ON DATABASIFYING BLOCKCHAINS

by

RUAN PINGCHENG
(B.S., Nanyang Technological University)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

NATIONAL UNIVERSITY OF SINGAPORE

2021

Supervisor:
Professor OOI Beng Chin

Examiners:
Associate Professor CHAR Kway Teow
Assistant Professor Yummy Bee Hoon CRAB
Professor BAK Kwa, Dessert University

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

RUAN PINGCHENG

1 September 2020

To my teachers, parents, peers and friends...

Acknowledgments

I would like to place my foremost and deepest gratitude to my supervisor, Distinguished Professor Beng Chin Ooi. Thank you for your guidance throughout my PhD years. In the early years when I feel lost from time to time, it is Professor Ooi who tirelessly guides me through the research process.

Two heads are better than one. This thesis is inseparable from the efforts of my collaborators. Chapter 3 is the joint work with Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Dumitrel Loghin, and Meihui Zhang. Chapter 4 is collaborated with Gang Chen, Tien Tuan Anh Dinh, and Qian Lin. Chapter 5 is a result of the contribution from Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, and Gang Chen. I would like to take this opportunity to thank the above co-authors, which offer me abundant tips to conduct first-class research.

It is also my privilege to work with my seniors, Luo Zhaojing, Zheng Kaiping, Xie Zhongle, Wang Ji, Ji Xin, and my fortune to grow with my peers, Cai Shaofeng, Feng Piaopiao.

I would also like to say thanks to my thesis committee members, Chee Yong Chan and Yong Meng Teo, for their valuable feedback on this thesis.

Doing PhD is hard. I still remember the tough time when I started my journey. Every day buried under hundreds of papers, it is easy to become frustrated at my research direction and then the desperation looms. Things turn around in my third year when my first publication wins the VLDB 2019 Best Paper Award. This award boosts my confidence and convinces me that my previous efforts pay off. And immediately next year, I, as the first author, published on the top-tiered database conference SIGMOD 2020, a moment hard to imagine at the beginning. Beyond the technical knowledge and research skill, this is the most valuable lesson from my PhD experience: no achievements accomplish at one stroke but they will definitely spur with long accumulation. And I would like to share this wisdom of life with all my dear readers of this thesis, especially to those junior PhD candidates.

Last but not the least, I never forget the continuous support from my family. For my parents, sorry for being even grumpier than normal, especially during my early PhD days. For my wife, Junna, I love you more than you can imagine. This thesis is dedicated to you and our bright future.

Contents

Acknowledgments **ii**

Abstract **vi**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Blockchain Overview 1

1.2 Vision, Motivation and Principle 2

1.3 Optimization Basis 5

1.4 Thesis Synopsis 6

2 Literature Review **8**

2.1 Data Model Layer 8

2.1.1 State Organization 8

2.1.2 Ledger Abstraction 10

2.2 Execution Layer 10

2.2.1 Order-execute Architecture 11

2.2.2 Execute-order-validate Architecture 12

2.3 Consensus Layer 13

2.3.1 PoW and its Analysis 13

2.3.2 The Enhancements on PoW 14

2.3.3 Byzantine Fault-tolerant Consensus 15

2.3.4 Committee-based Consensus 17

2.4 Application 18

2.5 Benchmarks and Surveys 20
2.6 Lessons learnt from the Review 20

3 Twin Study **22**

4 Fine-Grained, Secure and Efficient Data Provenance on Blockchains 24
4.1 Introduction 24
4.2 Preliminary 27
4.2.1 State Organization 29
4.2.2 *FabricSharp* Overview 30
4.3 Fine-Grained Provenance 30
4.3.1 Capturing Provenance 31
4.3.2 Smart Contract APIs 33
4.4 Secure Provenance Storage 35
4.4.1 Merkle DAG 35
4.4.2 Discussion 37
4.4.3 Support for Forward Tracking 38
4.5 Efficient Provenance Queries 39
4.5.1 Deterministic Append-Only Skip List 39
4.5.2 Discussion 41
4.6 Implementation 44
4.6.1 Storage Layer 45
4.6.2 Application and Execution Layer 46
4.7 Performance Evaluation 46

5 A Transactional Perspective on Execute-order-validate Blockchains **48**
5.1 Introduction 48
5.2 Background 51
5.2.1 EOV architecture in Fabric and Fabric++ 51
5.2.2 Optimistic Concurrency Control in Databases 54
5.3 Theoretical Analysis 54
5.3.1 Resemblance in Transaction Processing 54
5.3.2 Serializability Analysis 57

5.3.3 Reorderability Analysis 59
5.3.4 Fine-grained Concurrency Control 61
5.3.5 Security Analysis 63
5.4 Implementation 65
5.4.1 Overview 65
5.4.2 Snapshot Read 66
5.4.3 Dependency Resolution 66
5.4.4 Cycle Detection 67
5.4.5 Dependency Restoration 68
5.4.6 Dependency Graph Pruning 70
5.4.7 *FabricSharp* Specifics 71
5.5 Experiments 72

6 Conclusion and Future Directions **73**
6.1 Conclusion 73
6.2 Future Directions 74
6.2.1 Blockchain Interoperability 74
6.2.2 Declarative Language for Smart Contracts 74
6.2.3 Blockchain-like Verifiable Databases 75
6.2.4 Federated Learning on Blockchains 75

Bibliography **76**

Publications during PhD Study **94**

Abstract

On Databasifying Blockchains
by
RUAN PINGCHENG
Doctor of Philosophy in Computer Science
National University of Singapore

The success of Bitcoin brings enormous interest to its underneath technology, the blockchain. A blockchain is a decentralized system capable to settle disputes between mutually distrusted parties. Throughout the years, blockchain applications are mostly restricted to cryptocurrencies, without fully unleashing their potential. It is until the emergence of smart contracts then blockchains start their transformation from simple cryptocurrency platforms into general data processing systems. Unfortunately, most blockchains researches are still carried out in the security community. Only a few database researchers are aware of this trend. In this thesis, we focus on the enhancement and optimization of blockchains from the perspective of a data system. As an attempt to databasify blockchains, we not only demonstrate the vast opportunities in this area but also appeal to more system researchers.

First, we treat a blockchain also as a generic distributed system, and as such it shares some similarities with distributed database systems. Existing works that compare blockchains and distributed database systems focus mainly on high-level properties, such as security and throughput. They stop short of showing how the underlying design choices contribute to the overall differences. Our paper is to fill this important gap. To be particular, We perform a twin study of blockchains and distributed database systems as two types of transactional systems. We propose a taxonomy that helps illustrate their similarities and differences. The taxonomy is along four dimensions: replication, concurrency, storage, and sharding. We discuss how the design choices have been driven by the system's goals: the blockchain's goal is security, whereas the distributed database's goal is performance. We then conduct an extensive performance study on two blockchains, namely Quorum and Hyperledger Fabric, and three distributed databases, namely CockroachDB, TiDB

and etcd. We demonstrate how the different design choices in the four dimensions lead to different performances. And the experimental insight sheds light on our database-styled optimization on blockchains.

Secondly, with a tamper-evident ledger for recording transactions that modify some global states, a blockchain system captures the entire evolution history of the states. The management of that history, also known as data provenance or lineage, has been studied extensively in database systems. However, querying data history in existing blockchains can only be done by replaying all transactions. This approach is applicable to large-scale, offline analysis, but is not suitable for online transaction processing. We hence present *FabricSharp*, a fine-grained, secure and efficient provenance system for blockchains. *FabricSharp* exposes provenance information to smart contracts via simple and elegant interfaces, thereby enabling a new class of blockchain applications whose execution logics depend on provenance information at runtime. *FabricSharp* captures provenance during contract execution, and efficiently stores it in a Merkle tree. *FabricSharp* provides a novel skip list index designed for supporting efficient provenance query processing. We have implemented *FabricSharp* on top of Hyperledger Fabric v2.2 and a blockchain-optimized storage system called ForkBase. Our extensive evaluation of *FabricSharp* demonstrates its benefits to the new class of blockchain applications, its efficient query, and its small storage overhead.

Thirdly, catering for emerging business requirements, a new architecture called execute-order-validate has been proposed in Hyperledger Fabric to support parallel transactions and improve the blockchain's throughput. However, this new architecture might render many invalid transactions when serializing them. This problem is further exaggerated as the block formation rate is inherently limited due to other factors besides data processing, such as cryptography and consensus. In this work, we propose a novel method to enhance the execute-order-validate architecture, by reducing invalid transactions to improve the throughput of blockchains. Our method is inspired by state-of-the-art optimistic concurrency control techniques in modern database systems. In contrast to existing blockchains that adopt database's preventive approaches which might abort serializable transactions, our method is theoretically more fine-grained. Specifically, unserializable transactions are aborted before ordering and the remaining transactions are guaranteed to be serializable. For

evaluation, we implement our method on top of our *FabricSharp*. We compare the performance of *FabricSharp* with carefully-chosen baselines. The results demonstrate that *FabricSharp* achieves 25% higher throughput compared to the other systems in nearly all experimental scenarios.

List of Figures

1.1 Blockchain high-level architecture. 3
1.3 Primary evaluation results for Fabric. 6

2.1 The Unspent Transaction Output (UTXO) model vs the Account-based model. 9
2.2 The Order-execute vs. Execute-order-validate architecture. 11

4.1 A smart contract that manages for token management. 25
4.2 The example ledger with *gState* between the block interval 31
4.3 The provenance helper method for *Token* contract 32
4.5 The modified *Token* contract with provenance-dependent methods. . . 34
4.6 A Merkle DAG for storing provenance. 37
4.8 Forward tracking support for data provenance 38
4.10 Node structure that captures a state $s_{k,v}$ with value *val* 40
4.11 An example of the append procedure in Deterministic Append-only Skip List (DASL) 40
4.13 The *FabricSharp*'s instrumentation on Fabric to support data provenance. 44
4.15 ForkBase APIs to implement the *FabricSharp*'s storage. 45

5.1 Fabric's raw and effective throughput under both no-op transactions and single modification transactions with varying skewness 49
5.2 Background in Execute-order-validate architecture and Fabric++'s instrumentation. 51
5.4 An example of transactions reading across blocks 55
5.5 Concurrency of transactions within and across blocks 57
5.7 Six canonical dependencies between snapshot transactions. 58
5.9 The implication of reordering to concurrent dependencies 60

5.11 Transaction schedule reorderability 61
5.12 The integration of our concurrency control on EOV blockchains 65
5.13 An example of a dependency graph with new *c-ww* dependencies 70
5.15 An example that leads to the incorrect forward query 71

List of Tables

2.1 Blockchain-related surveys and benchmarks 20

5.1 The transaction summary in Figure 5.2. 52

Chapter 1

Introduction

1.1 Blockchain Overview

Blockchains shake the industry, academia, and the entire world with storms. The swing of the butterfly that initiates the storm is an unidentified hacker named Satoshi Nakamoto, who authored the Bitcoin whitepaper in 2008 [118]. His proposal makes the breakthrough by employing Proof-of-work (PoW) mechanism, which allows mutually distrusting parties to reach an agreement on the ledger. The ledger records the forever-appending monetary transactions, with the immutability guarantee. Along with other cryptographic techniques, such as the asymmetric encryption, the Merkle index, and the hashed chain structure, Bitcoin is the first-ever practical cryptocurrency that operates under a pure peer-to-peer network, without any central authority. And it immediately follows a series of alt-coins variants [168]. Bitcoin is ground-breaking, as no early design can reach such scalable Byzantine consensus while defying Sybil Attacks. Nakamoto overcomes it by relying on the built-in cryptocurrency to regulate the participant behavior via the economic incentive.

