# Snapshot: Friend or Foe of Data Management?

–

# On Optimizing Transaction Processing in Database and Blockchain Systems.

Ankur Sharma

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of the Saarland University

Saarbrücken, Germany
February, 2020

| | |
|---|---|
| Dean of the Faculty | Univ.-Prof. Dr. Thomas Schuster |
| Day of Colloquium | 03.06.2020 |

Examination Board:

| | |
|---|---|
| Chairman | Prof. Dr. Jörg Hoffmann |
| Adviser and First Reviewer | Prof. Dr. Jens Dittrich |
| Second Reviewer | Prof. Dr. Gustavo Alonso |
| Third Reviewer | Prof. Dr. Wolfgang Lehner |
| Academic Assistant | Dr. Felix Martin Schuhknecht |

*To my family*

# Acknowledgements

Firstly, I would like to express my sincere gratitude and thank my advisor Prof. Dr. Jens Dittrich, for his continuous support of my doctoral research, his patience, motivation, and excellent guidance. His continuous inspiration and invaluable mentorship have been one of the essential pillars of this research. I want to thank him for giving me enough freedom to pursue a system-oriented research direction, which required an exceptional level of patience from all perspectives. I have learned a lot from him over these years, be it his research style, his unmatched art of writing, or his unique presentation skill. He has been a great guide and friend during this journey.

I would also like to express my thanks to my reviewers, Prof. Dr. Gustavo Alonso and Prof. Dr. Wolfgang Lehner, for agreeing to review this dissertation. I could not have asked to get more prestigious and outstanding researchers into my doctoral committee.

Further, I would like to thank my co-authors Dr. Felix Martin Schuhknecht and Divya Agrawal. They are an integral part of this dissertation. I want to thank Felix, especially for his immense support in writing research papers and coming up with great visualizations of complex problems. I have acquired many skills from him in the past few years. I would like to express my thanks to Divya for his important suggestions and his support in developing complex projects. I would like to thank my colleagues and friends Immanuel, Marcel, and Joris, for all the insightful discussions we had in last few years. I would like to thank Daniel, Alvaro, and Max for the table-tennis matches that were always a great workout and stress busters. I would also like to thank Joris and Marcel for assisting me in translating the abstract into German.

I would also like to thank my soulmate and wife: Mansi, whose continuous support and love have motivated me to complete my doctoral research successfully. Last but not least, I would like to thank my parents and my brother for helping and supporting me throughout my life. I could not have achieved all of this without them.

The German Research Foundation funded the research work in this dissertation via the collaborative research center “Methods and Tools for Understanding and Controlling Privacy (SFB 1223).”

# Abstract

Data management is a complicated task. Due to a wide range of data management tasks, businesses often need a sophisticated data management infrastructure with a plethora of distinct systems to fulfill their requirements. Moreover, since snapshot is an essential ingredient in solving many data management tasks such as checkpointing and recovery, they have been widely exploited in almost all major data management systems that have appeared in recent years. However, snapshots do not always guarantee exceptional performance. In this dissertation, we will see two different faces of the snapshot, one where it has a tremendous positive impact on the performance and usability of the system, and another where an incorrect usage of the snapshot might have a significant negative impact on the performance of the system. This dissertation consists of three loosely-coupled parts that represent three distinct projects that emerged during this doctoral research.

In the first part, we analyze the importance of utilizing snapshots in relational database systems. We identify the bottlenecks in state-of-the-art snapshotting algorithms, propose two snapshotting techniques, and optimize the multi-version concurrency control for handling hybrid workloads effectively. Our snapshotting algorithm is up to $100$x faster and reduces the latency of analytical queries by up to $4$x in comparison to the state-of-the-art techniques.

In the second part, we recognize strict snapshotting used by Fabric as a critical bottleneck, and replace it with MVCC and propose some additional optimizations to improve the throughput of the permissioned-blockchain system by up to $12$x under highly contended workloads.

In the last part, we propose ChainifyDB, a platform that transforms an existing database infrastructure into a blockchain infrastructure. ChainifyDB achieves up to $6$x higher throughput in comparison to another state-of-the-art permissioned blockchain system. Furthermore, its external concurrency control protocol outperforms the internal concurrency control protocol of PostgreSQL and MySQL, achieving up to $2.6$x higher throughput in a blockchain setup in comparison to a standalone isolated setup. We also utilize snapshots in ChainifyDB to support recovery, which has been missing so far from the permissioned-blockchain world.

# Zusammenfassung

Datenverwaltung ist eine komplizierte Aufgabe. Aufgrund der vielfältigen Aufgaben im Bereich der Datenverwaltung benötigen Unternehmen häufig eine anspruchsvolle Infrastruktur mit einer Vielzahl an unterschiedlichen Systemen, um ihre Anforderungen zu erfüllen. Dabei ist Snapshotting ein wesentlicher Bestandteil in nahezu allen aktuellen Datenbanksystemen, um Probleme wie Checkpointing und Recovery zu lösen. Allerdings garantieren Snapshots nicht immer eine gute Performance. In dieser Arbeit werden wir zwei Facetten des Snapshots beleuchten: Einerseits können Snapshots enorm positive Auswirkungen auf die Performance und Usability des Systems haben, andererseits können sie bei falscher Anwendung zu erheblichen Performanceverlusten führen. Diese Dissertation besteht aus drei Teilen basierend auf drei unterschiedlichen Projekten, die im Rahmen der Forschung zu dieser Arbeit entstanden sind.

Im ersten Teil untersuchen wir die Bedeutung von Snapshots in relationalen Datenbanksystemen. Wir identifizieren die Bottlenecks gegenwärtiger Snapshottingalgorithmen, stellen zwei leichtgewichtige Snapshottingverfahren vor und optimieren Multi-Version Concurrency Control für das effiziente Ausführen hybrider Workloads. Unser Snapshottingalgorithmus ist bis zu $100$ mal schneller und verringert die Latenz analytischer Anfragen um bis zu Faktor vier gegenüber dem Stand der Technik.

Im zweiten Teil identifizieren wir striktes Snapshotting als Bottleneck von Fabric. In Folge dessen ersetzen wir es durch MVCC und schlagen weitere Optimierungen vor, mit denen der Durchsatz des Permissioned Blockchain Systems unter hoher Arbeitslast um Faktor zwölf verbessert werden kann.

Im letzten Teil stellen wir ChainifyDB vor, eine Platform die eine existierende Datenbankinfrastruktur in eine Blockchaininfrastruktur überführt. ChainifyDB erreicht dabei einen bis zu sechs mal höheren Durchsatz im Vergleich zu anderen aktuellen Systemen, die auf Permissioned Blockchains basieren. Das externe Concurrency Protokoll übertrifft dabei sogar die internen Varianten von PostgreSQL und MySQL und erreicht einen bis zu $2{,}6$ mal höhren Durchsatz im Blockchain Setup als in einem eigenständigen isolierten Setup. Zusätzlich verwenden wir Snapshots in ChainifyDB zur Unterstützung von Recovery, was bisher im Rahmen von Permissioned Blockchains nicht möglich war.

# Contents

# Chapter 1

# Introduction

With the explosion in the volume of data that is collected every minute, the advancement in the hardware, and an eruption in the number of data management solutions that are available in the market, making a choice on which data management system to use is harder than ever. Moreover, this trend is not something that is planning to subsidize soon. Every data management solution promises to sell us millions of features that we might never need. These additional options lead to an increase in the complexity of the company's infrastructure as well as costs. It is also impossible to find a silver bullet for our data management requirements, and because of this, we need to maintain multiple systems that are good in individual dimensions.

With an ever-increasing rise in our online activity, we are witnessing an unprecedented growth in the amount of data. Sensors, mobile phones, our actions on platforms like Facebook, Instagram, and Snapchat, everything is either generating new data or is continually changing it. This massive data explosion is asking us to develop more sophisticated data management platforms that can handle this evolving data and, at the same time, extract meaningful insights that help businesses to grow and provide a better user experience. Even though the research area around data management has been super active in the past few decades, publishing thousands of excellent research papers, we are yet to solve even the smallest fraction of data management tasks in an optimal way. Due to a wide range of data management tasks, researcher often tend to optimize their systems for a tiny subset to reduce the complexity of the system, while maintaining the performance requirement of the system.

A wide range of data management systems also tends to exploit a broad spectrum of techniques to extract every bit of performance from these systems. Snapshot is one such technique that emerges in many instances of optimization steps, helping us to fulfill

our hunger for more and more performance. Systems are known to utilize snapshots in transaction management and recovery algorithms extensively. So, does that mean using snapshots brings only good to data management? The answer can be yes but only with a careful system design to surround the snapshots. In this dissertation, our primary goal is to optimize the transaction management in data management platforms while simultaneously investigating the love-hate relationship of the data management and the snapshots.

This dissertation explores two distinct yet related class of data management platforms with a single vision of increasing the overall throughput of the system. The first class of systems is a transactional database system which covers the trusted setup, meaning that the host of the system fully trusts the system. Trust can have a significant impact, both on the system design itself, and the performance of the system. With trust, we mean that the host believes that the system is doing what it is supposed to do and not maliciously modifying the data in any way. The other class of system, which is still a transactional system, but in an untrusted setup is a blockchain system. This class of system has gained immense popularity in recent years. Blockchain systems, permissioned to be specific, provide a technology that enables sharing data in an untrusted environment utilizing cryptographic guarantees and proofs. They make strong guarantees concerning the tampering of the state, which means that a malicious change is impossible to hide.

In the first part, we look at how we can optimize the execution of hybrid transactional workloads in a main-memory database system that implements a variant of multi-version concurrency control. We evaluate different state-of-the-art snapshotting techniques, propose two distinct snapshotting techniques. First, that can efficiently snapshot data by manipulating virtual page to physical page mapping in the userspace, and second, that essentially is like `memcpy()` but supported by the Linux kernel. We also redesign the multi-version concurrency control protocol to utilize the snapshot and thus speed-up the execution of the hybrid transactional and analytical workload.

In the next part, we move ourselves to the untrusted setup of permissioned-blockchain systems. As a start, we investigate the transaction pipeline of Hyperledger Fabric, one of the most widely used permissioned-blockchain system developed by IBM. We amend the pipeline, getting rid of the pessimistic transaction execution (that uses strict snapshot isolation for isolating the endorsement and commit phase) that utilizes state locks. We also extend the execution model with mature database optimization to improve the transactional throughput under the contended workload.

In the last part, we showcase a powerful WLC model that can fundamentally transform an existing database infrastructure into a blockchain system. We show that it is a bad idea to design a permissioned-blockchain system from scratch, and it is possible to

achieve blockchain-level guarantees from an existing set of database nodes, with some small additions to the top. We also utilize database snapshots to support recovery in the permission-blockchain setup, which might be necessary if the node fails for some reason, or if the state changes due to malicious activity.

In the following sections, we will look at these three parts in more detail, exploring the background and contributions of each project that are a part of this dissertation. The next sections describes the background of the three projects, and the second section lists the contributions made by these projects, the list of publications, technical reports, grants, and patents that emerged out of these projects.

## 1.1 Background

### AnKerDB

Efficiently supporting complex analytical queries in conjunction with a transactional workload is a difficult task. Since these two variants of workload have an interfering nature, their efficient concurrent execution needs careful system design. If our use-case requires a very restrictive isolation level such as fully-serializable, the solutions and the system-design gets more and more complicated. If the system naively runs the analytical queries on a traditional transactional system implementing a variant of multi-version concurrency control, a simple scan over a column can be up to 6x slower (see Section 2.5.4). This degraded performance urges us to look out for other solutions.

Researchers and developers have proposed intuitive ways [83, 50, 71, 67, 35, 42] to execute these hybrid workloads without much performance problems. The most common and widespread mechanism to execute the hybrid workload is using a data warehouse in addition to the transactional system [44, 60, 55, 54]. In this approach, the transactional system handles the original, up-to-date data, and executes the transaction always on the most recent version of data. On the other hand, the other system which is optimized to execute analytical queries digests these transactional updates at some interval. This design allows us to maintain two optimally configured systems for two different types of workloads. While this is an important trait, it also has a severe bottleneck. Since the system that executes analytical workload synchronizes with the transactional system at some interval, the analytical queries potentially report stale state. These stale results might have a significant impact on businesses that rely on up-to-date real-time analytics. If we try to synchronize the two systems too frequently (for example, once every few milliseconds), it has a severe impact on the performance of the system.

One way to overcome the cons of having two systems for inherently different workloads and synchronizing it is to support both analytical and transactional queries in a single system. However, this requires a careful system design. As already mentioned, the two query classes are interfering in nature, especially in the presence of higher isolation levels. In order to efficiently support transactional and analytical queries in a single system, we need a smart concurrency control protocol that can coordinate resources efficiently between the hybrid workloads which might be a bit too complicated.

What if we take the data warehouse, and put it inside the transactional system? We would still need some synchronization, but it is way cheaper if it is inside the same process. In other words, if we can execute the analytical queries on a copy of the original transactional data, we do not need to worry about the concurrency control protocol.

Nevertheless, creating a copy of data can be expensive. Even if the data resides purely in main-memory, copying several gigabytes of data can take seconds. Researchers have proposed innovative techniques to snapshot data [30, 31, 66] in order to support hybrid transactional and analytical workload in a single system. The most prominent example is HyPer [50] which introduced an excellent approach to overcome the expensive copying of data by exploiting the virtual memory feature of the Linux kernel. Linux provides the `fork()` system call that allows users to copy the parent process's virtual memory space into a child process by copying over the page table instead of the actual pages containing the data. Even though this virtual memory snapshotting approach used by HyPer is way faster than a traditional copy, it can still be quite expensive. For a typical database process that allocates $100$ GB of memory, a `fork()` system call can take as long as $1$ second. While this is acceptable for many workloads, it is quite expensive for a much higher snapshotting frequency. What if we need to run some analytics on data that is no more than $10$ milliseconds old? None of the state-of-the-art snapshotting techniques are able to support such a high snapshotting frequency.

## Fabric++

Blockchain systems have gained immense popularity in the last few years. Their ability to bring trust to an environment that has been untrusted so far has a great significance, especially in a business environment. The database community has proposed several intuitive blockchain systems [97, 59, 53, 73, 105, 46, 63] in the last few years, which all claim to have either better performance or stronger cryptographic guarantees compared to all other available systems. There has been a significant amount of contribution also from the security and operating systems community in a similar direction [36, 90, 91, 40, 52, 89].

Nevertheless, are we trying to reinvent the wheel by proposing an entirely new system with blockchain level guarantees? Are blockchain systems related to another system that has existed for decades: Database Systems? If yes, how similar are the database and blockchain systems? Should we take an existing database system, and extend it to provide blockchain level guarantees? Or should we take a premature blockchain system designed from scratch and incorporate decades of database systems research into it? Is it even feasible to do so?

In this project, we investigate and analyze these questions and try to find out their answers. Firstly we try to blur the line of separation between the database and the blockchain systems by analyzing the similarity in their design. Later, we pick Hyperledger Fabric [18] a prevalent open-source permissioned blockchain system developed by IBM, and try to investigate the weakness of its transactional pipeline. We also integrate some mature database optimization into this pipeline to show the impact of utilizing decades of database research inside blockchain systems. The primary objective of this project is to identify the design space of the blockchain system, which can be optimized using database research.

## ChainifyDB

After looking at the ways and impact of incorporating database technology into a permissioned-blockchain system in Fabric++, in this project, we explore and analyze the transformation of a database system into a blockchain system. Researchers from the database community have already proposed some ways to transform an existing database system into a blockchain system [33, 41, 70]. However, these attempts require a profound modification to the transaction processing pipeline of the underlying database system. So, if a use-case requires to utilize the blockchain feature of the system, it needs to shift to an entirely new, potentially a beta version of these modified database systems, which is mostly a no-no.

In this project, we aim to extend an existing database infrastructure, potentially comprising of database systems from different vendors into a blockchain system. This choice of treating the underlying database system as a black-box brings unique challenges. These underlying database systems might behave differently, or they might treat isolation-levels differently, or they may support different subsets of SQL standards. In addition to this, they might implement concurrency control in entirely distinct ways. ChainifyDB handles all these challenges, additionally provides blockchain level guarantees, all of this without looking at the source code of the underlying system.

ChainifyDB provides an advanced network protocol and a transaction execution

model that sits on top of existing database infrastructure and provides cryptographic guarantees of a typical permissioned blockchain system. This non-invasive approach to adding a blockchain system to our technology stack can help achieve widespread commercial adoption of the blockchain technology.

## 1.2 Contribution

### 1.2.1 AnKerDB

In the following, we present a detailed list of contributions we made in this project.

1. We present a user-level snapshotting technique using rewiring [77]. In this technique, we exploit the mapping from the virtual page to a physical page by utilizing a main-memory filesystem. We create snapshots by manually rewiring the virtual page address to the physical page using rewiring. We also perform copy-on-write manually using the signal handlers in the user-space.
2. We analyze the power and shortcomings of rewired snapshotting and present an efficient and lightweight snapshotting mechanism implemented inside the Linux kernel. Unlike `fork()`, our snapshotting techniques can create snapshots at a page granularity.
3. We evaluate our snapshotting techniques against the state-of-the-art snapshotting techniques and show that our snapshot mechanism is significantly faster (up to 100x) than other state-of-the-art snapshot algorithms.
4. We present a variant of a multi-version concurrency control algorithm that incorporates our virtual memory snapshotting techniques to support the efficient execution of analytical queries while simultaneously executing the transactional workload.
5. We integrate other snapshotting techniques such as `fork()` inside AnKerDB and evaluate different snapshotting techniques under the hybrid analytical and transactional workload. We show that the version of AnKerDB that utilizes our system call is significantly faster (up to 4x) than the version that uses no or other snapshot techniques.
6. We also show that our MVCC implementation using virtual snapshots under the strongest serializable isolation level can execute analytical queries as fast as the weakest read-uncommitted isolation level.

### 1.2.2 Fabric++

We made the following contributions through this project:

1. We investigate the similarities and the difference between a traditional database system and permissioned blockchain systems. As an example, we explore the transaction processing pipeline of Hyperledger Fabric and showcase the overhead of non-transactional components of the pipeline. Due to this excessive overhead, it is tough to improve the end to end transactional throughput of the system without making considerable changes to the architecture of the transaction pipeline.

2. We optimize the transactional pipeline of Hyperledger Fabric to get rid of lock-based object synchronization. We replace the existing concurrency control implementation with multi-version concurrency control, which is similar in design to a traditional MVCC implementation in a relational database system.

3. In addition to integrating MVCC inside Hyperledger Fabric, we also integrate an early-abort mechanism in Fabric that helps us to clean the invalid transactions from the transaction pipeline.

4. We replace the standard FIFO-based ordering service used by Fabric by an advanced ordering service that analyzes the transactions in a block to find a close-to-optimal commit order.

5. We perform an extensive evaluation of Fabric++, which incorporates all suggested optimizations against standard Fabric to show the significance of age-old database research in improving permissioned-blockchain systems. Fabric++ achieves up to 12x improvement in the throughput of successful transactions in comparison to the standard version of Hyperledger Fabric.

### 1.2.3 ChainifyDB

We made the following contributions through this project:

1. We present an intuitive WhateverLedger Consensus model that pushes the consensus phase to the end of the transaction pipeline. The WLC model helps us in dropping interesting trust guarantees from the components like execution that process transactions before the consensus phase.

2. We present a concrete instance of the WhateverLedger Consensus model in the form of ChainifyDB. ChainifyDB essentially appends the blockchain guarantees to an existing database infrastructure. Hence block-**Chainify**-ing the **DB**.

3. We present a simple-yet-powerful recovery mechanism that can restore a diverged database replica. The state diversion can happen because of a loss of state due to the failure of the database system. Such a mismatch in the state can also happen due to an unintended malicious modification of the state.

4. We evaluate ChainifyDB under different database configurations. We also compare the performance of ChainifyDB to a standalone instance of the underlying database system to show that the overhead of appending cryptographic guarantees to an existing database infrasrtucture is minimal.

### 1.2.4 Personal Contributions

Table 1.1 lists my personal contributions in much more detail. Since, I joined the **Rewiring** project very late, my minor contribution to this project are merged with the Chapter 2.

### 1.2.5 Publications, Technical Reports, Patent, and Grant

Major part of this dissertation has been previously published, or is being submitted to different international conferences and a part of it is under an international patent filing by the Saarland University.

- **Chapter 2 – AnKerDB: Optimizing MVCC using Hyperfast Virtual Snapshotting**

  *Publications:*

  [77] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich.
  RUMA has it: Rewired User-space Memory Access is Possible!
  Proceedings of the VLDB 2016, New Delhi, India.

  [82] Ankur Sharma, Felix Martin Schuhknecht, Jens Dittrich.
  Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting.
  ACM SIGMOD 2018, Houston Tx, USA.

*Technical Report:*

[81] Ankur Sharma, Felix Martin Schuhknecht, Jens Dittrich.
Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting. arXiv:1709.04284 (2017).

- **Chapter 3 – Fabric++: Optimizing Transaction Processing in Hyperledger Fabric**

  *Publications:*

  [80] Ankur Sharma, Felix Martin Schuhknecht, Jens Dittrich, Divya Agrawal.
  Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric.
  ACM SIGMOD 2019, Amsterdam, Netherlands

  *Technical Report:*

  [79] Ankur Sharma, Felix Martin Schuhknecht, Jens Dittrich, Divya Agrawal.
  How to Databasify a Blockchain: the Case of Hyperledger Fabric. arXiv:1810.13177 (2018)

- **Chapter 4 – ChainifyDB: A Non-invasive Transformation of Database Systems into a Blockchain System**

  *Technical Report:*

  [78] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, Divya Agrawal.
  ChainifyDB: How to Blockchainify any Data Management System. arXiv:1912.04820 (2019). (Manuscript in preparation)

  *Grant:*

  German Ministry of Education and Science (BMBF) StartUp Secure (Phase I)
  Covering $850,000$ Euros over 1 year.

  *Patent:*

  Secure and Transparent Cross-organization Data Sharing via Permissioned Blockchain Technology. (filed provisionally by the Saarland University)

| **Project** | **Contribution** | Involvement | Details |
|---|---|---|---|
| Rewiring [77] | Applications of rewiring | Minor | I developed virtual snapshotting as an application of rewiring. |
| AnKerDB [82, 81] | `vm_snapshot()` system call | Major | I developed this on my own. |
| | Prototype database supporting different snapshotting mechanisms and isolation levels | Major | See above. |
| | Experimental analysis | Major | See above. |
| Fabric++ [80, 79] | Transactional pipeline optimization of Fabric | Major | See above. |
| | Optimized ordering service | Major | See above. |
| | Benchmarking tool | Major | See above. |
| | Experimental analysis | Major | See above. |
| ChainifyDB [78] | System architecture | Major | See above |
| | Formalization of the system architecture into WLC model | Minor | Developed by Felix Schuhknecht. I was involved in the discussions. |
| | Semantic query analysis | Major | Divya Agrawal implemented this for his Master-thesis. I heavily contributed with ideas and the overall design on this part of the project. |
| | Parallel query execution | Major | See above. |
| | Experimental analysis | Major | I developed this on my own. |

*Table 1.1: List of personal contributions to different projects.*

# Chapter 2

# AnKerDB: Optimizing MVCC using Hyperfast Virtual Snapshotting

Efficient transaction management is a delicate task. As systems face transactions of inherently different types, ranging from point updates to long-running analytical queries, it is hard to satisfy their requirements with a single execution engine. Unfortunately, most systems rely on such a design that implements its parallelism using multi-version concurrency control. While MVCC parallelizes short-running OLTP transactions well, it struggles in the presence of mixed workloads containing long-running OLAP queries, as scans have to work their way through vast amounts of versioned data. To overcome this problem, we reintroduce the concept of hybrid processing and combine it with state-of-the-art MVCC: OLAP queries are outsourced to run on separate virtual snapshots while OLTP transactions run on the most recent version of the database. Inside both execution engines, we still apply MVCC.

The most significant challenge of a hybrid approach is to generate the snapshots at a high frequency. Previous approaches heavily suffered from the high cost of snapshot creation. In our approach termed AnKer, we follow the current trend of co-designing underlying system components and the DBMS, to overcome the restrictions of the OS by introducing a custom system call `vm_snapshot`. It allows fine-granular snapshot creation that is orders of magnitudes faster than state-of-the-art approaches. Our experimental evaluation on an HTAP workload based on TPC-C transactions and OLAP queries show that our snapshotting mechanism is more than a factor of $100$x faster than fork-based snapshotting and that the latency of OLAP queries is up to a factor of $4$x lower than MVCC in a single execution engine. Besides, our approach enables a higher OLTP throughput than all state-of-the-art methods.

## 2.1 Introduction

Realizing fast concurrent transactional processing is a desirable but challenging design goal. A concurrency control technique is required to fully utilize the massive amount of hardware parallelization that is nowadays available even in commodity servers.

Interestingly, a large number of database systems, including major players like PostgreSQL [74], Microsoft Hekaton [26], SAP HANA [38], HyPer [71], MemSQL [6], MySQL [8], NuoDB [9], and Peloton [10] currently implement a form of multi-version concurrency control (MVCC) [95, 65, 21, 48] to manage their transactions. It allows for a high degree of parallelism as readers do not block writers. The core principle is straightforward: if a tuple is updated, a new physical version of this tuple is created and stored alongside the old one in a version chain, such that the old version is still available for readers that are still allowed to see the older version. Timestamps ensure that transactions access only the most recent version that existed when they entered the system.

### 2.1.1 Limitations of Multi-version Concurrency Control

In MVCC implementations that rely on a single execution engine, all transactions, no matter whether they are short running OLTP transactions or scan-heavy OLAP queries, are treated equally and are executed on the same (versioned) database. While this form of processing unifies the way of transaction management, it also has unpleasant downsides under HTAP workloads: First and foremost, scan-heavy OLAP queries heavily suffer when they have to deal with a large number of version chains [71]. During a scan, version chains must be traversed to locate the most recent version of each item that is visible to the transaction. It involves expensive timestamp comparisons as well as random accesses when going through the version chains. As column scans typically take significantly more time than short-running transactions, which touch only a few entries, a large number of OLTP transactions can perform updates in parallel to create such version chains. Apart from this, these version chains must be garbage collected from time to time to remove versions that are not visible to any transaction in the system. Garbage collection is typically done by a separate thread, which frequently traverses these chains to locate and delete outdated versions [22, 96, 99, 56]. This thread has to be managed and synchronized with the transaction processing, utilizing precious system resources.

HTAP workload, consisting of transactions of inherently different nature, does not fit the uniform processing in a single execution engine, which treats all incom-

ing transactions in the same way. Unfortunately, many state-of-the-art MVCC systems [71, 74, 26, 6, 8, 10] implement some variant of such a processing model.

### 2.1.2 Hybrid Processing

But why exactly do these systems rely on such a processing model, although it does not fit the faced workload? Why they do not implement *hybrid processing*, which classifies queries based on the type and executes them in separation?

To answer these questions, let us look at the development of the prominent HyPer [50, 71] system. Early versions of HyPer implemented hybrid processing [50, 66]: the queries were classified into the categories OLTP and OLAP and consequently executed on separate representations of the database. The short running modifying OLTP transactions were executed on the most recent version of the data while long-running OLAP queries were outsourced to run on *snapshots*. These snapshots were created from time to time on the up-to-date version of the database.

While this concept mapped the mixed workload to the processing system in a natural way, the developers faced a severe problem: the creation of snapshots turned out to be too expensive [71]. To snapshot, HyPer utilized the `fork` system call. This system call creates a child process that shares its virtual memory with the parent process. Both processes perform copy-on-write to keep changes locally, thus implementing virtual snapshotting. While this principle is cheaper than physical snapshotting, which eagerly creates a deep copy of the data, forking a process has a considerable overhead of replicating the entire virtual memory allocated by the parent process. Thus, the developers decided to move away from hybrid processing to a model with a single execution engine, relying entirely on MVCC in their current version [71].

### 2.1.3 Challenges

Despite the challenges one has to face when implementing a hybrid model, we believe it is the right choice. Matching the processing system to the workload is crucial for performance. This is the goal of our processing concept termed *AnKer*, which we will propose in the following. Still, to do so, we have to discuss two problems: **(a)** Obviously, MVCC is the state-of-the-art concurrency control mechanism in main-memory systems. We intend to apply it as well to parallelize transaction processing *within* our execution engines. But how to combine state-of-the-art MVCC with a hybrid processing model? **(b)** State-of-the-art snapshotting mechanisms are not capable of powering

a hybrid processing model. How to realize a fast snapshotting mechanism, that allows the creation of snapshots at a high frequency and with high flexibility?

## 2.2 Background

Classical systems implement MVCC in a single execution engine, where all queries are treated equally and executed on the same versioned database. In contrast to that, AnKer extends the capabilities of MVCC by reintroducing the concept of *hybrid processing*, where incoming OLTP transactions and OLAP queries are treated independently. By this, we can utilize the advantages of MVCC *while* avoiding its downsides.

### 2.2.1 MVCC in Hybrid Processing

The concept of hybrid processing works as follows: based on the classification, we separate the short-running OLTP transactions from the long-running (read-only) OLAP queries. Conceptually, the modifying OLTP transactions run concurrently on the most recent version of the database and build up version chains as in classical MVCC. In parallel, we outsource the read-only OLAP queries to run on separate (read-only) snapshots of the versioned database. These snapshots are created at a very high frequency to ensure freshness. Thus, instead of dealing with a single representation of the database that suffers from a large number of long version chains, we maintain a most recent version in an OLTP execution engine alongside with a set of snapshots, which are present in the OLAP execution engine. Naturally, each of the representations contains fewer and shorter version chains, which primarily reduces the main problem described in Section 2.1.1. Apart from that, using snapshots has the pleasant side-effect that the garbage collection of version chains becomes extremely simple: We remove the version chains automatically with the deletion of the corresponding snapshot if it is not visible to any transaction in the system. Other systems like PostgreSQL have to rely on a fine-granular garbage collection mechanism for shortening the version-chains, requiring precious resources. By using snapshotting, we can solve the problem of complex garbage collection techniques implicitly.

### 2.2.2 High-Frequency Snapshotting

With the high-level design of the hybrid processing model at hand, the question remains how to realize efficient snapshotting. The approach stands and falls with the ability to

generate snapshots at a *very high frequency* to ensure that transactions running on the snapshots have to deal only with few and short version chains. In this regard, previous approaches that relied on snapshotting suffered under the expensive snapshot creation phase and consequently moved away from snapshotting. As mentioned, early versions of HyPer [50], which also used a hybrid processing model, created virtual snapshots utilizing the system call `fork`. This call is used to spawn child processes which share their entire virtual memory with the parent process. The copy-on-write, which is carried out by the operating system on the level of memory pages ensures that changes remain local in the associated process. While this mechanism naturally implements a form of snapshotting, process forking is expensive. Thus, it is not an option for our case as we require a more lightweight snapshotting mechanism.

Unfortunately, all the existing solutions are not sufficient for our requirements on snapshot creation speed. Therefore, AnKer implements a more sophisticated form of virtual snapshotting. We do not limit ourselves by using the given general purpose system calls. Instead, we introduce our custom system call termed `vm_snapshot` and integrate the concept of rewiring [77] directly into the Linux kernel. Such a co-design of underlying system components and the DBMS has been demonstrated successfully in recent publications concerning both operating system [43, 62] and hardware [72, 86] customizations, as it enables a whole new level of optimization opportunities. Using our call, we can essentially snapshot arbitrary virtual memory areas within a single process at any point in time. The virtual snapshots share their physical memory until a write to a virtual page happens which allows us to create snapshots with a small memory footprint, allowing us to build them at a high frequency without much memory overhead. Consequently, the individual snapshots contain few and short version chains and enable efficient scans.

### 2.2.3 Structure & Contributions

Before we start with a detailed presentation of the system and the individual components, let us outline the contributions we make:

(I) We present AnKer, a hybrid storage model that is able to execute scan-heavy OLAP queries on a consistent snapshot while processing short running transactions over the most recent version of the database. We extend this model to redesign HyPer's MVCC engine [71] as an example, to show the benefits of a hybrid processing model over conventional MVCC implementations. We also show that the changes to the original implementation are minimal and can be easily adopted to other main-memory MVCC systems [51, 85, 94].

(II) We realize the snapshots in form of *virtual snapshots* and heavily accelerate the

snapshotting process by introducing a *custom system call* termed `vm_snapshot` to the Linux kernel. This call directly manipulates the virtual memory subsystem of the OS and allows for a significantly higher snapshotting frequency than state-of-the-art techniques. We demonstrate the capabilities of `vm_snapshot` in a set of micro-benchmarks and compare it against the existing physical and virtual snapshotting methods.

(III) We create snapshots on the *granularity of a column*, instead of snapshotting the entire table or database as a whole which is possible due to the flexibility of our custom system call `vm_snapshot`. Therefore, we can limit the snapshotting effort to those columns, which are accessed by the transactions.

(IV) We create *snapshots of versioned columns*. To create a snapshot, the current column as well as the timestamp information is virtually snapshotted using our custom system call `vm_snapshot`, and the current version chains are handed over. Running transactions can still access all required versions from the fresh snapshot. As the snapshot is read-only, all further updates happen to the up-to-date column, creating new version chains. As a side-effect, we avoid any expensive garbage collection as dropping an old snapshot drops all old version chains with it.

(V) We perform an extensive experimental evaluation of AnKer. We compare the hybrid processing model utilizing our system call `vm_snapshot` with a fork-based snapshotting approach. Additionally, we compare the hybrid models with a single execution engine under full serializability, snapshot isolation, and read uncommitted guarantees, executing mixed HTAP workloads based on TPC-C transactions and configurable OLAP queries. Our prototype implementation of the AnKer concept can be configured to support both a hybrid and a single execution engine (by disabling snapshotting) as well as the required isolation levels. We show that our approach offers faster snapshotting, lower OLAP latency and higher OLTP transaction throughput than the counterparts under mixed workloads.

The paper has the following structure: In Section 2.3, we describe the hybrid design of AnKer and motivate it with the problems of state-of-the-art MVCC approaches. As the hybrid execution engine requires a fast snapshotting mechanism, we discuss the currently available snapshotting techniques to understand their strengths and weaknesses in Section 2.4. In Section 2.5, we propose our snapshotting method based on our custom system call `vm_snapshot`. Finally, in Section 2.6, we evaluate AnKer in different configurations and show the superiority of hybrid processing using `vm_snapshot`.

## 2.3 AnKer

As outlined, the central component of AnKer is a hybrid processing model, which separates OLTP from OLAP processing using virtual snapshotting. Both in the up-to-date representation of the data as well as in the snapshots, we want to use MVCC as the concurrency control mechanism. To understand our hybrid design, let us first understand how MVCC works on a single execution engine.

### 2.3.1 Mechanisms of MVCC

To understand the mechanisms of MVCC, let us go through the individual components. Initially, the data is unversioned and present in the column. Thus, there does not exist any version chains. If a transaction updates an entry, we first store the new value locally inside the local memory of the transaction. Update are materialized in the column when the actual commit happens. Before applying the update to the in-place value, the system copies the old value to the version chain using atomic compare-and-swap instructions. We store the versions in a *newest-to-oldest* order. Other systems as, e.g., HyPer [71] rely on this order as well, as it favors younger transactions: they will find their version early on during the chain traversal. A version chain can become arbitrarily long if frequent updates to the same entry happen. Along with the version, we store a unique timestamp of the update that created the version which is necessary to ensure that the transactions that started before the (committed) update happened, do not see the new version of the entry but still the old one. Unfortunately, reading a versioned column can become arbitrarily expensive since the chain must be traversed using the comparison of timestamps to locate the proper version. In summary, if a large number of lengthy version chains is present and a transaction intends to read many entries, the version chain traversal cost becomes significant.

Besides the way of versioning the data, the guaranteed isolation level is an important aspect in MVCC. As a consequence of its design, MVCC implements snapshot isolation guarantees by default. During its lifetime, a transaction $T$ sees the committed state of the database, that was present at $T$'s start time. The updates of newer transactions, which committed during $T$'s lifetime, are not seen by $T$. Write-write conflicts are detected at commit time: if $T$ wants to write to an entry, to which a newer committed transaction already wrote, $T$ aborts. Still, under snapshot isolation, so-called write-skew [39] anomalies are possible. Fortunately, MVCC can be extended to support full serializability [71, 93]. To do so, we extend the commit phase of a transaction with additional checks. If a transaction $T$ wants to commit, it validates its read-set by inspecting if any other transaction, that committed during $T$'s lifetime changed an entry

in a way that would have influenced $T$'s result. If this is the case, $T$ has to abort as its execution was based on stale reads. To perform the validation, we adopt the efficient approach applied in HyPer [71], which is based on precision locking [95], a variant of predicate locking. Essentially, the system tracks the predicate ranges on which the transaction filtered the query result. During validation, we check whether any write of any recently committed transaction intersects with the predicate ranges. If an intersection is identified, the transaction aborts.

