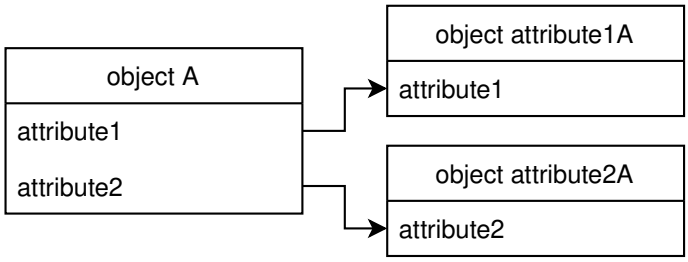

Figure 4.1: Object A is partitioned along its attributes

Practical considerations This was done to some extent by encouraging developers to follow OOP practices like separation of concerns. While this can result in fewer conflicts, the goal of such guidelines was predominantly to increase the maintainability of the code, not to decrease transaction failure rates. By following the above-described partitioning practice, more concurrent access is possible, however, code would become hard to maintain. For code maintainability, it is best if the logical data model is preserved and the mapping between the logical and ledger data models is done programmatically.

4.2 Total partitioning

As explained in 4.1, the logical data model must be divided into distinct non-overlapping subsets for a potential increase in effective throughput. One possible approach is to create a subset for each attribute of the object. 4.1. The algorithm is discussed in the following section.

4.2.1 Total partitioning algorithm

The algorithm creates a partition for all attributes of an object. These partitions can then be saved as separate objects on the ledger. The algorithm has a complexity *O(n)* where n

is the number of attributes the object and its attributes (if they are nested objects) have combined.

Algorithm 1 Total partitioning algorithm

```
1: procedure PARTITIONTOTALLY(asset)
   ▷ asset is an instance of a subclass of Asset class
   ▷ Asset class has the following property:
   ▷ uuid: string, a unique identifier of the object
   ▷ As asset is a subclass, it can have any number of attributes
   ▷ attributes are typed key-value pairs, where the key is a string
2:     result ← {}                                   ▷ empty key-value store
3:     id ← GETUUID(asset)
4:     serializedId ← SERIALIZE(id)
5:     PUT(id, serializedId)                ▷ saving the asset's id as a partition
6:     for attribute : asset do
7:         if attribute instanceof Asset then        ▷ attribute is nested asset
8:             nested ←PARTITIONTOTALLY(attribute)          ▷ call recursively
9:             for entry : nested do
10:                PUT(result, entry)      ▷ Put the key-value pair in the result map
11:            end for
12:        end if
13:        serialized ← SERIALIZE(attribute)
   ▷ getName returns the key of an attribute
14:        attributeName ← GETNAME(attribute)
15:        PUT(attributeName, serialized)
16:    end for
17:    return result                  ▷ The key-value pairs to be saved on ledger
18: end procedure
```

4.2.2 Benefits and potential shortcomings

The algorithm requires no input other than the logical data model, which is presumably readily available in the early stages of the design process. The approach is simple, but offers a maximum reduction in failure rates, assuming there are no transactions accessing only a subset of the data stored as an attribute. However, creating a partition for all attributes of an object might not be necessary, as there might be attributes that are never accessed independently. In such cases, the resulting ledger data model requires unnecessary database accesses, which might affect the throughput and the latency of the system negatively. Another potential drawback of this, and all partitioning approaches, is the increased storage requirements.

4.3 Affinity based partitioning

If data is available about the frequency of the transaction functions, the objects and attributes accessed by each function is known, it is possible to create more sophisticated ledger data models. (Example input data is provided at [Table 4.1].) By using the well-known techniques of Vertical Partitioning proposed by Navathe *et al.* [17], but only considering the transactions that update values as a semantic alteration to their approach,

ledger data models with a smaller number of partitions can be created that potentially offer the same degree of improvement in failure rates as the total partitioning approach.

4.3.1 Affinity-based partitioning algorithm

The algorithm works by constructing an affinity matrix (AA) from the input data, which is a diagonal matrix, where each field contains a value representing the similarity of each attribute in terms of the accesses to them. (Example AA at Table 4.2) The Bond Energy algorithm (BEA) [12] is then applied to this matrix to create clusters. (Example at Table 4.2) After the clustering step, the SPLIT_NON_OVERLAP algorithm [17] is applied to the matrix. This is done n times, once for each permutation of the matrix created by the SHIFT procedure, where n is the number of rows columns in the matrix. The SHIFT procedure places the leftmost column to the extreme right of the matrix and the bottom row to the top. From all the resulting binary partitions, only one is selected, such that the number of transactions that access only one of the two partitions is maximized, while the number of transactions that access both partitions is minimized. This is done by finding the binary partitioning with the maximal z value. The z value of a binary partition is calculated as such:

$$z = c_u c_l - c_i^2 \tag{4.1}$$

Where:

- c_u*:* is the number of transactions that access only the upper partition of the AA matrix,
- c_l*:* is the number of transactions that access only the lower partition of the AA matrix,
- c_i*:* is the number of transactions that access both partitions of AA matrix.

Only partitions with a positive z value are accepted. N-ary partitioning is achieved by repeatedly applying the algorithm to the affinity matrix of the resulting partitions.

Example input data about the updating functions				
Transaction function	attribute1	attribute2	attribute3	invocations
function1	1	0	1	30
function2	0	1	0	50

Table 4.1: Example input data for affinity-based partitioning. Accessed attributes are marked with a 1

attributes	attribute1	attribute2	attribute3
attribute1	30	0	30
attribute2	0	50	0
attribute3	30	0	30

Table 4.2: AA matrix created from table 4.1. (The algorithm only uses the numerical values)

attributes	attribute2	attribute1	attribute3
attribute2	50	0	0
attribute1	0	30	30
attribute3	0	30	30

Table 4.3: This AA is transformed from table 4.2 by the BEA.

Final partitions	
partitions	attributes
partition1	attribute2
partition2	attribute1, attribute3

Table 4.4: The result of partitioning the matrix in table 4.3.

4.3.2 Adaptations for application in HLF

Only updating transactions as input The input data is limited only to the updating transactions because the key's version only changes upon updates. Thus storing attributes that are frequently read together as separate objects does little to decrease the number of MVCC conflicts, as any transaction that reads only a subset of the updated attributes before the previous transaction is committed will be invalidated. Besides this, read-only transactions are encouraged not to be submitted for validation by the HLF community.

SPLIT_NON_OVERLAP as the splitting algorithm From the algorithms proposed in [17] for splitting the clustered AA matrix SPLIT_NON_OVERLAP was chosen despite the author's recommendations for algorithms to use in the case of a distributed database with replicated allocation because of the fundamental differences between traditional distributed databases and HLF. The cost of all factors examined by the paper is far outweighed by the cost of resubmission that is necessary in case of a rejected transaction. Besides this, all peers possess the state database; therefore, no data is fetched from other nodes in the endorsement step. Thus HLF is more akin to a single-site database in this regard.

4.3.3 Benefits and potential shortcomings

The benefits of affinity-based partitioning over the total partitioning approach are twofold. First, no unnecessary partitions are created, reducing the number of database accesses. Second, the maximum number of partitions and the minimum partition size are configurable, making the approach configurable for storage size. Compared to total partitioning, the drawback of this approach is that it requires input data, which might not be available at the design stage when data modelling decisions are made. However, the frequency input data does not need to be very precise, as the 20-80 rule specifies that a limited number of transactions are responsible for the majority of traffic, thus rough estimates should suffice.

4.4 Prototype framework

The prototype framework was implemented in Java, apart from the partitioning scheme creation for the affinity-based approach, which was created in Python. The design focuses on modularity and ease of use for the developers. The prototype has a fully functional

runtime implementation. However, a production-grade version of the framework could also utilize code generation, thereby further simplifying the developer experience beyond the one offered by the current implementation. (An example chaincode using the partitioning framework, and an example of the traditional approach is available in the appendix 7.2.)

4.4.1 Framework API and usage

A short description of the most notable classes and interfaces of the framework:

- *AssetBase:* All classes that need to be persisted in a partitioned manner must extend this class. It defines the core structure of the ledger objects.
- *DataLayer:* An interface that defines the expected API of the classes that deal with ledger access.
- *AssetCache:* This interface defines the expected functionality of a cache.
- *IAttributePartitioning:* Defines the expected API of classes that translate between the logical and ledger data models.
- *PartitionedCachedContext:* Custom transaction context that contains the application level cache. Data retrieved from the ledger is saved in its read cache, and data that needs to be written to the ledger is put in its write cache.
- *AssetContractInterface:* Provides default implementations for the before and after-Transaction lifecycle functions and creates the PartitionedCachedContext instances for each transaction.