To further unleash the power of blockchains beyond the cryptocurrency, there are two distinct directions. On the one hand, researchers preserve the incentive-based consensus to resolve the anonymity in the open setting. But it extends the system functionality from simple monetary flow into arbitrary data transformation, powered by smart contracts. A typical example is Ethereum, which allows to encode Turing-complete logic and execute it on an embedded virtual machine. We refer to this class of blockchains, featured with incentive-based consensus, built-in cryptocurrencies and the unauthenticated setup as *permissionless blockchains*.

On the other hand, to cater for applications where authenticity and auditability

are already mandated, blockchain designers take advantage of their close membership, and turn for more efficient and established state-machine replication [146] for the consensus. In addition, without the built-in cryptocurrencies, the smart contracts of these blockchains are more oriented towards their specific domains, such as Corda [73] for the financial sector and Hyperledger Fabric [9] for the enterprise. We refer to the above class of blockchains *permissioned*. Permissioned blockchains shows more potential to disrupt the industry and attract more interest from entrepreneurs.

Despite the above differences, both classes of blockchains share the identical high-level architecture, as proposed in BLOCKBENCH [51] and illustrated in Figure 1.1. The architecture is layered into four. Enumerating from the top, they are the application, consensus, execution, and data model layer. A typical processing pipeline for a generic blockchain constitutes of the following procedures. The consensus layer continuously drives participants to reach an agreement on the block at the ledger tip. Each participant then invoke the contract to mutate the state, based on the context in each transaction in the block. This step is conducted at the execution layer. If a transaction conforms to the blockchain protocol, the participant then persists its effect in the data model layer. And the top application layer hides all the underneath processing details but leaves interfaces to accept the request and query the ledger.

1.2 Vision, Motivation and Principle

Our system-wide optimization primarily focuses on permissioned blockchains. When compared with permissionless blockchains, permissioned blockchains resembles more to distributed databases and hence are more applicable to the database techniques. Their similarities and implication are summarized as follows:

Generic Workload Support. Smart contracts of permissioned blockchains support arbitrary data transformation, like stored procedures in databases. And both of their invocation result into a transaction in their respective context. This is in contrast with permissionless blockchains, which mostly restrict their attention to the ownership transfer. Inevitably, permissioned blockchains raises for more challenges due to their generality. Our optimization can no longer exploit the strong notion of asset ownership like in permissionless blockchains.

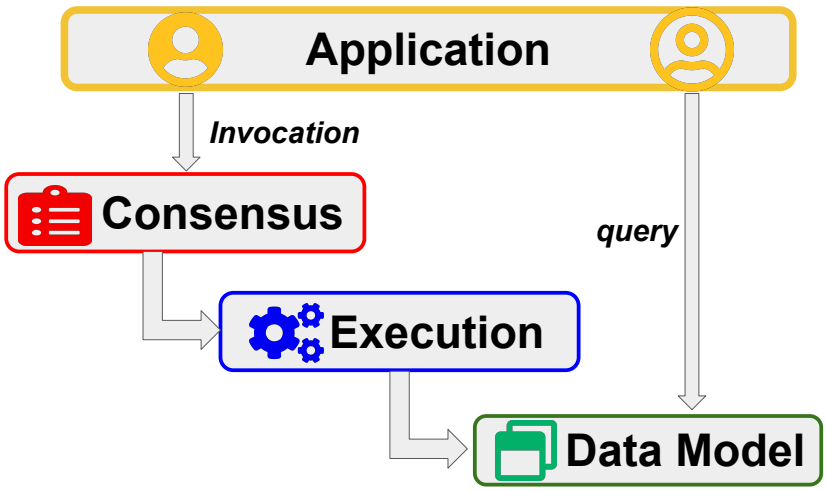


Figure 1.1: Blockchain high-level architecture.

State-mutating transactions must undergo the consensus and the execution components before persisting their effects at the data model layer, whereas ledger and state queries are directly answered by the storage component.

Authenticated Identity. Analogous to distributed databases, permissioned blockchains operate under the authenticated setup. Hence both categories of systems allow for more efficient state-machine replication consensus to withstand the byzantine failure. In comparison to permissionless blockchains, their PoW-like consensus (which is a must to mitigate Sybil Attacks) dominates the entire system performance. From this angle, the enhancement on other components of permissioned blockchains is necessary and worthwhile, as to side with their faster state-machine replication approach. Furthermore, the known membership relieves us from the identity problem. So that we will never get plagued by any Denial-of-the-service Attacks or Sybil Attacks throughout.

The above commonalities explain why a number of entrepreneurs are actively exploring permissioned blockchains to replace their enterprise-ready databases. They seek to harness their common data processing capability while enjoying the additional decentralization and security that permissioned blockchains uniquely provide. However, their attempts are primarily hindered by the following limitations of the permissioned blockchains:

Utility. Even though smart contracts open us opportunities for arbitrary transaction logic for blockchains, their provided utility is far from comparable to

that of databases. For example, mainstream blockchains only provide procedural languages to encode the data transformation, such as Ethereum with Solidity and Hyperledger Fabric with Golang. But current relational databases already adopt the more expressive declarative SQL language, not to mention the enriched query features that databases develop over decades. In the face of growing demand, permissioned blockchains call for more data processing functionalities, like databases.

Performance. Another challenge is the low processing volume of permissioned blockchain to accommodate the business load. Researchers in BLOCKBENCH evaluated three blockchains with database workloads on the same testbed. Their results show that blockchains lag far behind databases in around two magnitudes. We believe the extra security properties of blockchains shall not solely account for such a huge performance gap. There must exist abundant optimization room available for the speedup.

In the above, we lay out the optimization vision by showing vast similarities between permissioned blockchain and distributed databases. And we have also motivated such necessity by pinpointing the pain points for the adoption of blockchains in the industry. We now explain the three principles that our enhancement follows:

- We break no security properties. We believe security lies at the core of blockchains. Although relaxing security assumptions is a standard engineering approach for the performance speedup, we do not find it scientific. A typical approach is to improve the system throughput by simply switching from byzantine tolerant consensus to crash failure tolerant. In our thesis, this principle can be manifested in the following two ways. Firstly, for the utility enhancement, we must preserve security on the added features. For instance, in Chapter 4, we take special care to extend the tamper-evidence guarantee to the data provenance. On the other hand, any proposed procedures must be accompanied by their security analysis. This is why we dedicate Chapter 5.3.5 on the security implication of the transaction reordering.

- We adopt the modularized approach. We decouple a complex system into individual layers for the separate optimization. Modularization allows for the separation of concerns for the ease of reasoning. It also facilitates interoperability with independent optimization. We follow the proposed architecture

in BLOCKBENCH for our blockchain optimization in Chapter 4 and 5. In particular, we pinpoint our instrumentation with respect to each of four layers, as classified in Figure 1.1.

- Rather than building from the scratch, we ground our optimizations on Hyperledger Fabric v2.2, the most popular permissioned blockchain, for the demonstration. Building on an existing system not only reuses its well-examined components, but this action by itself proves our practicality. Moreover, the results from a full-fledged system, instead of a prototype, are more convincing. And the evaluation is more meaningful when directly comparing with the vanilla baseline.

1.3 Optimization Basis

We incrementally apply our optimization on Hyperledger Fabric 2.2.0 into *FabricSharp* and open source it [58]. We fork its codebase with the commit hash *2821cf*, the last commit on the branch *release-2.2* when we start preparing this thesis. In later paragraphs, Fabric, without any version specification, all refers to this codebase snapshot.

We now evaluate its throughput under its off-the-shelf configuration, which serves as the baseline. Unless otherwise mentioned, all our following experiments in this thesis are conducted in the same testbed as this experiment. The testbed consists of a local cluster of 16 nodes. Each node is equipped with E5-1650 3.5GHz CPU, 32GB RAM, and 2TB hard disk. The nodes are interconnected via 1Gbps Ethernet.

We present the primary results, after averaging over 3 times on the four-peer setup, in Figure 1.3. The first workload consists of no-op transactions that access no records. Fabric keeps around only 2300 tps throughput when the number of transactions per block is 2000 and beyond. In the second workload with 2000 transactions per block, we gradually increase the request skewness in the modification workload, in which each transaction reads and updates a single record out of 10k. The request distribution follows Zipfian distribution, where the larger coefficient θ implies for more skewness. When θ reaches 1.0, we observe its throughput reduces to 549 tps, 65% of that when $\theta = 0$. And more than half of transactions in the ledger do

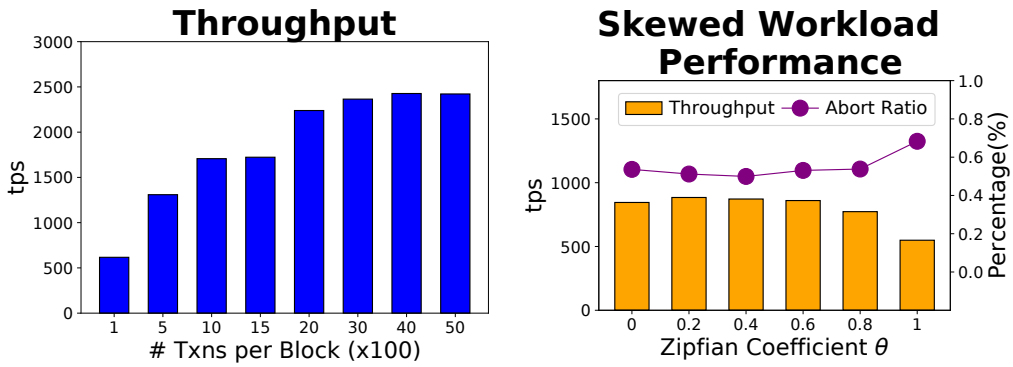


Figure 1.3: Primary evaluation results for Fabric.

not take effect due to the transactional conflicts. This primary experiment further motivates for the performance speedup of permissioned blockchains.