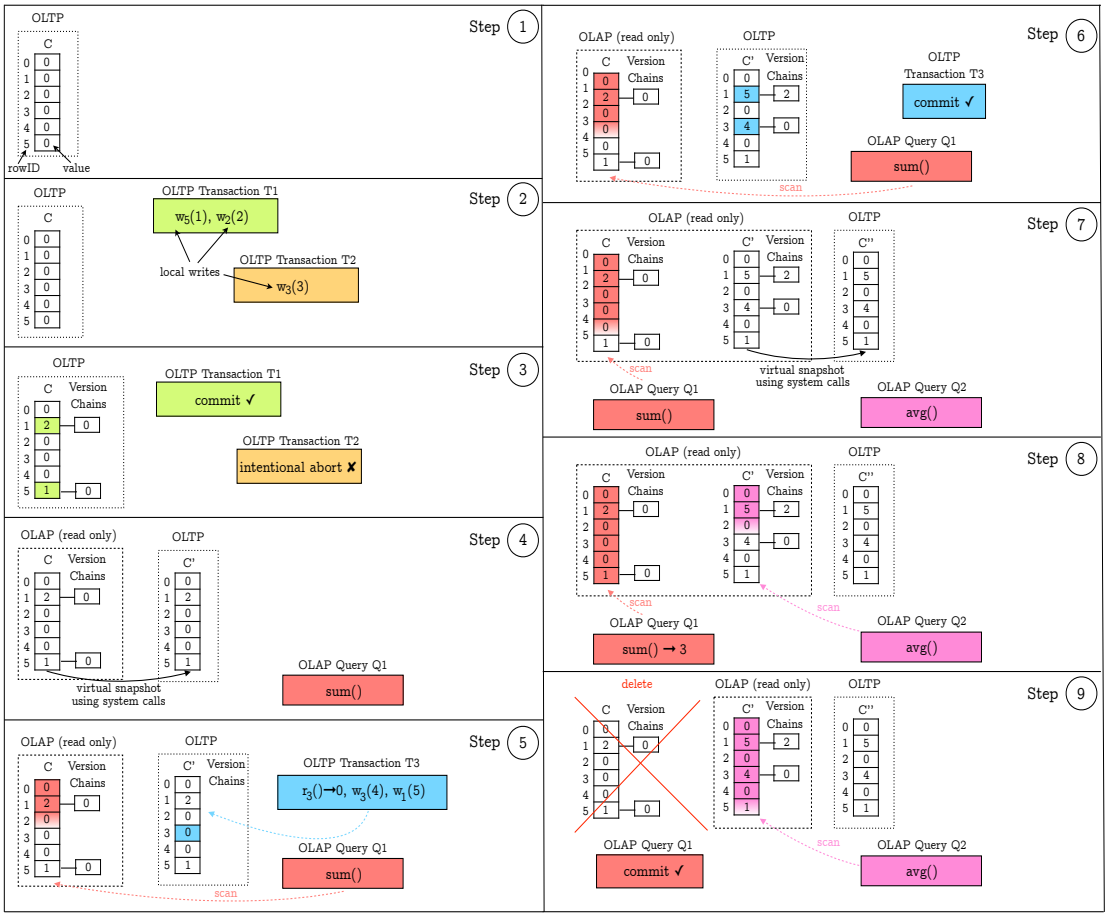



*Figure 2.1: Hybrid processing in AnKer.*

### 2.3.2 Hybrid MVCC

To overcome the limitations of MVCC implementations mentioned above, we realize a hybrid execution engine in AnKer. Two engines are present side by side: one engine is

responsible for the concurrent processing of short-running transactions (termed *OLTP execution engine* in the following), while the other one can perform long-running read-only transactions in parallel (termed *OLAP execution engine* from here on). Incoming transactions are marked as either an OLTP transaction or an OLAP query and sent to the respective engine for processing. The challenge is to combine the concept of hybrid processing with MVCC. Let us look at the engines in detail in the case of an example depicted in Figure 2.1.

In the example, we use the following set of operations:

| | |
|---|---|
| $w_i(v)$ | write to row $i$ the given value $v$ |
| $r_i() \rightarrow v$ | read from row $i$ and return the read value as $v$ |
| $sum() \rightarrow r$ | sum up all column values and return the result as $r$ |
| $avg() \rightarrow r$ | average all column values and return the result as $r$ |

1. For the following discussion, we assume that our table consists of a single column $C$ of $6$ rows, identified by rows $0$ to $5$, which all contain the value $0$ in the beginning. This column $C$ is located in the OLTP execution engine and is the up-to-date representation of the column. Since there are no snapshots present yet, the OLAP execution engine virtually does not exist.

2. Two OLTP transactions $T_1$ and $T_2$ arrive and intend to perform a set of writes. The first write $w_5(1)$ of $T_1$ intends to update at row $5$ the value $0$ with the new value $1$. However, instead of replacing the old value in the column with the new value, we store the new value locally inside the transaction $T_1$ and keep the column untouched as long as the transaction does not commit. In the same fashion, the remaining write $w_1(2)$ of $T_1$ as well as the write $w_3(3)$ of $T_2$ are performed only locally inside their respective transactions. Note that all three written values are uncommitted so far and are visible to the transactions that completed the individual writes.

3. Let us now assume that $T_1$ commits while $T_2$ intentionally aborts. The commit of $T_1$ now replaces the old value $0$ with the new value $1$ in column $C$ at row $5$. Of course, the old value $0$ is not discarded but stored in a newly created version chain for that row. Similarly, at row $1$ the old value $0$ is replaced by the new value $2$, and the old value stored in the version chain. Note that we implement a timestamp mechanism (logging both the start and end time of a transactions commit phase) to ensure that both writes of $T_1$ become visible atomically to other transactions. As no other transactions modified row $1$ and $5$ during the lifetime of $T_1$, the commit succeeds and satisfies full serializability, that we guarantee for all

transactions. In contrast to that, the abort of $T_2$ discards merely the local change of row 3. This strategy with no rollback makes aborts cheap.

4. An OLAP query $Q_1$ arrives, which intends to scan and sum up the values of all rows of the column, denoted by $sum()$. To execute $Q_1$, first, a snapshot of column $C$ is taken utilizing our custom system call (which we described in Section 2.5 in detail), resulting in a (virtual) duplicate of the column denoted as $C'$. It is essential to understand that this duplicate $C'$ will become the most recent version of the column in the OLTP engine. The "old" column $C$ along with its build-up version chains is logically moved to the OLAP engine and becomes read-only.

5. Another OLTP transaction $T_3$ arrives, that intends to perform a read $r_3()$ followed by the two writes $w_3(4)$ and $w_1(5)$. The read $r_3()$ is performed by accessing the current value of row 3 of the representation in the OLTP view, resulting in $r_3() \rightarrow 0$. The two successive writes are stored locally inside $T_3$ and are not visible for other transactions. In parallel to the depicted operations of $T_4$, our OLAP query $Q_1$, which sums up the column values, starts executing in the OLAP engine on $C$. As the snapshot is older than $Q_1$, it can directly scan $C$ without inspecting the version chains.

6. While the scan of $Q_1$ is running, $T_3$ decides to commit. This commit does not conflict with the execution of $Q_1$ in any way, as the $Q_1$ and $T_3$ run in different execution engines. The local write $w_3(4)$ and $w_1(5)$ are materialized in $C'$ after moving the old version to version chain.

7. Another OLAP query $Q_2$ arrives, which attempts to compute the average of the column, denoted by $avg()$ triggers the creation of a new snapshot. Again we use our system call and take a snapshot of column $C'$ that is located in the OLTP engine, resulting in a (virtual) duplicate of the column in form of $C''$. The new duplicate $C''$ becomes the most recent representation of the column in the OLTP engine, while $C'$ with its version chains re-labeled as the OLAP engine. Note that both $C'$ as well as $C$ is now present in the OLAP engine side by side, with $Q_1$ still running on $C$.

8. The new OLAP query $Q_2$ starts running on the new snapshot $C'$, while the older OLAP query $Q_1$ finishes its scan, returns the sum 3 and commits.

9. The finish of $Q_1$ makes $C$ obsolete, as a newer representation $C'$ already exists and no transaction accessing $C$ is running. Thus, we can safely delete the oldest snapshot $C$.

### Snapshot Synchronization

For simplicity, in the previous example, all transactions worked solely on a single column. However, a database usually consists of several tables, containing a large number of attributes and therefore, some form of *snapshot synchronization* is necessary. In this context, snapshot synchronization means that a transaction, which accesses multiple columns, has to see all columns consistent concerning a single point in time. The system could trivially snapshot all columns of all tables for each request of the snapshot. However, this causes unnecessary overhead as we might access only a small subset of the attributes. Therefore, in AnKer, we implement a lazy snapshot materialization approach. The system logs the snapshot-timestamp along with the list columns used by the snapshot for each snapshot request. The actual snapshot materialization happens for each column if an OLTP transaction or an OLAP query comes in, which accesses the columns requested by the snapshot. This lazy strategy ensures that columns, that are never used by OLAP queries or are never updated by transactions are also never materialized in snapshots.

### Snapshot Consistency

In the previous example, we created a new snapshot for each OLAP query. When this happens, and the previously described access triggers the actual materialization of the snapshot using our system call, we have to ensure that no other transactions modify the column while the snapshot is under creation. We ensure this using a shared lock on the column, which must be acquired by any transaction which performs an installation of updates to the respective column during the commit phase. When materializing a snapshot, an exclusive lock must be acquired. To grant an exclusive lock, the system blocks all further requests for the shared lock and the snapshot can be materialized once all already-acquired shared locks are released.

## 2.4 State-of-the-art Snapshotting

As stated before, our hybrid processing model stands and falls with an efficient snapshot creation mechanism. Only if we can create them at a high frequency without penalizing the system, we get up-to-date snapshots with short version chains. There exist different techniques to implement such a snapshotting mechanism, including *physical* and *virtual* techniques. While the former ones create costly physical copies of the entire memory, the latter ones lazily separate snapshots only for modified memory pages. Let us now

look at the state-of-the-art techniques in detail to understand why they do not suffice our needs and why we have to introduce an entirely new snapshotting mechanism in AnKer.

### 2.4.1 Physical Snapshotting

The most simple approach is *physical snapshotting*, where a deep physical copy of the database is created. On this physical copy, the reading queries can then run in isolation, while the modifying transactions update the original version. The granularity of snapshotting is an important design decision. It is possible to snapshot the entire database, a table, or just a set of columns. This way of snapshotting represents the *eager* way of doing it — at the time of snapshot creation, the snapshot and the source are entirely separated from each other. As a consequence, any modification to the source is not carried through to the snapshot.

Physical snapshotting is straightforward and is easy to apply. However, its effectiveness is directly bound to the amount of data that is updated on the source. If only a portion of the data is updated, the full physical separation of the snapshot and the source is unnecessary and just adds overhead to the snapshotting cost.

### 2.4.2 Virtual Snapshotting

Virtual Snapshotting overcomes this problem by following the *lazy* approach. The idea of virtual snapshotting is that initially the snapshot and the source are not separated physically. Instead, the separation happens lazily only for those memory pages that are modified. As we will see, there are multiple ways to perform this separation using virtual memory. To understand them, let us first go through some of the high-level concepts of the virtual memory subsystem of *Linux* (kernel 4.8).

#### Virtual vs Physical Memory

By default, the user perspective on memory is simple — only virtual memory is visible. To allocate a consecutive virtual memory area $b$ of size $s$ the system call `mmap` is used. For instance, the general purpose memory allocator `malloc` from the GNU C library internally uses `mmap` to claim large chunks of virtual memory from the operating system. The layer of physical memory is completely hidden and transparently managed by the operating system. After allocating the virtual memory area, the user can start accessing the memory area, e.g., via $b[i] = 42$. Apparently, the user perspective is relatively

simple. He does not have to distinguish between memory types at all. In comparison, the kernel perspective is significantly more complicated.

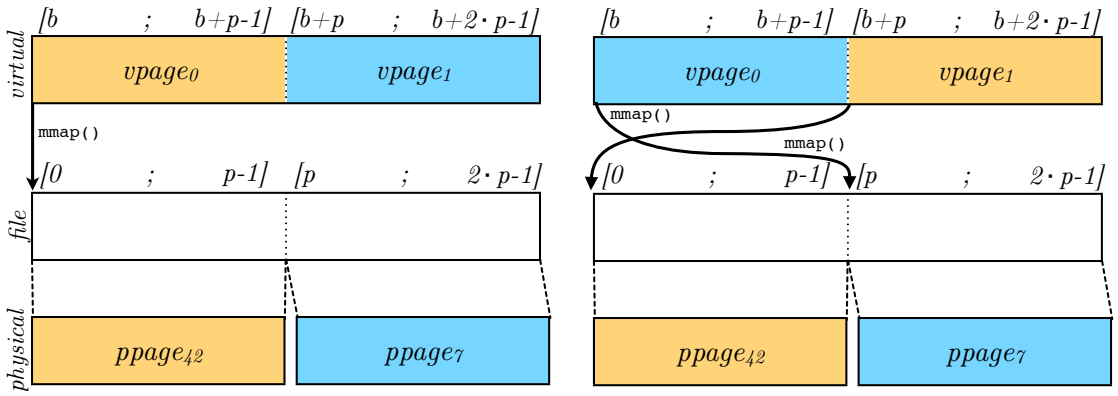



*Figure 2.2: Visualization of rewiring as shown in [77]. The start address of the virtual memory area is denoted as $b$ and the page size as $p$. A consecutive virtual memory area of two pages is mapped to a main-memory file, which is transparently mapped to two potentially scattered physical pages (left part). The system call `mmap` can be used to manipulate the mapping at runtime (right part).*

First of all, the previously described call to `mmap`, which allocates a consecutive virtual memory area, does not trigger the allocation of physical memory right away. Instead, the call only creates a so-called `vm_area_struct` (*VMA*), that contains all relevant information to *describe* this virtual memory area. For instance, it stores that the size of the area is $s$ and that the start address is $b$. Thus, the set of all VMAs of a process defines which areas of the virtual address space are currently *reserved*. Note that a single VMA can describe a memory area spanning over multiple pages. As an example, in Figure 2.3 we visualize two VMAs. They describe the virtual memory areas starting at address $b$ (spanning over four pages) and $c$ (spanning over three pages). In between the two memory areas is an unallocated memory area of size two pages. Besides the

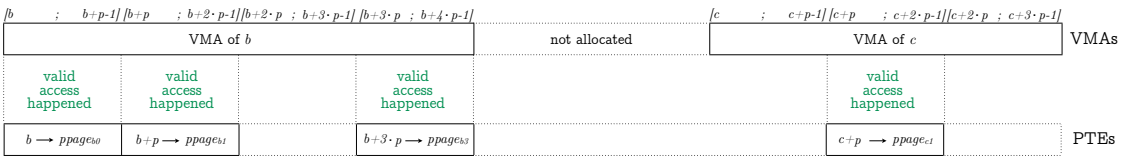



*Figure 2.3: Visualization of the relationship between VMAs and PTEs. The VMAs store the information about the currently allocated virtual memory areas alongside with all necessary meta-information.*

VMAs, there exists a *page table* within each process. A page table entry (*PTE*) that contains the actual mapping from a single virtual to a physical page is inserted after the first access to a virtual page, based on the information stored in the corresponding VMA. The example in Figure 2.3 shows the state of the page table after four accesses to four different pages. As we can see, there is one PTE per accessed page in the page table.

### Fork-based Snapshotting

With the distinction between the different memory types and the separation of VMAs and PTEs in mind, we are now able to understand the most fundamental form of virtual snapshotting: fork-based snapshotting [50]. It exploits the system call `fork`, which creates a child process of the calling parent process. This child process gets a copy of all VMAs and PTEs of the parent. In particular, this means that after a fork, the allocated virtual memory of the child and the parent share the same physical memory. Only a write[1] to a page of child or parent triggers the actual physical separation of that page in the two processes (called *copy-on-write* or *COW*). This concept can be exploited to implement a form of snapshotting. If the source resides in one process, one can merely fork it to create a snapshot. Any modification to the source in the parent process is not visible in the child process. As mentioned in Section 2.1.2, early versions of HyPer that implemented hybrid processing utilized `fork`.

### Rewired Snapshotting

While fork-based snapshotting has the convenient advantage, that the snapshotting mechanism is handled by the operating system in a transparent manner, it has two significant disadvantages. First, it requires the spawning and management of several processes at a time. Second, it always snapshots all allocated memory of the process, i.e., it cannot be used to snapshot a subset of the data. Both problems can be addressed using the technique of rewiring as described in [77].

To understand rewiring, let us again look at the mapping from virtual to physical memory as described in Section 2.4.2. This mapping is by default both *hidden* from the user as well as *static*, as the user sees only virtual memory by default. The authors of [77] manage to reintroduce physical memory to userspace in the form of so-called *main-memory files*. A main-memory file has the same properties as a file on disk, except that volatile main-memory instead of disk pages back it. It can be mapped to a virtual

[1] Assuming the virtual memory area written to is private (`MAP_PRIVATE`).

memory region using the system call `mmap` and accessed through it. As it is possible to manipulate the mapping from virtual memory to the main-memory files using `mmap` and main-memory files are internally backed by physical memory; they can establish a transitive mapping from virtual to physical memory. At any time, this mapping can be modified using `mmap`. Using rewiring memory, it is possible to establish a mapping that is both *visible* and *modifiable* in userspace. To understand the concept, consider the example in Figure 2.2 from the original paper [77] that swaps the content of two pages. On the left side, two virtual memory pages of size $p$ starting at virtual address $b$ are mapped to a main-memory file at offset $0$. Since the main-memory file is transparently backed by the two physical pages $ppage_{42}$ and $ppage_7$, the mappings $vpage_0 \rightarrow ppage_{42}$ and $vpage_1 \rightarrow ppage_7$ have been established. Using `mmap`, it is now easily possible perform the swapping by mapping $vpage_0$ to file offset $p$ and $vpage_1$ to file offset $0$. This changes the physical pages that are backing the virtual pages, resulting in a change of content.

In rewired snapshotting, we utilize this modifiable mapping. Let us assume there is a virtual memory area $b$, on which a snapshot should be created. To snapshot, we simply allocate a new virtual memory area $c$ and `mmap` (or rewire) it to the file, which represents the physical memory, in the same way, as $b$. Consequently, $b$ and $c$ share the same physical pages. If now a write to a page of $b$ is happening, the separation of the snapshot and original version must be performed manually on that page, before the write can be carried out. In the first place, the write must be detected. After detection, an unused page is claimed from the file (which serves as the pool for free pages), the page content is copied over, the write is performed, and $b$ is rewired to map to the new page. By this, it is possible to mimic the behavior of `fork` while staying within a single process. Further, the technique offers the flexibility of snapshotting only a fraction of the data. However, rebuilding the mapping can also be quite expensive as we will see.

### 2.4.3 Reevaluating the State-of-the-Art

As we have discussed the different state-of-the-art methods of physical and virtual snapshotting that are present, let us now try to understand their strengths and limitations. This analysis will point us directly to the requirements we have on our custom system call, that we will use in AnKer to power snapshotting. In the experiment we are going to conduct in Section 2.4.3, we evaluate the time to *create a snapshot* in the sense of establishing a separate view on the data. While for physical snapshotting, this means creating a deep physical copy of the data, for virtual snapshotting, it does not trigger any physical copy of the data. Still, virtual snapshotting has to perform a certain amount of work as we will see. We will perform the experiment as a stand-alone micro-benchmark to focus entirely on the snapshotting costs and to avoid interference with other components, that

are present in our prototype of the AnKer concept. We use a table with $n = 50$ columns, stored in a columnar fashion, where each column has a size of 200MB. The question remains which page size to use. To make snapshotting as efficient as possible, we want to back our memory with pages *as small as available*. This ensures that the overhead of copy-on-write on the level of page granularity is minimal. Consider the case where our 200MB column is either backed by 100 huge pages or 51,200 small pages. In the former case, 100 writes would cause a COW of the entire column (200MB) in the worst case, resulting in a full physical separation of the snapshotted column and the base column. In the latter case, 100 writes would trigger COW of at most 100 small pages (400KB), physically separating only $0.2\%$ of the snapshotted column from the base column.

### System Setup

Before the start of the evaluation, let us look at the setup. We perform all of the following experimental evaluations on a server consisting of two quad-core Intel Xeon E5-2407 running at 2.2 GHz. The CPU does neither support hyper-threading nor turbo mode. The sizes of the L1 and L2 caches are 32KB and 256KB, respectively, whereas the shared L3 cache has a capacity of 10MB. The processor can cache 64 entries in the fast first-level data-TLB for virtual to physical 4KB page address translations. In a slower second-level TLB, 512 translations can be stored. In total, the system is equipped with 48GB of main memory, divided into two NUMA regions of 24GB each. For the upcoming micro-benchmarks of this Section, we deactivate one CPU and the attached NUMA region to stay local on one socket. For the experimental evaluation in Section 2.6, we use both sockets. The operating system is a 64-bit version of Debian 8.16 with our customized Linux kernel (version 4.8.17), that has been extended with our `vm_snapshot` system call. The codebase is written in C++ and compiled using g++ 6.3.0 with optimization level O3.

### Creating a Snapshot

To simulate snapshotting on a subset of the data, we create a snapshot on the first $p$ columns of the table $T$. Let us precisely define how the individual snapshotting techniques behave in this situation:

(a) **Physical**: to create a snapshot of $p$ columns of table $T$, we allocate a fresh virtual memory area $S$ of size $p \cdot l$ pages, where $l$ denote the number of pages per column. Then, we copy the content of $p$ columns of $T$ into $S$ using `memcpy`. $S$ represents the snapshot.

(b) **Fork-based**: to create a snapshot of $p$ columns of table $T$, we create a copy of the process containing table $T$ using `fork`. Independent of $p$, this snapshots the entire table. The first $p$ columns of table $T'$ contained in the child process represent the snapshot. The virtual memory areas representing $T$ and $T'$ are declared as *private*, such that writes to one area are isolated from the other area.

(c) **Rewiring**: to create a snapshot of $p$ columns of table $T$, we first have to inspect by how many VMAs each column is actually described. As a VMA describes the characteristic properties of a consecutive virtual memory region, it is possible that a column is represented by only a single VMA (best case), by one VMA per page (worst case), or anything in between. The more writes happened to a column and the more copy-on-writes were performed, the more VMAs a column is backed by. Eventually, every page is described by its individual VMA. To create the snapshot, we first allocate a fresh virtual memory area $S$ of size $p \cdot l$ pages, where $l$ denote the number of pages per column. For each VMA that is backing a portion of the $p$ columns in $T$, we now rewire the corresponding part of $S$ to the same file offset. Additionally, we use the system call `mprotect` to set the protection of $S$ to read-only. This is necessary to detect the first write to a page to perform a manual copy-on-write. $S$ represents the snapshot.

| **Method** | **Pages Modified per Column** | **1 Col** [ms] | **25 Col** [ms] | **50 Col** [ms] |
|---|---|---|---|---|
| Physical | – | 108.09 | 2693.69 | 5382.87 |
| Fork-based | – | 108.28 | 108.28 | 108.28 |
| Rewiring | 0 | 0.02 | 0.39 | 7.72 |
| Rewiring | 500 | 1.22 | 30.90 | 61.87 |
| Rewiring | 5000 | 14.17 | 352.15 | 712.96 |
| Rewiring | 50000 | 169.28 | 4210.17 | 8459.67 |

*Table 2.1: Creating a snapshot using state-of-the-art techniques. We vary the number of columns on which we snapshot. For rewiring, the number of modified pages influences the runtime. Thus, we show the snapshotting cost after 0, 500, 5000, and 50000 pages were modified per column.*

Table 2.1 shows the results. We vary the number of columns to snapshot $p$ from $1$ column ($2\%$ of the table) over $25$ columns ($50\%$ of the table) to $50$ columns ($100\%$ of the table) and show the runtime in ms to create the snapshot. For rewiring, we vary the pages that have been modified (by writing the first $8$B of the page) before the snapshot is taken, as it influences the runtime. We test the case where no write has happened, and a single VMA backs each column. Further, we measure the snapshotting cost after $500$ pages,

$5000$ pages, and $50000$ pages have been modified. These number of writes lead to $995$, $9483$, and $51177$ number of VMAs backing a column. First of all, we can see that physical snapshotting is quite expensive, as it creates a deep copy of the columns already at snapshot creation time. As expected, we can observe a linearly increasing cost with the number of columns to snapshot. In contrast to that, fork-based snapshotting is independent of the number of requested columns, as it snapshots the entire process with the whole table in any case. When snapshotting $50\%$ of the table, fork-based snapshotting is over an order of magnitude faster than physical snapshotting, as it duplicates solely the virtual memory, consisting of the VMAs and the page table. The runtime of rewiring is highly influenced by the number of written pages, respectively the number of VMAs per column. The more VMAs we have to touch to create the snapshot, the higher the runtime. If we have as many VMAs as pages (the case after $50000$ writes), the runtime of rewiring is higher than the one of physical snapshotting. However, we can also see rewiring is significantly faster than the remaining methods if fewer VMAs need to be copied. For instance, after $500$ writes, rewiring is around two orders of magnitude faster for a single column and almost factor two faster for snapshotting the entire table.

#### Summary of Limitations

The performance of rewiring for snapshot creation is highly influenced by the number of VMAs per column. For every VMA, a separate `mmap` call must be carried out – a significant cost if the number of VMAs is large. Unfortunately, when using rewiring, an increase in the amount of VMAs over time is not avoidable.

Still, we believe in rewiring for efficient snapshotting. However, it can not show its full potential. If we carefully inspect the description of rewired snapshotting in Section 2.4.3 again, we can observe that rewiring implements a *workaround* of the limitations of the OS. We are forced to manually rewire the virtual memory areas described by the VMAs to create a snapshot — because there is no way to copy a virtual memory area. We have to perform another pass over the source VMAs to set the protection using the system call `mprotect` to read-only — instead of setting it directly when copying the virtual memory area. It is also expensive to keep track of shared physical pages in the presence of multiple snapshots.

Naturally, rewiring hits the limits of the vanilla kernel. Therefore, in the following Section, we will propose a custom system call that tackles these limitations — leading to a much more straight-forward and efficient implementation of virtual snapshotting, which we will finally use in AnKer.

## 2.5 System Call vm_snapshot

In the previous section, we have seen the limitations of the state-of-the-art kernel. Let us now discuss how we can overcome them by introducing our custom system call `vm_snapshot`. In our implementation of rewired snapshotting, we have experienced the need to snapshot virtual memory areas directly. By default, the kernel does not support this task. As a workaround, we had to rewire a new virtual memory area in the same way as the source area which is a costly process as it involves repetitive calls to `mmap`.

### 2.5.1 Semantics

To solve this problem, we have to introduce a new system call, that will be the core of our snapshotting mechanism. Before, let us precisely define what *snapshotting a virtual memory area* means in this context. Let us assume we have a mapping from $n$ virtual to $n$ physical pages starting at virtual address $b$. The first virtual page covering the virtual address space $[b; b + p - 1]$ ($vpage_{b0}$) is mapped to the physical page $ppage_{42}$. The second virtual page covering virtual address space $[b + p; b + 2 \cdot p - 1]$ ($vpage_{b1}$) is mapped to another physical page $ppage_7$ and so on.

Now, we want to create a new virtual memory area starting at a virtual address $c$, that maps to the *same* physical pages. Thus, the virtual page covering $[c; c + p - 1]$ should map to $ppage_{42}$, the virtual page $[c + p; c + 2 \cdot p - 1]$ should map to $ppage_7$ and so on. We define the following system call to encapsulate the described semantics:

```
void* vm_snapshot(void* src_addr, size_t length);
```

This system call takes the `src_addr` of the virtual memory area to snapshot and the `length` of the area in bytes. Both `src_addr` and `length` must be page aligned. It returns the address of a new virtual memory area of size `length`, that is a snapshot of the virtual memory area starting at `src_addr`. The new memory area uses the same update semantics as the source memory area, i.e., if the virtual memory area at `src_addr` has been declared using `MAP_PRIVATE | MAP_ANONYMOUS`, the new memory area is declared in the same way. Besides, the new memory area follows the same NUMA allocation policy as any virtual memory of the system – by default, the physical page serving a COW is allocated on the NUMA region of the socket, that executes the thread causing the COW.

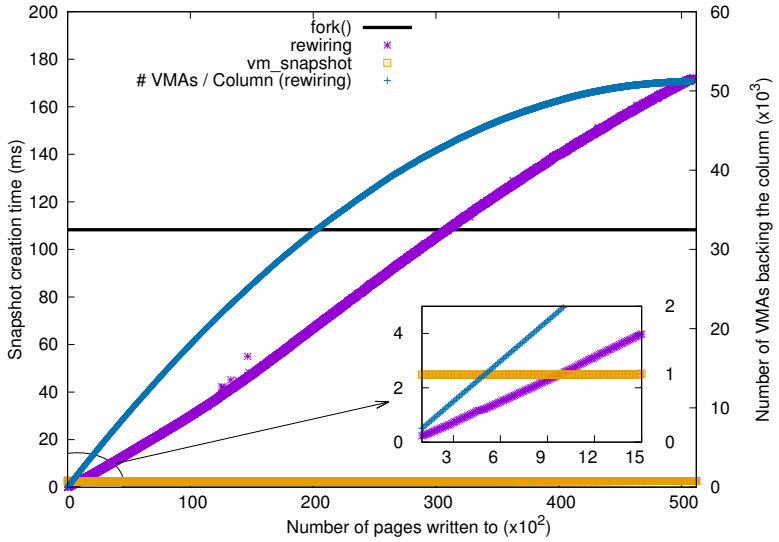
### 2.5.2 Implementation

Implementing a system call that modifies the virtual memory subsystem of Linux is a delicate challenge. In the following, we will provide a high-level description of the system call behavior. For the interested reader, we provide a full patch for Linux kernel v4.16 in the Appendix A. On a high level, `vm_snapshot` internally performs the following steps: (1) Identify all VMAs that describe the virtual memory area $[\texttt{src\_addr}, \texttt{src\_addr} + \texttt{length} - 1]$. (2) Reserve a new virtual memory area of size `length` starting at virtual address `dst_addr`. (3) Copy all of the previously identified VMAs and update them to describe the corresponding portions of virtual memory in $[\texttt{dst\_addr}, \texttt{dst\_addr} + \texttt{length} - 1]$. (4) For each VMA which describes a private mapping (which is the standard case in AnKer), additionally copy all existing PTEs and update them to map the corresponding virtual pages in $[\texttt{dst\_addr}, \texttt{dst\_addr} + \texttt{length} - 1]$. This system call `vm_snapshot` will form the core component of creating snapshots on columns in AnKer. It is the call that we use in Figure 2.1 in Step ④ and Step ⑦.

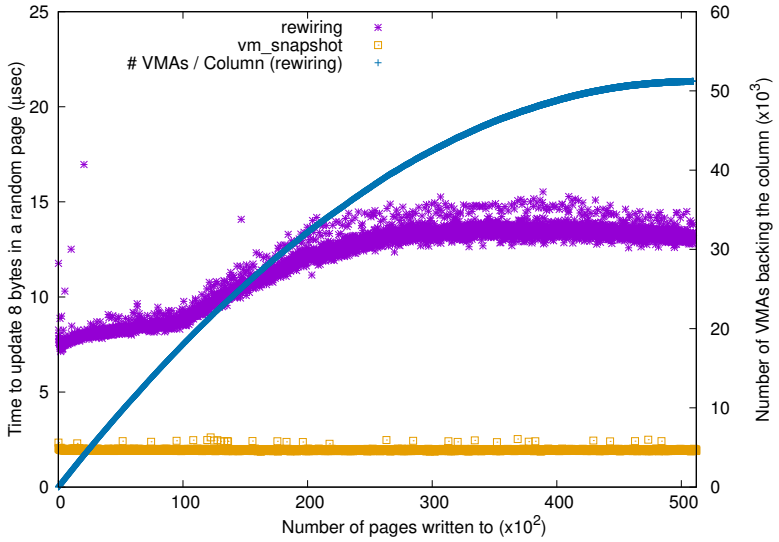
### 2.5.3 Evaluating Virtual Memory Snapshotting

Let us now see how our custom system call `vm_snapshot` performs in comparison with its direct competitor rewiring. We excluded the baseline of physical snapshotting, as it is already out of consideration for AnKer due to high cost and low flexibility. We first look at the snapshot creation time for a single column of 200MB. The previous experiment presented in Table 2.1 showed that rewiring is profoundly influenced by the number of VMAs that are backing the column to snapshot. To analyze this behavior in comparison with `vm_snapshot`, we run the following experiment: for each of the $51{,}200$ pages of the column, we perform precisely one write to the first 8B of the page. In the case of rewiring, this write triggers the COW of the touched page and thus, creates a separate VMA describing it. After each write, we create a new snapshot of the column and report the creation time.

In Figure 2.4(a), as predicted, the snapshot creation cost of rewiring is highly influenced by the number of VMAs that is increasing with every modified page. To visualize this correlation, we plot the number of VMAs per column for rewiring alongside with the snapshot creation time. In contrast to rewiring, our system call `vm_snapshot` shows both a stable and low runtime over the entire sequence of writes. After only around $1000$ writes have happened (see zoom-in of Figure 2.4(a)), the snapshotting cost of `vm_snapshot` already becomes lower than the one of rewiring. After all $51{,}200$ writes have been carried out, `vm_snapshot` is 68x faster than rewiring. This shows the tremendous effect of avoiding repetitive calls to `mmap`.




(a) **Comparison of snapshot creation times**. The time to snapshot a single column is shown on the left $y$-axis for rewiring respectively `vm_snapshot`. To enhance the visualization, we also show a zoom-in.




(b) **Comparison of writes** to the snapshotted column. On the left $y$-axis, the time to perform a write of 8B is shown.

*Figure 2.4:* ***Comparison of vm_snapshot and rewiring*** *in terms of snapshotting and write cost. After every write to a page, a new snapshot is taken. Additionally, we show the number of VMAs per column for rewiring on the right y-axis.*

However, we should also look at the actual cost of writing the virtual memory. In the case of rewiring, the triggered COW is handled by copying the page content to an unused page and rewiring that page into the column. In the case of `vm_snapshot`, which works on anonymous memory and relies on the COW mechanism of the operating system, no manual handling is necessary. This becomes visible in the runtime shown in Figure 2.4(b). Writing a page of the column snapshotted by `vm_snapshot` is up to 6x faster than writing to one created by rewiring, as the operating system handles the entire COW. No protection must be set manually, and no signal handler is necessary to detect the write to a page.

### 2.5.4 MVCC Scan Performance

In this part of our microbenchmarks, we use the TPC-H schema to highlight the impact of version chains on the performance of long-running analytical queries. We perform the experiment shown in Figure 2.5, which resembles executing mixed workloads under homogeneous processing.

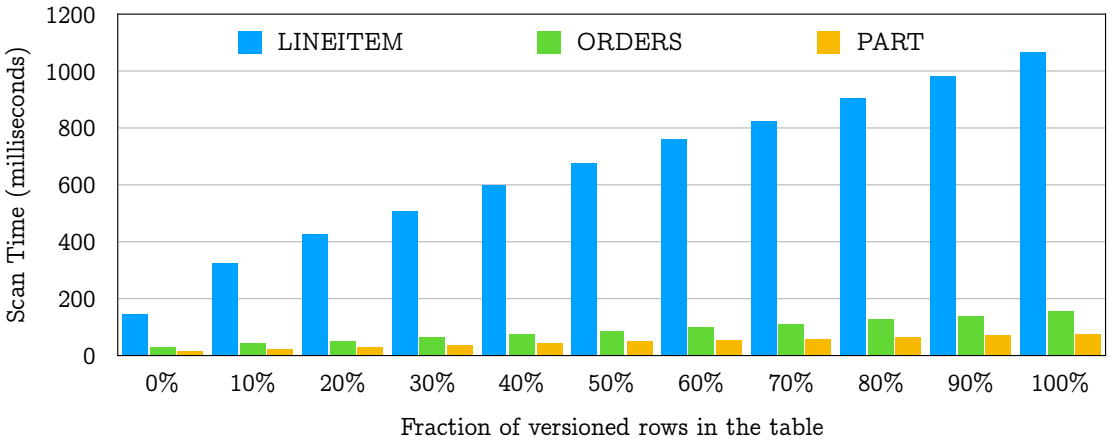



*Figure 2.5:* ***Runtime of scanning versioned tables****. We vary the amount of versioned rows and perform a full scan.*

In this experiment, we vary the number of rows that are versioned in the tables `LINEITEM`, `ORDERS`, and `PART` and measure the time it takes to perform a full scan of the table. The versioned rows are uniformly distributed across the table. To improve scan performance in the presence of versioned rows, we apply an optimization technique introduced by HyPer [71]: for every 1024 rows, we keep the position of the first and the last versioned row. With this information, it is possible to scan in tight loops between versioned records without performing any checks.

Nevertheless, in Figure 2.5 we can see that this optimization cannot defuse the problem entirely. With an increase in the number of versioned rows, we see a drastic increase in the runtime of the scan as well. Scanning a table that is completely versioned takes around 5 times longer than scanning an unversioned table. This unversioned table essentially resembles the situation when scanning in a snapshot under heterogeneous processing.

### 2.5.5 Snapshot Creation Cost

Let us now inspect the cost of snapshot creation in AnKer. Due to our flexible system call `vm_snapshot`, we are able to snapshot virtually at the granularity of individual columns.

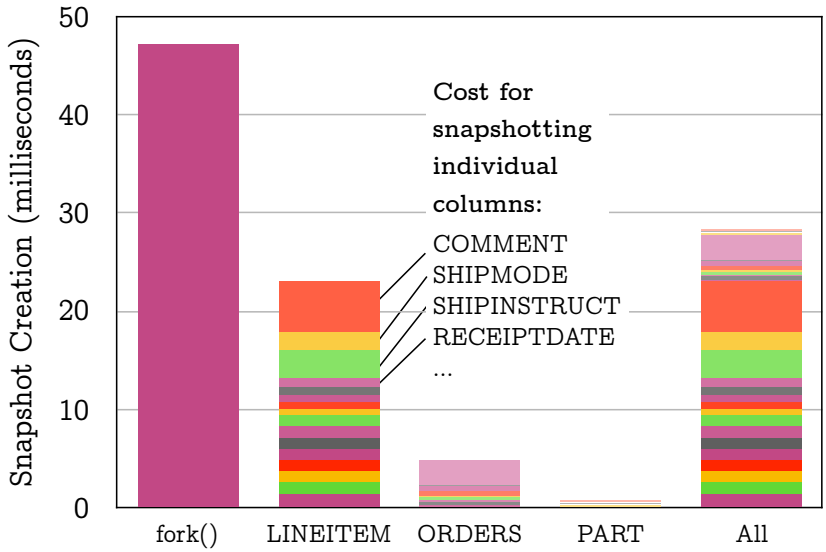



*Figure 2.6:* ***Snapshot creation cost*** *for the individual columns of* `LINEITEM`*,* `ORDERS`*, and* `PART` *utilizing our system call* `vm_snapshot` *in comparison with using* `fork`*.*

To demonstrate the benefit of this flexible approach, we present in Figure 2.6 the cost of snapshotting the individual columns of the `LINEITEM`, `ORDERS`, and `PART` table of the TPC-H benchmark inside of AnKer in form of stacked bars. Each layer in a bar resembles the cost of snapshotting a single column of the respective table. The bar *All* presents the cost of snapshotting all three tables. In comparison, we show the cost of forking the process in which AnKer is running using the system call `fork`. We make sure that when performing the `fork`, the process is in the same state as when performing the snapshotting using `vm_snapshot`. At this point in time, the AnKer process has a size of 5.2GB in terms of virtual memory.