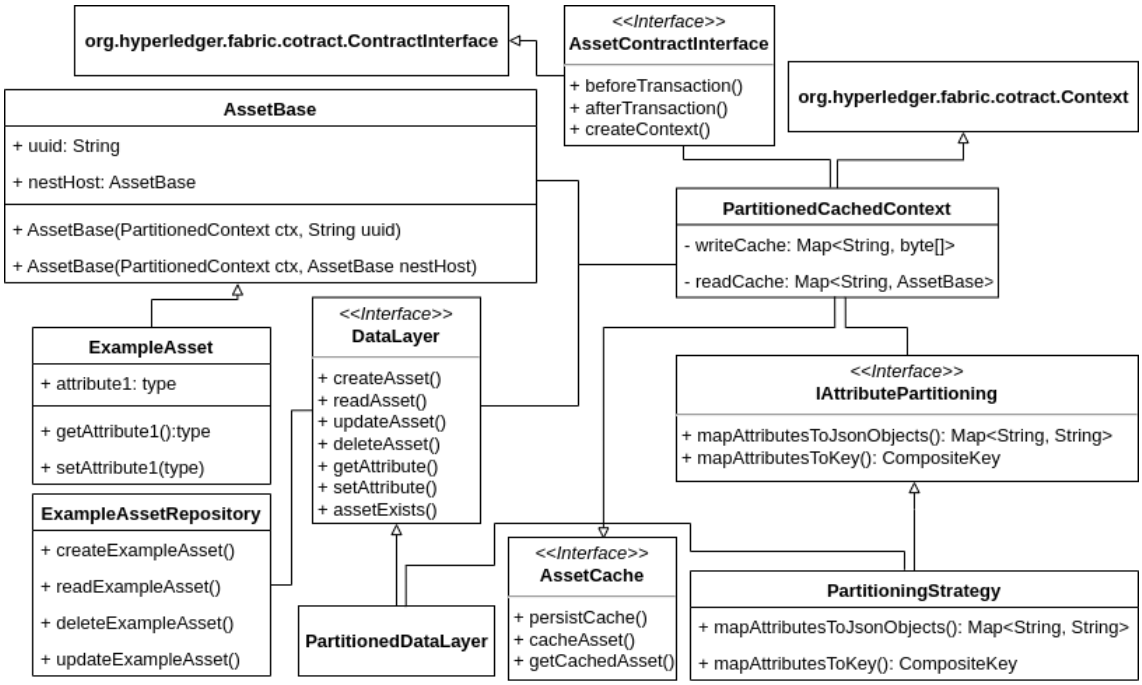

Figure 4.2: Simplified class diagram of the implementation

API The current implementation differs from the conventional development experience in 2 significant ways. First, immediate consistency (RYW) is supported using an application-level cache. Second, it is possible to use getters and setters on classes inheriting from the AssetBase class that read from the ledger and write to a cache. The getters perform tasks that are akin to loading each attribute of the asset lazily from the ledger. The changes done to the object by the setters are cached. The cache is then persisted after the transaction function is executed. It is important to note, that because the changes to the object are persisted on the ledger, AssetBase needs an instance of Context to be instantiated.

Usage The current implementation requires chaincode developers to write two classes for each class that holds data that needs to be persisted on the ledger with a partitioning approach. Consider a class named *ExampleAsset* that holds data that needs to be persisted on the ledger in a single attribute named *attribute1*. (Example illustrated on Figure 4.2) First, a class extending AssetBase needs to be created that defines the data model, annotates the attributes that need to be persisted and uses the getAttribute and setAttribute functions of DataLayer in its getter and setter implementations to get and set data on the object. Second, the *ExampleAssetRepository* class needs to be created that defines create, read, update and delete (CRUD) functions for the *ExampleAsset*. This class is a wrapper class and is expected to use the similarly named functions of PartitionedDataLayer in the implementation, as the logic is already implemented generically, and the functions require type arguments and are not convenient to use. These steps could be trivially automated in a production-ready implementation, as explained in subsection 4.4.3. After creating these simple classes, the developer can implement the business logic of the chaincode using the functions defined in the newly created classes.

4.4.2 Implementation

This subsection discusses the implementation details of the prototype partitioning framework. The classes and interfaces of the framework can be divided into two groups. The first one constitutes the rigid core of the framework, and the second is the extensible part responsible for implementing the different partitioning strategies. The core part is discussed in subsection 4.4.2.1, while the implemented approaches are discussed in subsections 4.4.2.2, 4.4.2.3.

4.4.2.1 Framework Core

The two main components responsible for translating between the ledger and logical data model are the classes that implement the DataLayer and IAttributePartitoning interfaces [Figure 4.2]. A new partitioning approach can be implemented by creating the two classes that extend the above-mentioned interfaces. The framework uses a custom context, that is the subclass of org.hyperledger.fabric.contract.Context and it implements the AssetCache interface. PartitionedDataLayer is implemented so that if an asset is read, it places it in the context's read cache, and if it is modified, it puts the serialized byte array of the new value in the writeCache. The writeCache is persisted after the transaction's business logic is completed via an invocation of the persistCache function in the afterTransaction lifecycle function of the AssetContractInterface [Figure 4.6]. For every transaction a new instance of PartitionedCachedContext is created, thus no new shared state is created between the transactions.

4.4.2.2 Total partitioning

As previously explained in subsection 4.4.2.1, partitioning approaches are implemented in a two-step process. First, a class that implements the IAttributePartitoning interface needs to be created, which handles the translation between the ledger and logical data model. Second, a class implementing the DataLayer interface needs to be created that is responsible for creating the CRUD operations that deal with the specifics of the ledger data model. In the case of Total Partitioning approach, the algorithm discussed in subsection 4.2.1 was implemented with the help of the Java.lang.reflection library, as part of a class that implements the IAttributePartitioning interface.

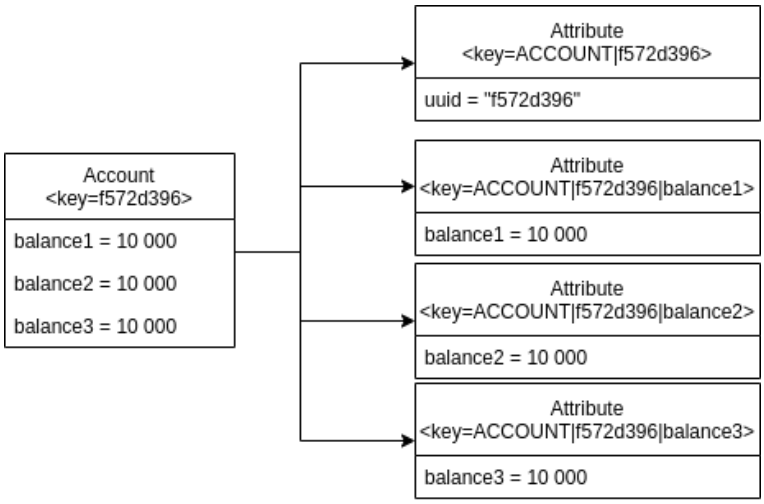

Figure 4.3: Example of the data model resulting from total partitioning (the ledger data model used in benchmarking [chapter 5])

4.4.2.3 Affinity based partitioning

As mentioned at the beginning of section 4.4, the affinity-based partitioning approach was implemented in Python. The python program takes files that contain the necessary metadata for running the partitioning algorithm as input. These are simple .csv files with data structured similarly to the example in Table 4.1. The output of the Python program is the partitioning scheme for all classes. The Java SDK then parses this, and the partitioning is performed by a class implementing the IAttributePartitioning interface accordingly. This flow is depicted on Figure 4.4.

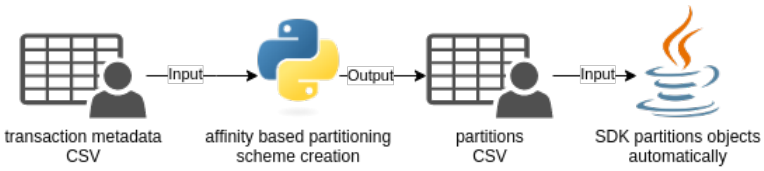

Figure 4.4: Flow depicting the usage of the Affinity-based partitioning approach

4.4.3 Code generation

The developer experience could be further enhanced with code generation in a production-grade implementation. As mentioned earlier, the code responsible for handling data per-

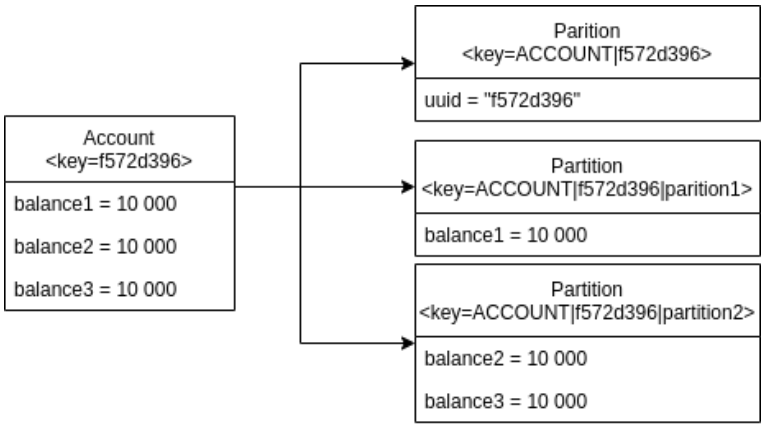

Figure 4.5: Example of the ledger data model resulting from affinity-based partitioning (the ledger data model used in benchmarking [chapter 5])

sistence could be generated. The data model could be parsed from an input UML diagram, and *ExampleAsset* and *ExampleAssetRepository* classes could be generated. This way the developers would not need to be concerned with the specifics of ledger access in HLF, as it could be treated similarly to conventional databases.