1.4 Thesis Synopsis

We structure the rest of this thesis as follows. Chapter [**ch:literature**] overviews the recent year progress on both permissioned and permissionless blockchains. The covered literature spans from the database, distributed computing, and security communities. For the modularized fashion, we organize their reviews according to the classified layers in Figure 1.1. This chapter ends with our critical analysis in this area.

Chapter 3 proposes a taxonomy that unifies blockchains and distributed databases. The study considers both systems as the same type of distributed transactional systems, for the joint analysis on their respective focus. According to each dimension in the taxonomy, we devise corresponding workloads for the evaluation. Our results reveal the implication of their design choices. This comprehensive study sheds light on the optimization opportunities on permissioned blockchain with database techniques.

Chapter 4 demonstrates our optimization for the utility. We first explain the added business value when data provenance is exposed to smart contracts. Then we introduce how lineage information in blockchains are captured, stored, and queried. The greatest contribution of our proposal is to extend the integrity property for the entire data evolution history. After implementing it into *FabricSharp* (or Fabric# for short), the empirical evaluation demonstrates the negligible performance and

storage overhead with this additional feature.

Chapter 5 demonstrates another optimization on the performance. The work originates from our following subtle observation: the execute-order-validate architecture in permissioned blockchains may over-abort transactions under the Serializable isolation level. Borrowed from the well-established transactional analysis from databases, we reason about the potential of transaction reordering to streamline the execution schedule. Based on the developed insight, then we adapt *FabricSharp* to attain the theoretical limits. And the improvement is empirically demonstrated with the remarkable speedup, compared with the vanilla Fabric and the state-of-the-art.

At last, we wrap up the thesis with the conclusion and future directions in Chapter 6.

Chapter 2

Literature Review

In this chapter, we lay out the foundation of blockchains and explore a systematic exposition on their recent progress. We organize the review based on the abstract layers as classified in Figure 1.1, before identifying the research gap.

2.1 Data Model Layer

The data model in blochchains concerns on how to organize data that reflect the latest states, and model the ledger that records the historical transactions.

2.1.1 State Organization

There are two state organizations in blockchains, the unspent transaction output-based (UTXO) and the account-based. Their key difference lies whether systems explicitly maintain the states. We illustrate both schemes in Figure 2.1.

UTXO Model. UTXO operates on the transaction basis. In their structure, a transaction consists of multiple inputs and outputs. Each output is associated with an amount of cryptocurrency and an cryptographic puzzle. Any future transaction can reclaim this amount in its input, by providing the puzzle answer and referencing the previously unspent output. An canonical puzzle and its solution can be an address in the format of a public key hash, and a digital signature from the corresponding private key. Notably, the UTXO model does not bookkeep the balance to addresses. All transactions in the ledger form a Direct Acyclic Graph that records the cryptocurrency flow, where identity hides under the anonymous addresses.

Bitcoins, due to its decentralization and anonymity, provide a terrain for financial crimes, such as drug dealing and the money laundry. There are a number of attempts

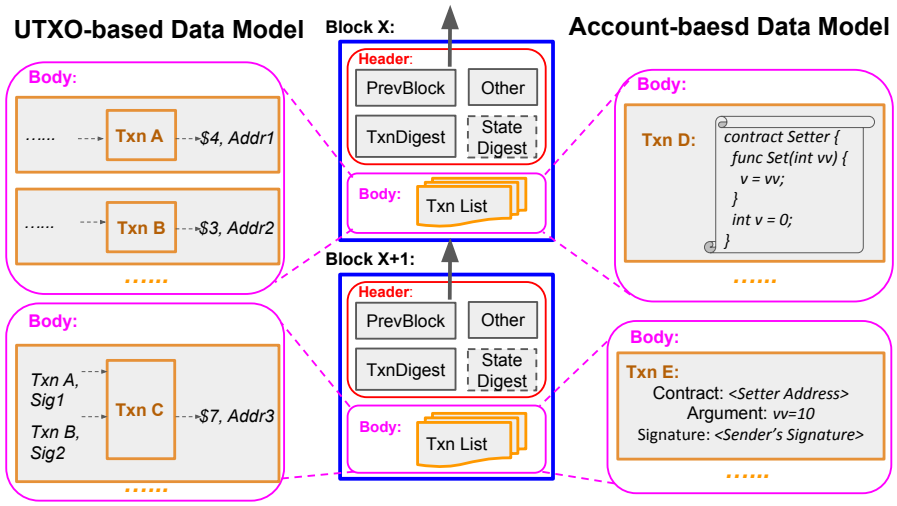


Figure 2.1: The Unspent Transaction Output (UTXO) model vs the Account-based model.

to exploit the transactional graph, and identify the pattern for the detection [62, 138, 166]. Some analysis relies on the graph linkage to discover the identity [125, 63, 117] or other information to predict the bitcoin price [70].

Account-based Model. Due to its simplicity, the UTXO model is solely applicable to cryptocurrency-based platforms. To support more general workloads, Ethereum introduces the smart contract to encode Turing-complete logic. In their design, a transaction either takes in the form of a contract deployment with the executable code. Or a transaction provides the execution context to invoke a contract. In all cases, each transaction is tagged with the digital signature of the sender. In addition, each blockchain peer must explicitly compute for contract states, including the cryptocurrency balance, in each account. Such requirement is enforced by the blockchain protocol that all peers shall reach consensus on a post-execution state digest in the block header. In contrast, the UTXO-based blockchain only requires a digest for the transaction integrity.

The account-based data model with smart contracts transition a blockchain into a more general processing platform. But it inevitably incurs more vulnerability from the additional complexity. For example, some malicious users might run an infinite loop in a transaction to waste system resources. To defer such Denial-of-service Attack in the permissionless setting, all blockchains are designed with an incentive-compatible mechanism to prevent the abusive usage. For example, Ethereum charges

transaction senders with the transaction fee, in a n amount proportional to the number and the complexity of contract operations [169]. Despite this, computational-heavy transactions may still render blockchains securely-flawed: some researchers reveal that rational block validators tend to skip their execution to gain an edge for the next block mining [107]. It is because all the transaction fee is credited to the block miner.

2.1.2 Ledger Abstraction

The term *blockchain* originates from the fact that the ledger takes in the form of a hash chain of blocks. Initially, the rationale for Bitcoin to batch transactions into blocks is to amortize the cryptographic overhead. However, this single-chain structure prohibits the concurrency, as all participants must sync up on the unique ledger tip. To address this problem, numerous studies have transformed a chain into a direct-acyclic graph (DAG), and then derive a total order of transactions [151, 86, 97, 152]. The exact ledger format is strongly tied to their consensus.

Meanwhile, batching trades off the latency for the throughput, as PoW bounds the block interval by the network delay. But from the perspective of permissioned blockchains. such tradeoff is not worthwhile: the state-machine replication consensus places no such restriction on the block interval. This observation accounts for why some researchers abandon the canonical block-based design but directly work upon transactions [80]. For a similar reason, we have observed a number of proposals known for their transaction-based DAG-typed ledger [96, 40, 52].

2.2 Execution Layer

The execution layer concerns on how to process transactions. Different blockchain platforms adopt their distinct execution platforms, such as the Docker environment for Hyperledger Fabric and Ethereum Virtual Machine (EVM) for Quorum. Despite their implementation differences, the execution architecture of any blockchains falls into either of the following categories. Figure 2.2 presents their distinction.

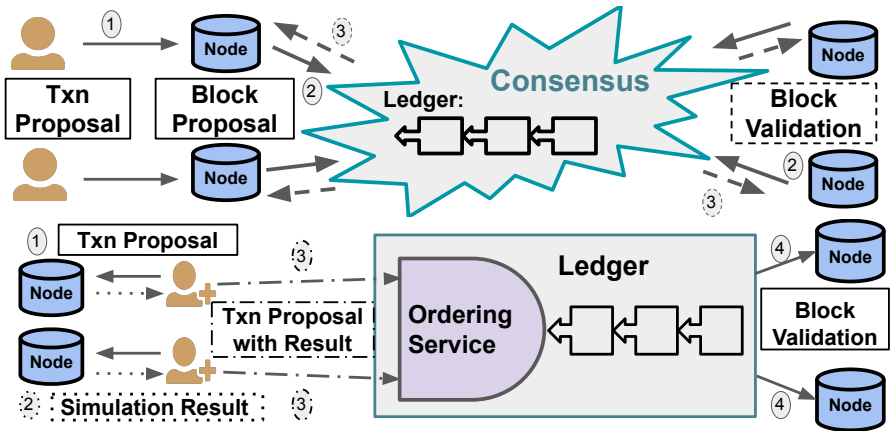


Figure 2.2: The Order-execute vs. Execute-order-validate architecture.

2.2.1 Order-execute Architecture

In the Order-execute architecture, each peer serially executes transactions, based on the established order according to the ledger by the consensus. Bitcoin as well as all cryptocurrency-based blockchain adopts this concurrency-free design, simply because the consensus, rather than the execution layer, decides on the system performance. Moreover, sequentiality makes it easy to reason about the system behavior.

Despite this, things still get convoluted when transactions deal with Turing-complete logic. For example, many security researchers have demonstrated that Solidity contracts in Ethereum are far more tricky than expected [104, 126, 11]. Due to some subtle misunderstanding on operation semantics on EVM, flawed contracts can be exploited by adversaries to gain profits. And the problem is further exaggerated given the transparency and the irreversibility of blockchains. The DAO hack shows that such an attack is not only a possibility in the theory but a true threat in reality [143]. In the meantime, there come along a series of empirical guidelines and practical tools to aid the contract development [54, 82, 15, 157].

Through the Order-execute architecture takes on the sequential approach, it is never insulated from the concurrency topic. For example, after remarking the reminiscence between contract bugs in blockchains and data races in shared-memory programming, researchers propose a novel viewpoint on the security issues of contracts, from the concurrency perspective in distributed computing [75, 147]. In [48], researchers explore to tentatively execute transactions in parallel and then fall into

serial if encountering a conflict. Instead of speeding up the entire system, the goal of the concurrency is to facilitate the block validation, so that validators can gain a competitive edge on the next block mining. Quantitative analysis of the transaction graph shows abundant concurrent chances in mainstream blockchains [134, 145].