As we can see in Figure 2.6, the cost of snapshotting individual columns of the TPC-H tables is negligibly cheap. Thus, if a transaction accesses only a portion of the attributes, the cost of preparing the snapshot stays as low as possible as well. Nevertheless, even when snapshotting all columns of all tables, our approach is considerably cheaper than using the `fork` system call. The problem of `fork` is that the virtual memory of the entire process containing $5.2$GB of virtual memory is replicated. Besides the tables, which consume only around $1.5$GB of memory, this includes the used indexes, the version chains, the timestamp arrays, and various meta-data structures.

## 2.6 Experimental Evaluation

After the description of the processing concept of AnKer and the introduction of `vm_snapshot` to efficiently snapshot virtual memory areas, let us now start with the experimental evaluation of the actual system. As AnKer relies on a hybrid processing model, we want to test it against MVCC using a single execution engine. Additionally, we want to test its snapshotting capabilities against fork-based snapshotting. Our prototype is designed in a way also to support both hybrid processing using `fork` as well as MVCC using a single execution engine by disabling snapshotting.

### 2.6.1 System Configurations

Let us define the precise configurations we are going to evaluate:

1. **MVCC in a Single Execution Engine, Full Serializability (abbreviated by SEE_FS)**. We configure our prototype such that *no snapshots* are taken at all. Thus, there is only a single execution engine with the most recent representation of the database. Both OLTP transactions and OLAP queries run on this execution engine under *full serializability* guarantees. A separate garbage collection mechanism cleans the version chains created by the updates. The system uses a thread that passes over the version chains every second and deletes all versions that are not visible to the oldest active transaction in the system. To speed up scanning over versioned data, we apply an optimization technique introduced by [71]: for every $1024$ rows, we keep the position of the first and the last versioned row. With this information, it is possible to scan in tight loops between versioned records without performing any checks.

2. **MVCC in a Single Execution Engine, Snapshot Isolation (abbreviated by SEE_SI)**. As in (1), *no snapshots* are taken. There is only a single execution engine with the most recent representation of the database. Both OLTP transactions and OLAP queries run in this component under *snapshot isolation* guarantees and thus, no read set validation is performed. The same garbage collection and scan optimization as in (1) are applied.

3. **MVCC in a Single Execution Engine, Read Uncommitted (abbreviated by SEE_RU)**. As in (1) and (2), *no snapshots* are taken. There is only a single execution engine with the most recent representation of the database. Both OLTP transactions and OLAP queries run in this component under *read uncommitted* guarantees and thus, running transactions/queries can see uncommitted changes. Not garbage collection is necessary since updates do not create versions. Scan optimization is not necessary as well.

4. **MVCC in a Hybrid Execution Engine using `vm_snapshot`, Full Serializability (abbreviated by HEE_AnKer)**. The OLTP transactions run in the OLTP execution engine, and the OLAP queries run in the OLAP execution engine. The creation of snapshots works in a lazy fashion using our system call `vm_snapshot` as described in Section 2.3.2. We additionally force the transactions, that are classified as OLTP to abort, as soon as they are forced to find the right tuple version from the version chain. This prevents these transactions from doing unnecessary work before they abort.

5. **MVCC in a Hybrid Execution Engine using `fork`, Full Serializability (abbreviated by HEE_fork)**. Same as (4), except that we use `fork` to perform the virtual snapshotting instead of `vm_snapshot`. A call to `fork` launches a new process of AnKer that runs the OLAP queries. To create a consistent snapshot, HEE_fork blocks all commits until the fork is complete. This can be replaced with log-based rollback similar to HyPer[50] to improve the OLTP throughput, but it adds additional cost to snapshot preparation.

### 2.6.2 Experimental Setup

To evaluate the system under a complex HTAP workload, we define the following mixture of OLTP transactions and OLAP queries:

On the OLTP side, we use three transactions from the TPC-C benchmark: `Payment`, `NewOrder`, and `OrderStat`. These three transactions access all nine tables of the TPC-C database and perform updates on the tables `stock`, `order_line`, `orders`, `new_order`, and `district`. For each transaction that is submitted to the

system, we pick the configuration parameters randomly within the bounds given in the TPC-C specification. We populate the database with 40 warehouses. We support two different types of access pattern for the transactions. For the first one, the accesses are *uniformly* distributed across warehouses and districts. For the second one, 50% of the accesses are *skewed* towards five given warehouse/district pairs. The remaining 50% follow a uniform distribution.

On the side of OLAP, we use eight synthetic queries, which are dominated by scanning, grouping, and aggregation. Figure 2.7 shows the precise queries. We pick the query `StockScan` (OLAP-Q1) as described in [96], which operates on the `warehouse` and `stock` tables. Further, we add the two single-table queries OLAP-Q2 and OLAP-Q3, which group and aggregate on `order_line`. To have scan-heavy queries, we add OLAP-Q4 and OLAP-Q5, which simply perform full table scans on `orders` respectively `new_order`. Finally, we add the three fast queries OLAP-Q6, OLAP-Q7, and OLAP-Q8, which perform scans and aggregations on the columns of a single table.

Unless mentioned otherwise, the upcoming experiments use 6 threads to process the stream of incoming OLTP transactions and 2 threads to answer OLAP queries.

**OLAP-Q1 StockScan [25]**

```
select w_id, count(*)
  from warehouse, stock
 where w_id = s_w_id
group by w_id;
```

**OLAP-Q2**

```
select ol_d_id,
       avg(ol_amount)
  from order_line
group by ol_d_id;
```

**OLAP_Q3**

```
select ol_w_id,
       sum(ol_quantity),
       avg(ol_amount)
  from order_line
group by ol_w_id;
```

**OLAP-Q4 FULL TABLE SCAN**

```
select *
  from orders;
```

**OLAP-Q5 FULL TABLE SCAN**

```
select *
  from new_order;
```

**OLAP Q6 COLUMN SCAN**

```
select avg(d_tax),
       avg(d_ytd)
  from district;
```

**OLAP Q7 COLUMN SCAN**

```
select avg(ol_amount)
  from order_line;
```

**OLAP Q8 COLUMN SCAN**

```
select avg(s_quantity)
  from stock;
```

*Figure 2.7: The **eight OLAP queries** we use in the evaluation.*

### 2.6.3 Snapshotting Cost and OLAP Latency

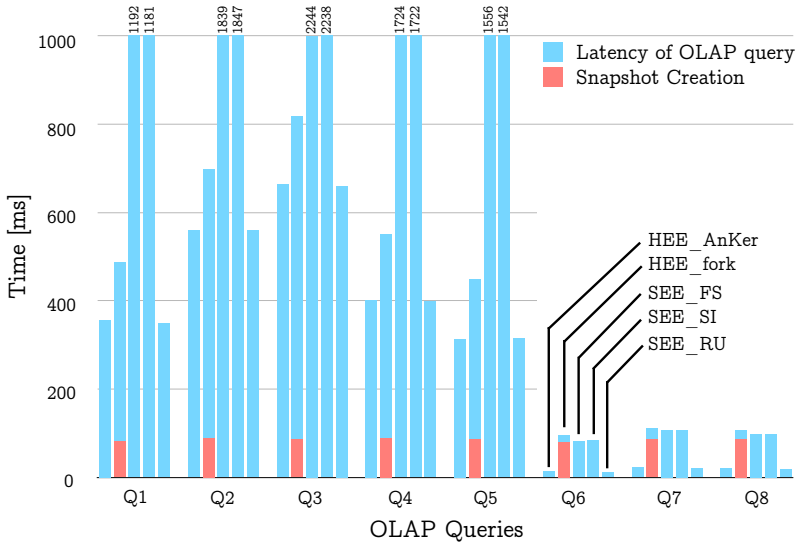
Let us start with an evaluation of the core mechanism of AnKer: the fast snapshotting using our system call `vm_snapshot`. Our initial motivation of this project was to enable virtual snapshotting without the overhead of fork-based snapshotting. Thus, let us now first see how hybrid processing using our system call (HEE_AnKer) competes with

hybrid processing using `fork` as originally done by HyPer (HEE_fork). Both run under full serializability guarantees. Additionally, we compare the two hybrid approaches with MVCC using a single execution engine under three different isolation levels (SEE_FS, SEE_SI, SEE_RU).

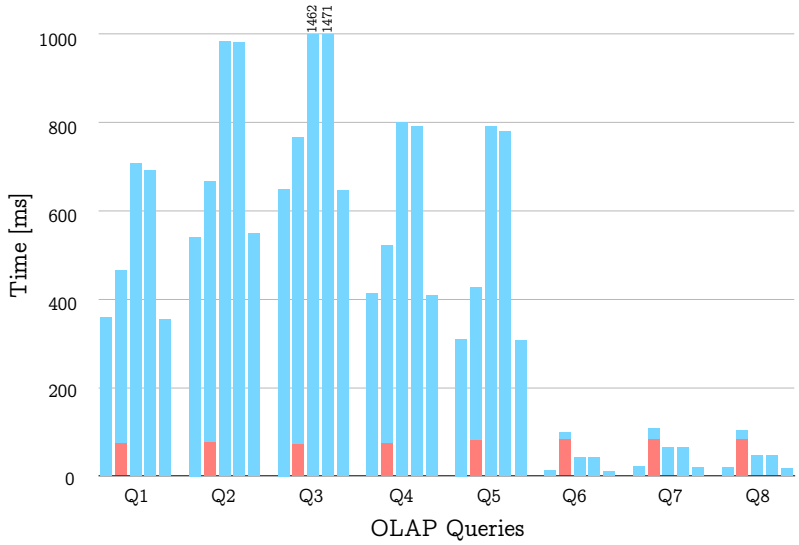
In the following experiment, we will answer two questions: First, how expensive is the snapshotting mechanism using our system call in comparison with the alternatives under a real-world HTAP workload? Second, what is the impact of the snapshotting mechanism and the hybrid processing under MVCC on the latency of the OLAP queries? To answer these questions, we perform the following experiment: To sustain the system, we fire an infinite stream of OLTP transactions randomly picked from the set of TPC-C transactions. After five seconds, we fire a random OLAP query from the set depicted in Figure 2.7 and repeat firing random OLAP queries every $500$ms. For every fired query, we create a new snapshot for the hybrid approaches HEE_AnKer and HEE_fork. After three minutes, we terminate the experiment and report the average of all observed snapshotting times and query latencies.

Figure 2.8 shows the results grouped by the OLAP queries. In Figure 2.8(a), we use the OLTP workload with a uniform access pattern while in Figure 2.8(b), we use the skewed OLTP workload focusing on five hot warehouse/district combinations. For each of the eight OLAP queries, we report the snapshot creation time as well as the latency in ms for each of the five tested methods.

Let us first have a look at the results on the uniform pattern in Figure 2.8(a). If we compare the baselines, we can see the significant cost of the snapshotting phase in HEE_fork, caused by replicating the process, followed by a fast query answering part. The approaches SEE_FS and SEE_SI using a single execution engine do not have an explicit snapshot creation phase but suffer from very high query execution times of up to $2200$ms for $Q_3$ (cut off at $1000$ms in the plot), as the OLAP query has to work its way through the version chains, which are build up by the OLTP stream. In comparison to the baselines, HEE_AnKer combines the best of both worlds: its snapshot creation time is so short that it is not even visible in the plots. This is caused by the fact that it snapshots only the columns that are touched by the respective query using `vm_snapshot`. In comparison to HEE_fork, the snapshotting phase of HEE_AnKer is more than $100$x faster. After snapshotting, the actual query answering part is equally fast as for HEE_fork. We want to point out that the total latency for HEE_AnKer (including snapshot creation and query answering) under full serializability guarantees almost equals the runtime of SEE_RU, which runs only on the isolation level of read-uncommitted. We can also observe that depending on the query, the expensive snapshotting phase of HEE_fork can have a drastic impact on the overall latency. For $Q_6$, $Q_7$, and $Q_8$, the snapshotting cost of HEE_fork dominates the latency, and HEE_AnKer achieves a speedup of $4.9$x to $7.5$x.




(a) System sustained using OLTP transactions with **uniform access pattern**.




(b) System sustained using OLTP transactions with **skewed access pattern**.

*Figure 2.8: Snapshotting Cost and Latency of OLAP queries.*

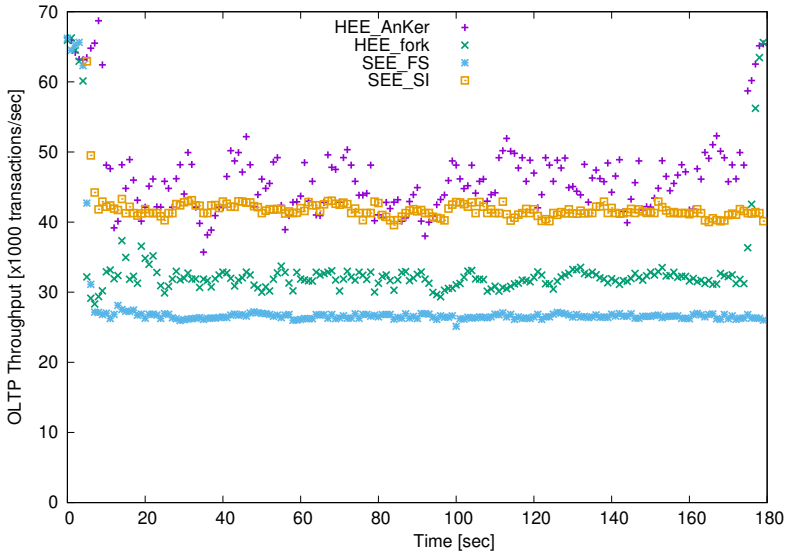
If we look at the results on the skewed pattern in Figure 2.8(b), we can observe that especially the approaches using a single execution engine and a higher isolation level (SEE_FS and SEE_SI) massively benefit from the skew. We can also see that for the three faster queries $Q_6$, $Q_7$, and $Q_8$, HEE_fork shows now the overall highest latency: the snapshotting mechanism using `fork` is more expensive than the query answering part of any competitor.

### 2.6.4 Transaction Throughput

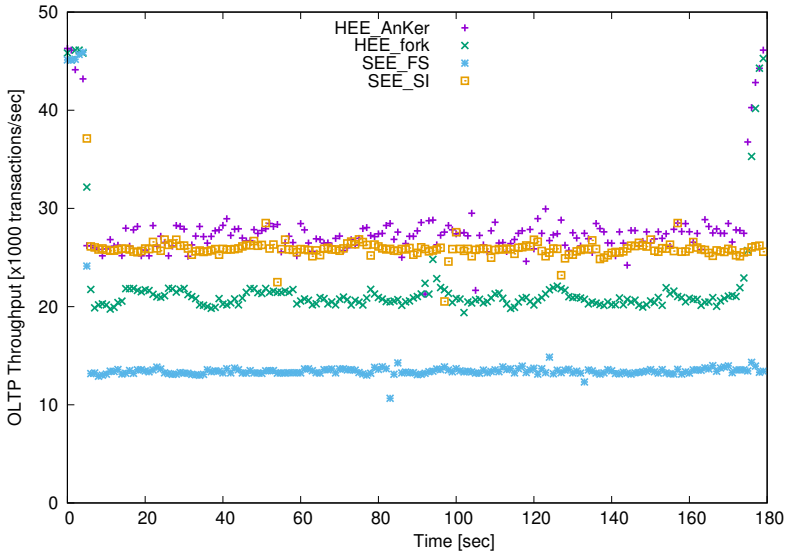
After inspecting the snapshotting cost and the latency of the OLAP queries, let us now investigate the previous experiment from Section 2.6.3 from the OLTP side. As we fire an infinite stream of OLTP transactions over a time interval of three minutes, we plot the OLTP throughput achieved per second. As before, starting after five seconds, we fire a random OLAP query every $500$ms and snapshot before every OLAP query for the hybrid methods. We exclude the results for SEE_RU, as the OLTP throughput is extremely high in comparison to the counterparts with higher isolation levels.

Figure 4.11 shows the results for the uniform OLTP access pattern (Figure 2.9(a)) and the skewed OLTP access pattern (Figure 2.9(b)). For the uniform case, in the first five seconds, no OLAP query is running, and we see the maximum OLTP throughput of the system which locates between $60$k and $70$k transactions per second. As soon as the first OLAP query arrives, the throughput significantly drops for all methods due to congestion. As expected, we observe the lowest throughput of around $28k$ transactions per second under SSE_FS because of the expensive commit phase validation that is performed, and a slightly higher throughput of around $30$k to $38$k transactions per second for HEE_fork. The costly snapshotting utilizing `fork` heavily throttles the OLTP throughput. The throughput of SEE_SI is very stable around $42$k transactions per second. To our surprise, this is a lower throughput than the average throughput achieved by HEE_AnKer with $47$k transactions per second which has one major reason: Before SEE_SI successfully reads a value that is stored in a column, it must validate whether the value is versioned or not. If it is versioned, it reads the timestamp of the most recent version and the pointer to the version chain. The read is successful if the current timestamp of the tuple is smaller than transaction's `begin-timestamp`. If not, the version chain is traversed to find the valid version. However, for the HEE_AnKer this is not the case. Since we only support full-serializability, the OLTP transactions are not allowed to read from the version chains. For every read, the transaction only reads the timestamp of the current tuple version. If the timestamp is smaller than transaction's `begin-timestamp`, the transaction can read the value. Otherwise, it prematurely aborts without doing the read. Due to fewer comparisons and metadata validation, a single read access for full-serializable OLTP transaction is faster than the reads performed by SI based protocol where the transaction can still commit after reading from the version chain. For HEE_AnKer, we observe a high variance in the throughput which is due to different snapshot creation time for different OLAP queries and the copy-on-write cost. We also observe a stable behaviour for HEE_Fork due to relatively stable `fork` cost.

For the skewed distribution in Figure 2.9(b), we see a lower throughput than for the uniform pattern across all methods. This is caused by update conflicts that must



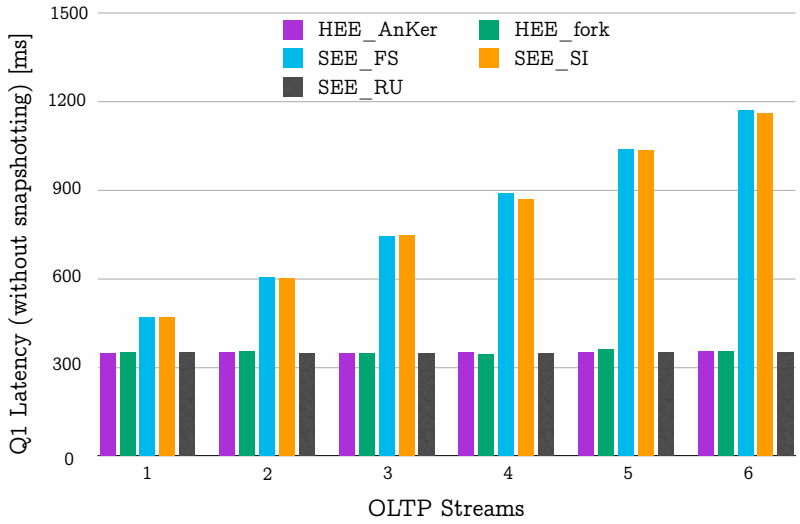

(a) Throughput of OLTP Transactions with **uniform access pattern**.



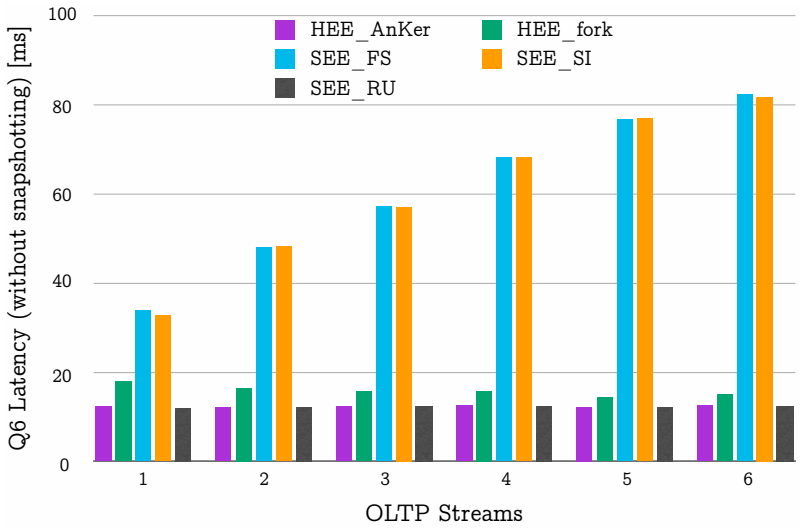

(b) Throughput of OLTP Transactions with **skewed access pattern**.

*Figure 2.9: Throughput of OLTP Transactions.*

be serialized for the contended transactions. We can also observe in this plot that the variance for HEE_AnKer and HEE_fork is smaller than in the uniform case since fewer copy-on-writes are performed.




(a) **Latency for Q1** while **varying the number of OLTP streams**. The number of OLAP streams is 2.




(b) **Latency for Q6** while **varying the number of OLTP streams**. The number of OLAP streams is 2.

*Figure 2.10: Varying the number of streams used for processing.*

### 2.6.5 Scaling

Our system essentially implements parallelism on two layers: On the first layer, we parallelize OLTP and OLAP execution by maintaining a hybrid execution engine. On the second layer, we apply MVCC inside each engine to ensure a high concurrency

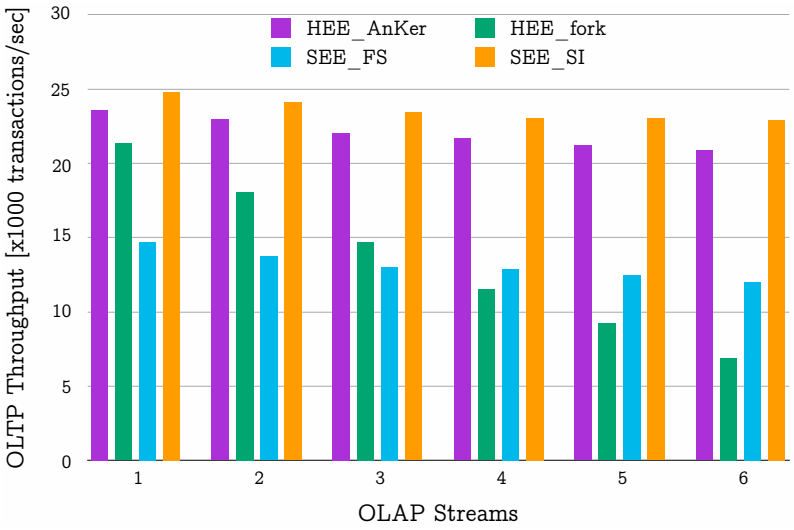



*Figure 2.11: Varying the number of streams used for processing.*

among transactions of a single type. In this regard, let us now investigate how well AnKer scales with the number of OLTP and OLAP streams, that are used to process the transactions and queries.

In Figure 2.10, we investigate the scaling capabilities along two dimensions. On the first dimension, we fix the number of OLAP streams and vary the number of available OLTP streams. On the second dimension, we fix the number of OLTP streams and vary the number of available OLAP streams. The experimental setup is the same as in Section 2.6.3 and Section 4.8.2. In Figure 2.10(a) and Figure 2.10(b), we report the latency of the OLAP queries $Q_1$ and $Q_6$ when fixing the number of OLAP streams to $2$ and varying the number of OLTP streams from $1$ to $6$ to sustain the system with varying OLTP load. In Figure 2.11, we fix the number of OLTP streams to $2$ and vary the number of OLAP streams from $1$ to $6$. Here, we show the average throughput over the run of $180$ seconds.

From Figure 2.10(a) and Figure 2.10(b), we can see that the hybrid approaches are largely unaffected by the number of OLTP streams. The reason for this is that the OLAP processing happens in isolation to the version chain building, that is happening in the OLTP execution engine. In contrast to that, the OLAP latency of SEE_FS and SEE_SI heavily decreases with an increase in OLTP streams, as more OLTP streams build up more version chains that must be traversed by an OLAP query. SEE_RU is again unaffected, as it simply reads the in-place version without traversing the version chains at all. In Figure 2.11, we can see that the OLTP throughput decreases for all methods with an increase of OLAP streams. However, some methods are more affected

than others. While the throughput of HEE_AnKer decreases only by $11.6\%$ from $1$ to $6$ OLAP streams, the throughput of HEE_fork decreases by $67.6\%$. This is because the expensive snapshotting phase using `fork` interrupts the processing of the OLTP stream for a significant amount of time and decreases the number of OLTP transactions that can be processed in $180$ seconds. Consequently, the effective throughput is decreased.

| **Method** | **1 WH** | **10 WH** | **20 WH** | **30 WH** | **40 WH** | **Slowdown 1WH→40WH** |
|---|---|---|---|---|---|---|
| HEE_AnKer | 51289 | 50525 | 49718 | 48637 | 47729 | 1.07x |
| HEE_fork | 46391 | 42741 | 39172 | 35678 | 31220 | 1.49x |
| SEE_FS | 32456 | 31810 | 31281 | 30299 | 28794 | 1.18x |
| SEE_SI | 48391 | 48027 | 47687 | 47033 | 46237 | 1.05x |

*Table 2.2: Varying the number of warehouses and observing the throughput decrease. The throughput is given in transactions per second. The last column shows the slowdown in throughput from 1 warehouse to 40 warehouses.*

Finally, let us vary the size of the used dataset and see the effect on the OLTP throughput. Additionally to $40$ warehouses, that we used in the previous experiments, we also evaluate $1$, $10$, $20$, and $30$ warehouses in the following and report the slowdown in throughput when increasing the size from $1$ to $40$ warehouses. As expected, HEE_fork is affected the most by an increase of the dataset size with a throughput slowdown of factor $1.49$x, as the process to fork heavily increases in size. For HEE_AnKer, the problem is not that severe as only the touched columns are snapshotted.

## 2.7 Future Work

Our system call `vm_snapshot` is the essential component that powers the hybrid execution engine of AnKer. It enables fast and fine-granular snapshotting in combination with a low memory footprint. Nevertheless, due to its flexibility and general design, it could be applied in a variety of other situations as well. From a more general perspective, `vm_snapshot` can essentially replace any larger **memcpy** operation. While `memcpy` duplicates *all* pages in a memory region in an eager fashion, `vm_snapshot` lazily duplicates only the *modified* pages. As `memcpy` is frequently used at essentially all levels of any software system, such a simple function swap can have a significant impact on performance. From a system perspective, the problem of efficient snapshotting is not limited to relational systems. For instance, **graph processing systems** throttle in the presence of concurrent updates and analytics. Our system call could be used to

snapshot (parts of) the graph and outsource the analytics as in AnKer. Apart from snapshotting, there is the related concept of **checkpointing** [95], where a consistent view of the database has to be stored to disk for recovery purposes. As this is more of a background task with respect to the query processing, it should be as transparent and lightweight as possible. We can ensure this with our system call: if a checkpoint is requested, we create a consistent view of the database using `vm_snapshot` with minimal effort. Afterwards, this view can be spilled to disk asynchronously. Consequently, a system, which supports the creation of snapshots and checkpoints at a high frequency, could be easily extended to run **time travel queries** efficiently: they would either run on snapshots, which are still available in the system, or on checkpoints, that are reloaded via `mmap`.

## 2.8 Conclusion

In this work, we introduced AnKer, a transactional processing concept implementing a hybrid execution engine in combination with MVCC, which works hand in hand with our customized Linux kernel to enable snapshotting at a very high frequency. We have shown that a hybrid design powered by a lightweight snapshotting mechanism fits naturally to the HTAP workloads and improves the throughput of OLAP queries by factors up to $4$x, as it enables fast scans in tight loops. Besides, due to the flexibility of our custom system call `vm_snapshot`, we can limit the snapshotting effort to those columns that are accessed by transactions, allowing a snapshotting speedup of more than factor $100$x over fork-based snapshotting.

Overall, in this chapter, we have seen that a fast snapshotting technique can have a significant impact on the performance of the database system that aims to provide a unique solution for transactional as well as analytical workloads. Therefore, highlighting the friendly nature of snapshots towards hybrid transactional and analytical workloads. In the upcoming chapter, we will look into another class of data management systems, i.e., permissioned-blockchain systems. We will investigate the transaction processing pipeline of Hyperledger Fabric and extend it with some optimizations from the relational database research, which will significantly improve the permissioned blockchain system's performance. We will also look at how we can hamper performance if we use snapshot isolation in an ineffective way as done by Hyperledger Fabric.

# Chapter 3

# Fabric++: Optimizing Transaction Processing in Hyperledger Fabric

Within the last few years, a countless number of blockchain systems have emerged on the market, each one claiming to revolutionize the way of distributed transaction processing in one way or the other. Many blockchain features, such as byzantine fault tolerance, are indeed valuable additions in modern environments. However, despite all the hype around the technology, many of the challenges that blockchain systems have to face are fundamental transaction management problems. These are largely shared with traditional database systems, which have been around for decades already.

These similarities become especially visible for systems, that blur the lines between blockchain systems and classical database systems. A great example of this is Hyperledger Fabric, an open-source permissioned blockchain system under development by IBM. By implementing parallel transaction processing, Fabric's workflow is highly motivated by optimistic concurrency control mechanisms in classical database systems. This raises two questions: (1) Which conceptual similarities and differences do actually exist between a system such as Fabric and a classical distributed database system? (2) Is it possible to improve on the performance of Fabric by transitioning technology from the database world to blockchains and thus blurring the lines between these two types of systems even further? To tackle these questions, we first explore Fabric from the perspective of database research, where we observe weaknesses in the transaction pipeline. We then solve these issues by transitioning well-understood database concepts to Fabric, namely transaction reordering as well as early transaction abort. Our experimental evaluation under the Smallbank benchmark as well as under a custom workload shows that our improved version Fabric++ significantly increases the throughput of successful transactions over the vanilla version by up to a factor of 12x, while decreasing the average latency to almost half.

## 3.1 Introduction

Blockchains are one of the hottest topics in modern distributed transaction processing. However, from the perspective of database research, one could raise the question: what makes these systems so special over classical distributed databases, that have been out there for a long time already?

The answer lies in byzantine fault tolerance: while classical distributed database systems require a trusted set of participants, blockchain systems are able to deal with a certain amount of *maliciously* behaving nodes. This feature opens lots of new application fields such as transactions between organizations, that do not fully trust each other.

Regarding the aspect of byzantine fault tolerance, blockchain systems have a clear advantage over distributed database systems. Unfortunately, with respect to essentially any other aspect of transaction processing, classical database systems are decades ahead of blockchain systems.

A great example for this is the *order-execute* transaction processing model, that prominent systems like Bitcoin [69] and Ethereum [3] implement: In the ordering phase, all peers first agree on a global transaction order, typically using a consensus mechanism. Then, each peer locally executes the transactions in that order on a replica of the state. While this approach is simple, it has two severe downsides: First, the execution of transactions happens in a sequential fashion. Second, as every transaction must be executed on every peer, the performance of the system does not scale with the number of peers. Of course, both parallel execution as well as scaling capabilities have been well-established properties of distributed database systems since many years.

### 3.1.1 Catching up

Still, there are blockchain systems that try to catch up. A prominent example for this is Hyperledger Fabric [18], a popular open-source blockchain system introduced by IBM. Instead of implementing the order-execute model, it follows a sophisticated *simulate-order-validate-commit* model. This model is highly influenced by optimistic concurrency control mechanisms in database systems: Transactions are simulated speculatively in parallel *before* actually ordering them. Then, after ordering, Fabric checks in the validation phase whether the order does not conflict with the previously computed simulation effects. Finally, the effects of non-conflicting transactions are committed. The advantages of this model are clear: parallel transaction execution and therefore the ability to scale — features, which will be mandatory for any upcoming blockchain

system, that aims at high performance.

We strongly believe that Fabric's ambitions in transitioning technology from the world of databases to blockchains are a step in the right direction. Unfortunately, its implementation of parallel transaction processing still suffers from certain problems which highly limit the gain, that can be achieved by concurrency. These problems can be identified easily in two simple experiments.

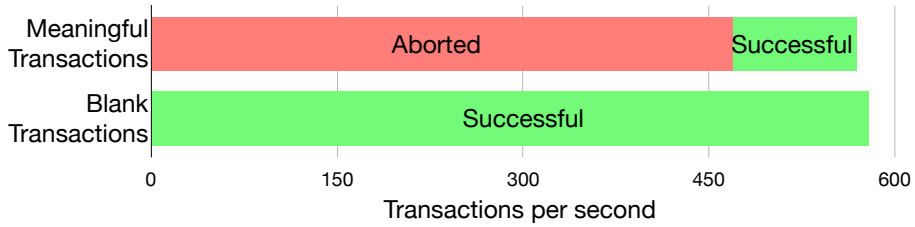



*Figure 3.1: Transactions per second of vanilla Fabric when meaningful transactions are fired as described in Section 3.6 for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1%. Additionally, we show the throughput when blank transactions are fired.*

In the first experiment (Figure 3.1, top bar), we submit a stream of meaningful transactions, which originate from an asset transfer scenario, and report the throughput, divided into aborted and successful transactions. This experiment reveals a severe problem of Fabric: a large number of transactions end up as being aborted. The reason for all these aborts are serialization conflicts, a negative side-effect of concurrent execution.

If we want to increase the number of successful transactions, we essentially have two options: Either we (a) increase the overall throughput of the system or (b) turn transactions, that would have been aborted by Fabric, to successful ones. Unfortunately, option (a) is hardly applicable in Fabric. We can see this in the second experiment (Figure 3.1, bottom bar), where we submit blank transactions without any logic. Interestingly, the total throughput of blank and meaningful transactions essentially equals. This reveals, that the overall throughput of the system is not dominated by the core components of transaction processing, but actually by other auxiliary factors: cryptographic computations and networking overhead.

### 3.1.2 Fabric++

Thus, option (b) is the key: we have to turn transactions, that would have been aborted by Fabric, to successful ones. We achieve this, by transitioning a well-known technique

from database systems to Fabric: *transaction reordering*. Instead of arbitrarily ordering transactions, we inspect the transaction semantics and arrange the transactions in a way such that the number of serialization conflicts is drastically reduced. Furthermore, we remove transactions, that have no chance to commit anymore, as early as possible from the pipeline. This *early abort of transactions* further improves the situation, as transactions, which have no chance to commit, are out of consideration for reordering.

In total, we carry out the following steps to further "databasify" Fabric:

1. To have a basis for the discussion, we first inspect the transaction flow of Hyperledger Fabric version 1.2 from a conceptual perspective. (Section 3.2).

2. We carefully inspect the related work in the area of concurrency control and discuss, which techniques are related to Fabric and which are out of consideration. (Section 4.2). This leads us directly to the techniques we are integrating.

3. Based on the analysis of the transaction flow in Fabric, we discuss its weaknesses in detail and describe, how database technology can be utilized to counter them. (Section 3.4).

4. We transition database technology to the transaction pipeline of Fabric. Precisely, we first improve on *the ordering of transactions*. By default, the system orders transactions arbitrarily after simulation, leading to unnecessary serialization conflicts. To counter this problem, we introduce an advanced *transaction reordering mechanism*, which aims at reducing the number of serialization conflicts between transactions within a block. This mechanism significantly increases the number of valid transactions, that make it through the system and therefore the overall throughput (Section 3.5.1).

5. Next, we advance the *abort of transactions*. By default, Fabric checks whether a transaction is valid right before the commit. This late abort unnecessarily penalizes the system by processing transactions, that have no chance to commit. To tackle this issue, we introduce the concept of *early abort* to various stages of the pipeline. We identify invalid transactions as early as possible and abort them, assuring that the pipeline is not throttled by transactions that have no chance to commit eventually. A requirement for this concept is a *fine-grained concurrency control mechanism*, by which we extend Fabric as well (Section 3.5.2). These modifications significantly extend the vanilla Fabric, turning it into what we call Fabric++.

6. We perform an extensive experimental evaluation of the optimizations of Fabric++ under the Smallbank benchmark as well as under a custom workload. We

show that we are able to significantly increase the number of successful transactions over the vanilla version. Additionally, we vary the blocksize, the number of channels, and clients to show that our optimizations also have a positive impact on the scaling capabilities of the system. Further, using the Caliper benchmark, we show that Fabric++ also produces a lower transaction latency than vanilla Fabric (Section 3.6).

## 3.2 Hyperledger Fabric

First, we have to understand the workflow of Fabric. Let us describe in the following section how it behaves in version 1.2.

### 3.2.1 Architecture

Fabric is a *permissioned* blockchain system, meaning all peers of the network are known at any point in time. Peers are grouped into *organizations*, which typically host them. Within an organization, all peers trust each other. Each peer runs a local instance of Fabric. This instance includes a copy of the *ledger*, containing the ordered sequence of all transactions that went through the system. This includes both valid and invalid transactions. Apart from the ledger, each peer also contains the *current state* in form of a state database, which represents the state after the *application* of all *valid* transactions in the ledger to the initial state. Apart from the peers, which play an important role both in the simulation phase and the validation phase, there is a separate instance called the *ordering service*, which is the core component of the ordering phase and assumed to be trustworthy.

### 3.2.2 High-level Workflow

At its core, Fabric follows a *simulate-order-validate-commit* workflow, as shown in Figure 3.2.

#### Simulation Phase

In the simulation phase, a client submits a transaction proposal to a subset of the peers, called the endorsement peers or *endorsers*, for simulation. This subset of endorsement

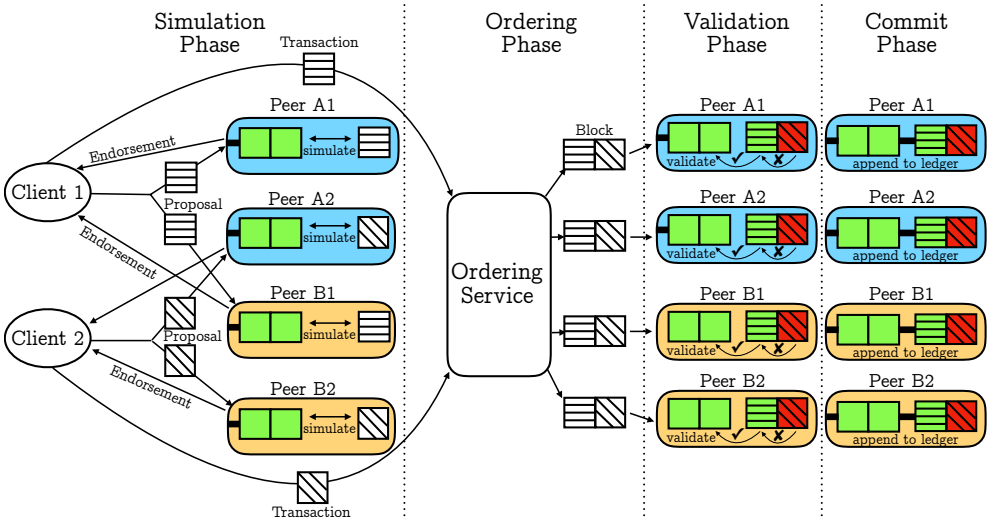



*Figure 3.2: High-level workflow of Fabric.*

peers is defined in a so called *endorsement policy*. Since organizations do not fully trust each other, it is typically specified, that at least one peer of each involved organization has to simulate the transaction proposal. The endorsers now simulate the transaction proposal against a local copy of the current state in parallel. As the name of this phase suggests, none of the effects of the simulation become durable in the current state at this point. Instead, each endorser builds up a read set and a write set during simulation to capture the effects. After simulation, each endorser returns its read and write set to the client. Along with that, the endorsers also return a cryptographic signature over the sets. If all returned read and write sets are equal, the client forms an actual transaction. It contains the previously computed read set and write set along with all signatures. The client then passes this transaction on to the ordering service.