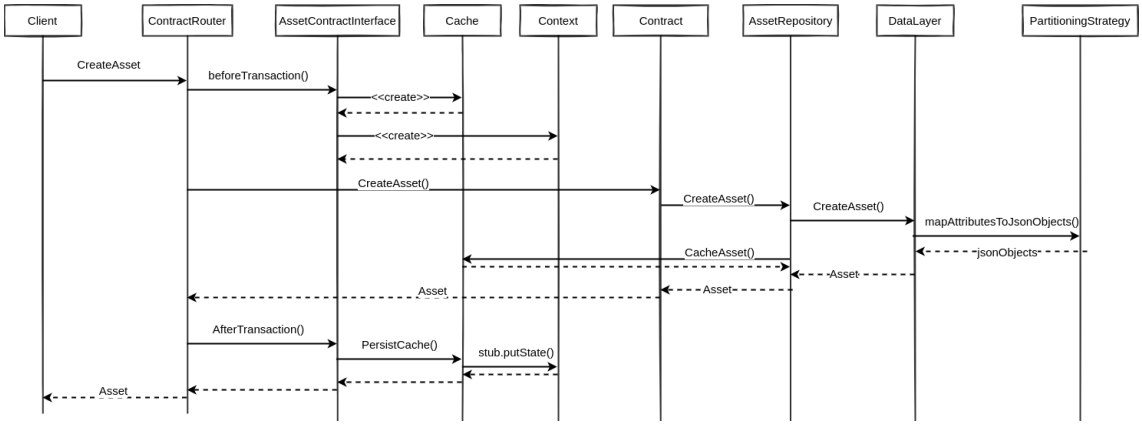

Figure 4.6: Example of a simplified asset creation sequence by the framework

Chapter 5

Empirical validation

The implementation of both partitioning approaches was tested to validate the viability of the techniques, as well as to assess the quantitative differences in performance between each of the data storage strategies. The tests were performed on the private cloud infrastructure of Budapest University of Technology and Economics, and the approaches are compared not only to each other but to the traditional solution as well. This chapter contains detailed explanations of the testing infrastructure, both software and hardware [section 5.1], the testing methodology and the benchmarking campaigns [section 5.2], as well as results and detailed analysis of them [section 5.3].

5.1 System under test

Finding optimal system configuration is not the goal of the tests, as those have been investigated thoroughly [4]. Since the partitioning solutions are applied at the application level, the effects of system configuration-based mitigations are not expected to change when partitioning is applied. However, to be able to compare the effects of my solution to others and ease the reproducibility of the results, it is necessary to detail the configuration of the system under test [Figure 5.1].

Hardware and Software environment The tests were performed in the Budapest University of Technology and Economics' private cloud on 7 QUEMU-based virtual machines (VM) running on eight-core Intel CPUs with 16GB of RAM. No RAMDisk-based Hyperledger Fabric configurations were tested. The operating system was Ubuntu 18.04[1], on which docker[2] 20.10 was installed, and the containerized HLF network components were orchestrated by a Docker Swarm.

Hyperledger Fabric configuration Hyperledger Fabric 2.22 was used, and the network consisted of 3 nodes; 2 peer nodes and an Ordering Service Node. All nodes were installed on separate virtual machines to limit interference. Both peers took part in the endorsement process, and each belonged to a different organization. A simple endorsement policy was chosen to avoid any possible overhead caused by it. A proposal conformed to the endorsement policy if any of the two organizations endorsed it. A batch timeout of 750ms and a max message count of 80 was chosen, as a reasonable middle ground between

[1] https://ubuntu.com/
[2] https://docker.com/

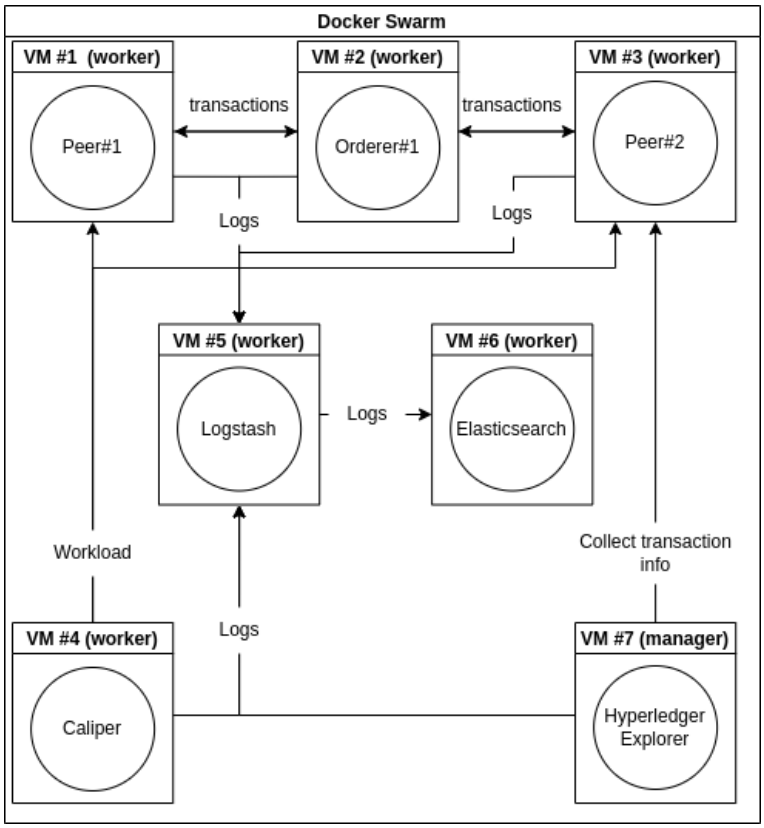

Figure 5.1: Test environment in the cloud

throughput and latency, that many applications might use. The maximum size of a block was set to 99MB, and the preferred max size was set to 30MB. GoLevelDB was used as the state, index and history databases. All remaining configuration options were left on their default values.

Workload generation For workload generation, Hyperledger Caliper [8] was chosen, as it offers easy configuration, and when using multiple workers, it can generate sufficient loads. The benchmarking campaigns utilized four caliper workers and a master placed on a single VM. This was not a limiting factor, however, as Caliper workers are single-threaded javascript processes, therefore the eight core VM was not overutilized.

Data collection Multiple sources of data were collected throughout the tests. An instance of Hyperledger Explorer was installed on a separate VM, which was used to gather data about the read/write sets of the transactions. A custom build of Hyperledger Caliper was utilized to collect data about the throughput, latency and validity of transactions. The partitioning framework implementations were instrumented with the help of a logging solution, and the logs were collected with Logspout[3] and processed with, Logstash[4] and Elasticsearch[5]. The utilization of system resources was monitored on all VMs with cmonitor[6], which was chosen as a lightweight alternative to more popular solutions to minimize overhead on the system.

[3]https://github.com/gliderlabs/logspout
[4]https://www.elastic.co/logstash/
[5]https://www.elastic.co/
[6]https://github.com/f18m/cmonitor

5.2 Case studies and benchmark campaigns

To evaluate the effectiveness of my implementations, I created two simple micro-benchmark scenarios. The scenarios were chosen not only because they showcase the behaviour of the mapping strategies well , but they represent a widespread data modelling pattern used in many real-world applications. Both scenarios simulate a very simple bank account with three balances. The three balances could represent different currencies and sub-accounts like trading accounts, savings accounts etc. The data model [5.1] is overly simplistic to be used in any real-world application, but that does not affect the key versioning. The three numbers could be swapped for more complex nested objects (a typical pattern in Object Oriented Programming), but the key versioning would remain the same. As explained in section 2.3, MVCC conflicts are the product of key versioning, and the scenarios test general access patterns; thus, an effect on performance on this synthetic workload is expected to translate to real-world usages with similar patterns reasonably well.

Account
float balance1
float balance2
float balance3

Table 5.1: Data model of the Bank account

5.2.1 Scenario 1. - Independent attribute access

Description In this scenario, three transacting functions each access a balance independently. These functions read the balance's previous value to ensure sufficient funds for the transaction execution and then update the value accordingly. The createAccount function is used to initialize the ledger with the accounts that will be the subject of the transacting functions. This scenario was chosen to investigate the hypothesis that storing the partitioned object on multiple keys provides more parallel access to the information and that this effect is not outweighed by the cost of more frequent database accesses.

Campaigns The test runs consisted of two phases. First, the ledger was initialized with the assets by calling the createAccount function at a constant rate of 50 transactions per second. The tests were performed with 200, 2000 and 20 000 accounts on the ledger, from which the accounts were selected for transacting in the second phase. The random selection was performed with uniform distribution. To investigate the effect of the framework for a wide range of workloads, the second phase was divided into four sub-phases, each with incrementally increasing loads. A range of accessed attributes was also tested by performing the tests three times by altering the second phase to call only a subset of the possible transaction functions. Only two partitioning frameworks were tested, as in the independent access scenario, the affinity-based strategy resulted in the same mapping as the total partitioning. A total of 36 different configurations were benchmarked on both strategies.

5.2.2 Scenario 2. - Shared attribute access

Description In contrast with the previous scenario, the attributes here can not be accessed independently, as balance1 and balance2 are accessed together by a single transac-

Functions and access patterns		
function name	accessed attribute	access type
createAccount	balance1, balance2, balance3	W
transactWithBalance1	balance1	R+W
transactWithBalance2	balance2	R+W
transactWithBalance3	balance3	R+W

Table 5.2: The transacting functions access the attributes independently

Control variables and possible values	
Variable name	Possible values
workload (tps)	30, 60, 90, 200
accessed attributes	balance1, OR(balance1, balance2), OR(balance1, balance2. balance3)
number of assets	200, 2000, 2000
partitioning framework	total partitioning, no partitioning

Table 5.3: All variables and their possible values in the independent access scenario

tion function. There are two transaction functions: transactWithBalance1, which accesses balance1 and transactWithBalance23 which accesses the remaining two balances. Similar scenarios are also commonplace in many applications, as this access pattern could represent a swap between two balances or a simultaneous update to two related values in any number of other use cases. This scenario was chosen to investigate the effectiveness of the affinity-based partitioning and to examine the potential drawbacks of the total partitioning strategy when the increased parallel accessibility can not be utilized.