2.2.2 Execute-order-validate Architecture

Execute-order-validate architecture is proposed in Hyperledger Fabric v1.0 [9]. Rather than taking a monolithic approach, the system is designed with two types of blockchain nodes: peers which execute smart contracts and validate blocks, and orderers which order transactions. A transaction pipeline is divided into three phases. In the Execute phase, a client requests a subset of peers to execute the transaction speculatively. The client collects the results and signatures from peers and sends them to the orderers. In the Order phase, orderers order the transactions and batch them into blocks. For modularity, orderers do not inspect the transaction details. In the Validate phase, each peer pulls blocks from the orderers and independently validates each block before persisting the results. The block validation process firstly verifies whether transactions satisfy the endorsement policy, i.e., enough number of peers show the endorsement by their signature. Then validation procedure checks for conflicts in the read/write sets for each transaction. The invalid transactions will not persist their effects, even though they are part of the ledger. Read-only queries only involve the Execute phase.

This architecture brings additional benefits compared to Order-execute architecture. Firstly, the endorsement policy decouples the trust condition of a contract from the consensus. For example, a transaction with the endorsement on execution results from only one of three peers can be considered valid. In contrast, the Order-execute architecture mandates the majority of peers to agree on the contract result. Secondly, it preserves confidentiality by restricting the execution to specify peers. Clients, by knowing results before transactions are effected, can also minimize the uncertainty. Lastly, speculative execution at the start fits well for the concurrency. It greatly facilitates computation-heavy transactions, which would queue up in the sequential Order-execute architecture.

However, such concurrency comes with the cost, which manifests as the aborted

transactions for the serializability. We have empirically demonstrated that in Figure 1.3 and will elaborate this issue in Chapter 5. In light of this, Fabric++ reorders transactions during the Order phase to minimize the abort [148]. OXII architecture is featured for an additional dependency resolution phase at the start [7]. So that it enables for a concurrency-friendly transaction schedule. OXII relies on the core assumption that the dependency can be extracted by inspecting the contract codebase. In the same spirit, XOX architecture runs a patch-up code to streamline a contended transaction [68]. The dependency captured during the transaction execution determines this snippet of code.

2.3 Consensus Layer

Byzantine-tolerant consensus differentiates blockchains from other distributed systems. Proof-of-work (PoW), initially proposed in Bitcoin, opens up a new horizon on incentive-based protocols. Meanwhile, permissioned blockchains revitalize the interest of state machine replication approach to address the arbitrary failure.

2.3.1 PoW and its Analysis

The essence of PoW is a hard-to-solve and easy-to-verify puzzle. The protocol prescribes that the first solver has the privilege to broadcast a new block on the tip of the ledger. By adjusting the puzzle difficulty such that the solving duration exceeds the maximum network delay, all blockchain participants can then sync up on the unique chain. In Bitcoin, the puzzle solution is a nonce to make the block hash value prefixed with enough zeros. The block proposer gets compensated for the hashing power with the newly-minted cryptocurrencies and the transaction fee.

In the context of Bitcoin, the solution-finding process is called *mining*. Due to the irreversibility of the hash function, mining is fair to all participants, including the adversaries. Ruling out the possibility that adversaries control the majority of the hash power, it results into the following two implications. If the adversaries pour all their resources to mine on a shorter chain fork, that fork is impossible to catch up the longest chain, where the rest of honest power concentrates. Nakamoto has bounded this possibility to be exponentially small in the original whitepaper [118]. More in-depth mathematical frameworks have been established to reason about the

mining behavior [66, 85, 135, 113, 65]. On the other hand, honest block proposers, aware that the longest chain is hardly revertible, are incentivized to extend on it. It is because their proposed block contains a coinbase transaction that credits the minted cryptocurrencies to proposers. Only the block is in the longest chain then this transaction can take into effect.

Inevitably, the longest-chain-rule may lead to two divergent forks during the asynchronous network period. When the network delay exceeds the block interval, two honest participants may generate two different blocks but on the same height. The longest-chain-rule will eventually resolve to a unique chain when the network partition heals. Hence, transactions in the ledger may not be secure given that they might reside on a shorter, to-be-pruned fork. Considering this, clients are advised to wait for their transactions until deep enough, before considering them committed. Intuitively, this depth, quantified by the number of blocks behind, balances between latency and security. In the meantime, the block interval and the block size controls the tradeoff between throughput and security. The shorter interval and the larger block imply greater system capacity. But it compromises the security, i.e., the network may not propagate a block to each participant before the next block is generated.

Moreover, [57] suggests that adversaries may not be incentivized to follow the default mining policy, i.e., mining on the longest chain and broadcast the block immediately. Instead, they may temporally withhold mined blocks and selectively publish them to gain extra profits. Since then, more mining policies are discovered which allows adversaries to gain a competitive edge [120, 42], even infiltrating other mining pools to waste their resources [106, 55]. Researchers employ a Markov Decision Process to find the optimal mining policy and explore the performance-security tradeoff [66, 144].

2.3.2 The Enhancements on PoW

PoW has long been plagued for its energy assumption. In 2019, Bitcoin's electricity consumption is reported to reach 45.8 TWh. Naturally, several researchers propose to transform PoW to be more eco-friendly, or at least dedicate resources to useful tasks. We refer to these PoW-based enhancements as Proof-of-X (PoX).

The major challenge of PoX is to replace the original computation-intensive puzzle and preserve its long-to-solve and easy-to-verify nature. For example, Proof-of-Stake (PoS) requires miners to stake a certain amount of cryptocurrencies [90]. The stake will be confiscated if the proposed blocks are found to be invalid. One can find the adoption of PoS in the following blockchains [87, 45, 21] and their security analysis in [121, 98, 27]. Proof-of-Elapsed-Time (PoET) replies on the Trusted Execution Environment to attest block validators that miners have waited enough time before the next block proposal [33]. To fully exploit the consumed resources to useful works, Proof-of-Retrievability is repurposed for the data archival [112], Proof-of-Prime-Number for the prime number searching [89], and others for matrix product problems [149].

Another enhancement direction is to optimize the primitive PoW. Researchers adapt the puzzle-computation to resist ASIC-equipped mining and deter power centralization [182, 38]. On the other hand, they refine the PoW mechanism to increase the system capacity. For example, the original PoW groups the leader election and transaction proposal together. Bitcoin-NG decouples these two tasks, by allowing a puzzle solver to continuously propose transactions until the next solver emerges [56]. GHOST incentives miners to extend on the heaviest sub-tree instead of the longest chain [151]. This is to better recycle orphaned blocks. Conflux furthers replies on GHOST to determine the pivot chain and use the pivot chain to decide on a direct-acyclic graph, whose topological order determines the overall transaction sequence [97]. The OHIE is renowned for its multiple-chain structure to harness the distributed network setup [179]. Its protocol drives each chain to grow at the same pace.

2.3.3 Byzantine Fault-tolerant Consensus

The state-machine replication method to address byzantine consensus resurges with the popularity of blockchains. The Byzantine-fault tolerant consensus problem involves a number of nodes with initial different proposals. The problem requires honest nodes to reach agreement on one of the proposals, while reserving the safety (no possibility of disagreement) and liveness (guarantee of termination) [95]. And this must hold true under the premise that a fraction of malicious nodes may perform

arbitrary action. Compared to Bitcoin, this problem modeling is more general, i.e., it is not targeting only on the ledger and no incentives can be hinged on.

Long before Bitcoin, PBFT has provided the first-ever practical approach. It achieves the consensus in $O(n^2)$ message complexity where n is the number of nodes. Quorum adopts a PBFT-variant, IBFT as one of its consensus options but still requires for $O(n^2)$ messages [142]. Tendermint reduces this bound to $O(n)$ but forgoes the network responsiveness [28]. In a word, unlike PBFT, the progress of Tendermint is dependent on the maximum network delay, rather than the actual message transmission delay [8]. Hotstuff augments on the two-phase approach of PBFT and Tendermint into three phases of the message exchange [177]. But it achieves both network linearity ($O(n)$ message complexity) and the responsiveness.

While the above series of researches are rooted from PBFT, researchers in VMWare address the problem from a new angle. They remark the possibility to reach consensus in a single phase, given more number of nodes have acknowledged the proposal than normal. Based on this insight, they start from Fast Byzantine Paxos [110] and Zyzzyva [71], and lead into a series of re-analysis [2] and re-design [3, 71]. Their novelty comes from a fast path if more-than-normal acknowlegements have been collected before a timeout. Otherwise, the consensus falls backs to the standard two-phase pipeline.

FLP Impossibility Result imposes no deterministic, safe and live protocols under the asynchronous network [61]. The above Byzantine-tolerant consensus protocols trade the liveness for the safety during the network partition. In other words, disagreement never occurs even the network loses the connection. Correspondingly, there also exist a number of approaches that rely on the network synchrony for the safety [4, 1]. In this manner, these protocols behave like PoW in Bitcoin, i.e., a chain will fork under the asynchronous period. In their design, the strict network assumption comes into play when to reliably broadcast the latest, majority-acknowledged block proposal. Notably, Flexible BFT mixes two types of deterministic protocols together, so that clients can draw their own commit decisions at their discretion on the network condition [109].

Randomness is another alternative route to break the Impossibilty, in order to reach agreement on the ledger. For example, HonestBadgerBFT combines a variety of randomized agreement protocols [114]. So that it achieves linear message

complexity, as PoW in Bitcoin. As the carry-on work, BEAT provides a diversity of randomized protocols that tailors for various application demands [53]. Such randomization takes in the form of the unpredictable network connectivity in [137]. In its design, each node repetitively queries for the proposals from the neighbors that it knows of. It then determines his own proposal from the majority of query results and uses it to respond for other's queries. Eventually, all honest nodes are probabilistically steered towards a uniform outcome, even though a limited fraction may cheat.

2.3.4 Committee-based Consensus

Blockchains with the committee-based consensus select a single or multiple representative committees, which then establish the ledger with the above state-machine replication consensus. This idea has been dated back to PeerCensus [46]. But their application may not be that straightforward. It is because these off-the-shelf protocols require for strong assumptions, including identity management, member constitution, and so on. To cater for these conditions, most blockchains must couple wth some additional procedures, especially on the fair committee formation. For example, Algorand determines a single committee based on the Verifiable Random Function (VRF) [67]. In particular, nodes must compute a value with VRF from its private key and a common seed. Only those with specific values are allowed to join the consensus with their public keys. VRF remains unpredictable without knowing the common seed beforehand. By starting from an unbiased random seed, Algorand guarantees that no adversaries may dominate a committee. Omniledger depends on the Randhound to achieve the decentralized and unbiased randomness, and power the multi-committee establishment [92]. Byzcoin repurposes PoW to determine the membership for a single committee [91]. PoW not only guarantees the randomness but also resists the Sybil attacks under the permissionless setting. RapidChain also uses PoW to determine participants, from which it breaks into multiple committees with a theoretical-involved method [181]. Elastico directly uses PoW nonce and group solvers with common prefixes into one of the committees [105]. Such randomness for the committee formation comes from Trusted Execution Environment in AHL [44]. Not that all the above systems are subject to the periodic

committee re-configuration between epochs. This prevents adaptive adversaries to exploit the setup and infiltrate honest nodes. The reconfiguration can be a full swap like in Elastico [105], Algorand [67], and Omniledger [92], or in a rotational manner like Byzcoin [91], RapidChain [181], and AHL [44].