### Ordering Phase

In the ordering phase, the trusted ordering service receives the transactions from the clients. Among all received transactions, it establishes a global order and packs them into blocks containing a certain number of transactions. By default, the transactions are essentially ordered in the way in which they arrive at the service, without inspecting the transaction semantics in any way. The ordering service then distributes each formed block to *all* peers of the network. Note that the system does not guarantee that all peers receive a block at the same time. However, it guarantees that all peers receive the same blocks in the same order.

#### Validation Phase

As soon as a block arrives at a peer, its validation phase starts. For each transaction within the block, the validation consists of two checks: First, Fabric tests whether the transaction respects the endorsement policy and whether all contained signatures fit to the read and write set. If this is not the case, it means that either an endorser or the client tampered with the transaction in some way. In this case, the systems marks the transaction as invalid. If a transaction passes the first test, Fabric secondly checks, whether any serialization conflicts occur. As the simulation of transactions happens in parallel *before* ordering them, it is possible that the effects of the simulation stand in conflict with the established order. Therefore, Fabric marks transactions, which conflict with previous transactions, as invalid as well.

#### Commit Phase

In the commit phase, each peer appends the block, which contains both valid and invalid transactions, to its local ledger. Additionally, each peer applies all changes made by the valid transactions to its current state.

## 3.3 Related Work

Before diving into the optimizations we apply, we have to discuss related work in the field.

In this work, we transition mature database techniques to the world of blockchains. As mentioned in the introduction, we essentially apply two prominent techniques from the field of database concurrency control to Fabric: *transaction reordering* and *early transaction abort*. Of course, concurrency control is a large and active area of research and the alerted reader might wonder, why we focus on precisely these techniques. The answer lies in the fact, that a blockchain system such as Fabric, despite being a parallel transaction processing system, differs from parallel database systems in four points:

(a) A blockchain system like Fabric commits at the granularity of blocks instead of committing at the granularity of individual transactions. The commit granularity has a significant effect on the type of applicable optimizations.

(b) A blockchain system like Fabric is a distributed system, where the state is fully replicated across the network and where transactions operate on all nodes. In con-

trast to that, parallel database systems are typically either installed locally on a single node or partition their state across a network, such that transactions operate on a subset of the network. The type of distribution and replication has a significant effect on the optimization options.

(c) In Fabric, a single transaction is simulated in parallel on multiple nodes to establish trust. The state, against which the transaction is simulated potentially differs across nodes. In a parallel database system, the situation is way simpler: a transaction is executed exactly once against the only state present in the system.

(d) The performance of a blockchain system like Fabric is largely dominated by cryptographic signature computations, network communication, and trust validation. In contrast, the performance of parallel database systems is highly influenced by low-level components, such as the choice of the locking mechanism for concurrency control. Therefore, despite being closely related, blockchain optimizations happen on a different level than database optimizations.

Let us now go through the related work. Essentially, concurrency control techniques can be divided into two classes: (1) methods, that aim at improving the overall transactional throughput [98, 75, 37, 57] and (2) methods, that try to turn aborted transactions into successful transactions [94, 100].

### 3.3.1 Class 1: Transaction Throughput

The works of [98], [75], and [37] all aim at improving parallel transaction execution. In [98], the authors propose a mechanism, which collects a batch of transactions and analyzes the access dependencies between these transactions. The resulting dependency graph is then partitioned into non-intersecting subgraphs. As the transactions in different subgraphs do not conflict with each other, they can be safely executed in parallel. [75] and [37] go a step further by not only analyzing dependencies between entire transactions, but actually between transaction-fragments. Precisely, they first split the transactions into possible fragments and then analyze the dependencies between these fragments, while respecting existing dependencies between entire transactions. By this, they are able to achieve a higher degree of parallelism. In general, a design as proposed by these three papers allows to equip systems, that would otherwise follow a purely sequential execution, with a partial parallel execution. In the context of blockchains, systems following an order-execute model could benefit from such a technique. Obviously, Fabric is not the right candidate for this method, as it already parallelizes the simulation phase by default.

[57] aims at improving concurrency control from a low-level perspective. It proposes interesting optimizations to MVCC components such as timestamp allocation, version storage, validation, index management, and recovery. In theory, these techniques could be applied to the underlying storage system of Fabric. However, improving low-level components of Fabric will not improve the overall performance, which is largely dominated by top-level components handling cryptography, networking, and trust validation.

Unfortunately, the techniques of class (1) are not suited to improve the transactional throughput of a blockchain system such as Fabric. We saw the reason for this in Figure 3.1 in the introduction. For blank transactions, the concurrency control mechanism essentially has no work to do. For meaningful transactions, simulation and validation must be synchronized. Still, the throughput equals. This means, that a technique that directly affects transactional processing, such as concurrency control, can *not* lead to an improvement in throughput. Instead, the system is dominated by factors, that are *not* directly related to transactional processing, such as cryptographical computations and networking. As a consequence, optimizations of class (1) are out of consideration.

### 3.3.2 Class 2: Transaction Abort & Success

[94] relates to our work, as it shares the same motivation: to reduce the amount of transactions, that are unnecessarily aborted due to serialization conflicts. In the context of a local parallel database system implementing multi-version concurrency control (MVCC) [82, 96, 99, 56], the authors propose to protect each frequently accessed entry additionally with a shared lock. The use of such a lock prevents unnecessary aborts due to read-write conflicts between transactions accessing these hot entries. Effectively, this need to acquire a lock assures for potentially conflicting transactions, that they commit in a non-conflicting order. Unfortunately, this strategy is hardly applicable to the distributed transaction processing model of Fabric. Since Fabric simulates and commits transactions in parallel on multiple nodes, this technique would require a trusted fine-grained distributed locking service to synchronize accesses, causing excessive network communication and coordination effort.

In [100], the authors also aim at reducing the number of aborted transactions by influencing the commit order. However, they follow a completely different strategy than [94]: When a transaction $T$ wants to commit and detects a read-write conflict with an already committed transaction, then it is not directly aborted. Instead, it is checked whether the commit time of $T$ can be simply changed retrospectively to its begin time. If this change does not trigger a read-write, write-write, or a write-read conflict with another concurrent transaction, then $T$ is allowed to commit at its begin time. While

this technique would be applicable in the ordering service of Fabric, its effects would be highly limited. This is caused by the simplicity of the method which allows the commit time of a transaction to be changed only to its begin time, not to other possible points in time within the commit sequence. This wastes a lot of optimization potential. In contrast, our transaction reordering mechanism considers commit reordering for all transactions within a block and aims at finding the best global order.

The benefit of transaction reordering has also recently been studied in [28]. In the context of OLTP systems, the authors identify that the number of successful transaction can be improved by up to a factor of 2.7x via reordering of transaction batches.

Thus, class (2) methods, which aim at increasing the number of successful transactions and clearing the ones that must be aborted, are the key for improving a blockchain system such as Fabric. Our optimizations of transaction reordering and early transaction abort fall into this class. In the following section, we will elaborate the importance of these techniques in detail.

## 3.4 Blurred Lines: Fabric vs Distributed Database Systems

With an understanding of the workflow of Fabric, we are able to discuss its architecture in relation to distributed database systems. In particular, we are interested in aspects of Fabric, that are (a) conceptually shared with distributed database systems, but (b) have potential for the application of database technology.

### 3.4.1 The Importance of Transaction Order

The first component we look at is the ordering mechanism. Such a component is also present in any distributed database system with transaction semantics and therefore a great candidate for transitioning database technology to Fabric.

As described in Section 3.2, Fabric relies on a single trustworthy ordering service for ordering transactions. Since Fabric simulates the smart contracts bound to proposals *before* performing the ordering, the order actually has an influence on the number of serialization conflicts between transactions. Again, this is a property shared with any parallel database system, that separates transaction execution from transaction commit.

In ordering transactions, various different strategies are possible: The simplest op-

tion is to arbitrarily order them, for instance in the order in which they arrive. While this arrival order is fast to establish, it can lead to serialization conflicts, that are *potentially unnecessary*. These conflicts increase the number of invalid transactions, which must be resubmitted by the client. Unfortunately, the vanilla Fabric follows exactly this naive strategy. This is caused by the design decision that the ordering service is not supposed to inspect the transaction semantics, such as the read and write set, in any way. Instead, it simply leaves the transactions in the order in which they arrive. This strategy can be problematic, as the example in Table 3.1 shows. In this example, four transactions are scheduled in the order in which they arrive, namely $T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_4$, where $T_1$ updates the key $k_1$ from version $v_1$ to $v_2$. Since the transactions $T_2$, $T_3$, and $T_4$ each read $k_1$ in version $v_1$ during their simulations, they have no chance to commit, as they operated on an outdated version of the value of $k_1$. They will be identified as invalid in the validation phase and the corresponding transaction proposals must be resubmitted by the client, resulting in a new round of simulation, ordering, and validation.

| Transaction | Read Set | Write Set | Is Valid? |
|---|---|---|---|
| 1. $T_1$ | — | $(k_1, v_1 \rightarrow v_2)$ | ✓ |
| 2. $T_2$ | $(k_1, v_1), (k_2, v_1)$ | $(k_2, v_1 \rightarrow v_2)$ | × |
| 3. $T_3$ | $(k_1, v_1), (k_3, v_1)$ | $(k_3, v_1 \rightarrow v_2)$ | × |
| 4. $T_4$ | $(k_1, v_1), (k_3, v_1)$ | $(k_4, v_1 \rightarrow v_2)$ | × |

*Table 3.1: For the order $T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_4$, only one out of four transactions is valid: $T_2$, $T_3$, and $T_4$ read the outdated version $v_1$ of key $k_1$, that has been updated by $T_1$ to $v_2$ before.*

Interestingly, for the four transactions from the previous example, there exists an order that is conflict free. In the schedule $T_4 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_1$, as shown in Table 3.2, all four transactions are valid, as their read and write sets do not conflict with each other in this order.

| Transaction | Read Set | Write Set | Is Valid? |
|---|---|---|---|
| 1. $T_4$ | $(k_1, v_1), (k_3, v_1)$ | $(k_4, v_1 \rightarrow v_2)$ | ✓ |
| 2. $T_2$ | $(k_1, v_1), (k_2, v_1)$ | $(k_2, v_1 \rightarrow v_2)$ | ✓ |
| 3. $T_3$ | $(k_1, v_1), (k_3, v_1)$ | $(k_3, v_1 \rightarrow v_2)$ | ✓ |
| 4. $T_1$ | — | $(k_1, v_1 \rightarrow v_2)$ | ✓ |

*Table 3.2: The order $T_4 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_1$ results in all four transactions being valid.*

This example shows that the vanilla orderer of Fabric misses a chance of removing *unnecessary* serialization conflicts. While this problem is new to the blockchain do-

main, as blockchains typically offer only a serial execution of transactions, within the database community, this problem is actually well known. There exist reordering mechanisms which aim at minimizing the number of serialization conflicts via a reordering of transactions [104, 58, 102, 28]. However, in a database system, it is typically avoided to buffer a large number of incoming transactions before processing as low latency is mandatory. Thus, reordering is not always an option in such a setup. Fortunately, as blockchain systems buffer the incoming transactions anyways to group them into blocks, this gives us the opportunity to apply sophisticated transaction reordering mechanisms without introducing significant overhead.

We will add such a transaction reordering mechanism to Fabric in Section 3.5.1, which significantly enhances the number of valid transactions, that make it through the system.

### 3.4.2 On the Lifetime of Transactions

The second aspect we look at from a database perspective tackles the lifetime of transactions within the pipeline. In Fabric, every transaction that goes through the system is either classified as valid or as invalid with respect to the validation criteria. In the vanilla version, this classification happens in the validation phase right before the commit phase. A severe downside of this form of *late abort* is that a transaction, that violated the validation criteria already in an earlier phase, is still processed and distributed across all peers. This penalizes the whole system with unnecessary work, throttling the performance of valid transactions. Besides, this concept also delays the abort notification to the client.

We have to distinguish in which phase a violation happens. First, a violation can occur already in the simulation phase, in form of so called *cross-block conflicts*, meaning a transaction from a later block, which is currently in the simulation phase, conflicts with a valid transaction from an earlier block. Second, a violation can occur as well as in the ordering phase, in form of *within-block conflicts* between conflicting transactions in a single block.

Let us look at these two scenarios in isolation in Section 3.4.2 and Section 3.4.2, respectively.

#### Violation in the simulation phase (cross-block conflicts)

To understand the problem in the simulation phase, let us look at the following situation and how the vanilla version of Fabric handles it. Let us assume there are four trans-

actions $T_1$, $T_2$, $T_3$, and $T_4$ that are currently in the ordering phase and that end up in a block of size four, which is shipped to all peers for validation. Before the validation of that block starts within a peer $P$, the smart contract of a transaction proposal $T_5$ starts its simulation in $P$. To do so, it acquires a read lock[1] on the *entire* current state. While the simulation is running, the block has to wait for the validation, as it has to acquire an *exclusive* write lock on the current state. The problem in this situation is: if $T_1$, $T_2$, $T_3$, or $T_4$ write the value of a key, that is read by $T_5$, then $T_5$ simulates on stale data. Therefore, in the moment of the read, the transaction becomes virtually invalid. Still, in the vanilla version of Fabric, this stale read is not detected before the validation phase of $T_5$. Thus, $T_5$ would continue its simulation and go through the ordering phase, just to be invalidated in the very end.

#### Violation in the ordering phase (within-block conflicts)

Apart from conflicts across blocks, there can be conflicts between transactions within a block. These conflicts appear after putting the transactions into a particular order in the ordering phase. For instance, the example from Table 3.1 in Section 3.4.1 showed a schedule, where the three transactions $T_2$, $T_3$, and $T_4$ individually conflict with the previously scheduled transaction $T_1$ of the same block. Unfortunately, these conflicts are not detected within the orderer of the vanilla version of Fabric. The block containing $T_2$, $T_3$, and $T_4$ would be distributed across all peers of the network for validation, although $3/4$ of transactions within the block are virtually invalid. As before, this originates from the design decision that the ordering service does not inspect transaction semantics.

The mentioned situations show that Fabric misses several chances to abort transactions right at the time of violation. In contrast to that, database systems are typically very eager in aborting transactions [47], as it decreases network traffic and saves computing resources. This concept of "cleaning" the pipeline as early as possible is called *early abort* in the context of databases, which apply this concept in various flavors. For instance, besides of the early abort of transactions, that violate certain criteria, database systems eliminate records from the query result set as early as possible by pushing down selection and projection operations in the query plan.

To overcome the mentioned problems, we will apply the concept of early abort at several stages of the transaction processing pipeline of Fabric. By this, we assure to utilize the available resources with meaningful work to the extend. We will detail this in Section 3.5.2.

[1] The read lock can be shared by multiple simulation phases, as they do not modify the current state.

## 3.5 Fabric++

We have outlined the problems of Fabric and how they relate to key problems known in the context of database systems. Let us now see precisely how we counter them. First, in Section 3.5.1, we introduce a transaction reordering mechanism, that aims at minimizing the number of unnecessary within-block conflicts. Second, in Section 3.5.2, we introduce early transaction abort to several stages of the Fabric pipeline. This also involves the introduction of a fine-grained concurrency control mechanism.

### 3.5.1 Transaction Reordering

When reordering a set of transactions $S$, multiple challenges must be faced. First, we have to identify which transactions of $S$ actually conflict with each other with respect to the actions they perform. Again, these conflicts can occur, as the transactions of $S$ simulated in complete isolation from each other. As the commits of $S$ happen in a later phase, they had no chance to see each other's potentially conflicting modifications. Precisely, we have a conflict between two transactions $T_i$ and $T_j$ (denoted as $T_i \nrightarrow T_j$), if $T_i$ writes to a key that is read by $T_j$. In this case, $T_i$ must be ordered *after* $T_j$ (denoted as $T_j \Rightarrow T_i$) to make the schedule serializable, as otherwise, the read of $T_j$ would be outdated. Unfortunately, the problem is typically more complex as *cycles of conflicts* can occur, such that simple reordering cannot resolve the problem. For example, if we have the cycle of conflicts $T_i \nrightarrow T_j \nrightarrow T_k \nrightarrow T_i$, there is no order of these three transactions that is serializable. Therefore, before reordering transactions, our mechanism must actually first remove certain transactions of $S$ to form a cycle-free subset $S' \subseteq S$, from which a serializable schedule can be generated.

From a high-level perspective, the following five steps must be carried out: (1) First, we build the conflict graph of all transactions of $S$. (2) Then, we identify all cycles within this conflict graph. (3) Based on that, we identify for each transaction, in which cycles it occurs. (4) Next, we incrementally abort transactions occurring in cycles, starting from the ones that occur in most cycles, until all cycles are resolved. (5) Finally, we build a serializable schedule from the remaining transactions. The pseudo-code of Algorithm 1 implements these five steps.

| Transactions | Read Set | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $K_0$ | $K_1$ | $K_2$ | $K_3$ | $K_4$ | $K_5$ | $K_6$ | $K_7$ | $K_8$ | $K_9$ |
| $T_0$ | **1** | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_1$ | 0 | 0 | 0 | **1** | **1** | **1** | 0 | 0 | 0 | 0 |
| $T_2$ | 0 | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 | 0 |
| $T_3$ | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | **1** | 0 |
| $T_4$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** |
| $T_5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Transactions | Write Set | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $K_0$ | $K_1$ | $K_2$ | $K_3$ | $K_4$ | $K_5$ | $K_6$ | $K_7$ | $K_8$ | $K_9$ |
| $T_0$ | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_1$ | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $T_2$ | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | **1** |
| $T_3$ | 0 | **1** | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 |
| $T_4$ | 0 | 0 | 0 | 0 | 0 | **1** | **1** | 0 | **1** | 0 |
| $T_5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** | 0 | 0 |

*Table 3.3: Ten unique keys that are accessed by six transactions, separated in read set and write set.*

### Example

To understand the principle and to discuss some of the implementation details, let us go through a concrete example. Let us assume we have a set $S$ of six transactions $T_0$ to $T_5$ to consider for reordering. These six transactions have read and write sets as shown in Table 3.3. In total, they access ten unique keys $K_0$ to $K_9$.

**Step (1)**: Based on this information, we now have to generate the conflict graph of the transactions as done by the function `buildConflictGraph()` in line 2 of Algorithm 1. To do so in an efficient way, we interpret the rows of Table 3.3 as bit-vectors of length 10. Let us refer to them as $vec_r(T_i)$ for the reading accesses and $vec_w(T_i)$ for the writing accesses of a transaction $T_i$. For each transaction $T_i$, we now perform a bitwise `&`-operation between $vec_r(T_i)$ and $vec_w(T_j)$ for all $j \neq i$. If the result of an `&`-operation is not 0, we have identified a read-write conflict and create an edge in the conflict graph between the corresponding transactions. As a result, we obtain the conflict graph $C(S)$ of our six transactions as shown in Figure 3.3.

Note that this algorithm has quadratic complexity on the number of transactions. Still, we apply it as the number of transactions to consider is very small in practice due to the

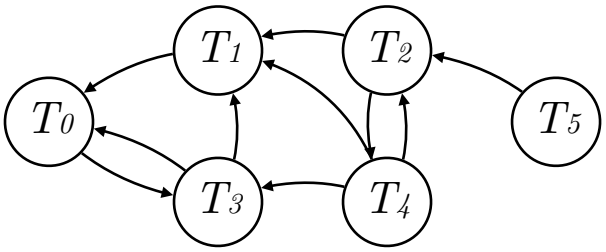



*Figure 3.3: Conflict graph $C(S)$ of the transactions in $S$.*

limitation by the block size and therefore, the overhead is negligible.

**Step (2)**: To identify the cycles, we apply Tarjan's algorithm [84] in the function `divideIntoSubgraphs()` in line 4 to identify all strongly connected subgraphs. In general, this can be done in linear time in $\mathcal{O}(N + E)$ over the number of nodes $N$ and number of edges $E$ and results in the three subgraphs as shown in Figure 3.4.

Using Johnson's algorithm [49], we then identify all cycles within the strongly connected subgraphs. Again, this step can be done in linear time in $\mathcal{O}((N+E)\cdot(C+1))$, where $C$ is the number of cycles. Therefore, if there are no cycles in the subgraphs, the overhead of this step is very small.

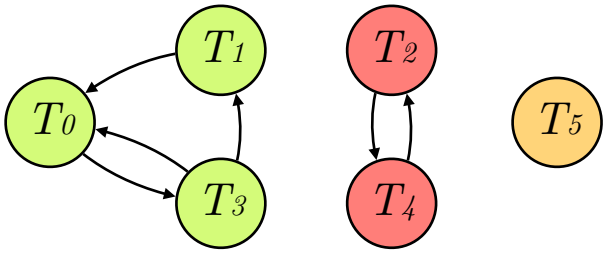



*Figure 3.4: The three strongly connected subgraphs of the conflict graph of Figure 3.3.*

We identify that the first subgraph (colored in green) contains the two cycles $c_1 = T_0 \nrightarrow T_3 \nrightarrow T_0$ and $c_2 = T_0 \nrightarrow T_3 \nrightarrow T_1 \nrightarrow T_0$. The second subgraph (colored in red) contains the cycle $c_3 = T_2 \nrightarrow T_4 \nrightarrow T_2$. The third subgraph (colored in yellow) contains only one node and is thus cycle-free.

**Step (3)**: From this information, we can build a table denoting for every transaction in which cycle it appears as shown in the lines 6 to 9 of Algorithm 1. Table 3.4 visualizes the result for our example. If a transaction $T_i$ is part of a cycle $c_j$, the corresponding cell is set to 1, otherwise 0. The last row of the table sums up for every transaction in how many cycles it is contained in total.

| Cycle | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|---|---|
| $c_1$ | 1 | 0 | 0 | 1 | 0 | 0 |
| $c_2$ | 1 | 1 | 0 | 1 | 0 | 0 |
| $c_3$ | 0 | 0 | 1 | 0 | 1 | 0 |
| $\sum$ | 2 | 1 | 1 | 2 | 1 | 0 |

*Table 3.4: If a transaction $T_i$ is a part of a cycle $c_j$, the corresponding cell is set to 1, otherwise 0. The last row contains for every transaction the total number of cycles, in which it appears.*

**Step (4)**: We now iteratively remove transactions, that participate in cycles, starting from the ones that appear in most cycles. The lines 17 to 24 of Algorithm 1 show the corresponding pseudo-code. As we can see, $T_0$ and $T_3$ both appear in two cycles, so we take care of them first. If we can choose between two transactions, such as $T_0$ and $T_3$, we pick the one with the smaller subscript. This assures that our algorithm is deterministic. We remove $T_0$, which clears all cycles in which $T_0$ appears, namely $c_1$ and $c_2$. The transactions $T_2$ and $T_4$ remain with a participation in cycle $c_3$ each. We remove $T_2$ which clears $c_3$ and thereby the last cycle.

From this we now know that from the set $S' = \{T_1, T_3, T_4, T_5\}$ we can generate a serializable schedule, leading to the cycle-free conflict graph $C(S')$ (line 28) as shown in Figure 3.5.

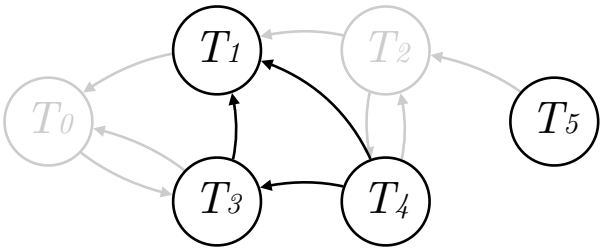



*Figure 3.5: The cycle-free conflict graph $C(S')$, containing only the transactions $T_1$, $T_3$, $T_4$, and $T_5$.*

**Step (5)**: Generating the final schedule is essentially a repetitive execution of two parts until all nodes are scheduled: (a) the locating of the source node in the current subgraph (lines 33 to 40) and (b) the scheduling of all nodes that are reachable from that source (lines 41 to 47).

We start part (a) at the node of $C(S')$ representing the transaction with the smallest

subscript, namely $T_1$. From this starting node, we have to find a source node, as sources have to be scheduled last. $T_1$ has two parents, namely $T_3$ and $T_4$, so it not a source. We follow the edge to $T_3$, which has not been visited yet but is also not a source, as it has $T_4$ as a parent as well. We follow the edge to $T_4$, which has not been visited yet and which is a source. Therefore, we can schedule $T_4$ safely at the last position in our schedule, to which we refer to as *position* $4$. Now, part (b) starts as all nodes that are reachable from $T_4$ must be scheduled before it. $T_4$ has two children, namely $T_1$ and $T_3$. We follow the edge to $T_1$, which has not been scheduled yet. However, as $T_1$ has an incoming edge from $T_3$, we also cannot directly schedule it. First, we visit $T_3$ and identify that it has a parent in form of $T_4$, the source at which we started. With this information, we know that $T_3$ must be scheduled at position $3$ and $T_1$ must be scheduled at position $2$. This ends part (b), as all reachable nodes have been scheduled. Next, we restart at the only remaining node $T_5$. As $T_5$ is not only a source but also a sink, we can schedule it instantly at position $1$. This results in the final schedule $T_5 \Rightarrow T_1 \Rightarrow T_3 \Rightarrow T_4$, which is returned to the orderer.

Please note that our reordering mechanism is not guaranteed to abort a minimal number of transactions, as this would be a NP-hard problem. However, it offers a very lightweight way to generate a serializable schedule with a small number of aborts.

### Batch Cutting

In the context of transaction reordering, we have to discuss and extend a mechanism within the ordering service, that we omitted for simplicity in the description of Fabric in Section 3.2, namely *batch cutting*. When the ordering service receives the transactions in form of a constant stream, it decides based on multiple criteria when to "cut" a batch of transactions to finalize it and to form the block. In the vanilla version, a batch is cut as soon as one of the following three conditions hold: (a) The batch contains a certain number of transactions. (b) The batch has reached a certain size in terms of bytes. (c) A certain amount of time has passed since the first transaction of this batch was received.

In Fabric++, we extend these criteria by one additional condition. We also cut the batch, if (d) the transactions within the batch access a certain number of unique keys. This condition ensures that the runtime of our reordering mechanism, in particular the time of step (1), remains bounded.

### Micro-Benchmark

To analyze the effectiveness of our reordering mechanism, we first evaluate it in a stand-alone micro-benchmark in isolation of Fabric. For a given sequence of input transac-

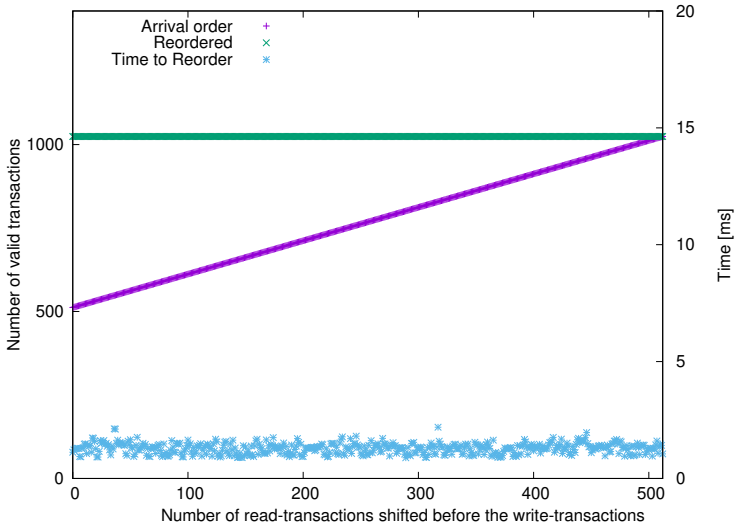



*Figure 3.6: Workload 1: Varying the number of conflicts.*

tions we compute the number of valid transactions for this particular sequence (called "arrival order" in the following plots) as well as for the sequence that is generated by our reordering mechanism (called "reordered" in the following plots). Additionally, we measure the time to compute the reordered schedule. In Figure 3.6, we test a workload pattern with varying number of conflicts. For the interested reader, we provide a second micro-benchmark in the Appendix 3.5.1 on the effect of varying the length of the cycles (Figure 3.7) and see how well our reordering mechanism performs in comparison to the naive arrival order.

### Micro-Benchmark 1: Interleave reads and writes to vary the number of conflicts

The first input sequence we test consists of two equal sized sub sequences, where one subsequence contains only transactions that perform writes (colored in red) and the other sequence only transactions that read (colored in blue). Each transaction performs only one operation (either read or write). Neither two writes nor two reads happen to the same key. For the example of $n = 6$ transactions, we start with the following sequence $S_1$:

$$S_1 = T[w(k_1)], T[w(k_2)], T[w(k_3)], T[r(k_1)], T[r(k_2)], T[r(k_3)]$$

To generate $S_i$, we move the last transaction of $S_{i-1}$ to the front, leading to the following sequences $S_2$, $S_3$, and $S_4$.

$$S_2 = T[r(k_3)], T[w(k_1)], T[w(k_2)], T[w(k_3)], T[r(k_1)], T[r(k_2)]$$

$$S_3 = T[r(k_2)], T[r(k_3)], T[w(k_1)], T[w(k_2)], T[w(k_3)], T[r(k_1)]$$
$$S_4 = T[r(k_1)], T[r(k_2)], T[r(k_3)], T[w(k_1)], T[w(k_2)], T[w(k_3)]$$

The more writing transactions happen before the corresponding reading transactions, the more conflicts happen. We want to find out whether our reordering mechanism can solve this problem.

Figure 3.6 shows the results for $n = 1024$ transactions. As we can see, our reordering mechanism is able to reorder the transactions for every input sequence in a way such that all transactions are valid. In contrast to that, the arrival order suffers under a lot of invalid reading transactions, if writing transactions happen before. We can also see that our reordering mechanism is computationally cheap: it takes only around 1 to 2 ms to rearrange the transactions on a Macbook Pro with Intel Core i7 running at 3.1 GHz.

#### Micro-Benchmark 2: Vary the length of cycles

In the following experiment, we want to analyze the impact of cycles on the arrival order and on our reordering mechanism. To do so, we again form a sequence of $n$ transactions, that contains $n/t$ cycles of size $t$ transactions of the form

$$T[r(k_0), w(k_0)], T[r(k_0), w(k_1)], T[r(k_1), w(k_2)], T[r(k_2), w(k_0)]$$

Again, we want to identify how many transactions are valid under the arrival order and when using our reordering mechanism. Figure 3.7 shows the results for 1024 transactions. For the arrival order, only half of transactions are valid, no matter of the cycle length. This is because aborting every second transaction breaks the cycles. In comparison to that, our reordering mechanism is able to achieve a high number of valid transactions, if the cycles are sufficiently long respectively, there are not too many cycles to cancel. Of course, our algorithm becomes more expensive with the length of the cycles to break. However, since extremely long cycles are very unlikely to occur in reality, the runtime of our mechanism will in general remain low in the ordering phase, as we will see in the full fledged evaluation later on.

### 3.5.2 Early Transaction Abort using Concurrency Control

The reordering mechanism previously described not only tries to minimize the number of unnecessary aborts, it also enables a form of *early abort*. Transactions, that are removed from $S$ because of their participation in conflict cycles can be aborted already in the ordering phase instead of later on the validation phase. This assures that less transactions are distributed across the network.

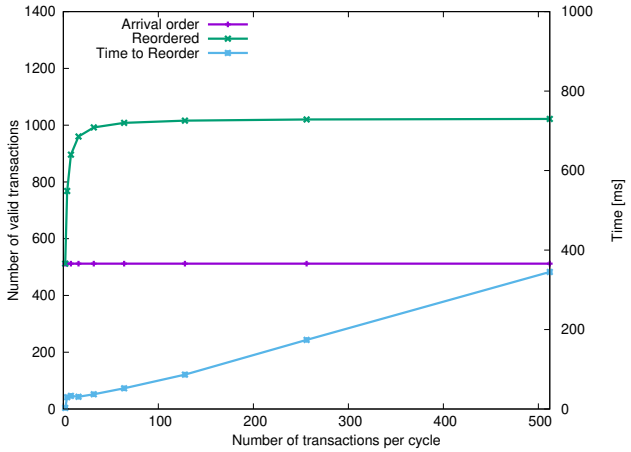



*Figure 3.7: Workload 2: Varying the length of the cycles.*

In the following, we want to push this concept of aborting transactions as early as possible in the pipeline to the limits. Additionally to early aborting transactions that occur in conflict cycles, we can integrate two more applications of early abort, as we will describe in Section 3.5.2 and Section 3.5.2. The first one is happening already in the simulation phase. Let us see in the following how this works.

### Early Abort in the Simulation Phase

To realize early abort in the simulation phase, we first have to extend Fabric by a more fine-grained concurrency control mechanism, that allows for the parallel execution of simulation and validation phase within a peer. With such a mechanism at hand, we have the chance of identifying stale reads *during* the simulation already.

To understand the concept, let us consider the example from Section 3.4.2 again. With a fine-grained concurrency control mechanism, the block containing $T_1$, $T_2$, $T_3$, and $T_4$ would not have to pend for validation while the smart contract bound to the proposal $T_5$ is simulating. Instead, the four transactions would apply their updates in an atomic fashion *while* $T_5$ is simulating. As a consequence of this design, for every read $T_5$ performs, we can check whether the read value is still up-to-date. As soon as we detect a stale read, we can abort the simulation of the transaction proposal. Additionally, we directly notify the corresponding client about the abort, such that it can resubmit the proposal without delay.

Let us discuss in the following, how exactly our fine-grained concurrency control mechanism works and how we realize it in Fabric++. In the context of modern database systems, advanced concurrency control mechanisms are well established [50, 93, 94,

99, 56, 82]. Instead of locking the entire store, these techniques typically perform a fine-grained locking on the record level or even at the level of individual cells/values. As there is conceptually no difference between the store of a database system and the store used within the Fabric peers, similar techniques can be applied here.

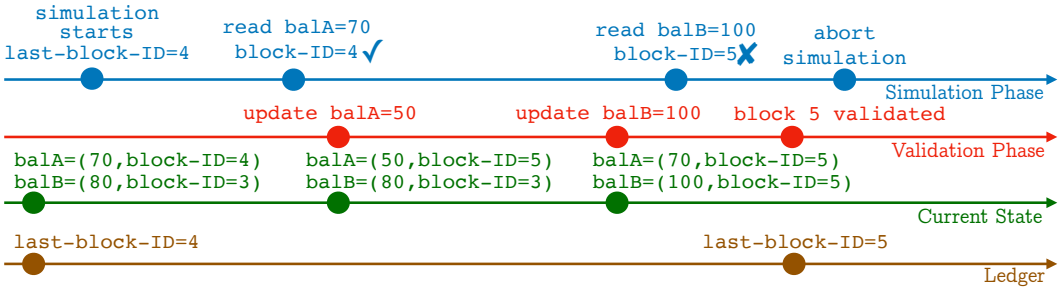



*Figure 3.8: Parallelization with early abort using our fine-grained concurrency control.*

As discussed in Section 3.2, Fabric implements its current state in form of a key-value store, which maps each individual key to a pair of value and version-number. The version-number is actually composed of the ID of the transaction, that performed the update, as well as the ID of the block that contains the transaction. In the original version of Fabric, the sole purpose of the version-numbers is to identify stale reads. In the validation phase, for every transaction we check whether the version-number of the read value still matches the one in the current state.

We can go one step further and exploit the available version-numbers to implement a lock-free concurrency control mechanism protecting the current state. To do so, in Fabric++, we first remove the read-write lock, that was unnecessarily sequentializing simulation and validation phase. The version-number, that is maintained with each value, is sufficient to ensure the same transaction isolation semantics as the vanilla version. As no lock is acquired anymore, we need a mechanism to ensure that updates performed by the validation phase are not seen by simulation phases running in parallel. To achieve this behavior, during simulation, we have to inspect the version-number of every read value and test whether it is still up-to-date.

Figure 3.8 visualizes this concept using a concrete example. At the start of the simulation phase, we first identify the `block-ID` of the last block that made it into the ledger. Let us refer to this `block-ID` as the `last-block-ID`. In our example, `last-block-ID = 4`. During the simulation of a smart contract bound to a transaction proposal $T_{exec}$, no read must encounter a version-number containing a `block-ID` higher than the `last-block-ID`. If it does see a higher `block-ID` it means that during the simulation phase, a validated transaction $T_{valid}$ in the validation phase modified a value in the read set of $T_{exec}$ and thus, the read set is outdated.

In our example, the read of $balA = 70$ in the simulation phase happens *before* the update of $balA$ to 50 in the validation phase. This is reflected by the version-number of $balA$, namely `block-ID = 4`. Therefore, this read is up-to-date and the simulation continues. In contrast to that, the read of $balB$ happens *after* the update of $balB$ to 100 in the validation phase. This is reflected by the version-number of $balB$, namely `block-ID = 5`. As `5` is higher than the `last-block-ID = 4`, we can directly classify $T_{exec}$ as invalid, as the transaction will not have a chance to pass the validation phase later on. Please note that the overall correctness of our lock-free mechanism is ensured by the atomic updates of the version-numbers.