Campaigns Similarly to the campaigns in paragraph 5.2.1, this test consisted of two phases. The first initialization phase was identical, and the second phase was divided into four subphases with incrementally increasing workloads again. The tests were conducted on the same range of accounts, and the transacting accounts were chosen in the same manner. However, the functions in the second phase were different, as described in paragraph 5.2.2. These two functions were called in a sequentially alternating manner. A total of 12 configurations were tested on each of the three partitioning options.

Functions and access patterns		
function name	accessed attribute	access type
createAccount	balance1, balance2, balance3	W
transactWithBalance1	balance1	R+W
transactWithBalance23	balance2, balance3	R+W

Table 5.4: Attributes balance1 and balance2 are accessed by a shared function

5.3 Evaluation of results

Throughout the two scenarios, a total of 48 configurations were tested, and the effectiveness of the partitioning approaches was evaluated on five key metrics. Transaction failure rates, throughput and latency, are widely used in the performance evaluation of Online Transaction Processing (OLTP) systems [6], as well as the HLF specific literature [22, 13,

Control variables and possible values	
Variable name	Possible values
workload (tps)	30, 60, 90, 200
number of assets	200, 2000, 2000
partitioning framework	total partitioning, affinity-based partitioning, no partitioning

Table 5.5: All control variables and their possible values in the shared access scenario

9, 7, 23, 21]. These metrics convey a great deal of information, even to the uninitiated reader. Two tool-specific metrics, Read-set and Write-set sizes will be used to examine the potential downsides of my approach.

The rest of this section is structured as follows: The used metrics are formally defined in subsection 5.3.1. In subsection 5.3.2, the results of the benchmarking campaigns for Scenario 1 are discussed, after which a similarly structured evaluation of Scenario 2 follows in subsection 5.3.3.

5.3.1 Used metrics

Transaction failure rate Transaction failure rate is the proportion of the total transaction count after which no update of the ledger occurred. In the following sections, this metric will be presented as a percentage.

Throughput Throughput is defined as the total number of transactions processed by the system over a period of time. This includes failed transactions as well as successful ones. Throughput values will be presented as transactions per second (tps).

Latency Latency is the total elapsed time from the submission of a transaction proposal to the receival of the result notification event by a client. All latency values in this paper are in milliseconds.

Read-set size Read set sizes are usually specified as the number of read keys. However, here the combined size of the business logic JSON[7] object and the key will be used instead, as it has more relevance when evaluating the potential downsides of the partitioning approach. All read set size values are to be interpreted as bytes.

Write-set size Write set size is the combined size of the business objects encoded in JSON and the associated keys. All write-set size values will be presented in bytes.

5.3.2 Scenario 1 Results

Transaction failure rate When all test configurations are examined, it can be concluded that in the case of the independent access scenario, the total partitioning approach was effective at conflict mitigation. The total partitioning approach performed comparably or significantly better than the traditional data model in all cases.

[7] https://www.json.org/

200 assets Generally, high transaction failure rates were observed in test runs with only 200 assets on the ledger [Figure 5.2]. This was expected, as even at the lowest send rate of 30 tps, the probability of an asset receiving two transactions in a single second is approximately 13%. Failure rates increased close to linearly with the increase in send rates. Without partitioning, the failure rates remained constant when the number of accessed attributes was changed. In the case of the total partitioning approach, however, the number of accessed attributes had an enormous impact on the results. At 2 and 3 accessed attributes, the number of conflicts was decreased by close to 1/2 and by 1/3, respectively. At lower send rates, the failures decreased by more than $1/n$, where n is the number of accessed attributes. These results are similar to what the theory in section 4.1 suggests might be possible. However, the factor of reduction shows a slight diminishing trend as the send rates are increased. It is important to note that while no significant reduction in failure rates occurred when only one attribute was accessed, the added complexity of the partitioning approach did not cause any deterioration relative to the conventional approach.

Figure 5.2: Transaction failures for 1, 2 and 3 written keys at 200 assets.

2000 assets With a 10x increase in assets on the ledger from 200 to 2000, a decrease of the same magnitude can be observed in the percentage of failed transactions. [Figure 5.3]. Despite the decrease in the number of conflicts, the improvements achieved by the total partitioning approach are similar in magnitude to one observed in the tests with 200 assets. At a single accessed attribute, performance was close to identical, while at 2 and 3 attributes the failure rates decreased by nearly 2x and 3x. Curiously, the same diminishing trend with the increase in send rates can not be observed in every case.

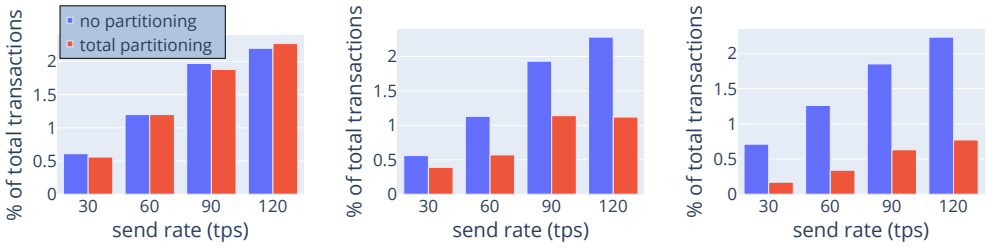

Figure 5.3: Transaction failures for 1, 2 and 3 written keys at 2000 assets.

20 000 assets In the case of 20000 assets, the number of failures due to MVCC conflicts was generally low, even without a partitioning approach. [Figure 5.4] When the number of assets was increased by 100x relative to the first tests, the amount of failed transactions

decreased proportionally. Although the probability of conflict was low with all send rates, the partitioning approach further decreased them.

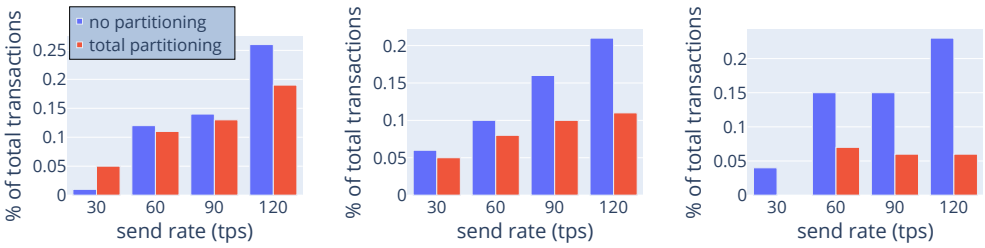

Figure 5.4: Transaction failures for 1, 2 and 3 written keys at 20000 assets.

Latency Despite the added complexity of the partitioning framework, latency remained unchanged throughout all test configuration options [Figure 5.5]. As observed by Thakkar *et al.* [22], latency substantially increases with increased send rates, only above the saturation rate of the system. In these benchmarking campaigns, this rate was avoided intentionally. In previous experiments I observed less consistent system performance at such high workloads, which could potentially interfere with the test results. As mentioned at the beginning of this section 5.1, finding the optimal system configuration for MVCC mitigation or performance modelling HLF is not the aim of my report.

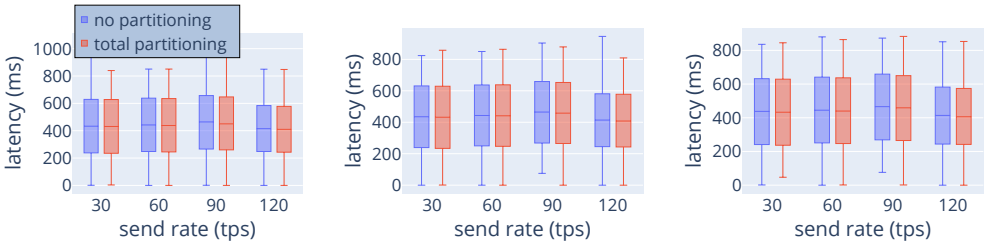

Figure 5.5: Latency for 1,2 and 3 accessed attributes across all asset counts

Throughput Throughput values were essentially identical for the tested approaches, suggesting that the added complexity of the total partitioning approach did not affect the throughput [Figure 5.6]. It is worth noting that the send rates were closely matched by the throughput values in all configurations, as the system was not loaded above the saturation level.

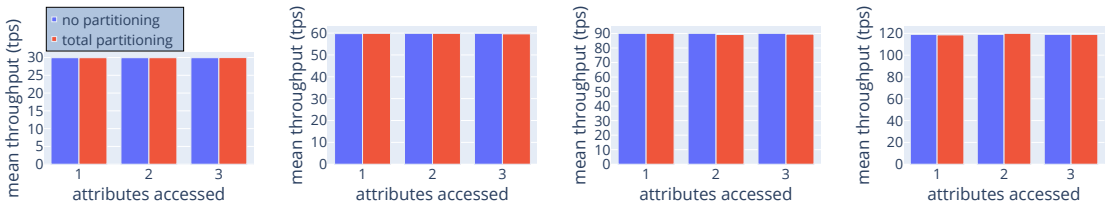

Figure 5.6: Mean throughput values at 30, 60, 90, 120 tps send rates, across all asset counts

Read-set sizes In the case of the transacting functions, read set sizes have increased by nearly 76%, while for the account creation, the same metric shows nearly 4.5x. The use of composite keys causes the 4.5x increase in size in the case of the creation function. Composite keys store metadata about the values that are stored under them. In the non-partitioning approach, composite keys were not utilized, as the complexity of the data structure does not require it. However, this significant proportional increase is not concerning, as when the accounts are created, the keys are read only to check for an account with the same key. If the account does not exist, which was always the case in this scenario, only the key is read, which might be proportionally large, but the absolute difference between the values is unlikely to be a limiting factor of system performance. This notion is further reinforced by the latency and throughput data, where the more significant absolute differences between the approaches showed no impact on performance.