We also observe several blockchains adopt a non-technical solution on membership management. For instance, a central authority decides on the validators' role in RSCoin [43]. The participants and their weight in Libra [20] and EoS [174] are pre-determined according to their initial investment. Chainspace completely sidesteps the committee formation problem [19]. It leaves the behavior of malicious committee audible under the premise that they can be enforced by an external entity.

2.4 Application

Blockchains provide unique decentralization, auditability, transparency, and irreversibility, adding abundant value to real-world applications. The most natural use case is to rely on the decentralized ledger as a trustworthy chronological logbook. For example, the Bitcoin protocol allows for a 40 byte of arbitrary data piggybacked at *OP_RETURN* opcode [153]. Several applications exploit this payload to use as a message commitment. So that one in the future can verify its existence, the creation time and the source of the origin. More advanced usage can be found in [108], where this payload is used to encode the right transfer of the external resource access, and in [171] for the e-voting. This survey investigates a broader range of applications and quantifies their the overhead of the piggybacking scheme [18].

Another popular application is the token management. Unlike the built-in cryptocurrency, the token is associated with external assets. The assets may range from vouchers, IOU, copyrights, and tangible objects. So that one may redeem the asset in the world, by showing the cryptographic proof of the token possession in the blockchain. For example, one may get a one-for-one exchange between Tether tokens and US dollars. We observe most of these applications operate on Ethereum, especially to hinge on the official provided ERC20 token standard [23]. In particular, ERC20 is a smart contract that defines a common list of rules on token operations. Any token issuer can deploy its own version of the ERC20 contract with the prevision of the initial ownership and the total of the supply amount. The transfer of tokens

takes in the form of the contract invocations. So that their transactions are secured by Ethereum. ERC20 token is getting widespread with a growing number of Initial Coin Offering (ICO). This study reveals that there are more than 64k distinct ERC20 tokens [160]. We refer readers to [160, 150, 35] for the analysis of their transaction pattern.

To further unleash the blockchain potential, plenty of business proposals explore its intermediary-free data processing capabilities to the banking industry [72], the supply chain management [159, 141, 93], Internet-of-Things(IoT) [124, 77, 84], the autonomous governance [47, 12], identity service [100, 102], medical record ownership control [14, 111], collaborative shared databases [128, 76] and so on. In all the above cases, blockchains provide a unique trust-building platform that enforces the business process. These processes usually involve multiple stakeholders with the conflict of interest. Along with the popularity, some researchers voice out their concern about the improper blockchain usage. [172] criticizes this worrying trend by critically analyzing several unreasonable cases. [176, 39, 172] provide empirical guidelines for applications to choose between blockchains and conventional databases. In their opinion, the absence of the central authority is the deciding factor for the blockchain adoption. This absence may be due to the entangled interest in the application, the trust deficit from the hostile environment or etc. Otherwise, the expense of dis-intermediation, in terms of the extra storage and performance, far outweighs other blockchain-provided features. Most similar business requirements, such as the provenance support and state integrity, can be more efficiently satisfied with other technical solutions. For example, QLDB, as a centrally administrated, in-cloud database, provides a verifiable storage [6]. In other words, the QLDB vendor is technically possible to tamper the data or refuse to deliver the service, despite the fact that all this action will leave evidence. Blockchains, however, is capable to prevent such arbitrary behavior due to their decentralized consensus. But it is questionable whether such additional security guarantee is worthwhile for most applications.

Table 2.1: Blockchain-related surveys and benchmarks

	Layer	**Focus**
Surveys	Execution	Ethereum Contract Security [99, 11, 32, 132]
	Consensus	[164, 122, 173, 64]
	Application	Internet-of-Things [124, 170, 165] Banking [72, 41, 130] Others [116, 81, 25, 186]
	Others	Technical Overview [185, 101, 50] Chain Interoperability and Sharding [183, 161, 178] Data Privacy [60, 83, 115]
Benchmarks		Hyperledger Fabric [51, 155, 119, 140, 131, 158] Ethereum[51, 131, 139, 34] Quorum [140, 17] Parity [51] Corda [129, 158]

2.5 Benchmarks and Surveys

Even though the aforementioned references cover most representative research works, they are far from comprehensive to draw an overall picture on the recent blochchain progress. For a wider range of exposition, we compile blockchain-related surveys with their focused layers in Table 2.1. We also tabulate reported benchmarks with their selected systems in the table.

2.6 Lessons learnt from the Review

Our above review reveals a mismatch between the focus of the industry and academia on blockchains. In particular, researchers place great emphasis on their security aspects. They restrict their attention on the incentive-based consensus, and UTXO-based cryptocurrency in permissionless blockchains. On the other hand, entrepreneurs in the industry show the most interest in the intermediary-free data processing capabilities from permissioned blockchains. In the meantime, we observe most researchers treat a blockchain as it is, but few of them realize its vast similarities to distributed databases. A few work with this awareness juxtapose permissioned blockchains and distributed databases for the joint study. But their analysis leaves much room for improvement: they either only show the blockchains' inferior performance to databases [51, 34]. Or they compare their distinct value-

addition properties for applications [39, 172, 176]. But they ignore a promising area, which is to optimize permissioned blockchains with database-specific techniques to support general workloads.

Our thesis moves a step forward along this direction. We first classify blockchains and distributed databases under the same taxonomy. Based on this framework, we jointly understand their design choices and behavior (Chapter 3). With the developed insights, we extend the provenance support to smart contracts to improve utility (Chapter 4). Then learning from databases, we explore reordering techniques to reduce the transaction abort. This performance enhancement applies to any blockchains with execute-order-validate execution pipeline (Chapter 5).

Chapter 3

Twin Study

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi

blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Chapter 4

Fine-Grained, Secure and Efficient Data Provenance on Blockchains

4.1 Introduction

Blockchains are disrupting many industries, including finance [154, 123], supply chain [93, 156], and healthcare [111]. These industries are exploiting two distinct advantages of blockchains over traditional data management systems. First, a blockchain is decentralized, which allows mutually distrusting parties to manage the data together instead of trusting a single party. Second, the blockchain provides integrity protection (tamper evidence) to all transactions recorded in the ledger. In other words, the complete transaction history is secure.

The management of data history, or data provenance, has been extensively studied in databases, and many systems have been designed to support provenance [36, 37, 29, 127, 5, 162]. In the context of blockchain, there is explicit, but only coarse-grained support for data provenance. In particular, the blockchain can be seen as having some states (with known initial values), and every transaction moves the system to new states. The evolution history of the states (or provenance) can be securely and completely reconstructed by replaying all transactions. This reconstruction can be done during offline analysis. During contract execution (or runtime), however, no provenance information is safely available to smart contracts. In other words, smart contracts cannot access historical blockchain states in a tamper-evident manner. The lack of secure, fine-grained, runtime access to provenance therefore restricts the expressiveness of the business logic the contract can encode.

Consider an example smart contract shown in Figure 4.1, which contains a method

```
contract Token {
    method Transfer(sender, recipient, amount) {
        bal1 = gState[sender];
        bal2 = gState[recipient];
        if (amount < bal1) {
            gState[sender] = bal1 - amount;
            gState[recipient] = bal2 + amount;
    }
  }
}
```

Figure 4.1: A smart contract that manages for token management.

for transferring a number of tokens from one user to another. Suppose user A wants to send tokens to B based on the latter's historical balance in recent months. For example, A only sends token if B's average balance per day is more than t. It is not currently possible to write a contract method for this operation. To work around this, A needs to first compute the historical balance of B by querying and replaying all on-chain transactions, then based on the result issues the `Transfer` transaction. Besides performance overhead incurred from multiple interactions with the blockchain, this approach is not *safe*: it fails to achieve transaction serializability. In particular, suppose A issues the `Transfer` transaction tx based on its computation of B's historical balance. But before tx is received by the blockchain, another transaction is committed such that B's average balance becomes $t' < t$. Consequently, when tx is later committed, it will have been based on stale state, and therefore fails to meet the intended business logic. In blockchains with native currencies, serializability violation can be exploited for Transaction-Ordering attacks that cause substantial financial loss to the users [104].

In this paper, we design and implement a fine-grained, secure and efficient provenance system for blockchains, called *FabricSharp*. In particular, we aim to enable a new class of smart contracts that can access provenance information at runtime. Although our goal is similar to that of existing works in adding provenance to databases [5, 156, 133], we face three unique challenges due to the nature of blockchain. First, there is a lack of data operators whose semantics capture provenance in the form of input-output dependency. More specifically, for general data management workloads (i.e., non-cryptocurrency), current blockchains expose

only generic operators, for example, `put` and `get` of key-value tuples. These operators do not have input-output dependency. In contrast, relational databases operators such as `map`, `join`, `union`, are defined as relations between input and output, which clearly capture their dependencies. To overcome this lack of provenance-friendly operators, we instrument blockchain runtime to record read-write dependency of all the states used in any contract invocation, which is then passed to a user-defined method that specifies which dependency to be persisted.

The second challenge is that blockchains assume an adversarial environment, therefore any captured provenance must be made tamper evident. To address this, we store provenance in a Merkle tree data structure that also allows for efficient verification. The final challenge is to ensure that provenance queries are fast, because a large execution overhead is undesirable due to the Verifier's Dilemma [107]. To address this challenge, we design a novel skip list index that is optimized for provenance queries. The index incurs small storage overhead, and its performance is independent of the number of blocks in the blockchain.

In summary, we make the following contributions:

- We introduce a system, called *FabricSharp* that efficiently captures fine-grained provenance for blockchains. It stores provenance securely, and exposes simple access interface to smart contracts.

- We design a novel index optimized for querying blockchain provenance. The index incurs small storage overhead, and its performance is independent of the blockchain size. It is adapted from the skip list but we completely remove the randomness to fit for deterministic blockchains.

- We implement *FabricSharp* for Hyperledger Fabric v2.2 [78]. Our implementation builds on top of ForkBase, a blockchain-optimized storage [163]. We conduct extensive evaluation of *FabricSharp*. The results demonstrate its benefits to provenance-dependent applications, and its efficient query and small storage overhead.

The remainder of the chapter is organized as follows. Chapter 4.2 provides necessary blockchain preliminaries for the understanding. Chapter 4.3 describes our design for capturing provenance, and the interface exposed to smart contracts.

Chapter 4.4 discusses how we store provenance, and Chapter 4.5 describes our new index. Chapter 4.6 presents our implementation. Chapter 4.7 reports the performance of *FabricSharp* on provenance-related matters.