#### Early Abort in the Ordering Phase

In addition to the early abort in the simulation phase, as explained in Section 3.5.2, we can transition a similar concept also to the ordering phase. As Fabric performs commits at the granularity of whole blocks, two transactions within the same block, that read the same key, must read the same version of that key. For example, let us consider two transactions $T_6$ and $T_7$, where $T_6$ is ordered before $T_7$ within the same block ($T_6 \Rightarrow T_7$). If $T_6$ read version $v_1$ of a key $k$ and $T_7$ read version $v_2$ of $k$ in their respective simulations, then $T_6$ is invalid. Such a version mismatch can happen, if between the simulations of $T_6$ and $T_7$, a change to the value of $k$ was committed by a valid transaction from a previous block. Therefore, as soon as we detect a version mismatch between transactions within the same block, we can early abort the earlier transaction. Again, this strategy assures that only those transactions end up in a block, that have a realistic chance of commit.

## 3.6 Experimental Evaluation

In the previous section, we have extended and modified core components of Fabric in several ways, turning it into Fabric++. It is now time to evaluate the modifications in terms of effectiveness. Primarily, we are interested in the throughput of valid/successful and invalid/failed transactions, that make it through the system. Secondarily, we are interested on the influence of certain system configurations and the workload characteristics on the system.

### 3.6.1 Setup

Before starting with the actual experiments, let us discuss the setup. Our cluster consists of six identical servers, that are located within the same rack and connected via gigabit-ethernet. Four machines serve as peers, one machine runs the ordering service, and one machine serves as the client, which fires transaction proposals. Each server consists of two quad-core Intel Xeon CPU E5-2407 (SandyBridge architecture) running at 2.2 GHz with 32KB of L1 cache, 256KB of L2 cache, and 10MB of a shared L3 cache. 24GB of DDR3 ram are attached to each of the two NUMA regions. The operating system used is a 64-bit Arch Linux with kernel version 4.17. Fabric is set up to use LevelDB as the current state database.

### 3.6.2 Benchmark Framework and Workload

In the database community, there exist numerous established benchmarking suites and workloads that can be used to test and to compare systems, such as TPC-C [15], TPC-H [12], or YCSB [16]. Unfortunately, since blockchains are still a relatively young field, there exist only very few benchmarks with standardized workloads.

#### Framework

First, we have to identify a framework that can be used to run a workload against Fabric. There are essentially three options available right now: Caliper [2], Gauge [13], and BlockBench [29]. Caliper feels like a natural candidate, as it originates from the Hyperledger project just like Fabric. While it is compatible with Fabric 1.2, it suffers from certain limitations: it supports only a single channel, it supports only one transaction type per run, it fires transactions non-uniformly with respect to time, and it is prone to failing with missed events at high firing rates [4]. As a consequence, Gauge was forked from Caliper, which addresses some of these problems. Unfortunately, it lacks compatibility with Fabric 1.2. The same incompatibility holds for BlockBench.

As none of the available frameworks is fully satisfying and since the framework is just a tool for running experiments, we decided to build our own benchmarking framework. It allows us to fire transaction proposals uniformly at a specified rate from multiple clients in multiple channels and reports the throughput of successful and aborted transactions per second. We use our framework for all main experiments in the upcoming evaluation. Still, we include an experimental run on Caliper with a compatible workload in Section 3.6.7 to ease for other groups the comparison with our work.

### Workload

In the following experiments, we use two different types of workloads.

The first one is the Smallbank [11] workload, which is perfectly suited to test a blockchain system, as it simulates a typical asset transfer scenario. Initially, it creates for a certain number of users a checking account and a savings account each and initializes them with random balances. The workload consists of six transactions, where five of them update the account balances in certain ways: `TransactSavings` and `DepositChecking` increase the savings account and the checking account by a certain amount respectively. `SendPayment` transfers money between two checking accounts. `WriteCheck` decreases a checking account, while `Amalgamate` transfers all funds from a savings account to a checking account. Additionally, there is a read-only transaction `Query`, which reads both the checking as well as the savings account of a user. During a single run, we repeatedly fire these six transactions in a random fashion, where we uniformly pick one of the five modifying transactions with a certain probability $P_w$, and the reading transaction with a probability $1 - P_w$. For each picked transaction, we determine the accounts to access by following a Zipfian distribution, which we can configure in terms of skewness by setting the s-value. Note that an s-value of $0$ corresponds to a uniform distribution.

| **Experiment Parameters** | **Values** |
|---|---:|
| Fired transaction proposals per second per client | 512 |
| Duration in which transaction proposals are fired | 90 sec |
| Number of channels | 1 |
| Number of clients per channel | 4 |

*Table 3.5: Experiment and system configuration.*

Our second workload consists solely of a single, highly configurable transaction, which performs a certain number of read and write accesses on a set of account balances. Initially, we create a certain number of accounts (N), each initialized with a random integer. Our transaction performs a certain number of reads and writes (RW) on a subset of these accounts. Among the accounts, there exist a certain number of hot accounts (HSS), that are picked for a read respectively write access with a higher probability. This probability for picking a hot account for reading (HR) respectively for writing (HW) can also be configured.

In a single run, we fire a constant stream of transactions for a certain amount of time at a certain firing rate. In the following experiments, we fix the experimental

and system configuration to the parameters as shown in Table 3.5 and Table 3.6. We identified these parameter values empirically with the goal to find a configuration, that sustains the system without overloading it.

| System Parameters | Values |
|---|---|
| Maximum time to form a block | 1 sec |
| Maximum number of keys accessed per block | 16384 |
| Maximum size per block | 2MB |
| Maximum number of transactions per block (BS) | 1024 (see Section 4.8.5) |

*Table 3.6: Experiment and system configuration.*

### 3.6.3 The Impact of the Blocksize

We start our evaluation by investigating the effect of the blocksize on Fabric and Fabric++. By default, Fabric's sample network limits the blocksize to only up to 10 transactions. In the following experiment, we vary the blocksize from $16$ transactions to $2048$ transactions in logarithmic steps and observe the impact on the number of successful transactions. As workload, we test Smallbank as defined above with $100{,}000$ users under a write heavy workload with $P_w = 95\%$, and a uniform distribution with s-value $= 0$. Figure 3.9 shows the average number of successful transactions per second over the entire run of $90$ seconds.

As we can see, increasing the blocksize also increases the throughput of successful transactions for both Fabric and Fabric++. This is due to the fact, that the usage of larger blocks causes less network communication. Obviously, Fabric's default setting of $10$ transactions per block is clearly too small and severely limits the throughput. We can also observe, that Fabric++ gains more over Fabric with an increase in the blocksize. This already gives us a first impression of the effectiveness of our reordering mechanism, which benefits from larger blocks. As we aim for a high overall throughput, for the remaining experiments, we use a blocksize of $1024$ transactions.

### 3.6.4 Transactional Throughput

Let us now test Fabric and Fabric++ under probably the most important criterium for a transaction processing system, namely the throughput of successful transactions.

### Throughput under Smallbank

First, as workload, we use Smallbank as defined in Section 3.6.2 and configure it as shown in Table 3.7. Again, we initialize 100,000 users, each equipped with a checking account and a savings account. However, this time we vary the probability of picking a modifying transaction over the reading transaction in three steps: We test $P_w = 95\%$ (write-heavy), $P_w = 50\%$ (balanced), and $P_w = 5\%$ (read-heavy). Further, we vary the skewness of the Zipf distribution, that is used to select the accounts: We go from an s-value $= 0.0$ (uniform) to an s-value $= 2.0$ (highly skewed) in steps of $0.2$. Table 3.7 summarizes the configuration again.

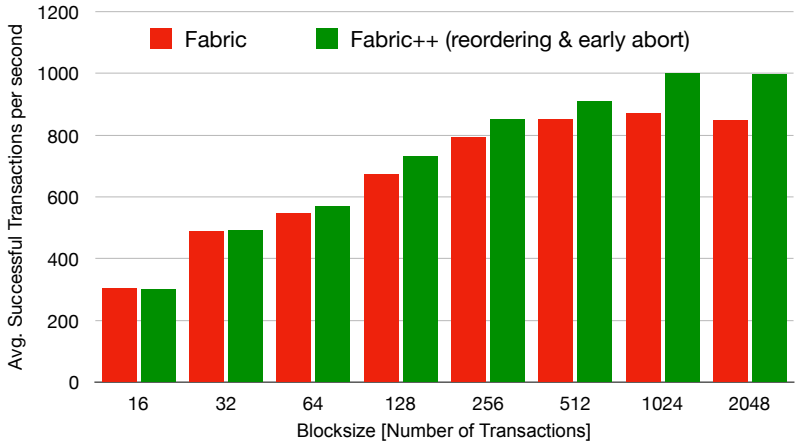
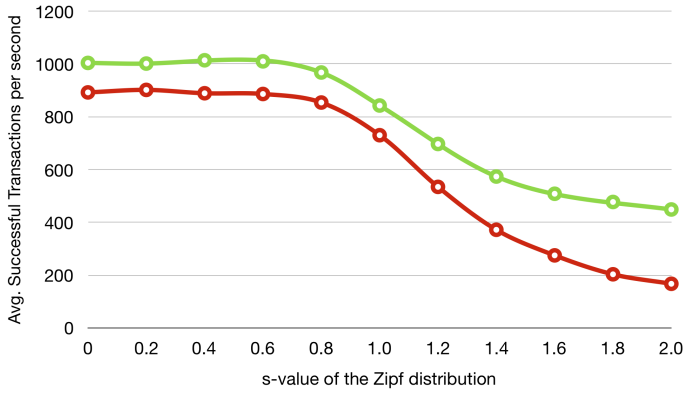



*Figure 3.9: Effect of the blocksize on the average number of successful transactions under Fabric and Fabric++.*

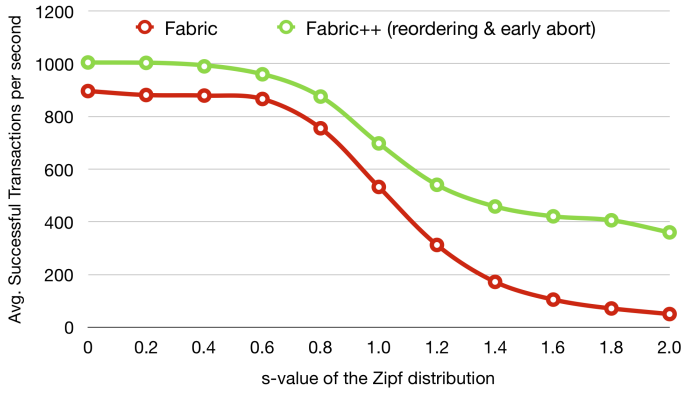
| **Workload Parameters** | **Values** |
|---|---|
| Number of users (two accounts per user) | 100,000 |
| Probability for picking a modifying transactions ($P_w$) | 95%, 50%, 5% |
| $s$-value of Zipf distribution | 0.0 – 2.0 in steps of 0.2 |

*Table 3.7: Smallbank workload configuration.*

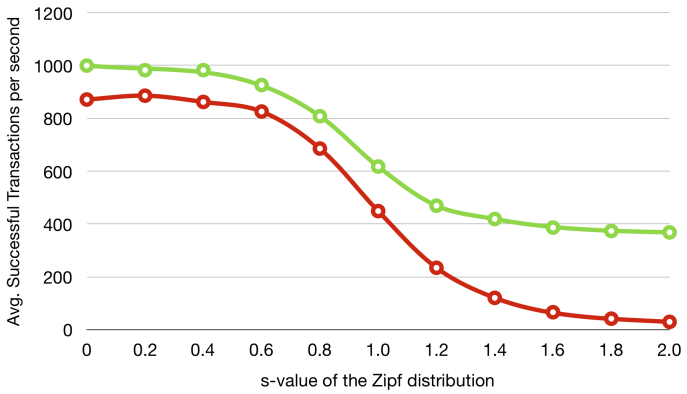
In Figure 3.10, we can see the results. We show one plot each for the read-heavy, balanced, and write-heavy workload. Within each plot, on the x-axis, we vary the s-value as described and report on the y-axis the average number of successful transactions per second over the run. Overall, we can see that Fabric++ shows a higher throughput of successful transactions for all tested runs. We can observe, that for little




(a) $P_w = 5\%$ (read-heavy)




(b) $P_w = 50\%$ (balanced)




(c) $P_w = 95\%$ (write-heavy)

*Figure 3.10: Average number of successful transactions per second of Fabric and Fabric++ under the Smallbank workload, as defined in Table 3.7.*

to no skew (up to an s-value of 0.6), the throughput of both Fabric and Fabric++ is relatively high, as the number of potential conflicts between transactions is small by default. Still, we see that with around 1000 successful transactions per second, the throughput of Fabric++ is a bit higher than for Fabric with around 900 transactions, which is mainly caused by cleaning the pipeline from transactions, that have no chance to commit. For higher skew (s-value $\geq 1.0$), we can see that Fabric++ drastically improves over Fabric, especially for the balanced and write-heavy workloads. For an s-value of 1.0, we observe improvement factors between 1.15x and 1.37x, while for an s-value of 2.0, Fabric++ shows an improvement between 2.68x and 12.61x. High skew in the access leads to a large number of potential conflicts, which can be resolved by our optimizations. For such a high contention, our optimizations make the difference between a system, that is essentially jammed (30 successful transactions per second for Fabric under $P_w = 95\%$, s-value $= 2.0$) and a system, that fluently processes transactions (370 successful transactions per second for Fabric++ under $P_w = 95\%$, s-value $= 2.0$).

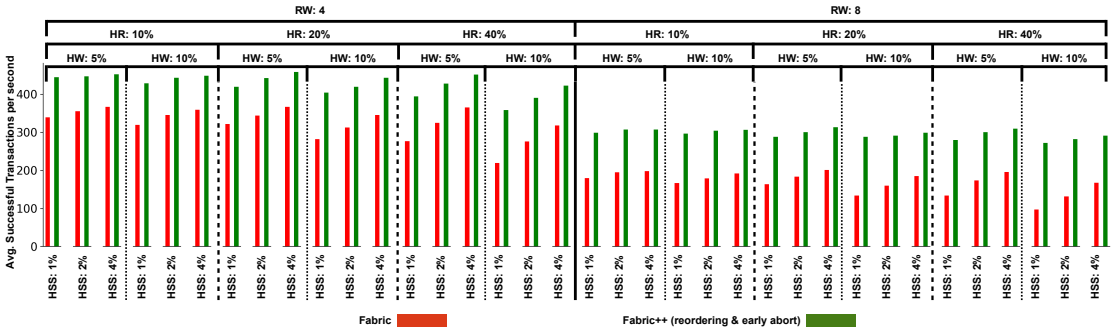



*Figure 3.11: Average number of successful transactions per second of Fabric and Fabric++ under 36 different configurations, as defined in Table 3.8. We vary the number of read & written balances per transaction (RW), the probability for picking a hot account for reading (HR) and writing (HW), and the number of hot account balances (HSS).*

### Throughput under custom workload

Let us now investigate the throughput under our custom workload. Again, we use the experimental configuration as well as the system configuration of Table 3.8. We configure our workload such that we use $N = 10{,}000$ accounts. We test both $RW = 4$ and $RW = 8$ read and write accesses per transaction. The probability of picking a hot account for reading is varied between $HR = 10\%$, $HR = 20\%$, and $HR = 30\%$. The probability of picking a hot account for a writing access is varied from $HW = 5\%$ to $HW = 10\%$. Additionally, we vary the number of hot accounts from $HSS = 1\%$

over $HSS = 2\%$ to $HSS = 4\%$ from the total number of accounts. In total, we test 36 configurations, which are summarized in Table 3.8.

| **Workload Parameters** | **Values** |
|---|---|
| Number of account balances (N) | 10,000 |
| Number of read & written balances per transaction (RW) | 4, 8 |
| Probability for picking a hot account for reading (HR) | 10%, 20%, 40% |
| Probability for picking a hot account for writing (HW) | 5%, 10% |
| Number of hot account balances (HSS) | 1%, 2%, 4% |

*Table 3.8: Custom workload configuration.*

Figure 3.11 shows summary of the results. We can see that Fabric++ significantly increases the throughput of successful transactions over Fabric for all tested configurations. The largest improvement of Fabric++ over Fabric in terms of successful transactions we observe is around factor 3x for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1%. See Figure 3.14 for a wide-spectrum evaluation of Fabric++ on all 108 tested configurations.

### Observations

We observe a significant decrease in the throughput of the successful transactions with the increase in the hotness of the transactions in both workloads. Each block $b_i$ roughly updates every hot key. This forces most of the transactions in the next block $b_{i+1}$ to abort because of read-write conflicts. In comparison to Fabric, which suffers heavily from this scenario, Fabric++ reorders the transactions within the block to remove the within-block conflicts to improve the overall throughput of successful transactions.

Fabric++ is also capable of improving the throughput of successful transactions significantly under workloads (Smallbank) which read and modify the same set of keys. Fabric++ prefers to select more transactions that access fewer keys rather than selecting fewer transactions with large number of accesses to improve the end-to-end throughput of successful transactions (as shown in Figure 3.10). For the workload that potentially has a non-overlapping read and write sets, Fabric++ is able to re-organize the transaction block to minimize the number of unnecessary aborts (as shown in Figure 3.11).

### 3.6.5 Optimization Breakdown

In Section 4.8.2, we measured the throughput of Fabric++ with both optimizations activated. Let us now see at a sample configuration, how much the individual optimizations of reordering and early abort contribute to the improvement. Figure 3.12 shows the improvement breakdown for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1% in comparison to standard Fabric. While Fabric achieves only a throughput of around 100 successful transactions per second, activating one of our two optimization techniques alone improves this to around 150 transactions per second. In comparison to that, activating both techniques at the same time results in the highest throughput of successful transactions with around 220 transactions per second. This shows nicely how both techniques work together: Transactions, that are already early aborted in the simulation phase do not end up in a block in the ordering phase. As a consequence, only transactions, that have a realistic chance of being successful, are considered in the reordering process.

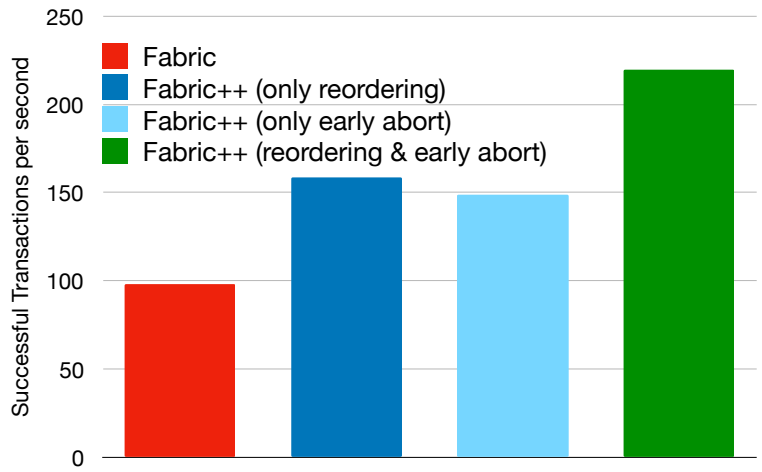
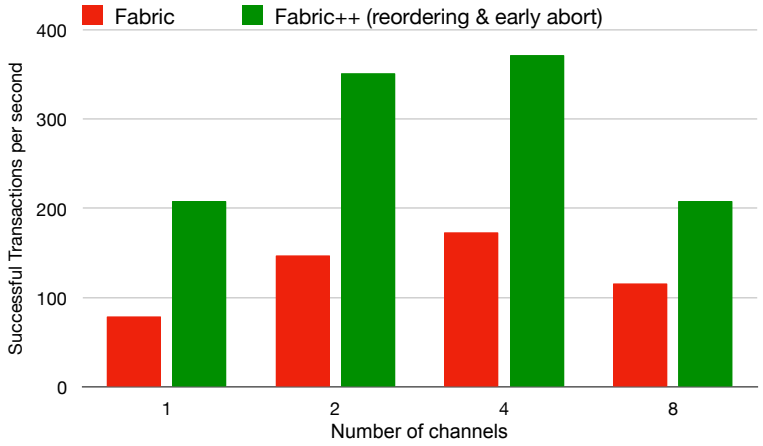



*Figure 3.12: Breakdown of the individual impact of our optimizations on the throughput of successful transactions for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1%.*

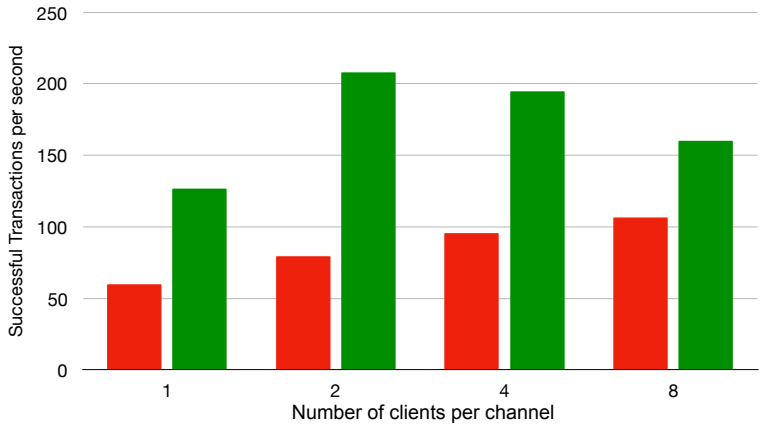
### 3.6.6 Scaling Channels and Clients

In all of our previous experiments we used four clients to fire transactions on a single channel. We now vary the number of channels, and the number of clients to see the effect on the throughput. We use the configuration BS=1024, RW=8, HR=40%, HW=10%,

HSS=1% to evaluate the average throughput of successful transactions for Fabric and Fabric++.




(a) **Varying the number of channels** from 1 to 8. Per channel, we use 2 clients to fire the transaction proposals.




(b) **Varying the number of clients per channel** from 1 to 8. All clients fire their transaction proposals in a single channel.

*Figure 3.13: The impact of the number of channels as well as the number of clients per channel on the throughput of successful transactions for the configuration BS=1024, RW=8, HR=40%, HW=10%, HSS=1%.*

First, we vary the number of channels in Figure 3.13(a) from 1 to 8. Per channel, we use 2 clients to fire transaction proposals. We can see that when going from 1 channel to 4 channels, the throughput of both Fabric and Fabric++ significantly increases. Obviously, the additional mechanisms of Fabric++ do not harm the scaling with the number of channels. Only when using 8 channels, the throughput decreases again for both Fabric and Fabric++. This is simply the case because individual channels start competing

for resources. This also increases the number of failed transactions: Scaling from 1 to 8 channels increases the number of failed transactions from 213 TPS to 837 TPS for Fabric and from 81 TPS to 704 TPS for Fabric++. Due to the competition for resources, individual simulations phase take longer and increase the chance of working on stale data.

After varying the number of channels, let us now vary the number of clients per channel in Figure 3.13(b). We test 1, 2, 4, and 8 clients, where all clients fire their transaction proposals into a single channel. Here, the picture is a slightly different to the behavior when scaling channels. The throughput of Fabric increases very gently with the number of clients, and we see an improvement from around 60 to 105 successful transactions per seconds when going from 1 to 8 clients. For Fabric++, we see the highest throughput with around 205 successful transactions per second already for 2 clients. For 8 clients, the throughput drops by around factor 2 to the throughput of Fabric, clearly showing that the firing clients also compete for resources. This is also visible in an increase in failed transactions when going from 1 to 8 clients per channel, which increase from 86 TPS to 928 TPS for Fabric and from 20 TPS to 841 TPS for Fabric++.

### 3.6.7 Hyperledger Caliper

For completeness, let us finally see how Fabric and Fabric++ perform under a run of the Hyperledger Caliper benchmarking framework. As said, Caliper severely struggles with high transaction firing rates, so we cannot use the configuration of Table **??** as before. Instead, we fire at a lower rate of 150 transactions per second per client, resulting in 600 transactions per second in total. As a consequence of this low firing rate, we also tune down the block size to 512 transactions. We test our custom workload with N = 10000, RW = 4, HR = 40%, HW = 10%, HSS = 1%. Table 3.9 shows the results.

| **Metric** | **Fabric** | **Fabric++** |
|---|---|---|
| Max. Latency [seconds] | 1.44 | 1.14 |
| Min. Latency [seconds] | 0.26 | 0.12 |
| Avg. Latency [seconds] | 0.47 | 0.28 |
| Avg. Successful Transactions per second | 188 | 299 |

*Table 3.9: Latency and Throughput as measured by Caliper for Fabric and Fabric++.*

Interestingly, Caliper also produces latency numbers additionally to the measured

throughput of successful transactions. We can see that the average latency of Fabric++ is almost half the latency of the vanilla Fabric. As less virtually invalid transactions trash the pipeline in Fabric++, valid transactions can commit earlier. The run of Caliper also confirms our findings on the throughput: Fabric++ significantly increases the number of successful transactions per second.

## 3.7 Conclusion

In this work, we identified strong similarities of the transaction pipeline of contemporary permissioned-blockchain systems at the case of Hyperledger Fabric and distributed database systems in general. We analyzed these similarities in detail and exploited them to transition mature techniques from the context of database systems to Fabric, namely transaction reordering to remove serialization conflicts as well as early abort of transactions, that have no chance to commit. In an extended experimental evaluation, where we tested Fabric++ and the vanilla version under the Smallbank benchmark as well as under a custom workload, we show that Fabric++ is able to significantly outperform Fabric by up to a factor of 12x for the number of successful transactions per second. Further, we are able to almost half the transaction latency, while keeping the scaling capabilities of the system intact.

Overall, in this chapter, we tried to bridge the gap between the relational database systems and the permissioned-blockchain system by utilizing well-established research from the former to optimize the latter class of data management systems. We saw significant improvements, with some great insights on the permissioned-blockchain system's architecture. We use the knowledge gained from this chapter to propose all new framework that aims to come up with a permissioned-blockchain system which has performance comparable to the modern database systems. To achieve such a high goal, we approach this problem in a different direction: aiming to transform a database system into a permissioned-blockchain system, rather than utilizing database technology inside permissioned-blockchain systems.

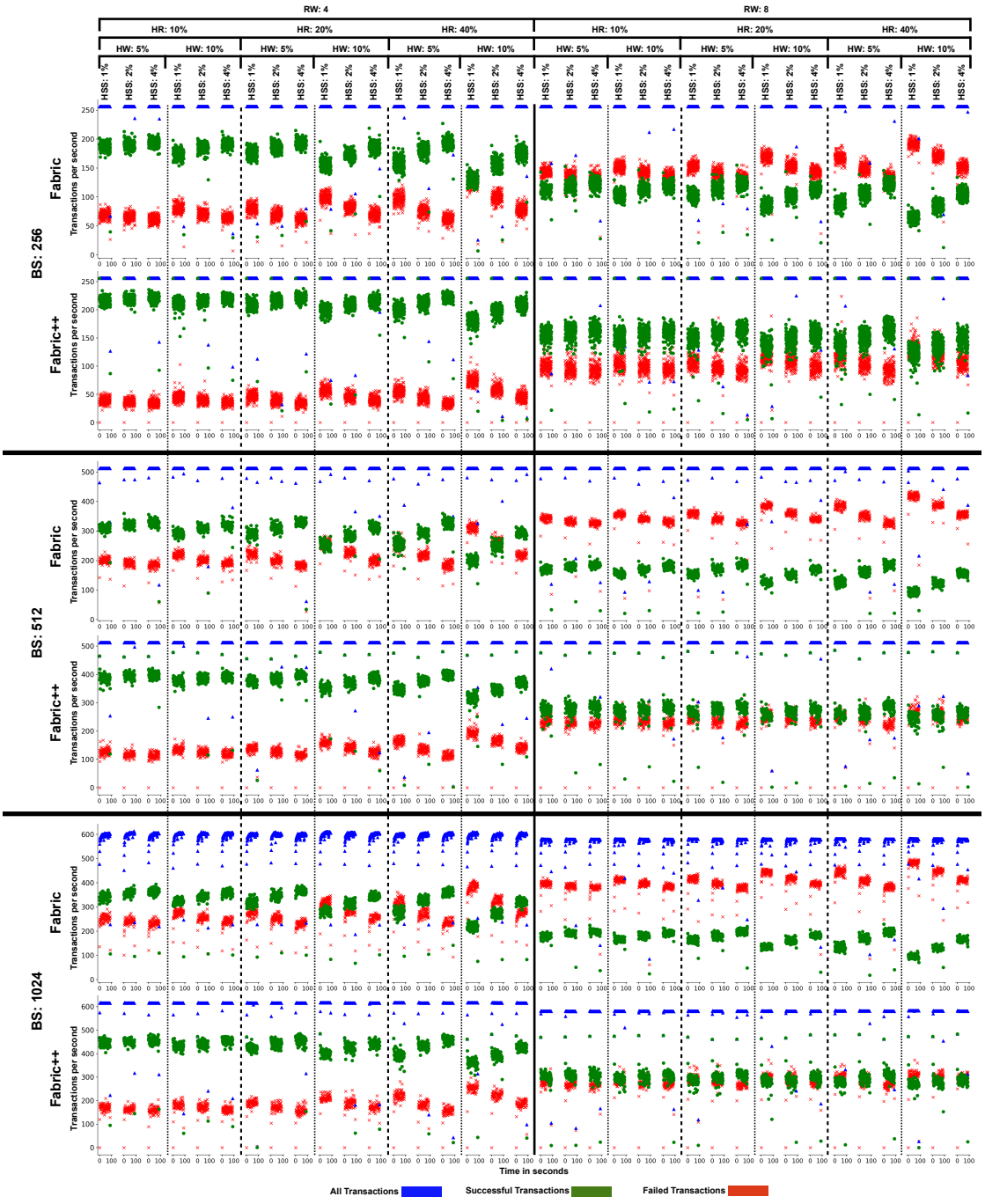



*Figure 3.14: Transactional Throughput of Fabric and Fabric++ under 108 different configurations.*

**Algorithm 1:** Reordering mechanism in pseudo code.

```
func reordering(Transaction[] S) {
   Graph cg = buildConflictGraph(S)

   Graph[] cg_subgraphs = divideIntoSubgraphs(cg)

   Cycle[] cycles = emptyList()
   foreach subgraph in cg_subgraphs:
      if(subgraph.numNodes() > 1):
         cycles.add(subgraph.getAllCycles())
   MaxHeap transactions_in_cycles = emptyMaxHeap()
   foreach Cycle c in cycles:
      foreach Transaction t in c:
         if transactions_in_cycles.contains(t)
            transactions_in_cycles[t]++
         else
            transactions_in_cycles[t] = 1
   Transaction[] S' = S
   while not cycles.empty():
      Transaction t = transactions_in_cycles.popMax()
      S'.remove(t)
      foreach Cycle c in cycles:
         if c.contains(t):
            c.remove(t)
            cycles.remove(c)
         foreach Transaction t' in c:
            transactions_in_cycles[t']--

   Graph cg' = buildConflictGraph(S')
   Transactions[] order = emptyList()
   Node startNode = cg'.getNextNode()
   while order.length() < cg'.numNodes():
      addNode = true
      if startNode.alreadyScheduled():
         startNode = cg'.getNextNode()
         continue
      foreach Node parentNode in startNode.parents():
         if not parentNode.alreadyScheduled():
            startNode = parentNode
            addNode = false
            break
      if addNode:
         startNode.scheduled()
         order.append(startNode)
         foreach Node childNode in startNode.children():
            if not childNode.alreadyScheduled():
               startNode = childNode
               break
   return order.invert()
}
```

# Chapter 4

# ChainifyDB: A Non-invasive Transformation of Database Systems into a Blockchain System

Today's permissioned blockchain systems come in a stand-alone fashion and require the users to integrate yet another full-fledged transaction processing system into their already complex data management landscape. This seems odd as blockchains and traditional DBMSs share large parts of their processing stack. Thus, rather than replacing the established data systems altogether, we advocate to simply '*chainify*' them with a blockchain layer on top.

Unfortunately, this task is far more challenging than it sounds: As we want to build upon *heterogenous* transaction processing systems, which potentially behave differently, we cannot rely on every organization to execute every transaction deterministically in the same way. Further, as these systems are already filled with data and being used by top-level applications, we also cannot rely on every organization being resilient against tampering with its local data.

Therefore, in this work, we will drop these assumptions and introduce a powerful processing model that avoids them in the first place: The so-called *Whatever-LedgerConsensus (WLC)* model allows us to create a highly flexible permissioned blockchain layer coined *ChainifyDB* that (a) is centered around bullet-proof database technology, (b) makes even stronger guarantees than existing permissioned systems, (c) provides a sophisticated recovery mechanism, (d) has an up to 6x higher throughput than the permissioned blockchain system Fabric, and (e) can easily be integrated into an existing heterogeneous database landscape.

## 4.1 Introduction

The vast majority of modern permissioned blockchain systems (PBS) [29, 1, 18, 7, 5, 70, 33], in which all organizations are known at any time, come as stand-alone end-to-end transaction processing systems. As a consequence, an organization that wants to utilize blockchain technology is currently forced to add yet another data management system to its infrastructure. However, since this infrastructure typically already consists of various established systems, which are filled with data and used by top-level applications, this approach is extremely troublesome. Data must be migrated, applications must be rewritten, personnel must be retrained — in general, the integration and maintenance of a new full-fledged system is associated with high costs and immense effort.

This raises the question, whether it is actually necessary to reinvent the wheel and design blockchain systems in a stand-alone fashion in the first place. Large parts of the transaction processing stack are conceptually shared with traditional database management systems [80]. Why not simply *reuse* these parts and build upon them? Precisely, instead of replacing the established database management systems altogether, we advocate to extend them with the missing and desired blockchain functionality.

Unfortunately, this task is far more challenging than it sounds at first. This has to do with two fundamental differences between stand-alone PBSs and our design.

First, due to their restrictive design, stand-alone PBSs can make the convenient assumption that:

1) *Every organization executes every transaction deterministically in the same way.*

As every organization runs the very same storage system and the very same transaction processing engine, it can be assumed comfortably that every transaction is interpreted in exactly the same way and results in the same effect across all organizations. In contrast to that, in a heterogeneous setup we don't have the luxury to rely on this assumption. As one organization of the network might build upon DBMS X while another one builds upon DBMS Y, the exact same transaction could result in different effects across organizations. Reasons for this are manifold: Systems might implement a different interpretation of the SQL standard or of the used data types.

The second fundamental difference is that, due to their restrictive design, stand-alone PBSs can assume that:

2) *Every organization is resilient against tampering with its local data.*

As stand-alone PBSs employ their own dedicated storage system, they fully control all access that is happening to it. Thus, in practice, it is unlikely that the data is corrupted externally. However, in our design we cannot make this assumption. As we want to utilize arbitrary database systems containing various forms of data, which are accessed from top-level applications aside from our blockchain layer, the chances of corruption are significantly higher in our case.

In summary, the processing model employed by the vast majority of stand-alone systems is obviously not powerful enough to handle the challenges of our highly flexible design. To understand the precise problem and to come up with a new and more powerful model, let us inspect the general workflow that is currently applied.

### 4.1.1 Order-Consensus-Execute

The model, that is currently implemented by the vast majority of stand-alone PBSs [29, 1, 18, 7, 5, 70, 33] is called *order-consensus-execute (OCE)*. First, in the order-phase, an order on a batch of transactions is proposed. Then, in the consensus-phase, the organizations try to globally agree on this ordered batch using some sort of consensus mechanism. If a consensus is reached, in the final execute-phase, the agreed-upon ordered batch is executed locally by every organization. As a result, all honest organizations reach the same state.

The problem lies in the assumption, that if a consensus is reached on the result of the order-phase, every honest organization will be in the same state after the execution-phase. In other words, it assumes that everything goes well after the consensus-phase. Figure 4.1 visualizes the problem: OCE assumes deterministic behavior on anything happening *after* the consensus-phase, namely on the execute-phase.

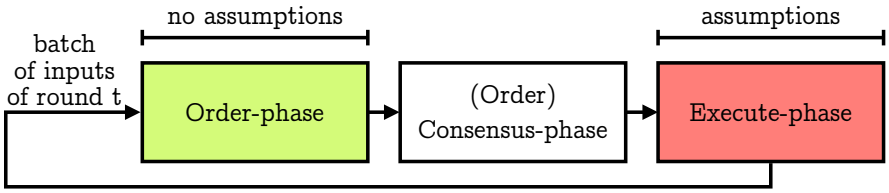



*Figure 4.1: The order-consensus-execute model (OCE). The consensus-phase sits between the order-phase and the execute-phase. As a consequence of this design, assumptions must be made on everything after the consensus-phase, namely on the execute-phase.*

Unfortunately, this assumption is not compatible with our design as we want to

chainify heterogeneous infrastructures, that potentially behave differently and that are potentially prone to tampering with the data.

### 4.1.2 Whatever-LedgerConsensus

To eliminate the need for assumptions, we argue that the consensus-phase must be pushed towards the end of the processing pipeline. Figure 4.2 shows the effect of doing so, resulting in the *order-execute-consensus model (OEC)*. If we reach consensus on the effects of the execute-phase instead of reaching consensus on the order established by the order-phase, no assumptions must be made on the order-phase and execute-phase.

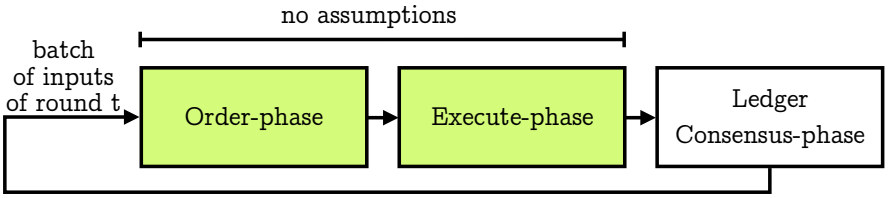



*Figure 4.2: The order-execute-consensus model (OEC). The consensus-phase sits at the end of the pipeline, after both order-phase and the execute-phase. As consensus is reached on the effects of the execute-phase, no assumptions must be made on any previous phase.*

From this perspective, we are actually able to abstract from the concrete order-phase and the execute-phase: Whatever happens in these phases, we are able to detect all differences in the produced effects in the consensus-phase afterwards.

This results in a new, highly flexible processing model we call the *Whatever-LedgerConsensus model* or *WLC* for short (pronounced "We'll see!"). Figure 4.3 visualizes the two phases of our WLC model:

1. **W**hatever-phase. Each organization does whatever it deems necessary to pass the LC-phase later on.

2. **L**edger**C**onsensus-phase. We perform a consensus round on the effects of the whatever-phase to check whether a consensus can be reached on them. If yes, the effects are committed to a ledger. If an organization is non-consenting, it must perform a recovery. If no consensus is reached at all, all organizations must try to recover.