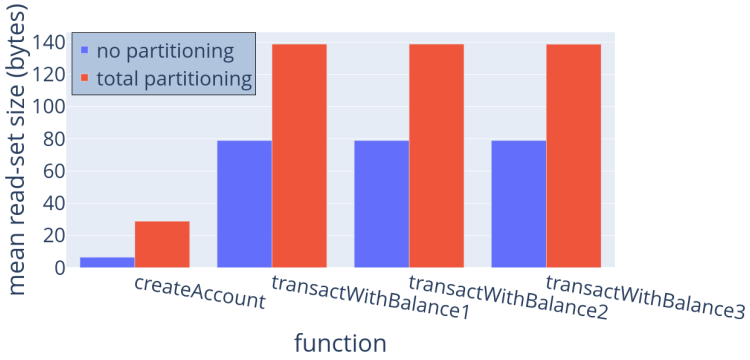

Figure 5.7: Read-set sizes of all transaction functions in Scenario 1

Write-set sizes The account creation function of the total partitioning approach saw a ~2.5x increase in mean write set sizes. This is a drawback of the total partitioning approach. By creating a different partition for each attribute, the same information is stored with increased metadata. If efficient storage is a necessity, other approaches should be considered. In the case of the transacting functions, a ~39% reduction in write sizes was observed. This is a benefit of using the total partitioning approach, as only the necessary subsets of the asset are updated. However, in a benchmark of this scale, none of the differences appeared to impact system performance. A more complex benchmark scenario might show a more significant effect of read and write sizes on performance.

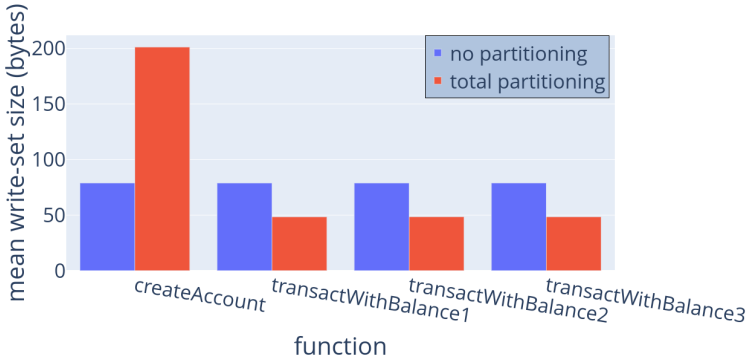

Figure 5.8: Write-set sizes of all transaction functions in Scenario 1

5.3.3 Scenario 2 Results

Transaction failure rate Like in the previous scenario, the failure rates decreased by 10x every time the same amount increased the asset count. [Figure 5.9] In the case of the traditional approach, essentially identical failure rates can be observed. In the case of the total partitioning approach, this scenario saw similar failure rates as the independent access scenario with two accesses attributes. This was expected, as the previous benchmarking campaigns showed that the added complexity did not have an adverse effect on latency, and in the case of total partitioning, the difference between the independent access scenario with two attributes accessed and this one is a single database access for every call to the transactWithBalance23 function. Considering these facts, it is not surprising to see that the affinity-based approach failed to significantly outperform the total partitioning approach in failure rate reduction. The difference in failure rates between the two approaches is negligible and is most likely the result of the random account selection during workload generation.

Figure 5.9: Transaction failures at 200, 2000 and 20 000 assets

Latency Similarly to the independent access scenario, the partitioning strategies did not significantly impact the latency. This was not surprising considering the facts discussed in paragraph 5.3.3. Interestingly, in the test run with 200 assets, there was an irregularity in latency with the traditional approach. This is most probably caused by a sudden demand on the university cloud. It is also worth noting that the affinity-based solution performed better, but this was not statistically significant and was well within the margin of error. The latency values did not increase with the increase in send rate, most probably for the reason discussed in paragraph 5.3.2.

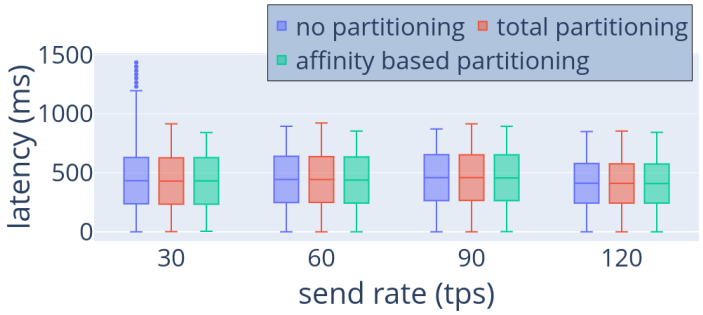

Figure 5.10: Latency for 1,2 and 3 accessed attributes across all asset counts

Throughput As observed in the previous scenario [Figure 5.6], there was no significant difference in throughput values when a partitioning approach was applied. This scenario was no different [Figure 5.11]. All approaches' and configurations' throughput closely matched the send rate.

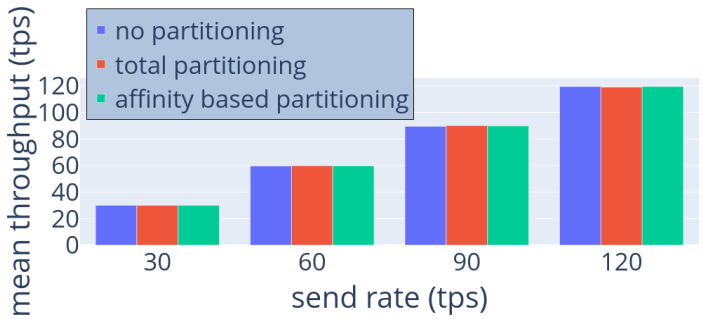

Figure 5.11: Mean throughput values at 30, 60, 90, 120 tps send rates, across all asset counts

Read-set sizes Read-set sizes are highly elevated. The more complex key structure can partly explain the larger read-set sizes. This is most visible at the createAccount function, where the difference in read-set sizes comes from the key sizes alone. In the case of the transacting functions, the cause of the elevated sizes comes from the increase in metadata. However, as seen previously [Figure 5.10], with a high bandwidth connection, the larger read-set sizes did not cause any deterioration in latency values.

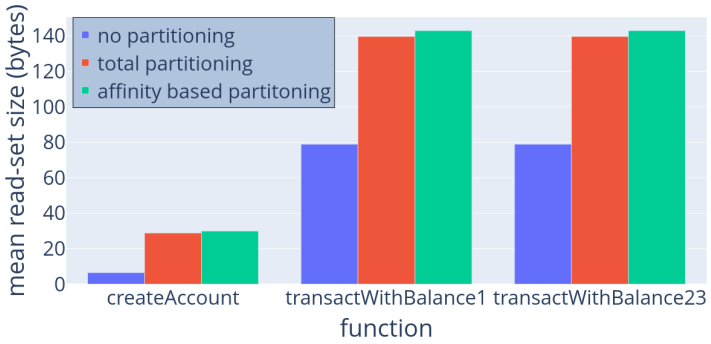

Figure 5.12: Read-set sizes of all transaction functions in Scenario 2

Write-set sizes Unlike read-set sizes [Figure 5.12], the size of the write-sets did not increase for all transaction types [Figure 5.13]. In the case of the createAccount function, significant write-set size increases were observed for both partitioning implementations. This is a potential drawback of the partitioning approaches, as the larger write-set sizes result in more used storage. In the cases of the transacting functions, the write set sizes did not increase nearly as much. On the contrary, write set sizes decreased when only a single balance was accessed. This is the result of updating only a subset of the data that is stored as a single object in the case of the traditional approach. This did not result in smaller write set sizes when two balances were accessed, as the increased amount of metadata had a more pronounced effect than the finer-grained access since the data model is simplistic.

5.3.4 Conclusion of benchmark results

Overview of the test results Both partitioning versions heavily outperformed the traditional approach in transaction failure rates in every scenario where more than one attribute was accessed. When only a single attribute is accessed, no improvements can

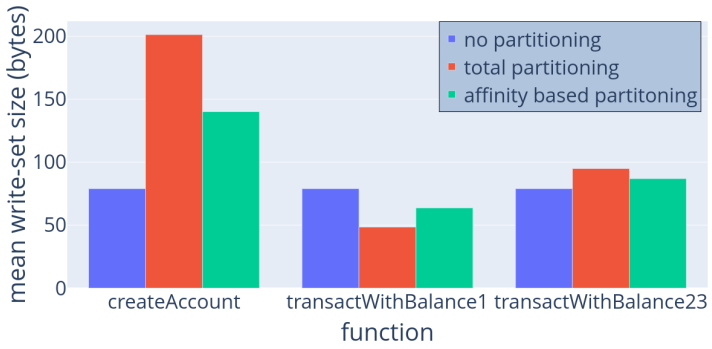

Figure 5.13: Write-set sizes of all transaction functions in Scenario 2

be made by partitioning along attributes, as explained in section 4.1. In the scenarios where more than 1 attribute was accessed, transaction failure rates generally decreased by $1/n$, where n is the number of accessed attributes. The throughput metrics (which included the failed transactions), along with the latency values, did not change significantly. Mean read-set sizes increased significantly in all configurations where a partitioning approach was utilized, while write-set sizes were comparable or reduced. The partitioning approaches effectively decreased the failure rates, and the overhead of the added complexity did not impact throughput and latency metrics. The affinity-based partitioning approach did not perform better than the total partitioning approach in conflict mitigation, which was expected. The affinity-based approach performed comparably to the total partitioning approach in latency and throughput metrics while using less storage. This can be considered a success, as it can not be expected to outperform the Total partitioning approach due to the theory discussed in section 4.1.