4.2 Preliminary

In this section, we present more background on blockchain systems. We especially focus on the block structure and the procedure to validate a block. And we explain the design choices of state organization that affect index structure requirements. At last, we present an overview of *FabricSharp*.

Data model. Different blockchains adopt different data models for their states. Bitcoin' states are unspent coins modeled as Unspent Transaction Outputs (UTXOs), as detailed in Chapter 2.1. More recent blockchains, namely Ethereum and Fabric, support general states that can be modified arbitrarily by smart contracts. They adopt an account-based data model, in which each account has its own local states stored on the blockchain. A smart contract transaction can write arbitrary data to the storage. This flexible data model comes at the cost of integrity protection and verification of the account states. In this paper, we focus on the account-based data model.

Block structure. A block in the blockchain stores the transactions and the global states. The block header contains the following fields.

- `PreviousBlockHash`: reference to the previous block in the chain.
- `Nonce`: used for checking validity of the block. In PoW consensus, `Nonce` is the solution to the PoW puzzle.
- `TransactionDigest`: used for integrity protection of the list of transactions in the block.
- `StateDigest`: used for integrity protection of the global states after executing the block transactions.

Both `TransactionDigest` and `StateDigest` are Merkle-tree roots. They allow for efficient block transfer, in which the block headers and block content can be

decoupled and transferred separately. In addition, they enable efficient verification of transactions and states.

Algorithm 1: Block verification in blockchain

```
   Input: A block Blk received from the network.
   Input: The global state gState, which maps state identifiers to their values.
   Output: True if the block is valid, False otherwise.
   // Step 1: Verify Nonce (PoW only).
 1: if checkNonce(Blk.Header.Nonce) then
 2:     return False;
   // Step 2: Verify transactions.
 3: txnDigest = computeDigest(Blk.Transactions);
 4: if txnDigest != Blk.Header.TransactionDigest then
 5:     return False;
   // Step 3: Tentatively execute transactions.
 6: oldState = gState;
 7: allUpdates = [];
 8: for txn in Blk.Header.Transactions do
        // Buffer the changes
 9:     updates = execute(txn);
10:     allUpdates.Append(update);
11: gState.apply(allUpdates);
   // Step 4: Verify new state.
12: stateDigest = computeDigest(state);
13: if stateDigest != Blk.Header.StateDigest then
        // Rollback
14:     gState = oldState;
15:     return False;
16: else
17:     return True;
```

Block verification. Algorithm 1 illustrates how a node uses the block header to verify if a block it receives from the network is valid. If the block is valid, it is appended to the chain. When PoW is used for consensus, the node first checks if `Nonce` is the correct solution to the PoW puzzle. This step is skipped if a deterministic consensus protocol, such as PBFT, is used. Next, it checks if the list of transactions has not been tampered with, by computing and verifying `TransactionDigest` from the list. It then tentatively re-executes the included transactions. During execution, the states are accessed via some index structures.

After the execution, the node checks if the resulting states match with `StateDigest`. If they do, the block is considered valid and the new states are committed to the storage. Otherwise, the states are rolled back to those before execution. We note that Algorithm 1 takes as input an object *gState* that represents the global states. If the blockchain does not have forks, e.g., Hyperledger Fabric, this object is the *latest* states. However, when there are forks, e.g., in Ethereum, this object may refer to the global states at a point in the past.

4.2.1 State Organization

The most important feature of blockchain is the guarantee of data integrity, which implies that the global states must be tamper evident. The block verification algorithm above is crucial for the security of blockchain. We note that the algorithm requires access to all history snapshots of the states, as well as the ability to update the states in batch. These requirements present new challenges in designing an index structure for organizing blockchain states. In particular, traditional database indices such as B+ tree cannot be used. We now elaborate on the requirements for a blockchain index, and explain how they are met in Ethereum and Hyperledger. *FabricSharp* builds on existing blockchain indices to ensure security for the captured provenance.

Tamper evidence.. A user may want to read some states without downloading and executing all the transactions. Thus, the index structure must be able to generate an integrity proof for any state. In addition, the index must provide a unique digest for the global states, so that nodes can quickly check if the post-execution states are identical across the network.

Incremental update.. The size of global states in a typical blockchain application is large, but one block only updates a small part of the states. For example, some states may be updated at every block, whereas other may be updated much more infrequently. Because the index must be updated at every block, it must be efficient at handling incremental updates.

Snapshot. A snapshot of the index, as well as of the global states, must be made at every block. This is crucial to realize the immutability property of blockchain which allows users to read any historical states. It is also important for block

verification. As explained earlier, when a new block is received that creates a fork, an old snapshot of the state must be used as input for verification. Even when the blockchain allows no forks, snapshots enable roll-back when the received block is found to be invalid after execution (step 4 in Algorithm 1).

Existing blockchains use indices that are based on Merkle tree. In particular, Ethereum implements Merkle Patricia Trie (MPT), and Hyperledger implements Merkle Bucket Tree (MBT). In a Merkle tree, content of the parent node is recursively defined by those of the child nodes. The root node uniquely identifies the content of all the leaf nodes. A proof of integrity can be efficiently constructed without reading the entire tree. Therefore, the Merkle tree meets the first requirement. This structure is also suitable for incremental updates (second requirement), because only the nodes affected by the update need to be changed. To support efficient snapshots, an update in the Merkle tree recursively creates new nodes in the path affected by the change. The new root then serves as index of the new snapshot, and is then included in the block header.

4.2.2 *FabricSharp* Overview

Given a smart contract on an existing blockchain, *FabricSharp* enriches it with fine-grained, secure and efficient provenance as follows. First, the contract can implement a helper method to define the exact provenance information to be captured at every contract invocation. By default, all read-write dependencies of all the states are recorded. Second, new methods can be added that make use of provenance at runtime. As far as a contract developer is concerned, these are the only two changes from the existing, non-provenance blockchain. The captured information is then stored in an enhanced blockchain storage that ensures efficient tracking and tamper evidence of provenance. On top of this storage, we build a skip list index to support fast provenance queries. These changes to the blockchain storage are invisible to the contract developer.

4.3 Fine-Grained Provenance

In this section, we describe our approach to capture provenance during smart contract execution. We present APIs that allow the contract to query provenance at

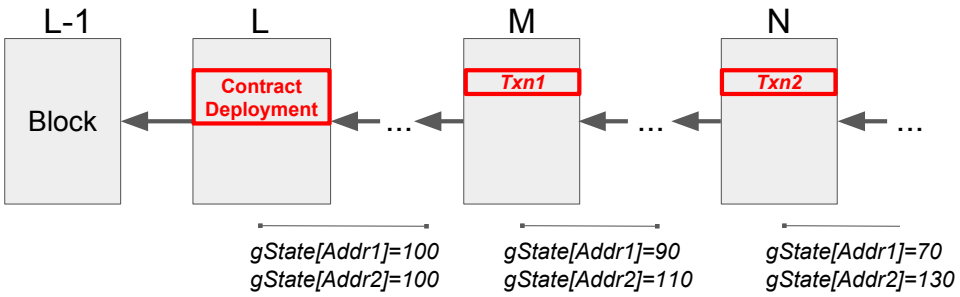


Figure 4.2: The example ledger with *gState* between the block interval

runtime.

Running example. Throughout this section, we use as running example the token smart contract shown in Figure 4.1. Figure 4.2 depicts how the global states are modified by the contract. In particular, the contract is deployed at block L^{th} in the blockchain. Two addresses *Addr1* and *Addr2* are initialized with 100 tokens. Two transactions *Txn1* and *Txn2* that transfer tokens between the two addresses are committed at block M and N respectively. The value of *Addr1* is 100 from block L to block $M-1$, 90 from block M to $N-1$, and 70 from block N. The global state *gState* is essentially a map of addresses to their values.

4.3.1 Capturing Provenance

Blockchains support only a small set of data operators for general workloads, namely *read* and *write*. These operators are not provenance friendly, in the sense that they do not capture any data association (input-output dependency). In contrast, relational databases or big data systems have many provenance-friendly operators, such as *map*, *reduce* and *join*, whose semantics meaningfully capture the association. For instance, the output of *join* is clearly derived from (or is dependent on) the input data.

In *FabricSharp*, every contract method can be made provenance-friendly via a *helper* method. More specifically, during transaction execution, *FabricSharp* collects the identifiers and values of the accessed states, i.e., ones used in `read` and `write` operations. The results are a read set `reads` and write set `writes`. For *Txn1*, `reads` $= \{Addr1 : 100, Addr2 : 100\}$, and `writes` $= \{Addr1 : 90, Addr2 : 110\}$. After the execution finishes, these sets are passed to a user-defined method

```
contract Token {
    method Transfer(...){...} // as above
    method prov_helper(name, reads, writes) {
      if name == "Transfer" {
        for (id,value) in writes {
            if (reads[id] < value) {
                recipient = id;
            } else {sender = id; }
        }
        // dependency list with a
        // single element.
        dep = [sender];
        return {recipient:dep};
      }
      ...
    }
}
```

Figure 4.3: The provenance helper method for *Token* contract

It defines dependency between the sender identifier and recipient identifier. This method is invoke after every invocation of the Token contract.

`prov_helper`, together with the name of the contract method. `prov_helper` has the following signature:

```
method prov_helper(name: string,
                   reads: map(string, byte[]),
                   writes: map(string, byte[]))
       returns map(string, string[]);
```

It returns a set of dependencies based on the input read and write sets. Figure 4.3 shows implementation of the helper method for the Token contract. It first computes the identifier of the sender and recipient from the read and write sets. Specifically, the identifier whose value in `writes` is lower than that in `reads` is the sender, and the opposite is true for the recipient. It then returns a dependency set of a single element: the recipient-sender dependency. In our example, for *Txn1*, this method returns $\{Addr2 : [Addr1]\}$

FabricSharp ensures that `prov_helper` is invoked immediately after every successfully contract execution. If the method is left empty, *FabricSharp* uses all identifiers in the read set as dependency of each identifier in the write set. Interested readers may observe that the vanilla Fabric already computes for the read/write set

during the endorsement phase. Orthogonal to ours, they are internally used for the concurrency control to achieve one-copy serializability. Instead, we allow contract developers to capture for their application-level provenance.

4.3.2 Smart Contract APIs

Current smart contracts can only safely access the *latest* blockchain state. In Hyperledger, for example, the `get(k)` operation returns the last value of k that is written or being batched. In Ethereum, on the other hand, when a smart contract reads a value of k at block b, the system considers the snapshot of states at block $b-1$ as the latest states. Although there may exist a block $b' > b$ on a different branch, the smart contract always treats what returned from the storage layer as the latest state.