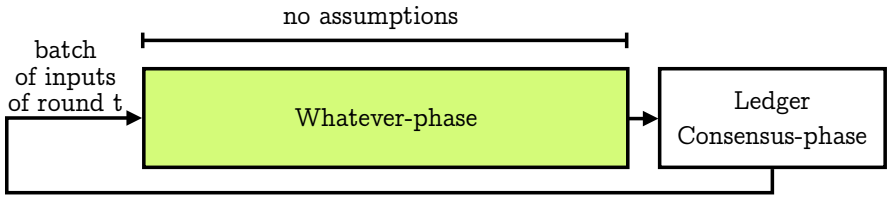



*Figure 4.3: Our Whatever-LedgerConsensus model (WLC). We do not make assumptions on the behavior of the whatever-phase. In the consensus-phase, consensus is reached on the effects of the whatever-phase.*

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

### 4.1.3 Contributions

**(1)** We present a new processing model coined the **Whatever-LedgerConsensus model (WLC)**. In contrast to existing processing models, our model does not make any assumptions on the behavior of the local engines. Still, we are able to detect any deviation of an organization, irrespective of the cause. Our model allows us to realize highly flexible blockchain systems while being able to express existing models like OCE as special cases.

**(2)** We present the **Whatever Recovery Landscape** and discuss the different classes of recovery algorithms possible in the WLC model, depending on the amount of information available regarding actions and effects.

**(3)** We present a concrete system instance of WLC coined **ChainifyDB**. We start with a set of heterogeneous database systems and show how to connect them to a network providing permissioned blockchain-properties. We only requires the DBMSs providing a trigger mechanism as defined in SQL 99 or similar.

**(4)** We show initial results with a **vendor-independent recovery algorithm** allowing ChainifyDB to efficiently recover non-consenting organizations. Notice that systems like Fabric [18] do not have *any* recovery mechanism.

**(5)** We perform an extensive **experimental evaluation of ChainifyDB** in comparison with the comparable state-of-the-art permissioned blockchain systems Fabric [18] and Fabric++ [80] and achieve an up to $6x$ higher throughput. Further, we show that ChainifyDB is able to fully utilize the performance of the underlying database systems and

demonstrate its robustness and recovery capabilities experimentally.

The paper is structured as follows. In Section 4.2, we first discuss related works, which sit at the intersection of databases and blockchains and contrast WLC and ChainifyDB to them from a conceptual perspective. In Section 4.3, we formalize our Whatever-LedgerConsensus model (WLC). In Section 4.4, we present the Whatever Recovery Landscape. In Section 4.5, we present the logical design of ChainifyDB and its components, including the recovery mechanism. In Section 4.6, we present interesting optimizations possible in ChainifyDB. In Section 4.7, we present concrete implementation details of ChainifyDB. Finally, in Section 4.8, we perform an extensive experimental evaluation.

## 4.2 Related Work

As stated, the vast majority of permissioned blockchain systems implement variants of the OCE model, with [1, 7, 5, 70, 33] being prominent examples. Even Fabric [18, 23], which uses a model the authors call *execute-order-validate*, implements in the end a form of OCE: In the execute-phase, the effect of transactions are computed. After ordering the transactions, the effects of non-conflicting transactions are used to update the state in the given order. Thus, the validate-phase in Fabric highly resembles the execute-phase in OCE. Interestingly, none of these system integrate any form of recovery mechanism: As soon as an organization deviates, it is considered as being malicious independent of the cause and implicitly banned from further transaction processing.

Apart from the execution model, there are other projects that sit at the intersection of databases and blockchains. In [70], the authors extend the relational system PostgreSQL with blockchain features. This results in a "blockchain relational database", which is capable of performing trusted transactions between multiple PostgreSQL instances. While this project is clearly a step in the right direction, it does not go far enough: the blockchain relational database still comes in a stand-alone fashion and forces the users to integrate a whole new system into their infrastructure. Further, they heavily modify the internals of PostgreSQL to integrate the blockchain features. In contrast to that, in ChainifyDB we install the blockchain features on-top of black-box DBMSs without changing them in any way. We intended to compare ChainifyDB against this system, however, the source code is not available.

A project with a similar attitude is BlockchainDB [33, 34]. In contrast to [70] and to ChainifyDB, the authors of [33, 34] install a database layer *on top of* an existing blockchain, such as Ethereum [3] or Hyperledger Fabric [18]. This database layer allows the user to manage and access the underlying shared data in a more convenient way

than directly communicating with the blockchain system. Unfortunately, the comfort is relatively limited: To support a variety of blockchain systems, the authors have to stick to a key-value data model and a simple put()/get() query interface.

Another project at the intersection of databases and blockchains is Veritas [41]. This visionary paper also proposes to extend existing database systems by blockchain features; however, they focus on a cloud infrastructure. To synchronize instances, they utilize log shipping. Therefore, this solution requires the underlying database system to provide log shipping in the first place and disallows the connection of *different* database systems in one network, if their log shipping mechanisms are not compatible with each other. Of course, they are not in general.

The project ChainSQL [68] takes the open-source blockchain system Ripple [19] and integrates relational and NoSQL databases into the storage backend. This enables it to run SQL-style respectively JSON-style transactions. However, by integrating database systems into the heavy-weight blockchain system Ripple, the authors limit the transaction processing performance to that of Ripple — thus overshadowing the high performance of the underlying database systems. In contrast to that, ChainifyDB is designed from the get-go to leverage the power of the underlying systems — in particular using highly parallel transaction execution, even across heterogeneous database systems.

The project BigchainDB [1] combines the blockchain framework Tendermint [14] with the document store MongoDB and therefore extends it with blockchain features. In contrast to ChainifyDB, the system is shipped in a stand-alone fashion and focuses on the query interface of MongoDB.

Apart from works aiming at closing the gap between databases and blockchains from an architectural perspective, there is a considerable amount of research improving the performance of permissioned blockchains. In [80], [45], [25], and [61], the authors apply various optimizations to improve the throughput of successful transactions in Fabric.

## 4.3 Whatever-Ledger Consensus

Let us now start by introducing our novel process model *Whatever-Ledger Consensus (WLC)* and formalize it.

### 4.3.1 Core Idea

In short, the core idea of WLC is to **not** seek consensus on *what should be done*-actions, but rather seek consensus on the effect of the actions *after they have been performed*. This allows us to drop assumptions on the concrete transaction processing behavior of the organizations. It also allows us to detect any external tampering with the data. The WLC model implies that if consensus on the effects of certain actions cannot be reached, the organizations must not commit the effects of their actions. Note that WLC is also more powerful than classical 2PC or 3PC-style protocols in the sense that they still assume deterministic behavior of the organizations without looking back at the produced effects. In WLC, we simply do not care about whether organizations claim to be *prepared* for a transaction and what they claim to *commit*. In contrast, we solely look at the outcome. If we can reach consensus on the outcome, it does not matter anymore which actions those organizations used to get to the same outcome in the first place. In summary, *we measure the outcome rather than the promises*.

### 4.3.2 Processing Model

In the following, let us formalize WLC in detail.

Let $O_1, \ldots, O_k$ be the $k$ participating organizations in the network. As always, these organizations do not trust each other. Still, they want to perform a mutual sequence of inputs, which is fed into the system batch-wise round by round, as visualized in Figure 4.3. The following description shows the W-phase and the LC-phase in round $t$.

**Whatever-phase**. Note that although the *same* sequence of inputs enters the system, each organization might actually receive a potentially different set of *actions* for processing, as we do not make any assumptions on what exactly is happening in the W-phase. For instance, an ordering service could distribute different actions to different organizations for whatever reason. Thus, we define $A_{l,1}, \ldots, A_{l,t}$ as the sequence of actions, that the individual organization $O_l$ receives till round $t$.

*Effect Functions:* Further, we assume that there is an *effect function* $F_l()$. Only if the effect contains the state, an action can be applied on the effect to generate a new effect. We compute for each organization $O_l$ the accumulated effect $E_{l,t}$ as:

$$E_{l,t} = F_l(E_{l,t-1}, [A_{l,t}])$$

with $E_{l,0} = \emptyset$ being the initial empty effect. Notice the iterative construction of this function (which will later on create our blockchain-style chaining of effects):

$$E_{l,t} = F_l(F_l(F_l(\ldots), [A_{l,t-1}]), [A_{l,t}]).$$

Notice that we use $E_{l,t} = F_l(\emptyset, [A_{l,1}, \ldots, A_{l,t}])$, as a shorthand, still there will be a separate effect function call for each of the $t$ actions.

**LedgerConsensus-phase**. On the accumulated effects $E_{l,t}$ for $1 \leq l \leq k$, a consensus must be reached (and not necessarily on the entire state!). Otherwise, the system is not allowed to proceed to round $t + 1$. To decide whether consensus is reached, a *consensus policy* $c$ specifies how many organization must at least have reached the same accumulated effect. Thus, consensus in round $t$ is reached if

$$\exists O_{cons} = \{O_i, ..., O_j\} : |O_{cons}| \geq c : (E_{i,t} = \ldots = E_{j,t}) = E_{cons,t}.$$

We summarize the effect on which consensus has been reached as $E_{cons,t}$. If consensus has been reached, each organization $O_l$ has to decide on its own whether its local effect $E_{l,t}$ matches the consensus effect $E_{cons,t}$. If it matches, the effect can be committed to a *ledger* and round $t$ ends. Otherwise, $O_l$ can not proceed to round $t + 1$ and tries to recover. If no consensus can be reached at all, i.e. $E_{cons,t} =$ undefined, then no organization can proceed to round $t + 1$. In this case, all organizations try to recover. We will discuss in Section 4.4 the detailed recovery behavior and introduce a concrete database vendor-independent recovery algorithm in Section 4.5.

## 4.4 Whatever Recovery

| | | Actions | | |
|---|---|---|---|---|
| | | **not accessible** | **accessible (blackbox)** | **accessible (whitebox)** |
| **Effects** | **State not contained** | — | + full replay | + optimized full replay |
| | **State contained** | + restore state | + partial replay | + optimized partial replay |

*Table 4.1: The 2 × 3 Whatever Recovery Landscape. The two-dimensions of Whatever recovery (accessibility of effects vs actions) and their implications on the classes of recovery algorithms possible*

Let us now see how we perform recovery and what levels of recovery are actually possible in the WLC model.

### 4.4.1 Non-Consenting Organization Scenario

As described formally in Section 4.3.2, we must perform recovery on an organization $O_l$ if $E_{l,t} \neq E_{cons,t}$. The reasons for this are multitude: It could be that $O_l$ simply interpreted $A_{l,t}$ differently than the others or that non-determinism is hidden in $A_{l,t}$. It could also be that someone, e.g. an administrator, tampered with $E_{l,t}$.

Irrespective of the cause, during recovery, we have to compute a new effect $E'_{l,t}$. If $E'_{l,t} \neq E_{l,t}$, then the computed effect differs from the original effect, which was used for consensus, and $O_l$ has a chance to recover. If now $E'_{l,t} = E_{cons,t}$, then $O_l$ has recovered and can proceed with round $t + 1$. If not, then it can not recover and is excluded from the network.

If no consensus has been reached at all, i.e. $E_{cons,t} = \textit{undefined}$, then we perform a new consensus round on the new effects. The network can recover, if

$$\exists O'_{cons} = \{O_i, ..., O_j\} : |O'_{cons}| \geq c : E'_{i,t} = \ldots = E'_{j,t}.$$

### 4.4.2 The 2 × 3 Recovery Landscape

There are tradeoffs and optimizations involved in practical recovery. For example, instead of starting with an empty state and replaying the entire sequence of actions there are many more options. Table 4.1 introduces the whatever recovery landscape. That landscape has two dimensions *Effects* and *Actions*. For the dimension *Effects*, we distinguish between whether the *state is not contained* and the *state is contained*. For the dimension *Actions*, we distinguish between *not accessible*, *blackbox actions* (we have access but do not understand the semantics in any way), and *whitebox actions* (we have access **and** understand the semanics, i.e. we see the individual operations carried out). Like that we receive six different classes of recovery algorithms. Note that each cell in that landscape includes all cells with weaker accessibility levels (i.e. all cells that are further left and/or up). In the following we discuss each cell individually. We label each subsection with a visual notation of its position in the whatever recovery landscape, e.g. ▣ for (State: accessible, Actions: whitebox).

### 4.4.3 No Recovery

**Accessibility:** ▣ State: na, Actions: na.

If neither state nor actions are available to us, we cannot recover a non-consenting organization.

**Recovery:** —

### 4.4.4 Recovery from a State

**Accessibility:** ◧ State: contained, Actions: na.

We have access to the state computed by the individual organizations but do not understand their semantics. As we don't have access to actions, we cannot apply them in any way.

**Recovery:** In order to recover organization $O_l$ in round $t$, we overwrite its local effect $E_{l,t}$ with the effect $E_{i,t}$ of *any* other organization $O_{i \neq l}$, that is matching the consensus effect $E_{cons,t}$

$$E'_{l,t} = F_l(E_{i,t}, []).$$

If $E_{cons,t} =$ undefined, i.e. no consensus was reached in round $t$, then $O_l$ cannot recover.

### 4.4.5 Full Replay

**Accessibility:** ◨ State: na, Actions: blackbox.

The state is not contained, but we have access to the sequence of blackbox actions.

**Recovery:** In order to recover organization $O_l$ in round $t$, we can (blindly) replay the entire history of blackbox actions $A_{l,1}, \ldots, A_{l,t}$ from the very beginning to restore the accumulated effect

$$E'_{l,t} = F_l(\emptyset, [A_{l,1}, \ldots, A_{l,t}]).$$

If $E'_{l,t} = E_{cons,t}$, then $O_l$ may rejoin the network.

### 4.4.6 Partial Replay from a State

**Accessibility:** ◧◨

State: contained, Actions: blackbox.

The state is contained. Further, we have access to blackbox actions but do not understand their semantics.

**Recovery:** In order to recover $O_l$ in round $t$, we can perform a partial replay. In other words, we start with an older effect $E_{l,s<t}$ and partially replay the blackbox actions $A_{l,s+1}, \ldots, A_{l,t}$ (*redo* in database lingo):

$$E'_{l,t} = F_l(E_{l,s<t}, [A_{l,s+1}, \ldots, A_{l,t}])$$

If $E'_{l,t} = E_{cons,t}$, then $O_l$ may rejoin the network.

### 4.4.7 Optimized Full Replay

**Accessibility:** ⊞■

State: na, Actions: whitebox.

The effect does not contain the state, but we have access to the sequence of whitebox actions and understand the precise semantics of the actions.

**Recovery:** In order to recover organization $O_l$ in round $t$, we can replay the entire history of actions from the very beginning to restore the accumulated effect. However, as we have *whitebox* actions available, we can also optimize the replay: For example, if we know that an action $A_{l,t}$ consists of a sequence of three transactions $A_{l,t} = [T_1, T_2, T_3]$, where all three transactions update the same record, then we could safely drop $T_1$ and $T_2$ and use $A'_{l,t} = [T_3]$ for the replay. Further possible optimizations include changing the order of operations and to allow for parallel execution of non-conflicting operations [98], as we show in Section 4.6.3. Thus, we restore the effect using the optimized actions $A'_{l,1}, \ldots, A'_{l,t}$:

$$E'_{l,t} = F_l(\emptyset, [A'_{l,1}, \ldots, A'_{l,t}])$$

If $E'_{l,t} = E_{cons,t}$, then $O_l$ may rejoin the network.

### 4.4.8 Optimized Partial Replay from a State

**Accessibility:** ⊞■

State: contained, Actions: whitebox.

We have access to the state as well as access to whitebox actions.

**Recovery:** In order to recover $O_l$ in round $t$, we start with an older effect $E_{l,s<t}$ and partially replay the optimized actions $A'_{l,s+1}, \ldots, A'_{l,t}$ (*redo* in database lingo):

$$E'_{l,t} = F_l(E_{l,s<t}, [A'_{l,s+1}, \ldots, A'_{l,t}])$$

If $E'_{l,t} = E_{cons,t}$, then $O_l$ may rejoin the network.

### 4.4.9 Abstraction vs Implementation

Notice that we use the concepts of 'effect' and 'action' to abstract from the details of a concrete implementation. For example, conceptually, it is not strictly necessary to physically materialize an effect: Replaying the entire history of actions is sufficient to restore an effect. This is similar to log-only databases ("the log is the database") [32, 88, 92] that regard the database store as a performance optimized representation of the database log.

In ChainifyDB, which we will describe in detail in the next section, effects are materialized in form of database states and snapshots of those to enable high performance transaction processing and recovery. Actions represent blocks of transactions, which modify the database state and thereby generate new effects. Let us now see how it works in detail.

## 4.5 Chainify DB

In Section 4.3 we have introduced and formalized our novel WLC model which is able to detect deviation of effects without making any assumptions on the behavior of the whatever-phase. In Section 4.4, we presented the different recovery options in the WLC landscape depending on the accessibility of effects and actions.

In this section we present ChainifyDB, a concrete system that instantiates the WLC model. The core feature of ChainifyDB is to equip *established* infrastructures, which already consist of several database management systems, with blockchain functionality as a layer on top. The challenge is that these infrastructures can be highly heterogeneous, i.e. every participant potentially runs a *different* DBMS where each system can interpret a certain transaction differently. As a result, the effects across participants might differ.

As mentioned earlier, the classical OCE model, which relies on the previously discussed strong assumptions, is not capable of handling such a heterogeneous setup: The execution across participants is neither guaranteed to be deterministic nor equal. In contrast to that, our WLC model is perfectly suited to handle such heterogeneous scenarios, where no assumptions on the behavior of the organizations are made.

### 4.5.1 Overview on our WLC-Implementation

Let us now see in detail how we implement the W-phase as well as the LC-phase in ChainifyDB. Just like our model, the implementation operates in rounds and consumes a batch of input transactions, which have been proposed by clients to the system, in every round.

**Whatever-phase**. In its simplest variant, ChainifyDB instantiates the W-phase of the WLC model with two subphases: the *order-subphase* and the *execute-subphase*. Figure 4.4 visualizes the instantiation.

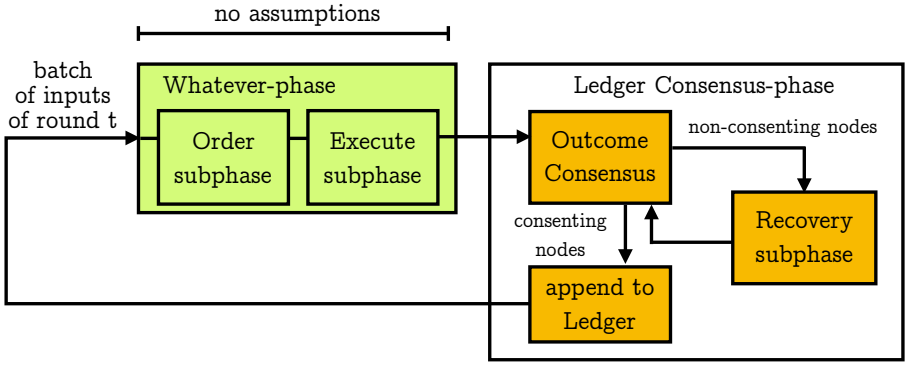



*Figure 4.4: ChainifyDB as a concrete instance of the Whatever-LedgerConsensus model (WLC).*

*Order-subphase*. In the order-subphase of round $t$, a batch of input transaction is globally ordered and grouped in a block. Note that an action in our earlier formalization resembles a block of transactions here, i.e. $A_{l,t} = [T_1, T_2, T_3]$. Packing transactions into blocks is merely a performance optimization in order to amortize the costs for consensus later on. There is *no* conceptual need to form blocks. Notice that our order-subphase fully resembles the order-phase in the classical OCE-model. However, in strong contrast to the OCE-model, we do not perform a consensus round on the established order afterwards, even if we do not trust the ordering service in any way. For now, we simply take whatever the ordering service outputs.

*Execute-subphase*. In the execute-subphase, each organization $O_l$ receives an action $A_{l,t} = [T_1, T_2, T_3]$ produced by the order-phase. Each transaction of that block, that has valid signatures, is then executed against the local relational database. This execution potentially updates the database and thereby produces an effect $E_{l,t}$. In Section 4.5.2 and Section 4.5.3, we will outline in detail how this effect looks like in ChainifyDB. For now, let us simply assume that the effect captures all modifications done by the valid transactions of the block to the database.

Again, we want to point out that in contrast to the OCE-model, we do not assume deterministic execution across all organizations: The different DBMSs of two organizations could have interpreted a transaction slightly differently or two organizations could have received different blocks from the order-phase altogether.

**LedgerConsensus-phase**. In the ledger consensus phase, all organizations have to reach consensus on the effects produced by the whatever-phase of the individual organizations. Thus, in a consensus round, which we will describe in Section 4.5.4 in detail, the organizations first try to agree on one particular effect. Then, each organization whose effect is consenting commits it to its local ledger and proceeds with round $t + 1$. Again, only if consensus on an effect is reached, we consider it as globally committed.

*Recovery-subphase*. If the effect of an organization is non-consenting, the organization must at least try to recover from this situation. This is done using a variant of *Optimized Partial Replay from a (Logical) Snapshot* as introduced in Section 4.4. We will explain our recovery mechanism in detail in Section 4.5.5.

### 4.5.2 Logical per Block Digests

In the previous section, we mentioned that the execution of a block on the local database produces an effect as a side-product of execution. On this effect, the consensus round is performed. It is also eventually committed to the ledger. Thus, let us see in the following how we precisely generate the effect.

In ChainifyDB we assume SQL-99 compliant relational DBMSs to keep the state at each organization. This has two reasons: On the one hand, we want to allow for existing organizations with existing DBMS-products to be able to easily build WLC-networks with blockchain-style guarantees. On the other hand, we can utilize SQL 99 triggers to realize a vendor-independent digest versioning mechanism, that specifically versions the data of ChainifyDB in form of a *digest table*.

The digest table is computed per block. We instrument every shared table in our system with a trigger mechanism to automatically populate this digest as follows: for every tuple changed by an `INSERT`, `UPDATE`, and `DELETE`-statement, we create a corresponding *digest tuple*. A digest tuple has the following schema: `[PK:<as of Foo>, serial:int, hash:int, T]`. Here, `PK` is the primary key of the original table (which may of course also be a compound key), `serial` is a strictly monotonously rising counter used to distinguish entries with the same `PK` (every new version of a tuple increases this counter), `hash` is the digest of the values of the tuple after it was changed (for a delete: its state before removing it from the table), `T` is the type of change that was performed, i.e. (I)nsert, (U)pdate or (D)elete. Notice that in contrast to recovery

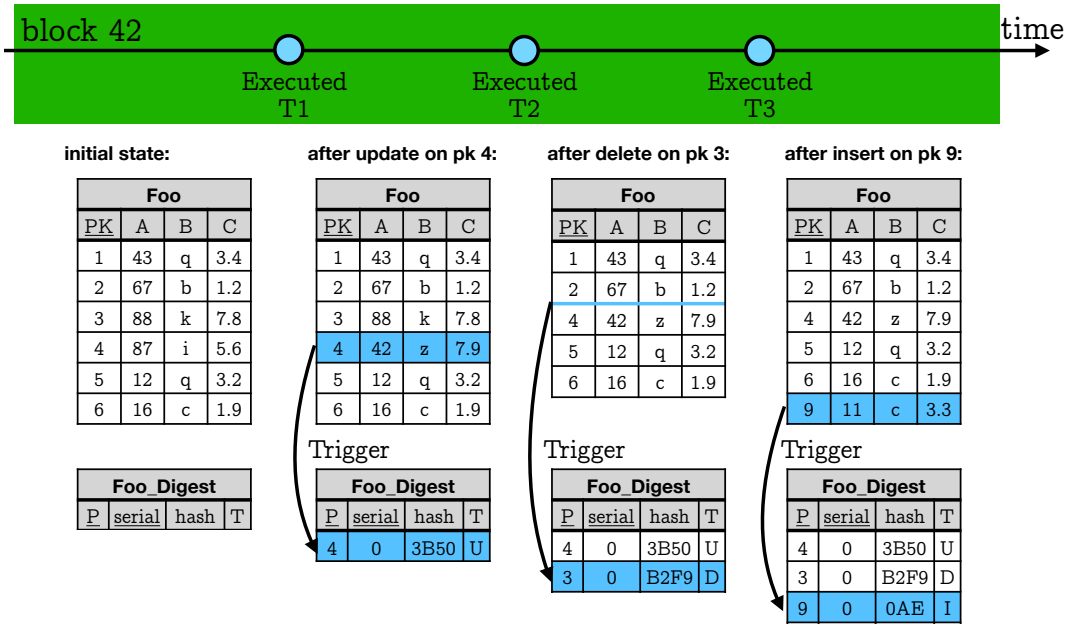



*Figure 4.5: Logical tuple-wise per block digest computation on an example table Foo. All changes are automatically tracked and digested through SQL 99 triggers.*

algorithms like ARIES [64], in those tuples we do not preserve the information how to undo/redo changes, as we simply do not need that information.

Figure 4.5 shows an example how to process a block of three transactions: we start with a particular state of table `Foo` and an empty digest table `Foo_Digest`. Now, we perform an update on tuple with `PK`=4. As a result, the tuple in `Foo` is changed to (4, 42, z, 7.9) and we insert a new digest tuple (4, 0, 3B50, U) into `Foo_Digest`. After that a delete of record with PK=3 is performed. The tuple in `Foo` is deleted and we insert a new digest tuple (3, 0, B2F9, D) into `Foo_Digest`. We proceed until we processed all three transactions TA1–TA3 available in this particular block. For the next block to process we start with an empty digest table.

### 4.5.3 LedgerBlocks

Although the digest table captures all changes done to a table by the last block of transactions, it does not represent the effect yet. The actual effect is represented in form of a so called `LedgerBlock`, which consists of the following fields:

1. `TA_list`: The list of transactions that were part of this block. Each transaction is represented by its SQL code.

2. `TA_successful`: A bitlist flagging the successfully executed transactions. This is important as transactions may of course fail and that behavior should be part of consensus across all organizations.

3. `hash_digest`: A hash of the logical contents of the digest table. In our case, this is a hash over the hash-values present in the diff table. The hash values are concatenated in lexicographical [PK, serial]-order and then input into SHA256.

4. `hash_previous_LedgerBlock`: A hash value of the entire contents of the previous `LedgerBlock` appended to the ledger, again in form of a SHA256 hash. This backward chaining of hashes allows anyone to verify the integrity of the ledger in linear time.

This `LedgerBlock` now leaves the W-phase and enters the LC-phase to determine whether consensus can be reached.

### 4.5.4 Consensus Algorithm

In our permissioned setup, we can safely make the assumption, that all organizations of the network are known at any time and that no organization can join the network during a consensus round. This allows us to use a lightweight voting algorithm for this purpose, instead of having to rely on more heavyweight consensus algorithms such as [24, 103, 101, 76]. To determine whether consensus was reached, a *consensus policy* $c$ must be specified by all organizations in advance during the bootstrapping process of the network. The constant $c$ specifies how many organizations must have reached the same effect.

In the first step of the consensus algorithm, the individual organization has to count how often each `LedgerBlock` occurred in the network. To do so, it requests the so called `LedgerBlockHashes` from all other organizations and counts the occurrences, including its own local `LedgerBlockHash`. This `LedgerBlockHash` is essentially a compressed form of the contents of a `LedgerBlock`.

Consensus is reached if two conditions hold: (a) There must be a `LedgerBlockHash` that occurred at least $c$ times. (b) This consensus `LedgerBlockHash` equals the local `LedgerBlockHash`. If both hold, then the organization can append/commit its `LedgerBlock` to its local ledger.

### 4.5.5 Logical Checkpointing and Recovery

If an organization is non-consenting, it must enter recovery as outlined in Section 4.5.1 and presented in Figure 4.4.

For recovery, ChainifyDB implements *Optimized Partial Replay from a (Logical) Snapshot* as introduced in Section 4.4.8. Again, like the digests (see Section 4.5.2) our approach is *DBMS-system independent*: we do not need access to the source code of the DBMS. The Figures 4.6 and 4.7 show an example run of our checkpointing and recovery algorithm.

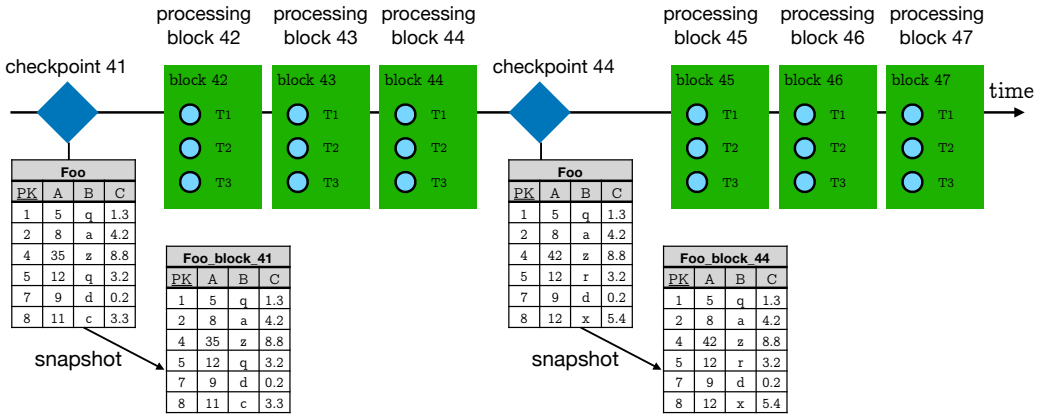



*Figure 4.6: ChainifyDB's checkpointing mechanism. Here, a checkpoint is created after every three blocks.*

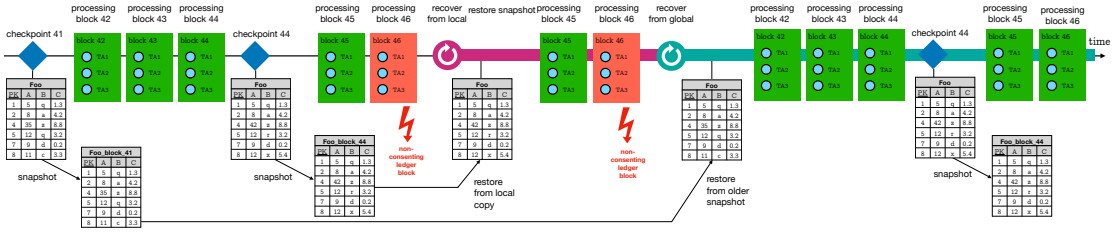



*Figure 4.7: ChainifyDB's Recovery using checkpoints. As block 46 is non-consenting it has to enter the recovery phase. It will first try to recover using the most recent local checkpoint. This fails in this example and hence recovery from an older checkpoint is performed.*

Figure 4.6 shows the normal operation mode of a single organization that is consenting: we create a checkpoint by creating a snapshot of table `Foo` after every $k$ blocks

($k = 3$ in the example). Snapshots are created on the SQL-level through either a non-maintained materialized view or by a `CREATE TABLE` command. If the source code of the DBMS and the operating system is available, the snapshotting support from the operating system could be exploited at this point as well [82, 50] – however, we do not make this assumption in our design. The snapshot is created for all tables that were changed since the last consistent snapshot. Creating such a snapshot is surprisingly fast: Snapshotting the accounts table with $1{,}000{,}000$ rows, which is part of the Smallbank [11] benchmark used in our experiment evaluation, takes only $827$ms in total on our machine of type $2$ running PostgreSQL (see Section 4.8). Notice that there is no need to store this checkpoint in the ledger, as done in ARIES [64] for instance: As the information contained in a checkpoint can be fully recomputed from the ledger, it has to pass the LC-phase anyways again.

Figure 4.7 shows an organization switching to recovery mode. This organization is in normal (consenting) mode for all blocks shown up to and including block 45. For block 46 this organization is non-consenting. Hence, it must enter the recovery phase.

First, we have to reset `Foo` to the state of the latest consistent snapshot `Foo_block_44`. Then, we replay block 45 which is consenting. Then, we replay block 46 which is unfortunately *again* non-consenting. In this situation we have to assume that this local snapshot has an issue, i.e. it was corrupted externally. Thus, we reset the local table `Foo` to the second latest snapshot `Foo_block_41`. Now, we replay all blocks starting from block 42 up to block 46. This time block 46 is consenting.

In our implementation, we keep three committed snapshots per organizations. If replaying from all of these snapshots does not lead to a consenting organization, then we can try to replay the *entire* history. If even this fails, we have to assume that the ordering service acts maliciously and can try again after starting a fresh ordering service, possibly by a different organization.

Notice that, of course, there is no 100% guarantee that any of these measures will lead to a consenting organization. Severe problems such as a hardware error (in particular an error that is not detected and transparently fixed by hardware or operating system itself) are out of reach for repair by ChainifyDB. The important point here is that we detect the problem early on (after every block of transactions and not only eventually), try to "rehabilitate" this organization through recovery, and reintegrate it into the network.

## 4.6 Optimizations

Before we come to the experimental evaluation of ChainifyDB, let us discuss a set of interesting optimizations.

### 4.6.1 Transaction Agreement

A powerful feature of ChainifyDB is that clients can propose arbitrary transactions to the system. These transactions are then simply executed against the local DBMSs of the individual organizations without any restrictions. However, there might be situations where such a plain execution without any restrictions is highly undesired by the organizations.

For example, consider the scenario where two organizations would like to log their mutual trades in a shared table. A transaction inserting a trade could look as follows

```
INSERT INTO Trades(TID,product,amount,totalprice)
VALUES (42,"Gearbox",5,60000):
```

Obviously, this transaction is only meaningful if certain integrity constraints hold: The selling organization must have enough products in stock (at least five gearboxes in our example) and the buying organization must have enough money available (at least $60{,}000$ in our example). To enforce such integrity constraints, we prepend an optional *agreement-subphase* to our pipeline, through which any transaction proposed to ChainifyDB must go first. Only if all involved organizations agree to the proposed transaction, the transction may enter the subsequent order-phase.

To enable the optional agreement phase, two steps must be carried out by the organizations: First, an *agreement policy* must be installed in consent when creating the shared table. It specifies for the shared table which organizations have to agree upon a transaction operating on that table. For our trading example, the policy of the table *Trades* could look as in Algorithm 2, enforcing both involved organizations to decide for agreement.

**Algorithm 2:** Agreement policy on the table *Trades*.

```
AgreementPolicy(Trades) =
   { SellingOrganization, BuyingOrganization }
```

Second, each organization has to implement its individual integrity constraints, which are evaluated against each proposed transaction. Note that these constraints could be formulated to compare the transaction against local data. For example, the agreements for the selling organization and the buying organization could look as in Algorithm 3. Only if both organizations agree to a transaction operating on the *Trades* table, it is passed on for further processing.

**Algorithm 3:** Agreement of two organizations.

```
SellingOrganization.agree(T) =
  { return T.amount <= Stocks.amount
          WHERE T.product == Stocks.product }
BuyingOrganization.agree(T) =
  { return T.totalprice <= Fund.availableMoney }
```

### 4.6.2 Iterative WLC-Setups

From a 10,000 feet perspective an agreement can be considered a different WLC-iteration where the participants agree upon the string describing the SQL-transaction to be executed rather than the outcome of running that transaction. This is visualized in Figure 4.8. In WLC 1 the organizations must first agree upon a trade to be done (represented as an SQL-transaction). If the organizations agree, a SQL string is inserted into a table `PlannedTrades`:

```
INSERT INTO PlannedTrades(TID,SQL_string)
VALUES (42,"UPDATE line SET..."):
```

If there is consent, that the tuple was inserted, i.e. that this trade should be done, the corresponding SQL string will be send to WLC 2 which executes the SQL string.

In summary, mapping subphases to incremental rounds of the WLC model enables interesting abstractions. In future work, we plan to investigate these mappings in depth.

### 4.6.3 Parallel Transaction Execution

Apart from the interplay of the phases, we did not precisely specify how the execute-subphase actually runs transactions in the underlying database system.

Naively, we could simply execute all valid transactions of a block one by one in a

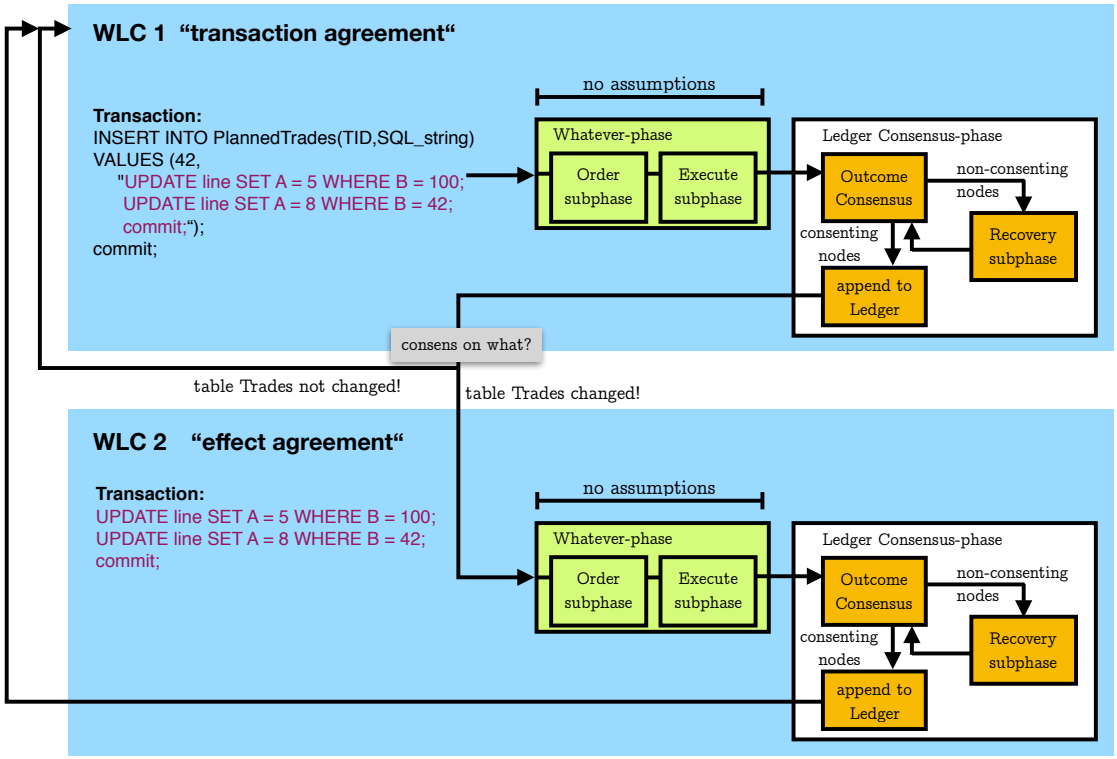



*Figure 4.8: Transaction agreement can be regarded as running a separate pre-WLC phase on transaction agreement before executing the actual transaction.*

sequential fashion. However, this strategy drastically wastes performance if the underlying system is able to execute transactions in parallel.