Shortcomings of the benchmarking campaigns Potential shortcomings of the benchmark campaigns are the simplistic data model and synthetic workload. While more complex data models would have likely resulted in larger comparative storage use for the partitioning approaches, the metadata would likely also be a smaller portion of the read and write-sets, and the finer-grained access to attributes would also further decrease the read and write-set sizes relative to the traditional approach. A non-synthetic workload might see the affinity-based approach fall behind the total partitioning one in failure rate reduction, as there might be more overlap between the transaction functions regarding accessed partitions. The micro-benchmark successfully showed the approach's viability on often-used patterns, but more comprehensive testing is needed with workloads more representative of real-world use. For this reason, I plan to implement and test TPC-C [5] as a Java chaincode. TPC-C is a standardized benchmark for OLTP systems which was designed to be representative of real-world usage.

Shortcomings of the partitioning approaches A potential shortcoming of the partitioning approaches might be the storage used. They create more ledger objects, which result in more world state entries leading to longer database searches when objects are retrieved. However, my tests did not observe the adverse effects of the longer database searches.

Chapter 6

Dynamic endorsement delaying

6.1 Motivation behind the proposal

Numerous techniques try to mitigate the effect of MVCC conflicts, most focusing on optimizing a single aspect. Even when used alone, these efforts can provide great optimizations, but a layered approach combining the solutions could result in even larger improvements. However, most of these solutions are incompatible with each other and the unaltered protocol. While specialized use cases which require total optimization of a single aspect exist, most applications do not fall into this category. Most applications would benefit more from an approach that combines the effect of the optimizations. Remaining compatible with the unaltered protocol has further benefits regarding adoption. A solution that can be integrated iteratively and does not require the overhaul of existing systems is more likely to be adopted by businesses. Motivated by these reasons, I propose an approach which is compatible with the unaltered protocol and can be used as part of a multi-layered MVCC conflict mitigation strategy.

6.2 Core idea

In HLF v2.4 Fabric Gateway (FG) was introduced, which provides a simplified interface for writing client applications. By creating a service which offers the same API as the FG, to which clients can connect transparently, it is possible to build an organization-level optimization layer on top of the HLFs network. The proposed service forwards clients' proposals to peers for endorsement, exactly like the FG, but the possibility of delaying the endorsement is a powerful tool. By delaying the endorsement of a proposal, it can be executed on a world state that contains updated versions of the keys accessed by the proposal. This would result in fewer failed transactions. The service can inspect the inputs passed to the chaincode function and decide to delay the endorsement if it detects a high probability of conflict. To provide accurate and reliable predictions, there are multiple data sources available to the service besides the current transaction:

- *Past transactions:* By connecting to the network as a client itself, the service is able to retrieve all past transaction data.
- *Pending transactions of the Organization:* Transactions that are proxied through the service are stored in a cache.

These can be used as inputs to different algorithms that estimate the probability of a transaction being an MVCC conflict.

6.3 Architecture

The Optimization layer is situated between the client and the unaltered HLF network. The original architecture of HLF remains the same, while the organizations' clients connect to the optimization layer's proxy gateways. The optimization layer consists of two main components, the Proxy gateway and the Cache service. The components will be discussed in their subsections below.

6.3.1 Proxy Gateway

The Proxy Gateway manages the communication between the client, the Cache service and the HLF network. There can be many instances of the proxy gateway; it is not a centralized service. It exposes the full gRPC API of the FG and forwards most calls to the FG unchanged. The simplified version of the original communication flow can be seen on [Figure 6.1], and the altered version is depicted on [Figure 6.2]. Whenever it receives a proposal submitted for endorsement, it forwards it to the Cache service. The cache service saves the transaction and processes it to estimate the probability of conflict. After the necessary time has passed according to the probability, the Cache service signals to the proxy service that the proposal can be submitted for endorsement. The proposal is forwarded to the FG, where the FG gathers the endorsements from the endorsing peers. The result is then returned to the Proxy Gateway, which forwards it to the client. After this step, no changes are made. All client requests are forwarded to FG, and the responses of the FG are sent back to the client.

6.3.2 Cache service

The Cache service is responsible for storing the pending transactions and estimating the probability of conflict for the incoming ones. After the decision has been made to delay it or not and the potential delay time has passed, the Cache service notifies the Proxy Gateway that the delay duration has expired. The Cache service removes a proposal from storage once it has received the Commit event of the block that contains a transaction. This is done

6.4 Estimating MVCC conflict probability

Chaincodes usually have a limited number of functions available for invoking, and their execution must be deterministic, or they will fail validation. By their nature, chaincode functions usually perform the same types of ledger accesses when invoked. This means the invoked function's name and parameters passed in can be used to create rules that probabilistically define when a transaction function depends upon another one.

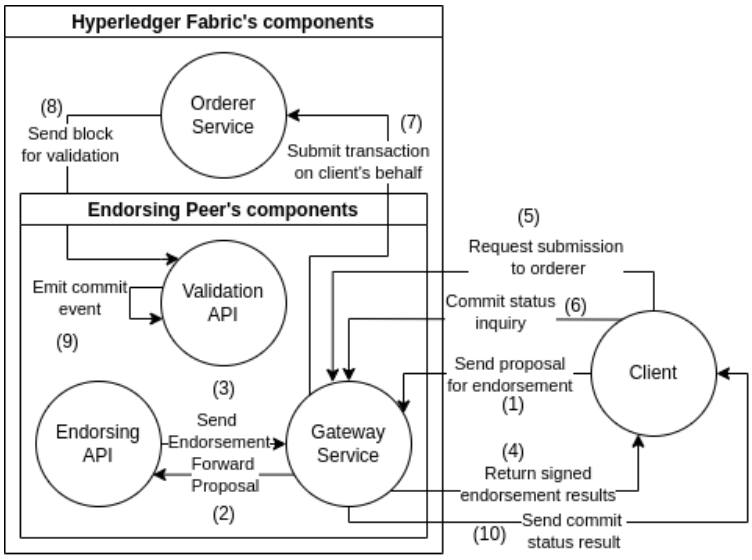

Figure 6.1: Simplified flow of communication between HLF's components and the client

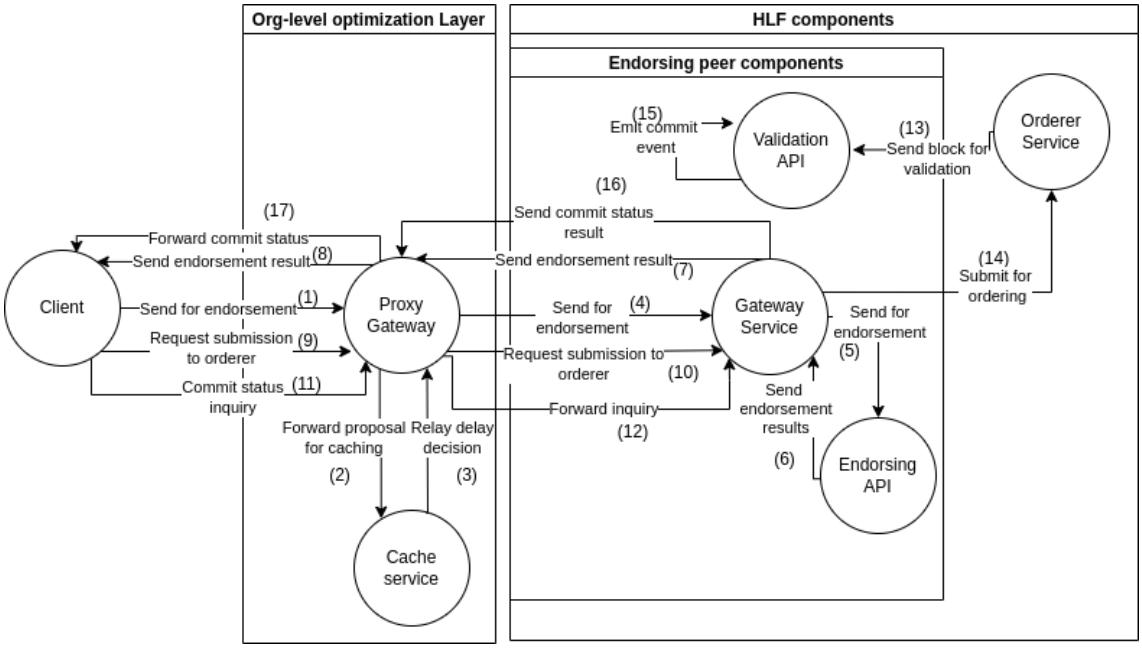

Figure 6.2: Simplified flow of information between the components of the Org-level Optimization layer and HLF. Note that only steps 2 and 3 are new. Steps 12,10,4,8,17 are forwarded requests; other steps are unchanged.