The main limitation of the current APIs is that the smart contract cannot tamper-evidently read previous values of a state. Instead, the contract has to explicit track historical versions, for example by maintaining a list of versions for every state. This approach is costly both in terms of storage and computation. *FabricSharp* addresses this limitation with three additional smart contract APIs.

- `Hist(stateID, [blockNum])`: returns the tuple `(val, blkStart)` where `val` is the value of `stateID` at the snapshot of block `blockNum`. If `blockNum` is not specified, the latest block is used. `blkStart` is the index of the block that contains a transaction setting `stateID`.

- `Backward(stateID, blkNum)`: returns a list of tuples `(depStateID, depBlkNum)` where `depStateID` is the dependency state of `stateID` at block `blkNum`. `depBlkNum` is the block number at which the value of `depStateID` is set. It also returns `txID` which is the identifier of the transaction that sets `stateID`. In our example, `Backward(Addr2, N)` returns `(Addr1, M)` and `txID` is equal to `Txn2`.

- `Forward(stateID, blkNum)`: similar to the `Backward` API, but returns the states of which `stateID` is a dependency and the corresponding transaction identifier. For example, `Forward(Addr1, L)` returns `(Addr2, M, Txn1)`.

```
contract Token {
  ...
  method Refund(addr) {
    blk := last block in the ledger
    first_blk := first block in this month
    sum = count = 0;
    while (first_blk < blk) {
      val, startBlk = Hist(addr, blk);
      blk = startBlk - 1;
      sum += val;
      count += 1;
    }
    avg = sum / count;
    refund_amount := refund amount based on avg
    gState[addr] += refund_amount;
  }

  method Suspect(addr) {
    blk := last block in the ledger
    suspected = false;
    iterate 5 times {
      val, startBlk = Hist(addr, blk);
      for (depAddr, depBlk)
            in (Backward(addr, startBlk)
            or Forward(addr, startBlk)) {
        if depAddr in gState["suspect"] {
          gState["suspect"].append(addr);
          return;
        }
      }
      blk = startBlk - 1;
    }
  }
}
```

Figure 4.5: The modified *Token* contract with provenance-dependent methods.

Figure 4.5 demonstrates how the above APIs are used to express smart contract logics that are currently impossible in the secure manner. We add two additional methods to the original contract, both of which use the new APIs. The *Refund* method examines an account's average balance in the recent month and makes the refund accordingly. The *Suspect* method marks an address as suspected if one of its last 5 transactions is with a suspected address.

4.4 Secure Provenance Storage

In this section, we discuss how *FabricSharp* enhances existing blockchain storage layer to provide efficient tracking and tamper evidence for the captured provenance. Our key insight is to reorganize the flat leaf nodes in the original Merkle tree into a Merkle DAG. We first describe the Merkle DAG structure, then discuss its properties. Finally, we explain how to exploit the blockchain execution model to support forward provenance tracking.

4.4.1 Merkle DAG

Let k be the unique identifier of a blockchain state, whose evolution history is expected to be tracked. Let v be the unique version number that identifies the state in its evolution history. When the state at version v is updated, the new version v' is strictly greater than v. In *FabricSharp*, we directly use the block number as its version v. Let $s_{k,v}$ denote the value of the state with identifier k at version v. We drop the subscripts if the meaning of k and v are not important. For any $k \neq k'$ and $v \neq v'$, $s_{k,v}$ and $s_{k',v'}$ represent the values of two different states at different versions. s_k^b represents the state value with identifier k at its latest version before block b. In our example, for $k = Addr1$ and $v = M$, $s_{k,v} = 90$.

Definition 1. *A transaction, identified by tid which is strictly increasing, reads a set of input states S^i_{tid} and updates a set of output states S^o_{tid}. A valid transaction satisfies the following properties:*

$$\forall s_{k_1,v_1}, s_{k_2,v_2} \in S^o_{tid}. \quad k_1 \neq k_2 \wedge v_1 = v_2 \tag{4.1}$$

$$\forall s_{k_1,v_1} \in S^i_{tid}, s_{k_2,v_2} \in S^o_{tid}. \quad v_1 < v_2 \tag{4.2}$$

$$\forall s_{k,v} \in S^i_{tid}, s_{k,v'} \in S^i_{tid'}. \quad tid < tid' \Rightarrow v \leq v'. \tag{4.3}$$

$$tid \neq tid' \Rightarrow S^o_{tid} \cap S^o_{tid'} = \emptyset \tag{4.4}$$

Property (4.1) means that the versions of all output states of a transaction are identical, because they are updated by the same transaction in the same block. Property (4.2) implies the version of any input state is strictly lower than that of the output version. This makes sense because the blockchain establishes a total order

over the transactions, and because the input states can only be updated in previous transactions. Property (4.3) specifies that, for all the states with the same identifier, the input of later transactions can never have an earlier version. This ensures the input state of any transaction must be up-to-date during execution time. Finally, Property (4.4) means that every state update is unique.

Definition 2. *The dependency of state s is a subset of the input states of the transaction that outputs s. More specifically:*

$$dep(s) \subset S^i_{tid} \text{ where } s \in S^o_{tid}.$$

We note that *dep*, which is returned by `prov_helper` method, is only a subset of the read set.

Definition 3. *The entry $E_{s_{k,v}}$ of the state $s_{k,v}$ is a tuple containing the current version, the state value, and the hashes of the entries of its dependent state. More specifically:*

$$E_{s_{k,v}} = \langle v, s_{k,v}, \{hash(E_{s'}) | s' \in dep(s_{k,v})\}\rangle$$

An entry uniquely identifies a state. In *FabricSharp*, we associate each entry with its corresponding hash.

Definition 4. *The set of latest states at block b, denoted as $S_{latest,b}$ is defined as:*

$$S_{latest,b} = \bigcup_k \{s^b_k\}$$

Let U_b be the updated states in block b. We can compute $S_{latest,b}$ by recursively combining U_b with $S_{latest,b-1} \setminus U_b$.

Definition 5. *χ_b is the root of a Merkle tree built on the map S_b where*

$$S_b = \{k : hash(E_{s^b_k}) | \forall s^b_k \in S_{latest,b}\}.$$

FabricSharp stores χ_b as the state digest in the block header. Figure 4.6 illustrates how state DAG evolves with the blockchain.

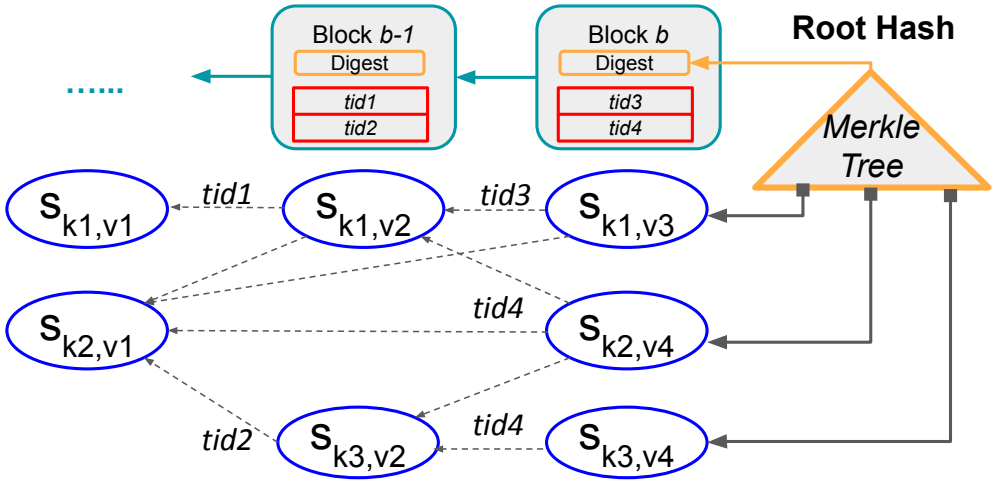


Figure 4.6: A Merkle DAG for storing provenance.

s_{k_2,v_4} and s_{k_3,v_4} updated by the same transaction (tid_4), but their dependencies are different. b contains two transactions, tid_3 and tid_4. Its latest states include s_{k_1,v_3}, s_{k_2,v_4}, s_{k_3,v_4}, from which a Merkle tree is built.

4.4.2 Discussion

Our new Merkle DAG can be easily integrated to existing blockchain index structures. In particular, existing Merkle index such as MPT stores state values directly at the leaves, whereas the Merkle DAG in *FabricSharp* stores the entry hashes of the latest state versions at the leaves. By adding one more level of indirection, we maintain the three properties of the index (tamper evidence, incremental update and snapshot), while enhancing it with the ability to traverse the DAG to extract fine-grained provenance information.

Recall that the state entry hash captures the entire evolution history of the state. Since this hash is protected by the Merkle index for tamper evidence, so is the state history. In other words, we add integrity protection for provenance without any extra cost to the index structure. For example, suppose a client wants to read a specific version of a state, it first reads the state entry hash at the latest block. This read operation can be verified against tampering, as in existing blockchains. Next, the client traverses the DAG from this hash to read the required version. Because the DAG is tamper evident, the integrity of the read version is guaranteed.

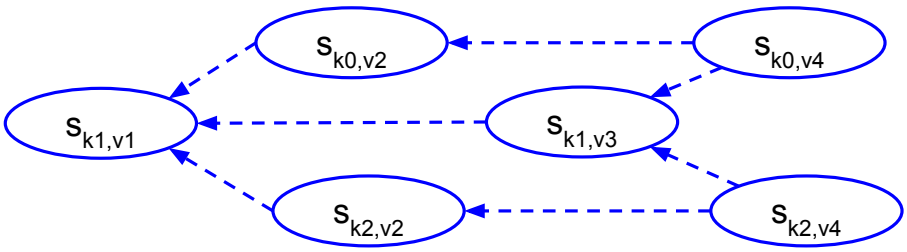


Figure 4.8: Forward tracking support for data provenance

After s_{k_1,v_1} is updated, there can only be s_{k_0,v_2} and s_{k_2,v_2} that are dependent on s_{k_1,v_1}. Future states can only depend on $s_{k_1,v3}$. Forward pointers of s_{k_1,v_1} are stored in the entry of s_{k_1,v_3}.

4.4.3 Support for Forward Tracking

One problem of the above DAG model is that it does not support forward tracking, because the hash pointers only reference *backward* dependencies. When a state is updated, these backward dependencies are permanently established, so that they belong to the immutable derivation history of the state. However, the state can be read by future transaction, therefore its forward dependencies cannot be determined at the time of update.