This leads us to an alternative strategy, where we could simply submit all valid transactions of a block to the underlying (potentially parallel) database system in one batch and let it execute them concurrently. While this strategy leverages the performance of the underlying system, it creates another problem: it is very likely that every DBMS schedules the same batch of transactions differently for parallel execution. As a consequence, the commit order of the transactions likely differs across organizations, thus increasing the likelihood of non-consent.

The strategy we apply in ChainifyDB sits right between the previously mentioned strategies and is inspired by the parallel transaction execution proposed in [98] and relates to the ideas of [37, 28, 75]. When a block of transactions is received by the execute-subphase, we first identify all existing conflict dependencies between transactions. This allows us to form mini-batches of transactions, that can be executed safely in parallel, as they have no conflicting dependencies.

Let us see in detail how it works. The process can be decomposed into three phases:

(1) **Semantic Analysis**. First, for a block of transactions, we do a semantic analysis of each transaction. Effectively, this means parsing the SQL statements and extracting the read and write set of each transaction. These read and write sets are realized as intervals on the accessed columns to support capturing both point query and range query accesses. For instance, assume the following two single-statement transactions:

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

The extracted intervals for these transactions are:

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

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

Let us extend the example from our Semantic Analysis Phase and let us assume, that T1 has been added to the dependency graph already. By inspecting T2 we can determine that PK[42, inf] overlaps with PK[100,100] of T1. As T2 is an update transaction, there is a conflict between T2 and T1 and add a dependency edge from T1 to T2. Figure 4.9 presents an example dependency graph for $9$ transactions.

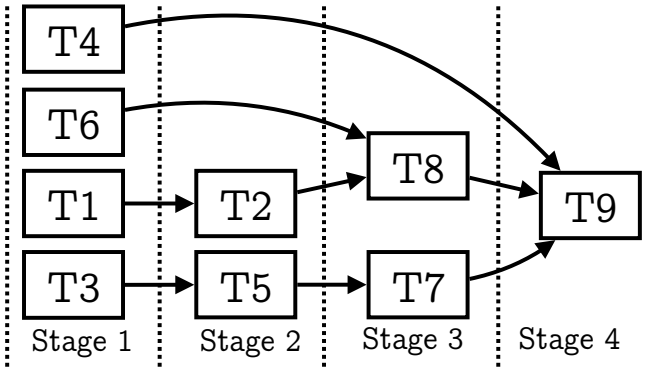



*Figure 4.9: A topological sort of the dependency graph with $k = 9$ transactions yielding four execution stages.*

(3) **Executing the Dependency Graph**. Finally, we can start executing the transactions in parallel. To do so, we perform topological sorting, i.e. we traverse the execution stages of the graph, that are implicitly given by the dependencies. Our example graph in Figure 4.9 has four stages in total. Within one stage, all transactions can be executed in parallel, as no dependencies exist between those transactions.

The actual parallel execution on the underlying database system is realized using $k$ client connections to the DBMS. To execute the transactions within an execution stage in parallel, the $k$ clients pick transactions from the stage and submit them to the underlying system. As our method is conflict free, it guarantees the generation of serializable schedules.

Therefore we can basically switch off concurrency control on the underlying system. This can be done by setting the isolation level of the underlying system to READ UNCOMMITTED[1] [20] to get the best performance out of the DBMS.

## 4.7 System Architecture

With a good knowledge of the logical design of the Whatever-LedgerConsensus model, let us now map this logical model to a concrete implementation of ChainifyDB. We implemented the ChainifyDB majority in Golang, with some additional bits of C++ in just under $11,000$ LOC. Different components of ChainifyDB, both within and across the organization, interact using remote procedure calls (RPC). Moreover, ChainifyDB uses the elliptic package of Golang to sign every message that flows in the network to preserve integrity.

ChainifyDB consists of three loosely coupled components: ChainifyServer, ExecutionServer, and the CommitServer. A single organization of a ChainifyDB network is entitled to run as many instances of these components. The components can be scaled in order to sustain the incoming workload or can be entirely left out if the organization does not require its functionality. Let us now briefly explore the function of each distinct component of the ChainifyDB.

**ChainifyServer:** It is an entry point of a transaction in the ChainifyDB network. It is responsible for authenticating the client, validating the integrity of the Proposal, assessing the agreement policy for the Proposal, and creating a so-called ChainedTransaction which wraps the Proposal and other transaction-specific metadata. We will learn more about ChainedTransaction in the upcoming running example in Section 4.7.1.

[1]Note however, that typical MVCC-implementations do not provide this level, e.g. in PostgreSQL the weakest isolation level possible is READ COMMITTED.

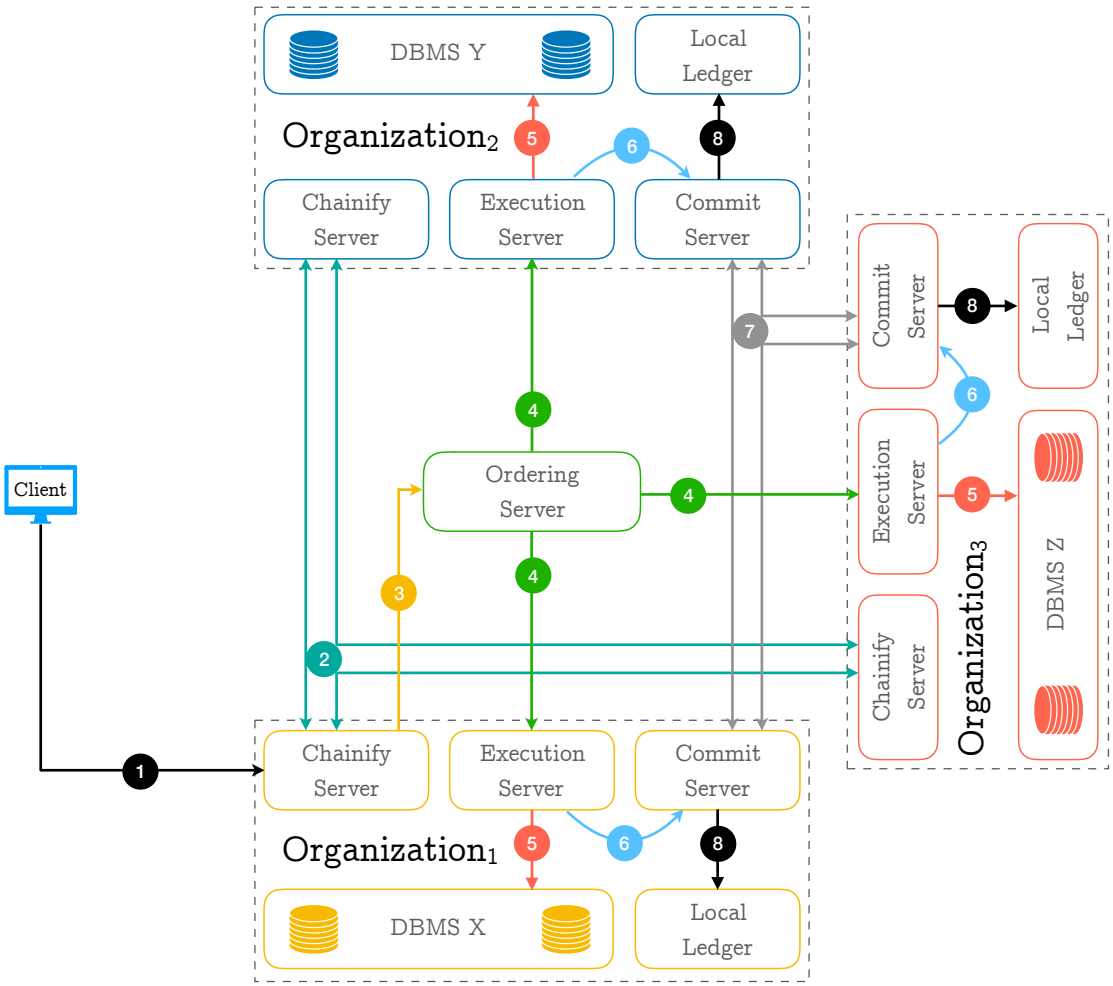



*Figure 4.10: Architecture of a sample ChainifyDB network: A concrete instance of the Whatever-LedgerConsensus model (WLC).*

**OrderingServer.** The ChainifyServer transmits the ChainedTransaction to the OrderingServer. Then the OrderingServer batches the received ChainedTransactions and forwards them to every ExecutionServer in the network. Please note that the OrderingServer in ChainifyDB is entirely untrusted. If the OrderingServer is malicious in any way (sends a different block to the ExecutionServers), organizations will fail to reach the consensus for the misformed block and elect a new OrderingServer.

**ExecutionServer.** Once the ExecutionServer receives the block of ChainedTransactions from the OrderingServer, it computes an execution graph for the block using transaction dependency analysis described in Section 4.6.3. The ExecutionServer then exe-

cutes the graph with the maximum possible parallelism and computes the diff between the initial and the final state. It forwards the digest of the diff and the block to the local CommitServer in Step 6.

**CommitServer.** The CommitServer hashes the `LedgerBlockHash` of the previous block, the current block, and the diff together to compute the `LedgerBlockHash` for the current block. In other words, the CommitServer encodes the previous consistent state, the effect of how the current block changed the state, and the next state into a single `LedgerBlockHash`. The `LedgerBlockHash`, along with the block, is then sent to the CommitServer.

The CommitServer then runs a simple voting-based consensus algorithm to verify whether a majority of nodes in the network computed the same `LedgerBlockHash`. If the `LedgerBlockHash` satisfies the commit policy, the block appends to the ledger. If not, this node must recover.

```
struct Proposal {
  struct Payload {
    ... header;
    String sql_tx;
  }

  Payload payload;
  Byte[] signature;
}
```

*Listing 4.1: Proposal format.*

```
struct Agreement {
  struct Payload {
    ... header;
    Byte[] proposal_hash;
    Bool status;
  }

  Payload payload;
  Byte[] signature;
}
```

*Listing 4.2: Agreement format.*

```
struct ChainedTx {
  struct Payload {
    ... header;
    Proposal proposal;
    Agreement[] agreements;
  }

  Payload payload;
  Byte[] signature;
}
```

*Listing 4.3: ChainedTx format.*

```
struct Block {
  struct Payload {
    ...  header;
    ChainedTx[] txns;
  }

  Payload payload;
  Byte[] signature;
}
```

*Listing 4.4: Block format.*

```
struct CommitBlock {
  struct Payload {
    ...  header;
    Block block;
    Byte[] ledger_block_hash;
    Bool[] tx_commit_status;
  }

  Payload payload;
  Byte[] signature;
}
```

*Listing 4.5: CommitBlock format.*

### 4.7.1 Running Example

Let us now look at a running example of how the transaction flows in a sample ChainifyDB network shown in Figure 4.4. In Step 1, the client wraps a SQL transaction into a Proposal. The Proposal, as shown in Listing 4.1, holds the original transaction in SQL format along with some metadata information such as client_id encoded in the header. The client also signs the transaction and appends the valid signature into the Proposal. The client then sends the Proposal to the ChainifyServer of Organization 1.

The ChainifyServer of Organization 1 has access to the authentication policy of the client, the public key of the client, and the policy that defines the subset of organizations that must agree to this transaction. For simplicity, let us assume that the agreement policy for this example is ALL, meaning that every organization must agree to execute every transaction in the system. If the ChainifyServer of Organization 1 successfully authenticates the client, and the verification of the Proposal is successful, it forwards the Proposal to the Organizations 2 and 3 in Step 2. Every ChainifyServer in the network executes the agreement logic for the transaction and computes the Agreement as specified in Listing 4.2. It must incorporate the digest of the Proposal into the Agreement to guarantee a one to one mapping between the Proposal and the Agreement. We must make sure that a malicious ChainifyServer cannot pack an Agreement of some Proposal as an Agreement for some another Proposal. The ChainifyServers of Organizations 2 and 3, then send the Agreement back to the ChainifyServer of Organization 1.

In Step 3, the ChainifyServer of Organization 1 creates a ChainedTx (see Listing 4.3) using the original Proposal and the Agreements, and send it to the OrderingServer. The OrderingServer queues this ChainedTx into a batch.

When the batch has adequate size, or enough time has elapsed, the OrderingServer produces a block (as shown in Listing 4.4) from this batch. In Step 4, the dispatch service of OrderingServer forwards the block to every ExecutionServer in the network.

In Step 5, the ExecutionServer of each organization computes the near-optimal execution graph and executes the transactions in the block using maximum possible parallelism. After execution, it computes the `LedgerBlockHash` and forwards the block and the `LedgerBlockHash` to the local CommitServer in Step 6.

In Step 7, the CommitServer of each organization acquires the votes for consensus from other organizations. If the `LedgerBlockHash` is same for the majority of the organizations, the CommitServer generates the CommitBlock (See Listing 4.5) and appends it to the local ledger in Step 8. If in case the consensus `LedgerBlockHash` is different to the local `LedgerBlockHash`, a recovery process is instantiated. If consensus fails globally, a new OrderingServer must be elected.

In summary, the ChainifyServer and the ExecutionServer cover the W-phase, whereas the CommitServer falls under the LC-phase of the Whatever-Ledger Consensus model. Note that if we purely follow the WLC model, we can exclude the transactions of the block from the consensus round and the ledger, since technically, we do not care about *how* an organization reaches a certain state. However, we still keep the information about the transactions per block to enable a recovery phase.

## 4.8 Experimental Evaluation

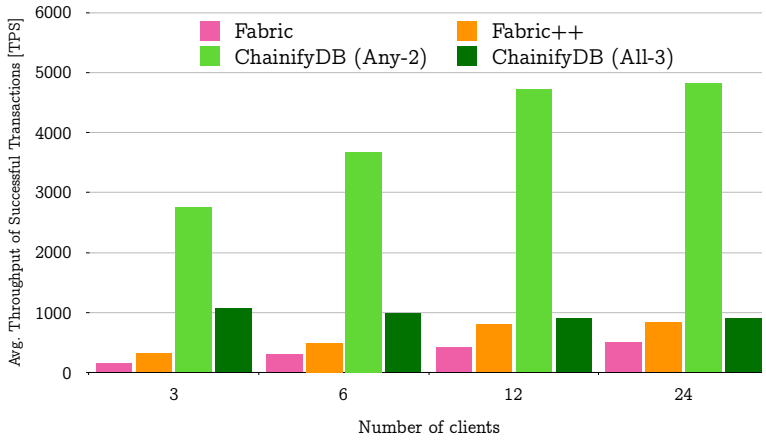
To evaluate ChainifyDB we use the following system setup.

### 4.8.1 Setup and Workload

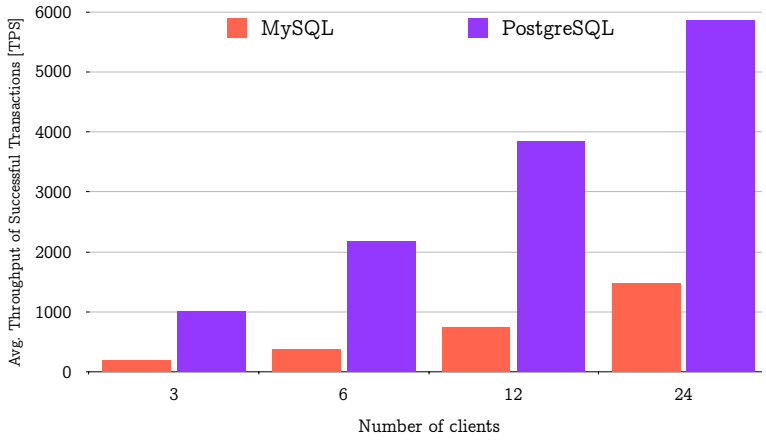
**Type 1 (small)**: Two quad-core Intel Xeon CPU E5-2407 running at 2.2 GHz, equipped with 48GB of DDR3 RAM.

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

Unless stated otherwise, we use a heterogeneous network consisting of three independent organizations $O_1$, $O_2$, and $O_3$. Organization $O_1$ owns two machines of type 1, where PostgreSQL 11.2 is running on one of these machines. Organization $O_2$ owns two machines of type 1 as well, but MySQL 8.0.18 is running on one of them. Finally, orga-




(a) Throughput of ChainifyDB with *Any-2* and *All-3* policy for varying number of clients. Additionally, we evaluate Fabric [18] and Fabric++ [80]. We use the Smallbank workload following a Zipf distribution.




(b) Throughput of standalone MySQL and PostgreSQL for varying number of clients. The same workload as in Figure 4.11(a) is fired using OLTP-bench. Note that OLTP-bench follows a uniform distribution.

*Figure 4.11: Throughput of successful transactions for the heterogeneous setup as described in Section 4.8.1.*

nization $O_3$ owns two machines of type 2, where again PostgreSQL is running on one of the machines. The individual components of ChainifyDB, as described in Section 4.7, are automatically distributed across the two machines of each organization. Additionally, there is a dedicated machine of type 2 that represents the client firing transactions to ChainifyDB as well as a type 2 machine that solely runs the ordering service.

As consensus policy, we configure two different options: In the first option *Any-2* we set $c = 2$ such that at least two out of our three organizations have to produce the same effect to reach consensus. In the second option *All-3* we set $c = 3$ and consensus is reached only if all three organizations produce the same effect. In any case, all three organizations have to agree to every transaction. Besides, empirical evaluation revealed a block size of $4096$ transactions to be a good fit (see Section 4.8.5). Of course, we also activate parallel transaction execution as described in Section 4.6.3.

As workload we use transactions from Smallbank [11], which simulate a typical asset transfer scenario. To bootstrap the test, we create for $100{,}000$ users a checking account and a savings account each and initialize them with random balances. The workload consists of the following four transactions: *TransactSavings* and *DepositChecking* increase the savings account and the checking account by a certain amount. *SendPayment* transfers money between two given checking accounts. *WriteCheck* decreases a checking account by a certain amount. During a single run, we repeatedly fire these four transactions at a fire rate of $4096$ transactions per client, where we uniformly pick one of the transactions in a random fashion. For each picked transaction, we determine the accounts to access based on a Zipfian distribution with a $s$-value of $1.1$ and a $v$-value of $1$, unless stated otherwise.

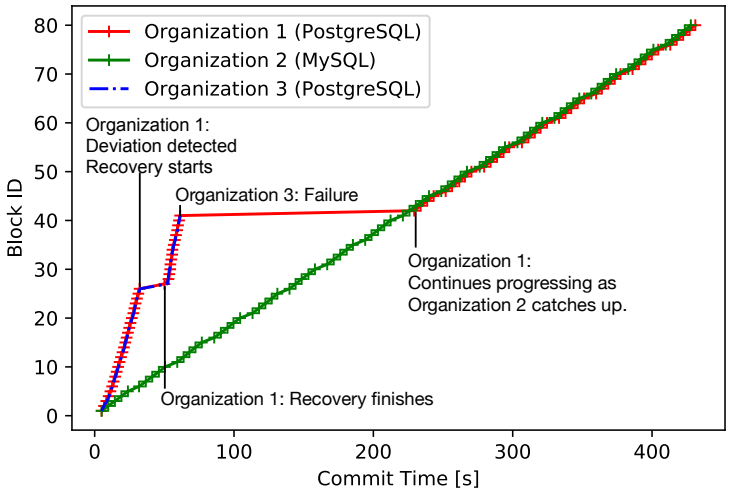
### 4.8.2 Throughput

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

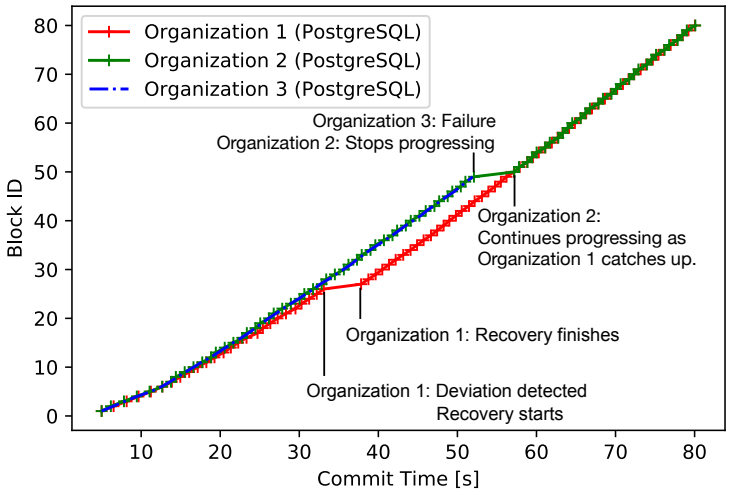
Therefore, we first inspect the throughput of ChainifyDB in our heterogeneous setup under our two different consensus policies Any-2 and All-3. Additionally to ChainifyDB, we show the following two PBS baselines: (a) Vanilla Fabric [18] v1.2, probably the most prominent PBS system currently. (b) Fabric++ [80], an improved version of Fabric v1.2. Both Fabric and Fabric++ are also distributed across the same set of machines and the blocksize is set to $1024$.

Figure 4.11(a) shows the results. On the $x$-axis, we vary the number of clients firing transactions concurrently from $3$ clients to $24$ clients. On the $y$-axis, we show the average throughput of successful transactions, excluding a ramp-up phase of the first five seconds. We can observe that ChainifyDB using the Any-2 strategy shows a significantly higher throughput than Fabric++ with up to almost $5000$ transactions per second. In comparison, Fabric++ achieves only around $1000$ transactions per second, although it makes considerably more assumptions than ChainifyDB: First, it assumes the order-

ing service to be trustworthy. Second, it assumes the execution to be deterministic and therefore does not perform any consensus round on the state.




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




(b) Homogeneous Setup (3x PostgreSQL).

*Figure 4.12: Robustness and recovery of ChainifyDB under the Any-2 consensus policy.*

Regarding ChainifyDB, we can also observe that there is a large performance gap between the Any-2 and the All-3 strategy. The reason for this lies in the heterogeneous setup we use. The two organizations running PostgreSQL are able to process the workload significantly faster than the single organization running MySQL. Thus, under the Any-2 strategy, the two organizations using PostgreSQL are able to lead the progress,

without having to wait for the significantly slower third organization. Under the All-3 strategy, the progress is throttled by the slowest organizations running MySQL.

The difference in processing speed also becomes visible, if we inspect the throughput of the stand-alone single-instance database systems in Figure 4.11(b) under the same workload. This time, we fire the transactions using OLTP-bench [27]. Note that both system are configured with a buffer size of 2GB to keep the working set in main memory. As we can see, PostgreSQL significantly outperforms MySQL under this workload independent of the number of clients.

There is one more observation we can make: By comparing Figure 4.11(a) and Figure 4.11(b) side-by-side, we can see that ChainifyDB introduces only negligible overhead over the raw database systems. In fact, for 3, 6, and 12 clients, ChainifyDB under the Any-2 policy actually produces a slightly higher throughput than raw PostgreSQL. The reasons for this lies in our optimized parallel transaction execution, which exploits the batch-wise inflow of transactions, and executes the transaction at the lowest possible isolation level.

For completeness, we also show in Table 4.2 the throughput for Smallbank, where the accounts are picked following a uniform distribution. As we can see, the throughput under a uniform distribution is even higher with up to 6144 transactions per second than under the skewed Zipf distribution, as it allows for a higher degree of parallelism during execution due to less conflicts between transactions.

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

*Table 4.2: Average throughput of successful transactions for ChainifyDB (Any-2) under Smallbank following a Zipf distribution and a uniform distribution.*

### 4.8.3 Robustness and Recovery

Apart from the transaction processing performance, the robustness and recovery capabilities are crucial properties of ChainifyDB as well. To put these capabilities to the test, in the following experiment, we will disturb our ChainifyDB network in two different ways: First, we forcefully corrupt the database of one organization and see whether ChainifyDB is able to detect and recover from it. Afterwards, we bring down one organization entirely and see whether the network is able to continue progressing. Of course, we are also interested in the performance of the recovery processes.

Precisely, we have the following setup for this experiment: In the first phase, we sustain our ChainifyDB network with transactions of the Smallbank workload. These do not cause the organizations to deviate in any way. Consequently, this phase essentially resembles the standard processing situation of ChainifyDB. Then, after a certain amount of time, we manually inject an update to the table of organization $O_1$ and see how fast $O_1$ is able to recover from the deviation. Note that we do not perform this update through a ChainifyDB transaction, but externally by directly modifying the table in the database. Finally, we simulate a complete failure of one organization by removing it from the network. The remaining two organizations then have to reach consensus to be able to progress under the Any-2 policy.

In Figure 4.12(a), we visualize the progress of all organizations for our typical heterogeneous setup. Additionally, in Figure 4.12(b), we test a homogeneous setup, where all three organizations run PostgreSQL. On the $x$-axis, we plot the time of commit for each block. On the $y$-axis, we plot the corresponding block IDs. Every five committed blocks, each organizations creates a local checkpoint.

Let us start with our typical heterogeneous setup in Figure 4.12(a). First of all, we can observe that the organizations $O_1$ and $O_3$, which run PostgreSQL, progress much faster than organization $O_2$ running MySQL. Shortly after the update has been applied to $O_1$, it detects the deviation in the consensus round and starts recovery from the most recent checkpoint. Interestingly, this also stops the progression of organization $O_3$, as $O_3$ is not able to reach consensus anymore according to the Any-2 policy: $O_1$ is busy with recovery and $O_2$ is too far behind. As soon as $O_1$ recovers, which takes around $17$ seconds, $O_3$ also restarts progressing, as consensus can be reached again. Both $O_1$ and $O_3$ progress until we let $O_3$ fail. Now, $O_1$ can not progress anymore, as $O_3$ is not reachable and $O_2$ still too far behind due its slow database system running underneath. Thus, $O_1$ halts until $O_2$ has caught up. As soon as this is the case, both $O_1$ and $O_2$ continue progressing at the speed of the slower organization, namely $O_2$.

In Figure 4.12(b), we retry this experiment on a homogeneous setup, where all organization run PostgreSQL. Thus, this time there is no drastically slower organization throttling the network. Again, at a certain point in time, we externally corrupt the database of organization $O_1$ by performing an update and $O_1$ starts to recover from the most recent checkpoint. In contrast to the previous experiment, this time the recovery of $O_1$ does not negatively influence any other organization: $O_2$ and $O_3$ can still reach consensus under the Any-2 policy and continue progressing, as none of the two is behind the other one. Recovery itself takes only around $4$ seconds this time and in this case, another organization is ready to perform a consensus round right after recovery. When organization $O_3$ fails, $O_2$ has to halt processing for a short amount of time, as organization $O_1$ has to catch up.

In summary, these experiments show (a) that we can detect state deviation and recover from it, (b) that the network can progress in the presence of organization failures, (c) that all organizations respect the consensus policy at all times, and (d) that recover neither penalizes the individual organizations nor the entire network significantly.

### 4.8.4 Cost Breakdown

In Section 4.8.2, we have seen the end-to-end performance of ChainifyDB. In the following, we want to analyze how much individual components contribute to this performance.

Precisely, we want to investigate: (a) The cost of all cryptographic computation, such as signing and validation, that is happening at several stages in the pipeline (see Section 4.7 for details). (b) The impact of parallel transaction execution on the underlying database system (see Section 4.6.3 for details).

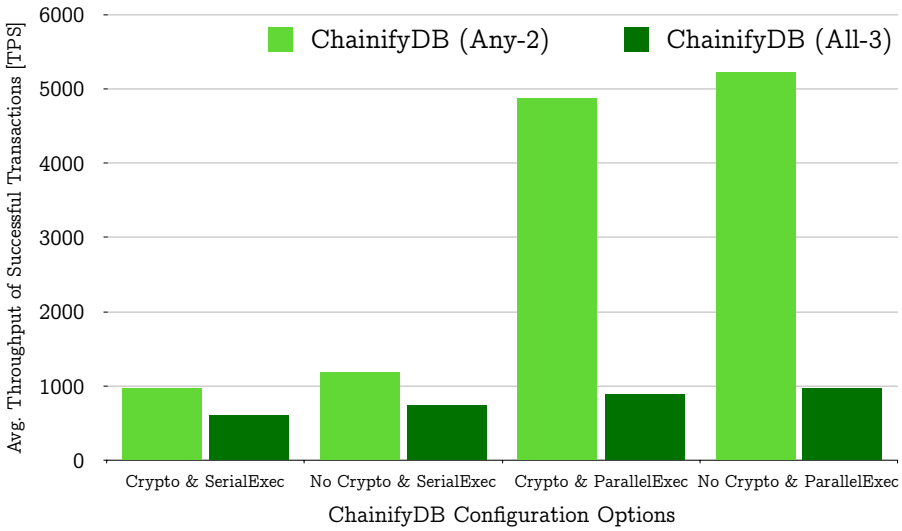



*Figure 4.13: Cost breakdown of ChainifyDB.*

Figure 4.13 shows the results. We can observe that the overhead caused by cryptographic computation is surprisingly small. Under the Any-2 policy, turning on all cryptographic components decreases the throughput only by $7\%$ for parallel execution. Under the All-3 policy, the decrease is only $8.5\%$. While our cryptographic components have little negative effects, our parallel transaction execution obviously has a very positive one. With activated cryptography, parallel transaction execution improves the throughput by up to $5$x.

### 4.8.5 Varying Blocksize

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

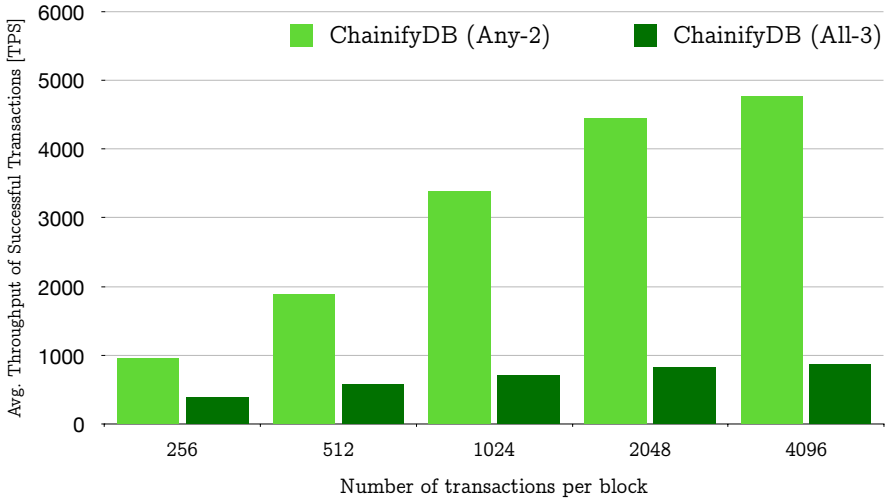



*Figure 4.14: The effect of varying the blocksize.*

Figure 4.14 shows the results. We can see that both under the Any-2 and All-3 policy, the throughput increases with the blocksize. This increase is mainly caused by our parallel transaction execution mechanism, which analyzes the whole block of transactions and schedules them for parallel conflict-free execution.

## 4.9 Conclusion

In this work, we introduced a highly flexible processing model for permissioned blockchain systems called the Whatever-LedgerConsensus model. WLC avoids making assumptions on the deterministic behavior of individual organizations by reaching consensus on the effects instead of the actions. We clearly formalized WLC and discussed in detail the recovery options in that landscape. To showcase the strengths of WLC, we proposed ChainifyDB, an implementation of a blockchain layer, which is able to chainify arbitrary data management systems and connect them in a network. In an extensive experimental evaluation, we showed that ChainifyDB does not only offer a 6x higher throughput than comparable baselines, but also introduces a robust recovery mechanism, which grant organizations the chance to rehabilitate.

# Chapter 5

# Conclusion

Data management requirements have grown immensely in the last decade to support evolving business requirements. And with the growing diversity of workloads, enterprises are faced with the necessity to add new systems to their technology stack, adding to the complexity and cost of their business. If firms keep selecting a new data management system for every new problem, soon they will have a technology stack that is infeasible both in terms of maintenance and cost. This problem demands the necessity of a unified system that can handle at least some distinct data management workload efficiently. In this work, we argued that a single system can efficiently serve both the transactional and the analytical workloads. We also propose a platform to transform an existing database infrastructure in a non-invasive manner to support transactions with a blockchain-level guarantee without adding an all-new system into the technology stack.

In the first part of this thesis, we investigated and introduced different solutions to unify transactional and analytical workloads. We recognized only one way to achieve this without hampering the performance of the system, which is by running these workloads in isolation. And this degree of separation requires an efficient snapshotting mechanism that helps the system to isolate these workloads efficiently. We proposed two distinct ways to create a snapshot of a virtual memory area. The first user-space technique exploits the concept of main-memory files using rewiring to create a copy of a virtual memory area. We utilized signal handlers provided by the Linux kernel to manually perform the copy-on-write operation and keep the copies isolated. We observed that the manual copy-on-write is expensive, and the cost of creating the snapshot using rewiring is highly dependent on the fragmentation of the virtual memory area. To overcome the snapshot management cost, we propose an alternative snapshotting system call implemented inside the Linux kernel. The system call allows us to handover the snapshot maintenance and copy-on-write mechanism to the operating system. So, the user uses

the snapshotting system call like a simple `memcpy()`, and everything else is hidden underneath by the OS. While our system call has a similar performance in terms of GB/s to `fork()`, it gives us the flexibility to create snapshots at a granularity of a virtual page.

To show the impact of fast snapshotting, we developed a prototype main-memory database system that implements a variant of the multi-version concurrency control. We evaluated our database system using different snapshotting strategies to show the impact of efficient snapshotting on the execution of the hybrid workload. We observed that a simple scan can be as much as 6x slower while running it simultaneous to the transactional workload. For the evaluated workload, even while creating a fresh snapshot for every incoming analytical query, we achieved up to 4x lower latency for OLAP queries even in the presence of concurrency transaction processing. We also showed that using our snapshotting technique, we can execute an analytical query under full-serializability as fast as if the isolation level is read-uncommitted, giving much stronger guarantees for the transactional system. Hence, achieving the goal of integrating hybrid workload processing into a single system without any significant impact on the performance of the system.

Another category of data management system that has recently gained immense popularity is the blockchain system. While these systems bring trust into an untrusted environment, making the state and history tamper-proof, they are way off in terms of performance in comparison to the relational database systems. And above all, they are mostly an all-new system in our technology stack. In Chapter 3, we investigated the transaction processing pipeline of a state-of-the-art permissioned-blockchain system to show its weaknesses and pinpoint the components of the transaction-pipeline that can learn a great deal from the mature database research. We extended Fabric with database technology to show the impact of using relational database research on the performance of the system under contended workloads, improving the throughput by up to 12x when compared to the vanilla Fabric. On our journey through this project, we observed the naiveness of the design of these systems and realized that bringing the permissioned-blockchain system on the same page to the relational database systems in terms of performance requires a much deeper modification to these systems.

Lastly, we looked at an entirely new system architecture for blockchain systems: the WhateverLedger Consensus model. It aims to drop some critical assumptions that make the permissioned-blockchain systems weak. Instead of running consensus on the order of the transactions, like other blockchain systems, our proposed architecture delays the consensus until the very end. This model allows us to ignore what was done by the nodes, and accept the transactions if all nodes are in the same state after execution. We also proposed ChainifyDB, an instance of a WL-C model that can transform a set of database nodes into a blockchain system without adding much overhead in terms

of performance. Our evaluation showed that we achieve up to 6x higher throughput in comparison to our state-of-the-art baselines, and our concurrency control mechanism implemented outside the database system can outperform the concurrency control protocol implemented by the evaluated database systems. ChainifyDB gives us high throughput, stronger blockchain-level guarantees, and much more without adding any new system to the technology stack. Thus businesses can keep all their data in one place inside the database system and still get all the functionalities of a blockchain system.

In summary, we analyzed transaction processing pipelines of main-memory database systems and the permissioned-blockchain systems, proposed innovative optimizations, carefully examined the impact of snapshots on the performance of these systems, and presented a platform to transform an existing database infrastructure into a blockchain infrastructure.

## 5.1 Future Work

Proper research is all about opening new chapters rather than closing them. So, let us discuss some possible extensions of the contributions made by this thesis.

**Snapshot Manager.** Now that we can create snapshots at a much faster rate, we get into the problem of managing the created virtual-snapshots. Also, we should keep in mind that the snapshotting cost is still dependent on the size of the snapshot itself. So, as soon as the size of the virtual-snapshot is too large, we might have to think about alternatives. It might also be infeasible to snapshot for every individual analytical query. Here comes the task of the snapshot manager. We can develop the snapshot manager to make some decisions based on the workload, like when to snapshot? Or what to snapshot? It may decide to snapshot the optimal subset of columns or even a horizontal slice of the table, which might be sufficient for executing the incoming analytical query. We might also utilize machine learning or even deep learning to learn the best-snapshotting strategy. This strategy may try to optimize the model for the cost of snapshots, the SLA concerning the real-time analytics guarantee of the system, or even the overall memory consumption of the system. We have to be careful about the memory because as we keep more and more prepared virtual-snapshots, we increase the total memory consumption of the system.

**Transactions, analytics, and blockchain in a single system.** This thesis presents two distinct systems AnKerDB and ChainifyDB, which provide us with two very different

features. One of the next tasks can be to combine the two projects to create one system with OLTP, OLAP, and the blockchain features into a single system. Enabling OLAP on blockchain systems brings a lot of exciting challenges concerning the guarantees of the blockchain system [87]. Having a fast snapshotting mechanism can also help us in efficiently identifying state modifications, presenting a significant optimization for the ChainifyDB.

**Optimizing ChainifyDB's external concurrency control.** As shown already in Chapter 4, our external concurrency control implementation outperforms the inbuilt concurrency control of PostgreSQL and MySQL. However, it is not perfect. It is tough to predict the conflicting nature of two transactions that select on two completely different attributes, adding an edge in their dependency graph. Our next goal can be to move from pessimistic to optimistic execution of these two transactions and identify the conflicts after executing the transaction. The goal of this project is to execute transactions in a pessimistic way if we can identify the conflict and optimistically for the case where we cannot say anything about the nature of the conflict. We may also use machine learning here to learn the correlation between different attributes to filter out the optimistic execution of transactions that have a very high chance of conflict. Solving this problem has its challenges since, efficiently validating the transactions for conflict from outside of the database is not so trivial.