6.4.1 Core idea

As mentioned in section 6.2, the full transaction history is available for the cache service, along with the read and write sets of the past transactions and the proposal input that resulted in the generation of the read and write-sets. In the full transaction history, it is possible to find for every failed conflicting transaction the transactions it conflicted with. This can be done by finding every valid transaction which the failed transaction depended upon, whose commit height is newer than the version of the key that is present in both sets. By finding every failure causing dependency and examining which functions were invoked, it is possible to find the most problematic functions the failed transaction's

function depends on. Creating rules based only on the functions' names is not sufficient for reliable predictions. For such rules to be created, the parameters the function was invoked with are essential data. Consider the following example of two functions:

Transaction function *buyCar* takes three arguments $(carID, buyerID, price)$, while *takeLoan* takes $(loanerID, loaneeID, amount)$ as inputs. Although the parameter names are different, the two functions can deal with the same ledger object, which is a wallet of a person. In this case, if $buyerID$ and $loaneeID$ are the same, the functions will modify the balance of the person represented by $buyerID$ and $loaneeID$. Thus function *buycar* is dependent upon function *takeLoan* and vice versa. To avoid MVCC conflicts, in this case, a rule can be created that states that if there is a pending *takeLoan* function with a $loaneeID$ that equals the $buyerID$ parameter of the incoming *buyCar* transaction, *buyCar* needs to be delayed until the result of *takeLoan* is committed. In the case of this example, the resulting transactions were guaranteed to change the value of the balance if the transaction was successful. However, if the *buyCar* function were to contain conditional logic that would result in the balance only getting updated in some cases, then the previous rule would result in the delayment of *buyCar* in cases where it is unnecessary. To avoid such cases, a confidence value c can be used:

$c = n_c/n_d \times n_c/n_m$

where :

- n_c: is the number of cases where a *buyCar* transaction was invalidated because of a *takeLoan* transaction and the parameters $loaneeID = buyerID$.
- n_d: is the total number of cases where *buyCar* was dependent on a recent *takeLoan* transaction and the parameters $loaneeID = buyerID$.
- n_m: is the total number of cases where a *buyCar* transaction was invalidated because of a *takeLoan* transaction.

Recency here is the number of past blocks that are examined for the dependency. The optimal value of this is different for every system configuration, and this is an input parameter of the algorithm. The usage of a confidence value is advantageous because, in the case of multiple matching parameters, finding the meaningful match is not possible without further metadata. In the previous example, if *price* and *amount* were to match without using a confidence value, transactions working on unrelated business objects with the same price would get delayed. However, in this example, since the third parameters of the functions are not the ones leading to conflicts, it is reasonable to assume that given a large enough sample of transaction history, the rule resulting from the parameters *price* and *amount* matching would have a lower confidence value, as the parameters might match in cases where there are no conflicts and might not match in cases with conflicts. This effect is shown in table 6.1, where a single conflicting transaction where $price = amount$ results in a rule with a confidence value of 1, but after three more conflicting transactions where $price \neq amount$ the confidence value of the "incorrect" rule is only 0.25. By assigning a confidence value to every rule, the algorithm becomes configurable, as system operators can choose a threshold confidence value p, where if $p > c$, the delay is not applied. This is beneficial, as the approach can be tailored to use cases with different degrees of tolerance for the added latency and the number of failing transactions.

n_m			1
parameters	$carID$	$buyerID$	$price$
$loanerID$	0	0	0
$loaneeID$	0	1	0
$amount$	0	0	1

n_m			4
parameters	$carID$	$buyerID$	$price$
$loanerID$	0	0	0
$loaneeID$	0	4	0
$amount$	0	0	1

n_m			1
parameters	$carID$	$buyerID$	$price$
$loanerID$	0	0	0
$loaneeID$	0	1	0
$amount$	0	0	1

n_m			4
parameters	$carID$	$buyerID$	$price$
$loanerID$	0	0	0
$loaneeID$	0	4	0
$amount$	0	0	1

n_m			1
parameters	$carID$	$buyerID$	$price$
$loanerID$	0	0	0
$loaneeID$	0	1	0
$amount$	0	0	1

n_m			4
parameters	$carID$	$buyerID$	$price$
$loanerID$	0	0	0
$loaneeID$	0	1	0
$amount$	0	0	0.25

Table 6.1: The n_c (1st row), n_d (2nd row) and c values (3rd row) of each rule after one (left) and four (right) conflicts

6.4.2 Input matching approach

By creating rules between every possible parameter pairing of every dependent function in a transaction history, a knowledge base can be defined that contains rule-sets for every function. The rule-sets can be queried with proposal input pairs. A proposal input pair consists of the incoming transaction's proposal input and one of the pending transaction's proposal inputs. If a query returns a rule with a confidence value higher than the threshold value set by the operators for any of the pending proposal inputs, no more queries are necessary; the incoming proposal should be delayed. Only the querying needs to be executed for every incoming transaction, which has an algorithmic complexity of $O(n \times m \times l)$ if a linear search is used, where n is the number of functions the incoming transaction function depends upon, m is the maximum number of parameters any function has in the knowledge base, and l is the number of pending transactions.

- Let f_i and f_j be transaction functions such that f_i depends upon f_j.
- Let n_i be the number of input parameters f_i has and n_j be the number of parameters of f_j.
- Let p_i be a parameter of f_i and p_j be a parameter of f_j
- Let r_{p_i,p_j} be a rule declaring an incoming proposal of f_i should be delayed with confidence value $c^{f_i,f_j}_{r_{p_i,p_j}} = n^{f_i,f_j}_{c_{p_i,p_j}} / n^{f_i,f_j}_{d_{p_i,p_j}} \times n^{f_i,f_j}_{c_{p_i,p_j}} / n^{f_i,f_j}_{m}$ where:
  - $n^{f_i,f_j}_{c_{p_i,p_j}}$ is the number of times a transaction of f_i was invalidated because of a dependency on f_j such that $p_i = p_j$.
  - $n^{f_i,f_j}_{d_{p_i,p_j}}$ is the number of times function f_i depended on a recent f_j such that $p_i = p_j$.
  - n^{f_i,f_j}_{m} is the number of times a transaction of f_i was invalidated because of a dependency on f_j.

- Let rule-set R_{f_i,f_j} be a set of rules containing $n_i \times n_j$ rules, one for each possible $\{p_i, p_j\}$ pair.
- Let knowledge base B be a set of rule-sets for every possible f_i,f_j dependent pairs of functions in a chaincode.

$$D(f_i, t_j) = \begin{cases} 1 \leftarrow \exists \rho \in R_{f_i,f_j} \text{with matching parameterization and } c_{r_{p_i,p_j}}^{f_i,f_j} \geq \tau, \\ 0 \text{ otherwise} \end{cases}$$

where τ is a threshold parameter specified by the system configurator

The knowledge base can be stored in a structure similar to the one depicted on [Listing 6.1]. On the depicted example, *function1* and *function2* are both transaction functions with two parameters. The rules are the numbers in the "confidences" matrix, where each entry belongs to a parameter pair. The conflicts and the dependencies between the functions with matching parameters are both counted, as well as the total number of conflicts, from which the confidence values can be calculated by the equation shown earlier in this subsection. The structure is stored in memory for querying performance. Because of the combinatorial nature of the problem, a large number of rules are created for a relatively small number of functions. However, since chaincodes are not extremely complex, by my estimations, the memory requirements are not expected to be a problem. The hierarchical structure of the knowledge base results in faster queries; instead of one linear search of $O(n^2 \times m)$ algorithmic complexity, three linear searches are performed, each with a complexity of $O(n), O(n), O(m)$, where n is the number of defined functions and m is the maximum number of parameters any function has. Building the knowledge base is computationally intensive, as it requires $O(n \times m \times l)$ iterations, where n is the number of transactions in the transaction history, m is the recency block, parameter and l is the maximum number of transactions in each block.

```
{
"function1":{
        "function1":{
            "total_conf":4,
            "conflicts":[[0,0],[2,1]],
            "dependencies":[[0,0],[2,2]],
            "confidences":[[0,0],[0.5,0.125]]
                    },
        "function2":{
            "total_conf":5,
            "conflicts":[[0,0],[1,0]],
            "dependencies":[[0,0],[4,0]],
            "confidences":[[0,0],[0.05,0]]
        }
    },
"function2":{...}
}
```

Listing 6.1: Example of the structure of the knowledge bases

6.4.3 Further possibilities

The input matching approach presented in subsection 6.4.2 is a general approach that can be used without any information about the data model and structure used by the chaincode. However, it is possible to implement a solution that utilizes the metadata used in the Affinity-based partitioning approach presented in section 4.4. This could provide more accurate predictions with potentially less computational effort. It is also possible to use machine learning algorithms to provide predictions. In the future, I intend to implement all three of these approaches and empirically validate them to compare their effectiveness.

Chapter 7

Conclusions and future work

7.1 Conclusions

In my report, I created a taxonomy of the current MVCC conflict mitigation literature and identified a previously overlooked class of mitigation techniques. I contributed two possible approaches to this category of data storage-based techniques. The approaches prioritize different aspects and are suitable for a wide range of use cases. The total partitioning approach prioritizes the reduction of failure rates above storage efficiency. The affinity-based partitioning approach results in more efficient storage, while it can potentially match the failure reduction effects of the partitioning approach. I implemented the proposed approaches as part of a prototype framework and evaluated them on multiple micro-benchmark scenarios. The results of the evaluations are promising; however, they require a more complex test scenario. I contributed to the protocol extension category of mitigation techniques with the proposal of a novel endorsement delaying framework. I provide a potential solution for both the architectural structure of the framework as well as the algorithm to select the potentially problematic transactions for delayment.

7.2 Future work

In the future, I plan to implement TPC-C [5] as a Java chaincode and perform comprehensive testing on the improved prototype implementations. I plan to implement code generation as a part of the framework, resulting in a better chaincode development experience than the current implementation. I plan to release this improved version as open-source software for public use. I also intend to implement the proposed endorsement delaying framework with multiple estimation algorithms. The empirical evaluation of these implementations is also a possible future work.

Acknowledgements

I am deeply thankful for the help of my advisors, Attila Klenik and dr. Imre Kocsis. Their advice was invaluable and my work would not have been possible without their unrelenting support.

List of Figures

List of Tables

Bibliography

[1] Ali Alzubaidi, Karan Mitra, and Ellis Solaiman. Smart contract design considerations for sla compliance assessment in the context of iot. 2021. DOI: 10.1109/SmartIoT52359.2021.00021.

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Srinivasan Muralidharan, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Chet Murthy, Christopher Ferris, Gennady Laventman, Yacov Manevich, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. volume 2018-January. Association for Computing Machinery, Inc, 4 2018. ISBN 9781450355841. DOI: 10.1145/3190508.3190538.

[3] Vitalik Buterin. *Ethereum: A next-generation smart contract and decentralized application platform.* 2014. URL https://github.com/ethereum/wiki/wiki/White-Paper.

[4] Jeeta Ann Chacko, Ruben Mayer, and Hans Arno Jacobsen. Why do my blockchain transactions fail?: A study of hyperledger fabric. 2021. DOI: 10.1145/3448016.3452823.

[5] Transaction Processing Performance Council. *TPC-C benchmark.* 2010. URL https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[6] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment*, 12:169–182, 10 2018. ISSN 2150-8097. DOI: 10.14778/3282495.3282502. URL https://dl.acm.org/doi/10.14778/3282495.3282502.

[7] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. *Proceedings of the 2017 ACM International Conference on Management of Data*, Part F127746: 1085–1100, 3 2017. ISSN 07308078. DOI: 10.1145/3035918.3064033. URL http://arxiv.org/abs/1703.04057.

[8] Hyperledger Foundation. Hyperledger caliper 2022, 2022. URL https://hyperledger.github.io/caliper/. [Online;accessed 26-October-2022].

[9] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. 2019. DOI: 10.1109/BLOC.2019.8751452.

[10] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. Xox fabric: A hybrid approach to blockchain transaction execution. 2020. DOI: 10.1109/ICBC48266.2020.9169478.

[11] Zsolt István, Alessandro Sorniotti, and Marko Vukolić. Streamchain: Do blockchains need blocks? 2018. DOI: 10.1145/3284764.3284765.

[12] William T. McCormick, Paul J. Schweitzer, and Thomas W. White. Problem decomposition and data reorganization by a clustering technique. *Operations Research*, 20, 1972. ISSN 0030364X. DOI: 10.1287/opre.20.5.993.

[13] Takuya Nakaike, Qi Zhang, Yohei Ueda, Tatsushi Inagaki, and Moriyoshi Ohara. Hyperledger fabric performance characterization and optimization using goleveldb benchmark. pages 1–9. IEEE, 5 2020. ISBN 978-1-7281-6680-3. DOI: 10.1109/ICBC48266.2020.9169454. URL https://ieeexplore.ieee.org/document/9169454/.

[14] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. December 2008. URL https://bitcoin.org/bitcoin.pdf.

[15] Qassim Nasir, Ilham A. Qasse, Manar Abu Talib, and Ali Bou Nassif. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks*, 2018, 2018. ISSN 19390122. DOI: 10.1155/2018/3976093.

[16] Pezhman Nasirifard, Ruben Mayer, and Hans Arno Jacobsen. Fabriccrdt: A conflict-free replicated datatypes approach to permissioned blockchains. 2019. DOI: 10.1145/3361525.3361540.

[17] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9:680–710, 12 1984. ISSN 15574644. DOI: 10.1145/1994.2209.

[18] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. Performance analysis of private blockchain platforms in varying workloads. Institute of Electrical and Electronics Engineers Inc., 9 2017. ISBN 9781509029914. DOI: 10.1109/ICCCN.2017.8038517.

[19] Pingcheng Ruan, Dumitrel Loghin, Quang Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. 2020. DOI: 10.1145/3318464.3389693.

[20] Ankur Sharma, Divya Agrawal, Felix Martin Schuhknecht, and Jens Dittrich. Blurring the lines between blockchains and database systems: The case of hyperledger fabric. 2019. DOI: 10.1145/3299869.3319883.

[21] Harish Sukhwani, Nan Wang, Kishor S. Trivedi, and Andy Rindos. Performance modeling of hyperledger fabric (permissioned blockchain network). 2018. DOI: 10.1109/NCA.2018.8548070.

[22] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. 2018. DOI: 10.1109/MASCOTS.2018.00034.

[23] Arnold Woznica and Michal Kedziora. Performance and scalability evaluation of a permissioned blockchain based on the hyperledger fabric, sawtooth and iroha. *Computer Science and Information Systems*, 19, 2022. ISSN 24061018. DOI: 10.2298/CSIS210507002W.

[24] Lu Xu, Wei Chen, Zhixu Li, Jiajie Xu, An Liu, and Lei Zhao. Solutions for concurrency conflict problem on hyperledger fabric. *World Wide Web*, 24, 2021. ISSN 15731413. DOI: 10.1007/s11280-020-00851-6.

[25] Xiaoqiong Xu, Xiaonan Wang, Zonghang Li, Hongfang Yu, Gang Sun, Sabita Maharjan, and Yan Zhang. Mitigating conflicting transactions in hyperledger fabric-permissioned blockchain for delay-sensitive iot applications. *IEEE Internet of Things Journal*, 8, 2021. ISSN 23274662. DOI: 10.1109/JIOT.2021.3050244.

[26] Shenbin Zhang, Ence Zhou, Bingfeng Pi, Jun Sun, Kazuhiro Yamashita, and Yoshihide Nomura. A solution for the risk of non-deterministic transactions in hyperledger fabric. pages 253–261. IEEE, 5 2019. ISBN 978-1-7281-1328-9. DOI: 10.1109/BLOC.2019.8751453. URL https://ieeexplore.ieee.org/document/8751453/.

Appendix

```java
public class DemoContract implements AssetContractInterface {
    private static final DemoAssetRepository repo = DemoAssetRepository.getInstance();

    @Transaction()
    public boolean demoAssetExists(Context ctx, String uuid) {
        return repo.demoAssetExists(ctx, uuid);
    }

    @Transaction()
    public void createDemoAsset(Context ctx, String uuid, Double pocket1, Double pocket2,
    Double pocket3) {
        repo.createDemoAsset(ctx, uuid, pocket1, pocket2, pocket3);
    }

    @Transaction()
    public void transactWithPocket1(Context ctx, String uuid, Double amount) {
        DemoAsset asset = repo.readDemoAsset(ctx, uuid);
        if (asset.getPocket1Attribute() + amount < 0) {
            throw new ChaincodeException("Not enough funds in pocket1 to finish
    transaction");
        }
        asset.setPocket1Attribute(asset.getPocket1Attribute() + amount);
    }
}
```

Listing A.0.1: Contract implementation using the partitioning approach

```java
@AssetType(type = "DemoAsset")
public class DemoAsset extends AssetBase {

    @Attribute
    public Double pocket1;

    DemoGeneratedAsset(SparseCachedContext ctx, String uuid) {
        super(ctx, uuid);
    }

    DemoGeneratedAsset(AssetBase nestHost) {
        super(nestHost);
    }

    public Double getPocket1Attribute() {
        try {
            return dataLayer.getAttribute(DemoAsset.class, this, "pocket1", () -> pocket1,
     null);
        } catch (NoSuchFieldException | SecurityException | JsonProcessingFailureException
     e) {
            throw new ChaincodeException(e);
        }
    }
```

```
}
```

Listing A.0.2: Asset with implementing the logical data model for the partitioning approach

```
// the asset used in this contract is a standard Java class with getters and setters
public class DemoAssetContract implements ContractInterface {

    @Transaction()
    public boolean demoAssetExists(Context ctx, String demoAssetId) {
        byte[] buffer = ctx.getStub().getState(demoAssetId);
        return (buffer != null && buffer.length > 0);
    }

    @Transaction()
    public void createDemoAsset(Context ctx, String uuid, Double pocket1, Double pocket2,
    Double pocket3) {
        boolean exists = demoAssetExists(ctx, uuid);
        if (exists) {
            throw new RuntimeException("The asset " + uuid + " already exists");
        }
        DemoAsset asset = new DemoAsset();
        asset.setUuid(uuid);
        asset.setPocket1(pocket1);
        asset.setPocket2(pocket2);
        asset.setPocket3(pocket3);
        ctx.getStub().putState(uuid, asset.toJSONString().getBytes(UTF_8));
    }

    @Transaction()
    public void transactWithPocket1(Context ctx, String uuid, Double pocket1) {
        DemoAsset asset = readDemoAsset(ctx, uuid);
        if ((asset.getPocket1() + pocket1) >= 0) {
            updateDemoAsset(ctx, uuid, asset.getPocket1() + pocket1, null, null);
        }
    }
}
```

Listing A.0.3: Contract implementation using the traditional approach

```
transactions,1,2,3,accesses
t1,0,1,0,30
t2,1,0,1,50
```

Listing A.0.4: Example input csv used by the python script

The prototype framework code is available here[1].

[1] https://drive.google.com/drive/folders/108YiSVFuVORp9drXjYTGlvDham7gwtbG?usp=sharing