# Appendix A

# AnKer's System Call Implementation

We have attached the patchfile [17] for the Linux kernel v4.16 for interested readers. Please note that this code and technology is provided for academic use only. The snapshotting mechanism can be used commercially only after receiving a written permission form the authors.

```
diff -uNr linux.vanilla/arch/x86/anker/anker.c linux.anker/arch/x86/anker/anker.c
--- linux.vanilla/arch/x86/anker/anker.c        1970-01-01 01:00:00.000000000 +0100
+++ linux.anker/arch/x86/anker/anker.c  2018-05-20 18:42:33.698529492 +0200
@@ -0,0 +1,119 @@
+/**
+ * File              : arch/x86/anker/anker.c
+ * Author            : Ankur Sharma <ankur.sharma@uni-saarland.de>
+ * Date              : 09.07.2017
+ * Last Modified Date: 09.10.2017
+
+ * Copyright (c) 2018 Ankur Sharma <ankur.sharma@uni-saarland.de>
+ */
+
+#include <linux/mm.h>
+#include <linux/mman.h>
+#include <linux/mm_types.h>
+#include <linux/syscalls.h>
+
+#include <asm/tlbflush.h>
+#include <asm/syscalls.h>
+
+#include "../mm/internal.h"
+
+#define PAGE_ALIGNMENT ((1 << PAGE_SHIFT) - 1)
+
+/*
+ * SYSCALL_NR 333
+ * anker: vm_snapshot(void *, unsigned long);
+ */
+SYSCALL_DEFINE2(vm_snapshot, unsigned long, src_addr, unsigned long, len)
```

```
+{
+    struct mm_struct *mm = current->mm;
+       struct vm_area_struct *vma;
+       unsigned long ret = -EINVAL;
+       bool locked = false;
+       struct vm_userfaultfd_ctx uf = NULL_VM_UFFD_CTX;
+       unsigned long map_flags = 0;
+    unsigned long new_addr = 0, charged = 0;
+    unsigned long new_len = PAGE_ALIGN(len);
+       LIST_HEAD(uf_unmap_early);
+       LIST_HEAD(uf_unmap);
+
+       if (offset_in_page(src_addr))
+               return ret;
+
+    /*
+        * We allow a zero old-len as a special case
+        * for DOS-emu "duplicate shm area" thing. But
+        * a zero new-len is nonsensical.
+        */
+       if (!new_len)
+               return ret;
+
+       if (down_write_killable(&mm->mmap_sem))
+               return -EINTR;
+
+       /*
+        * We weren't able to just expand or shrink the area,
+        * we need to create a new one and move it..
+        */
+       ret = -ENOMEM;
+
+    vma = find_vma(mm, src_addr);
+
+    if (!vma) goto out;
+
+       if (vma->vm_flags & VM_MAYSHARE)
+               map_flags |= MAP_SHARED;
+
+       new_addr = get_unmapped_area(vma->vm_file, 0, new_len,
+                                    vma->vm_pgoff +
+                                    ((src_addr - vma->vm_start) >> PAGE_SHIFT),
+                                    map_flags);
+
+    if (offset_in_page(new_addr))
+    {
+               ret = new_addr;
+               goto out;
+       }
+
+    for (; vma && charged < new_len; vma = vma->vm_next)
+    {
+        len = vma->vm_end - (src_addr + charged);
+        len = (new_len - charged < len) ? new_len - charged : len;
+
+        ret = anker_copy_vma(vma, src_addr + charged,
+                 new_addr + charged, len, &locked,
+                &uf, &uf_unmap);
+
+        /* unmap new copy if anything failed */
+        if (ret == -ENOMEM) {
+            up_write(&current->mm->mmap_sem);
```

```
+             vm_munmap(new_addr, new_addr + charged);
+             return ret;
+         }
+
+         charged += len;
+     }
+
+out:
+        up_write(&current->mm->mmap_sem);
+        if (locked)
+                mm_populate(new_addr, new_len);
+        return ret;
+}
diff -uNr linux.vanilla/arch/x86/anker/Makefile linux.anker/arch/x86/anker/Makefile
--- linux.vanilla/arch/x86/anker/Makefile        1970-01-01 01:00:00.000000000 +0100
+++ linux.anker/arch/x86/anker/Makefile 2018-05-20 18:42:33.698529492 +0200
@@ -0,0 +1 @@
+obj-$(CONFIG_ANKER_VMSNAPSHOT) := anker.o
diff -uNr linux.vanilla/arch/x86/entry/syscalls/syscall_64.tbl linux.anker/arch/x86/
    entry/syscalls/syscall_64.tbl
--- linux.vanilla/arch/x86/entry/syscalls/syscall_64.tbl         2018-05-20
    18:18:22.315552266 +0200
+++ linux.anker/arch/x86/entry/syscalls/syscall_64.tbl  2018-05-20 18:42:33.708529325
    +0200
@@ -339,6 +339,8 @@
330     common  pkey_alloc              sys_pkey_alloc
331     common  pkey_free               sys_pkey_free
332     common  statx                   sys_statx
+333 64      vm_snapshot          sys_vm_snapshot
+

#
# x32-specific system call numbers start at 512 to avoid cache impact
diff -uNr linux.vanilla/arch/x86/Makefile linux.anker/arch/x86/Makefile
--- linux.vanilla/arch/x86/Makefile      2018-05-20 18:18:22.315552266 +0200
+++ linux.anker/arch/x86/Makefile        2018-05-20 18:42:33.698529492 +0200
@@ -271,7 +271,7 @@
libs-y  += arch/x86/lib/

# See arch/x86/Kbuild for content of core part of the kernel
-core-y += arch/x86/
+core-y += arch/x86/ arch/x86/anker/

# drivers-y are linked after core-y
drivers-$(CONFIG_MATH_EMULATION) += arch/x86/math-emu/
diff -uNr linux.vanilla/include/linux/huge_mm.h linux.anker/include/linux/huge_mm.h
--- linux.vanilla/include/linux/huge_mm.h        2018-05-20 18:18:22.315552266 +0200
+++ linux.anker/include/linux/huge_mm.h 2018-05-20 18:42:35.605164185 +0200
@@ -43,6 +43,9 @@
extern bool move_huge_pmd(struct vm_area_struct *vma, unsigned long old_addr,
unsigned long new_addr, unsigned long old_end,
pmd_t *old_pmd, pmd_t *new_pmd, bool *need_flush);
+extern bool anker_copy_huge_pmd(struct vm_area_struct *vma, struct vm_area_struct *
    new_vma,
+         unsigned long old_addr,unsigned long new_addr, unsigned long old_end,
+         pmd_t *old_pmd, pmd_t *new_pmd, bool *need_flush);
extern int change_huge_pmd(struct vm_area_struct *vma, pmd_t *pmd,
unsigned long addr, pgprot_t newprot,
int prot_numa);
diff -uNr linux.vanilla/include/linux/mm.h linux.anker/include/linux/mm.h
--- linux.vanilla/include/linux/mm.h     2018-05-20 18:18:22.318885538 +0200
+++ linux.anker/include/linux/mm.h       2018-05-20 18:42:35.625163849 +0200
```

```
@@ -1312,6 +1312,9 @@
unsigned long end, unsigned long floor, unsigned long ceiling);
int copy_page_range(struct mm_struct *dst, struct mm_struct *src,
struct vm_area_struct *vma);
+extern unsigned long anker_copy_vma(struct vm_area_struct *vma,
+               unsigned long old_addr, unsigned long new_addr, unsigned long len,
+               bool *locked, struct vm_userfaultfd_ctx *uf, struct list_head *uf_unmap
    );
int follow_pte_pmd(struct mm_struct *mm, unsigned long address,
unsigned long *start, unsigned long *end,
pte_t **ptepp, pmd_t **pmdpp, spinlock_t **ptlp);
diff -uNr linux.vanilla/include/linux/syscalls.h linux.anker/include/linux/syscalls.h
--- linux.vanilla/include/linux/syscalls.h      2018-05-20 18:18:22.318885538 +0200
+++ linux.anker/include/linux/syscalls.h        2018-05-20 18:42:35.658496625 +0200
@@ -940,5 +940,6 @@
asmlinkage long sys_pkey_free(int pkey);
asmlinkage long sys_statx(int dfd, const char __user *path, unsigned flags,
unsigned mask, struct statx __user *buffer);
-
+/* anker */
+asmlinkage long sys_vm_snapshot(unsigned long, unsigned long);
#endif
diff -uNr linux.vanilla/include/uapi/asm-generic/unistd.h linux.anker/include/uapi/asm-
    generic/unistd.h
--- linux.vanilla/include/uapi/asm-generic/unistd.h     2018-05-20 18:18:22.318885538
    +0200
+++ linux.anker/include/uapi/asm-generic/unistd.h       2018-05-20 18:42:35.698495953
    +0200
@@ -732,9 +732,10 @@
__SYSCALL(__NR_pkey_free,     sys_pkey_free)
#define __NR_statx 291
__SYSCALL(__NR_statx,     sys_statx)
-
+#define __NR_vm_snapshot 333
+__SYSCALL(__NR_vm_snapshot, sys_vm_snapshot)
#undef __NR_syscalls
-#define __NR_syscalls 292
+#define __NR_syscalls 293

/*
* All syscalls below here should go away really,
diff -uNr linux.vanilla/init/Kconfig linux.anker/init/Kconfig
--- linux.vanilla/init/Kconfig  2018-05-20 18:18:22.318885538 +0200
+++ linux.anker/init/Kconfig    2018-05-20 18:42:35.728495451 +0200
@@ -37,7 +37,6 @@
and put_task_stack() in save_thread_stack_tsk() and get_wchan().

menu "General setup"
-
config BROKEN
bool

@@ -97,8 +96,7 @@
release tree by looking for git tags that belong to the current
top of tree revision.

-         A string of the format -gxxxxxxxx will be added to the localversion
-         if a git-based tree is found.  The string generated by this will be
+         A string of the format -gxxxxxxxx will be added to the localversion if a git-
    based tree is found.  The string generated by this will be
appended after any matching localversion* files, and after the value
set in CONFIG_LOCALVERSION.
```

```
diff -uNr linux.vanilla/kernel/sys_ni.c linux.anker/kernel/sys_ni.c
--- linux.vanilla/kernel/sys_ni.c       2018-05-20 18:18:22.318885538 +0200
+++ linux.anker/kernel/sys_ni.c 2018-05-20 18:42:35.751828393 +0200
@@ -259,3 +259,6 @@
cond_syscall(sys_pkey_mprotect);
cond_syscall(sys_pkey_alloc);
cond_syscall(sys_pkey_free);
+
+/* anker: vm_snapshot ~ defined in mm/mremap.c */
+cond_syscall(sys_vm_snapshot);
diff -uNr linux.vanilla/mm/huge_memory.c linux.anker/mm/huge_memory.c
--- linux.vanilla/mm/huge_memory.c      2018-05-20 18:18:22.318885538 +0200
+++ linux.anker/mm/huge_memory.c        2018-05-20 18:42:35.775161334 +0200
@@ -7,7 +7,7 @@

#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

-#include <linux/mm.h>
+#include <linux/mm.h>
#include <linux/sched.h>
#include <linux/sched/coredump.h>
#include <linux/sched/numa_balancing.h>
@@ -1836,6 +1836,77 @@
return false;
}

+bool anker_copy_huge_pmd(struct vm_area_struct *vma, struct vm_area_struct *new_vma,
+        unsigned long old_addr, unsigned long new_addr, unsigned long old_end,
+        pmd_t *old_pmd, pmd_t *new_pmd, bool *need_flush)
+{
+       spinlock_t *old_ptl, *new_ptl;
+       pmd_t pmd;
+    struct page *src_page;
+       struct mm_struct *mm = vma->vm_mm;
+       bool force_flush = false;
+
+       if ((old_addr & ~HPAGE_PMD_MASK) ||
+           (new_addr & ~HPAGE_PMD_MASK) ||
+           old_end - old_addr < HPAGE_PMD_SIZE)
+               return false;
+
+       /*
+        * The destination pmd shouldn't be established, free_pgtables()
+        * should have release it.
+        */
+       if (WARN_ON(!pmd_none(*new_pmd))) {
+               VM_BUG_ON(pmd_trans_huge(*new_pmd));
+               return false;
+       }
+
+       /*
+        * We don't have to worry about the ordering of src and dst
+        * ptlocks because exclusive mmap_sem prevents deadlock.
+        */
+       old_ptl = __pmd_trans_huge_lock(old_pmd, vma);
+       if (old_ptl) {
+               new_ptl = pmd_lockptr(mm, new_pmd);
+               if (new_ptl != old_ptl)
+                       spin_lock_nested(new_ptl, SINGLE_DEPTH_NESTING);
+               pmd = *old_pmd;
+               if (pmd_present(pmd) && pmd_dirty(pmd))
```

```
+                       force_flush = true;
+               VM_BUG_ON(!pmd_none(*new_pmd));
+
+       if (is_cow_mapping(vma->vm_flags)) {
+           pmdp_set_wrprotect(mm, old_addr, old_pmd);
+           pmd = pmd_wrprotect(pmd);
+       }
+
+       if (vma->vm_flags & VM_SHARED)
+           pmd = pmd_mkclean(pmd);
+
+       pmd = pmd_mkold(pmd);
+
+       src_page = pmd_page(pmd);
+
+       if (src_page) {
+           get_page(src_page);
+           page_add_anon_rmap(src_page, new_vma, new_addr, true);
+       }
+
+       // add_mm_counter(mm, MM_ANONPAGES, HPAGE_PMD_NR);
+               set_pmd_at(mm, new_addr, new_pmd, pmd);
+
+       if (new_ptl != old_ptl)
+                       spin_unlock(new_ptl);
+               if (force_flush)
+                       flush_tlb_range(vma, old_addr, old_addr + PMD_SIZE);
+               else
+                       *need_flush = true;
+               spin_unlock(old_ptl);
+               return true;
+       }
+       return false;
+}
+
+
/*
* Returns
*  - 0 if PMD could not be locked
diff -uNr linux.vanilla/mm/Kconfig linux.anker/mm/Kconfig
--- linux.vanilla/mm/Kconfig    2018-05-20 18:18:22.318885538 +0200
+++ linux.anker/mm/Kconfig      2018-05-20 18:42:35.775161334 +0200
@@ -422,6 +422,13 @@
benefit.
endchoice

+config ANKER_VMSNAPSHOT
+    bool "vm_snapshot() support"
+    depends on TRANSPARENT_HUGEPAGE
+    default n
+    help
+      Use kernel based virtual memory copy support.
+
config ARCH_WANTS_THP_SWAP
def_bool n

diff -uNr linux.vanilla/mm/mmap.c linux.anker/mm/mmap.c
--- linux.vanilla/mm/mmap.c     2018-05-20 18:18:22.318885538 +0200
+++ linux.anker/mm/mmap.c       2018-05-20 18:42:35.781827889 +0200
@@ -3170,6 +3170,87 @@
}
```

```
diff -uNr linux.vanilla/mm/mremap.c linux.anker/mm/mremap.c
--- linux.vanilla/mm/mremap.c    2018-05-20 18:18:22.318885538 +0200
+++ linux.anker/mm/mremap.c      2018-05-20 18:42:35.781827889 +0200
@@ -643,3 +643,235 @@
userfaultfd_unmap_complete(mm, &uf_unmap);
return ret;
}
+
+/*
+ * AnKer: copy ptes to new virtual address.
+ */
+static void anker_copy_ptes(struct vm_area_struct *vma, pmd_t *old_pmd,
+               unsigned long old_addr, unsigned long old_end,
+               struct vm_area_struct *new_vma, pmd_t *new_pmd,
+               unsigned long new_addr, bool need_rmap_locks, bool *need_flush)
+{
+       struct mm_struct *mm = vma->vm_mm;
+       pte_t *old_pte, *new_pte, pte;
+       spinlock_t *old_ptl, *new_ptl;
+       bool force_flush = false;
+    struct page *page;
+       unsigned long len = old_end - old_addr;
+
+       if (need_rmap_locks)
+               take_rmap_locks(vma);
+
+       /*
+        * We don't have to worry about the ordering of src and dst
+        * pte locks because exclusive mmap_sem prevents deadlock.
+        */
+    old_pte = pte_offset_map_lock(mm, old_pmd, old_addr, &old_ptl);
+       new_pte = pte_offset_map(new_pmd, new_addr);
+       new_ptl = pte_lockptr(mm, new_pmd);
+
+    if (new_ptl != old_ptl)
+               spin_lock_nested(new_ptl, SINGLE_DEPTH_NESTING);
+
+    flush_tlb_batched_pending(vma->vm_mm);
+       arch_enter_lazy_mmu_mode();
+
+       for (; old_addr < old_end; old_pte++, old_addr += PAGE_SIZE,
+                                  new_pte++, new_addr += PAGE_SIZE) {
+               if (pte_none(*old_pte))
+                       continue;
+
+        pte = *old_pte;
+
+               /*
+                * If we are remapping a dirty PTE, make sure
+                * to flush TLB before we drop the PTL for the
+                * old PTE or we may race with page_mkclean().
+                *
+                * This check has to be done after we removed the
+                * old PTE from page tables or another thread may
+                * dirty it after the check and before the removal.
+                */
+
+        if (pte_present(pte) && pte_dirty(pte))
+                       force_flush = true;
+
+        /*
+         * If it's a COW mapping, write protect both parent and child
```

```
+         *
+         */
+        if (is_cow_mapping(vma->vm_flags)) {
+            ptep_set_wrprotect(mm, old_addr, old_pte);
+            pte = pte_wrprotect(pte);
+        }
+
+        /*
+         * If its a shared mapping, mark it clean in the child
+         */
+        if (vma->vm_flags & VM_SHARED)
+            pte = pte_mkclean(pte);
+
+        pte = pte_mkold(pte);
+
+        page = vm_normal_page(vma, old_addr, pte);
+
+        if (page) {
+            get_page(page);
+            inc_mm_counter(mm, mm_counter(page));
+                if (vma_is_anonymous(new_vma))
+                 page_add_anon_rmap(page, new_vma, new_addr, false);
+                else
+                        page_add_file_rmap(page, false);
+        }
+
+        // set the pte at new address
+        set_pte_at(mm, new_addr, new_pte, pte);
+    }
+
+       arch_leave_lazy_mmu_mode();
+
+       if (new_ptl != old_ptl)
+               spin_unlock(new_ptl);
+       pte_unmap(new_pte - 1);
+
+    if (force_flush)
+               flush_tlb_range(vma, old_end - len, old_end);
+       else
+               *need_flush = true;
+
+    pte_unmap_unlock(old_pte - 1, old_ptl);
+
+    if (need_rmap_locks)
+               drop_rmap_locks(vma);
+}
+
+/*
+ * AnKer: copy the page tables backing the given VMA
+ */
+unsigned long anker_copy_page_tables(struct vm_area_struct *vma,
+        struct vm_area_struct *new_vma, unsigned long old_addr,
+        unsigned long new_addr, unsigned long len, bool need_rmap_locks)
+{
+       unsigned long extent, next, old_end;
+       pmd_t *old_pmd, *new_pmd;
+       bool need_flush = false;
+       unsigned long mmun_start;       /* For mmu_notifiers */
+       unsigned long mmun_end;         /* For mmu_notifiers */
+
+       old_end = old_addr + len;
+       flush_cache_range(vma, old_addr, old_end);
```

```
+
+       mmun_start = old_addr;
+       mmun_end   = old_end;
+       mmu_notifier_invalidate_range_start(vma->vm_mm, mmun_start, mmun_end);
+
+       for (; old_addr < old_end; old_addr += extent, new_addr += extent) {
+        cond_resched();
+
+        next = (old_addr + PMD_SIZE) & PMD_MASK;
+
+        /* even if next overflowed, extent below will be ok */
+               extent = next - old_addr;
+
+        if (extent > old_end - old_addr)
+                       extent = old_end - old_addr;
+
+        old_pmd = get_old_pmd(vma->vm_mm, old_addr);
+
+        if (!old_pmd)
+                       continue;
+
+        new_pmd = alloc_new_pmd(vma->vm_mm, vma, new_addr);
+
+        if (!new_pmd)
+                       break;
+
+        if (is_swap_pmd(*old_pmd) || pmd_trans_huge(*old_pmd)) {
+                       if (extent == HPAGE_PMD_SIZE) {
+                               bool moved = false;
+                               /* See comment in move_ptes() */
+                               if (need_rmap_locks)
+                                       take_rmap_locks(vma);
+
+                moved = anker_copy_huge_pmd(vma, new_vma, old_addr, new_addr,
+                         old_end, old_pmd, new_pmd, &need_flush);
+
+                               if (need_rmap_locks)
+                                       drop_rmap_locks(vma);
+
+                if (moved)
+                                       continue;
+                       }
+
+            split_huge_pmd(vma, old_pmd, old_addr);
+
+            if (pmd_trans_unstable(old_pmd))
+                               continue;
+               }
+
+        if (pte_alloc(new_vma->vm_mm, new_pmd, new_addr))
+                       break;
+
+        next = (new_addr + PMD_SIZE) & PMD_MASK;
+
+        if (extent > next - new_addr)
+                       extent = next - new_addr;
+
+        if (extent > LATENCY_LIMIT)
+                       extent = LATENCY_LIMIT;
+
+        anker_copy_ptes(vma, old_pmd, old_addr, old_addr + extent, new_vma,
+                new_pmd, new_addr, need_rmap_locks, &need_flush);
```

```
+       }
+
+       if (need_flush)
+               flush_tlb_range(vma, old_end-len, old_addr);
+
+       mmu_notifier_invalidate_range_end(vma->vm_mm, mmun_start, mmun_end);
+
+       return len + old_addr - old_end;        /* how much done */
+}
+
+/*
+ * AnKer: copy all VMAs backing the given virtual memory area.
+ * */
+unsigned long anker_copy_vma(struct vm_area_struct *vma,
+               unsigned long old_addr, unsigned long new_addr, unsigned long len,
+               bool *locked, struct vm_userfaultfd_ctx *uf, struct list_head *uf_unmap
    )
+{
+       struct mm_struct *mm = vma->vm_mm;
+       struct vm_area_struct *new_vma;
+       unsigned long vm_flags = vma->vm_flags;
+       unsigned long new_pgoff;
+       unsigned long moved_len;
+       bool need_rmap_locks;
+
+    /*
+        * We'd prefer to avoid failure later on in do_munmap:
+        * which may split one vma into three before unmapping.
+        */
+       if (mm->map_count >= sysctl_max_map_count - 3)
+               return -ENOMEM;
+
+       new_pgoff = vma->vm_pgoff + ((old_addr - vma->vm_start) >> PAGE_SHIFT);
+       new_vma = copy_vma(&vma, new_addr, len, new_pgoff, &need_rmap_locks);
+
+       if (!new_vma)
+               return -ENOMEM;
+
+       moved_len = anker_copy_page_tables(vma, new_vma, old_addr, new_addr,
+            len, need_rmap_locks);
+
+    if (moved_len < len) {
+               return -ENOMEM;
+       }
+
+       vm_stat_account(mm, vma->vm_flags, len >> PAGE_SHIFT);
+
+       if (vm_flags & VM_LOCKED) {
+               mm->locked_vm += len >> PAGE_SHIFT;
+               *locked = true;
+       }
+
+       return new_addr;
+}
diff -uNr linux.vanilla/README linux.anker/README
--- linux.vanilla/README        2018-05-20 18:18:22.315552266 +0200
+++ linux.anker/README  1970-01-01 01:00:00.000000000 +0100
@@ -1,18 +0,0 @@
-Linux kernel
-============
-
-This file was moved to Documentation/admin-guide/README.rst
```

```
-
-Please notice that there are several guides for kernel developers and users.
-These guides can be rendered in a number of formats, like HTML and PDF.
-
-In order to build the documentation, use ``make htmldocs`` or
-``make pdfdocs``.
-
-There are various text files in the Documentation/ subdirectory,
-several of them using the Restructured Text markup notation.
-See Documentation/00-INDEX for a list of what is contained in each file.
-
-Please read the Documentation/process/changes.rst file, as it contains the
-requirements for building and running the kernel, and information about
-the problems which may result by upgrading your kernel.
diff -uNr linux.vanilla/README.md linux.anker/README.md
--- linux.vanilla/README.md      1970-01-01 01:00:00.000000000 +0100
+++ linux.anker/README.md        2018-05-20 18:42:33.178538211 +0200
@@ -0,0 +1,50 @@
+Linux kernel
+============
+Actual linux documentation is available in `Documentation/admin-guide/README.rst`
+Commit hash of first commit 48c0dcc537 (needed to create a patch).
+
+#### Build requirements for ubuntu 18.04
+`sudo apt-get install git build-essential kernel-package fakeroot libncurses5-dev
    libssl-dev ccache bison flex`
+
+#### syscall specification:
+```c
+SYSCALL_NR 333
+
+void* vm_snapshot(void* src, unsigned long length);
+
+// use syscall defined in unistd.h to invoke the system call or,
+// use the header file (anker.h) defined in rapido/tests/include/
+
+void *copy = syscall(333, src, length);
+```
+#### Configure kernel with VMSNAPSHOT support
+1. Use `make menuconfig` to generate a `.config` file.
+2. Enable transparent hugepages by editing `.config` file.
+  * Set `CONFIG_TRANSPARENT_HUGEPAGE=y` and `CONFIG_TRANSPARENT_HUGEPAGE_MADVISE=y`.
+  * These options are necessary to run the tests specified under rapido/tests/src/
+3. Enable `vm_snapshot()` by setting `CONFIG_ANKER_VMSNAPSHOT=y`.
+4. Currently `TRANSPARENT_HUGEPAGE` must be enabled to use `vm_snapshot()`.
+
+#### Testing the kernel
+This kernel was developed using the testbed backed by `Qemu`. I have used a trimmed
    down version of [Rapido](https://github.com/rapido-linux/rapido) that provides a
    set of scripts to quicky generate `VM Image` and necessary modules using Dracut
    and boots the image using `Qemu`.
+
+#### Build instructions
+1. Generate relevant configure file as mentioned above.
+2. make -j
+3. INSTALL_MOD_PATH=./mods make modules_install
+
+#### Running tests
+1. `cd rapido/tests`
+2. `cmake -DCMAKE_CXX_FLAGS=-O2 .`
+3. `make -j4`
+4. `cd ..`
```

```
+5. `./cut_anker.sh`
+6. Boot the `VM` using `./vm.sh`.
+7. Tests are installed in path `/tests/` inside the VM.
+8. Use `shutdown` to powerdown the `VM`.
+
+#### Patching linux
+1. Create patch using `diff`.
+   * `diff -uNr linux.vanilla linux.new > patchfile`
+2. Apply patch using `patch`.
+   * `cd linux && patch -p1 < ../patchfile`
\ No newline at end of file
```

# List of Figures

# List of Tables

# Bibliography


[1] Bigchaindb: https://www.bigchaindb.com.

[2] Caliper: https://github.com/hyperledger/caliper.

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

[4] Guage github: https://github.com/persistentsystems/gauge/blob/master/docs/caliper-changes.md.

[5] https://github.com/jpmorganchase/quorum.

[6] Memsql: http://www.memsql.com.

[7] Multichain: https://www.multichain.com/download/multichain-white-paper.pdf.

[8] Mysql: http://www.mysql.com.

[9] NuoDB: http://www.nuodb.com.

[10] Peloton: http://www.pelotondb.org.

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

[12] Tpc-h: http://www.tpc.org/tpch/.

[13] Guage: https://github.com/persistentsystems/gauge, 2019.

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

[15] Tpc-c: http://www.tpc.org/tpcc/, 2019.

[16] Ycsb: https://github.com/brianfrankcooper/ycsb, 2019.

[17] Anker: https://github.com/sh-ankur/anker, 2020.

[18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. *CoRR*, abs/1801.10228, 2018.

[19] Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In *TRUST*, volume 9229 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2015.

[20] Philip A. Bernstein. SQL isolation levels. In *Encyclopedia of Database Systems (2nd ed.)*. Springer, 2018.

[21] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[22] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory MVCC systems. *PVLDB*, 13(2):128–141, 2019.

[23] Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310, page 4, 2016.

[24] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25*, pages 173–186, 1999.

[25] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *SIGMOD Conference*, pages 123–140. ACM, 2019.

[26] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254, 2013.

[27] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

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

[29] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, et al. Untangling blockchain: A data processing view of blockchain systems. *IEEE Trans. Knowl. Data Eng.*, 30(7):1366–1385, 2018.

[30] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. Indexing moving objects using short-lived throwaway indexes. In Nikos Mamoulis, Thomas Seidl, Torben Bach Pedersen, Kristian Torp, and Ira Assent, editors, *Advances in Spatial and Temporal Databases, 11th International Symposium, SSTD 2009, Aalborg, Denmark, July 8-10, 2009, Proceedings*, volume 5644 of *Lecture Notes in Computer Science*, pages 189–207. Springer, 2009.

[31] Jens Dittrich, Lukas Blunschi, and Marcos Antonio Vaz Salles. MOVIES: indexing moving objects by shooting index images. *GeoInformatica*, 15(4):727–767, 2011.

[32] Jens Dittrich and Alekh Jindal. Towards a one size fits all database architecture. In *CIDR*, pages 195–198. www.cidrdb.org, 2011.

[33] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. Blockchaindb - a shared database on blockchains. *PVLDB*, 12(11):1597–1608, 2019.

[34] Muhammad El-Hindi, Martin Heyden, Carsten Binnig, Ravi Ramamurthy, Arvind Arasu, and Donald Kossmann. Blockchaindb - towards a shared database on blockchains. In *SIGMOD Conference*, pages 1905–1908. ACM, 2019.

[35] Aaron J. Elmore, Jennie Duggan, Mike Stonebraker, Magdalena Balazinska, Ugur Çetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, Samuel Madden, David Maier, Timothy G. Mattson, Stavros Papadopoulos, Jeff Parkhurst, Nesime Tatbul, Manasi Vartak, and Stan Zdonik. A demonstration of the bigdawg polystore system. *PVLDB*, 8(12):1908–1911, 2015.

[36] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoining: A scalable blockchain protocol. In Katerina J. Argyraki and Rebecca Isaacs, editors, *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 45–59. USENIX Association, 2016.

[37] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5):613–624, 2017.

[38] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[39] Alan Fekete, Elizabeth J. O'Neil, and Patrick E. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Record*, 33(3):12–14, 2004.

[40] Zhenfeng Gao, Yushun Fan, Cheng Wu, Jia Zhang, and Chang Chen. DSES: A blockchain-powered decentralized service eco-system. In *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 25–32. IEEE Computer Society, 2018.

[41] Johannes Gehrke, Lindsay Allen, Panagiotis Antonopoulos, Arvind Arasu, Joachim Hammer, James Hunter, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Srinath T. V. Setty, Jakub Szymaszek, Alexander van Renen, Jonathan Lee, and Ramarathnam Venkatesan. Veritas: Shared verifiable databases and tables in the cloud. In *CIDR*. www.cidrdb.org, 2019.

[42] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Shareddb: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[43] Jana Giceva, Gerd Zellweger, Gustavo Alonso, and Timothy Rosco. Customized os support for data-processing. In *DaMon' 16*, pages 2:1–2:6, New York, NY, USA, 2016. ACM.

[44] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengießer, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. Towards scalable real-time analytics: An architecture for scale-out of olxp workloads. *PVLDB*, 8(12):1716–1727, 2015.

[45] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20, 000 transactions per second. In *IEEE ICBC*, pages 455–463. IEEE, 2019.

[46] Siyuan Han, Zihuan Xu, and Lei Chen. Jupiter: A blockchain platform for mobile devices. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1649–1652. IEEE Computer Society, 2018.

[47] Zhengyu He and Bo Hong. Impact of early abort mechanisms on lock-based software transactional memory. In *16th International Conference on High Performance Computing, HiPC 2009, December 16-19, 2009, Kochi, India, Proceedings*, pages 225–234, 2009.

[48] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *PVLDB*, 13(5):629–642, 2020.

[49] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.

[50] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE Computer Society, 2011.

[51] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1675–1687, 2016.

[52] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 839–858. IEEE Computer Society, 2016.

[53] Alan G. Labouseur and Carolyn C. Matheus. Dynamic data quality for static blockchains. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019*, pages 19–21. IEEE, 2019.

[54] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil MacNaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zaït. Oracle database in-memory: A dual format in-memory database. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1253–1258. IEEE Computer Society, 2015.

[55] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-time analytical processing with SQL server. *PVLDB*, 8(12):1740–1751, 2015.

[56] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[57] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM*

*International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 21–35, 2017.

[58] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael Watzke. Transaction reordering. *Data Knowl. Eng.*, 69(1):29–49, 2010.

[59] Sujaya Maiyya, Victor Zakhary, Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Database and distributed computing foundations of blockchains. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 2036–2041. ACM, 2019.

[60] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. Batchdb: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 37–50. ACM, 2017.

[61] Hagar Meir, Artem Barger, and Yacov Manevich. Increasing concurrency in hyperledger fabric. In *SYSTOR*, page 179. ACM, 2019.

[62] Qingzhong Meng, Xuan Zhou, Shiping Chen, and Shan Wang. Swingdb: An embedded in-memory DBMS enabling instant snapshot sharing. In *ADMS/IMDM Workshop 2016*, pages 134–149, 2016.

[63] C. Mohan. Blockchains and databases: A new era in distributed computing. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1739–1740. IEEE Computer Society, 2018.

[64] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[65] C. Mohan, Hamid Pirahesh, and Raymond A. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *SIGMOD 1992*, pages 124–133, 1992.

[66] Henrik Mühe, Alfons Kemper, and Thomas Neumann. How to efficiently snapshot transactional data: hardware or software controlled? In *DaMoN 2011, Athens, Greece*, pages 17–26, 2011.

[67] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Scyper: A hybrid oltp&olap distributed main memory database system for scalable real-time analytics. In Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen, editors, *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, volume P-214 of *LNI*, pages 499–502. GI, 2013.

[68] Muhammad Muzammal, Qiang Qu, and Bulat Nasrulin. Renovating blockchain with distributed databases: An open source system. *Future Generation Comp. Syst.*, 90:105–117, 2019.

[69] Satoshi Nakamoto. Bitcoin: https://bitcoin.org/bitcoin.pdf.

[70] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. *PVLDB*, 12(11):1539–1552, 2019.

[71] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD 2015*, pages 677–689, 2015.

[72] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid CPU-FPGA databases. In *FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017*, pages 211–218, 2017.

[73] Zhe Peng, Haotian Wu, Bin Xiao, and Songtao Guo. VQL: providing query efficiency and data authenticity in blockchain systems. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019*, pages 1–6. IEEE, 2019.

[74] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *PVLDB*, 5(12):1850–1861, 2012.

[75] Thamir M. Qadah and Mohammad Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*, pages 13–25, 2018.

[76] Yingyao Rong, Jingjing Zhang, Jing Bian, and Weigang Wu. ERBFT: efficient and robust byzantine fault tolerance. In *HPCC/SmartCity/DSS*, pages 265–272. IEEE, 2019.

[77] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. RUMA has it: Rewired user-space memory access is possible! *PVLDB*, 9(10):768–779, 2016.

[78] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, and Divya Agrawal. Chainifydb: How to blockchainify any data management system. *arXiv preprint arXiv:1912.04820*, 2019.

[79] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. How to databasify a blockchain: the case of hyperledger fabric. *arXiv preprint arXiv:1810.13177*, 2018.

[80] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *SIGMOD Conference*, pages 105–122. ACM, 2019.

[81] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting. *arXiv preprint arXiv:1709.04284*, 2017.

[82] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. Accelerating analytical processing in MVCC using fine-granular high-frequency virtual snapshotting. In *SIGMOD 2018, Houston, TX, USA, June 10-15, 2018*, pages 245–258, 2018.

[83] Yihan Sun, Guy E. Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *PVLDB*, 13(2):211–225, 2019.

[84] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[85] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 18–32, 2013.

[86] Annett Ungethüm, Dirk Habich, Tomas Karnagel, Sebastian Haas, Eric Mier, Gerhard Fettweis, and Wolfgang Lehner. Overview on hardware optimizations for database engines. In *BTW 2017, 6.-10. März 2017, Stuttgart, Germany, Proceedings*, pages 383–402, 2017.

[87] Hoang Tam Vo, Ashish Kundu, and Mukesh K Mohania. Research directions in blockchain data management and analytics. In *EDBT*, pages 445–448, 2018.

[88] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. Logbase: A scalable log-structured database system in the cloud. *PVLDB*, 5(10):1004–1015, 2012.

[89] Marko Vukolic. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In Jan Camenisch and Dogan Kesdogan, editors, *Open Problems in Network Security - IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*, volume 9591 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2015.

[90] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 95–112. USENIX Association, 2019.

[91] Qinshen Wang, Hongzhi Wang, and Bo Zheng. An efficient distributed storage strategy for blockchain. In *Proceedings of the ACM Turing Celebration Conference - China, ACM TUR-C 2019, Chengdu, China, May 17-19, 2019*, pages 54:1–54:5. ACM, 2019.

[92] Sheng Wang, David Maier, and Beng Chin Ooi. Lightweight indexing of observational data in log-structured storage. *PVLDB*, 7(7):529–540, 2014.

[93] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal*, 26(4):537–562, Aug 2017.

[94] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[95] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[96] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792, 2017.

[97] Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of*

*the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 141–158. ACM, 2019.

[98] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Trans. Knowl. Data Eng.*, 28(10):2635–2650, 2016.

[99] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[100] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. BCC: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *PVLDB*, 9(6):504–515, 2016.

[101] Maciej Zbierski. Parallel byzantine fault tolerance. In *ACS*, volume 342 of *Advances in Intelligent Systems and Computing*, pages 321–333. Springer, 2014.

[102] Bo Zhang, Binoy Ravindran, and Roberto Palmieri. Reducing aborts in distributed transactional systems through dependency detection. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 13:1–13:10, 2015.

[103] Wenbing Zhao. Optimistic byzantine fault tolerance. *IJPEDS*, 31(3):254–267, 2016.

[104] Ningnan Zhou, Xuan Zhou, Xiao Zhang, et al. Reordering transaction execution to boost high-frequency trading applications. *Data Science and Engineering*, 2(4):301–315, 2017.

[105] Yanchao Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, and Ying Yan. SEBDB: semantics empowered blockchain database. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 1820–1831. IEEE, 2019.