Our key insight here is that only forward dependencies of the *latest state* are mutable. Once the state is updated, due to the execution model of blockchain smart contract, in which the latest state is always read, forward dependencies of the previous state version becomes permanent. As a result, they can be included into the derivation history. Figure 4.8 illustrates an example, in which forward dependencies of s_{k_1,v_1} becomes fixed when the state is updated to s_{k_1,v_3}. This is because when the transaction that outputs s_{k_0,v_4} is executed, it reads s_{k_1,v_3} instead of s_{k_1,v_1}.

In *FabricSharp*, for each state $s_{k,v}$ at its latest version, we buffer a list of forward pointers to the entries whose dependencies include $s_{k,v}$. We refer to this list as $F_{s_{k,v}}$, which is defined more precisely as follows:

$$F_{s_{k,v}} = \{hash(E_{s'}) | s_{k,v} \in dep(s')\}$$

When the state is updated to $s_{k,v'}$ for $v' > v$, we store $F_{s_{k,v}}$ at the entry of $s_{k,v'}$.

4.5 Efficient Provenance Queries

The Merkle DAG structure supports efficient access to the latest state version, since the state index at block b contains pointers to all the latest versions at this block. To read the latest version of s, one simply reads χ_b, follows the index to the entry for s, and then reads the state value from the entry. However, querying an arbitrary version in the DAG is inefficient, because one has to start at the DAG head and traverse along the edges towards the requested version. Supporting fast version queries is important when the user wants to examine the state history only from a specific version (for auditing purposes, for example). It is also important for provenance-dependent smart contracts because such queries directly affect contract execution time.

In this section, we describe a novel index that facilitates fast version queries. The index is designed for permissioned blockchains. We discuss its efficiency and how to extend it to permissionless blockchains.

4.5.1 Deterministic Append-Only Skip List

We propose to build an index on top of a state DAG to enable fast version queries. The index has a skip list structure, which we call Deterministic Append-only Skip List (or DASL). It is designed for blockchains, exploiting the fact that the blockchain is append-only, and randomness is not well supported [30]. More specifically, a DASL has two distinct properties compared to a normal skip list. First, it is append-only. The index keys of the appended items, which are versions in our case, are strictly increasing. Second, it is deterministic, that is, the index structure is uniquely determined by the values of the appended items, unlike a stochastic skip list. For ease of explanation, we assume that version numbers are positive integers.

Definition 6. *Let $V_k = \langle v_0, v_1, ...\rangle$ be the sequence of version numbers of states with identifier k, in which $v_i < v_j$ for all $i < j$. A DASL index for k consists of N linked lists $L_0, L_1, .., L_{N-1}$. Let v^i_{j-1} and v^i_j be the versions in the $(j-1)^{th}$ and j^{th} node of list L_i. Let b be the base number, a system-wide parameter. The content of L_i is constructed as follows:*

1) $v_0 \in L_i$

```
struct Node {
  Version v;
  Value val;
  List<Version> pre_versions;
  List<Node*> pre_nodes;
}
```

Figure 4.10: Node structure that captures a state $s_{k,v}$ with value *val*

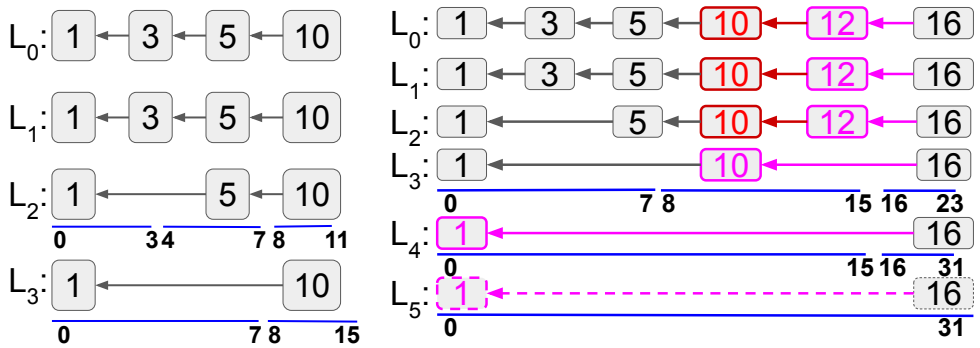


Figure 4.11: An example of the append procedure in Deterministic Append-only Skip List (DASL)

(a) A DASL containing versions 1, 3, 5 and 10. The base b is 2. The intervals for L_2 and L_3 are shown in blue lines. (b) The new DASL after appending version 12 and 16. L_4 is created when appending version 16. L_5 is created, then discarded.

2) Given v^i_{j-1}, v^i_j is the smallest version in V_k such that:

$$\left\lfloor \frac{v^i_{j-1}}{b^i} \right\rfloor < \left\lfloor \frac{v^i_j}{b^i} \right\rfloor \tag{4.5}$$

Figure 4.10 shows how DASL is stored with the state in a data structure called Node. This structure (also referred to as node) consists of the state version and value. A node belongs to multiple lists (or levels), hence it maintains a list of pointers to some version numbers, and another list of pointers to other nodes. Both lists are of size N, and the i^{th} entry of a list points to the previous version (or the previous node) of this node in level L_i. For the same key, the version number uniquely identifies the node, hence we use version numbers to refer to the corresponding nodes.

We can view a list as consisting of continuous, non-overlapping intervals of certain sizes. In particular, the j^{th} interval of L_i represents the range $R^i_j = [jb^i, (j+1)b^i)$. Only the smallest version in V_k that falls in this range is included in the list.

Figure 4.11a gives an example of a DASL structure with $b = 2$. It can be seen that when the version numbers are sparsely distributed, the lists at lower levels are identical. In this case, b can be increased to create larger intervals which can reduce the overlapping among lower-level lists.

A DASL and a skip list share two properties. First, if a version number appears in L_i, it also appears in L_j where $j < i$. Second, with $b = 2$, suppose the last level that a version appears in is i, then this version's preceding neighbour in L_i appears in L_j where $j > i$. Given these properties, a query for a version in the DASL is executed in the same way as in the skip list. More specifically, the query traverses a high-level list as much as possible, starting from the last version in the last list. It moves to a lower level only if the preceding version in the current list is strictly smaller than the requested version. In DASL, the query for version v_q returns the largest version $v \in V_k$ such that $v \leq v_q$ (the inequality occurs when v_q does not exist). This result represents the value of the state which is visible at time of v_q.

We now describe how a new node is appended to DASL. The challenge is to determine the lists that should include the new node. Algorithm 2 details the steps that find the lists, and subsequently the previous versions, of the new node. The key idea is to start from the last node in L_0, then keep increasing the list level until the current node and the new node belong to the same interval (line 9 - 18). Figure 4.11b shows the result of appending a node with version 12 to the original DASL. The algorithm starts at node 10 and moves up to list L_1 and L_2. It stops at L_3 because in this level node 10 and 12 belong to the same interval, i.e., $[8, 16)$. Thus, the new node is appended to list L_0 to L_2. When the algorithm reaches the last level and is still able to append, it creates a new level where node 0 is the first entry and repeats the process (line 21 - 24). In Figure 4.11b, when appending version 16, all existing lists can be used. The algorithm then creates L_4 with node 1 and appends the node 16 to it. It also creates level L_5, but then discards it because node 16 will not be appended since it belongs to same interval of $[0, 32)$ with node 1.

4.5.2 Discussion

Integrating to Merkle DAG. The DASL is integrated to the Merkle DAG as follows. The node structure (Figure 4.10) is stored in the state entry (Definition 3).

Algorithm 2: DASL Append

Input: version v and last node *last*
Output: previous versions and nodes

```
1: level=0; // list level
2: pre_versions = [];
3: pre_nodes = [];
4: finish = false ;
5: cur = last ;
6: while not finish do
7:     l = cur->pre_versions.size() ;
8:     if l > 0 then
9:         for j=level; j<l; ++j do
10:            if cur->version / b^j < v / b^j then
11:                pre_versions.append(cur->version);
12:                pre_nodes.append(cur);
13:            else
14:                finish = true;
15:                break;
16:        if not finish then
17:            cur = cur->pre_versions[l-1];
18:            level = l
19:    else
           /* We have reached the last level                    */
20:        finish = true;
21:        while cur->version / b^level < v / b^level do
22:            ++level;
23:            pre_versions.append(cur->version);
24:            pre_nodes.append(cur);
25: return pre_version, pre_nodes;
```

The node pointers are implemented entry hashes. The Merkle tree structure remains unchanged.

Speed-storage tradeoff. As a skip list variant, DASL shares the same lineage space complexity and logarithmic query time complexity. Suppose there are v^* number of versions and the base of DASL is b. The maximum number of required pointers is $\frac{bv^*-1}{b-1}$. (There are at most $\lceil \log_b v^* \rceil$ levels and the i-th level takes at most $\lceil \frac{v^*}{b^i} \rceil - 1$ pointers.) Suppose the queried version is v^q and the query distance $d = v^* - v^q$, the maximum number of hops in such query is capped at $2b\lceil \log_b d \rceil$. (A

query traversal from the end will undergo two stages, one stage towards lower levels and the other stage back towards upper levels. In each stage, the traversal will take at most b hops on the same list before moving to the next level. And there are at most $\lceil \log_b d \rceil$ levels to traverse.) Hence, b controls the tradeoff between the space overhead and query delay. One benefit of this property is that DASL queries favor more recent versions, i.e. d are small. It is useful for smart contracts that work on recent rather than old versions. Another benefit is that the performance of such recent-version queries does not change as the state history grows.

Extending to permissionless blockchains. We note that DASL incurs storage overhead. The version query also incurs some computation cost, even though it is more efficient with DASL than without it. These costs may be small enough such that they do not affect the performance of a permissioned blockchain, as we demonstrate in Chapter 4.7. However, they need to be carefully managed in a permissionless blockchain where any overhead directly translates to monetary cost to the miners. In particular, any additional cost to the miner triggers the Verifier Dilemma [107] and compromises the incentive mechanism of the blockchain.

A malicious user could issue a transaction that references a very old version. Reading earlier versions is more expensive, because there are more hops involved. This overhead is born by all nodes in the network, since every node in the network has to execute the same transaction. Current public blockchains prevent such denial-of-service attack by explicitly charging a fee for each operation in the transaction. In Ethereum, the transaction owner pays for the resource consumption in *gas*. A transaction that writes more data or consumes more CPUs has to pay more *gas*. Thus, rational users are deterred from running too complex transactions on the blockchain.

As DASL consumes resources, its costs must be explicitly accounted for in permissionless blockchains. More specifically, during deployment, the contract owner specifies which states require DASL support. Alternatively, DASL support can be automatically inferred from the contract's source code. The deployment fee should reflect this extra storage cost for DASL. If the fee is too high, the owner can lower it by increasing b. During contract execution, the execution engine must charge the cost of DASL queries to the transaction fee. In particular, a query that requires more hops to find the requested version incurs a higher transaction fee. Users

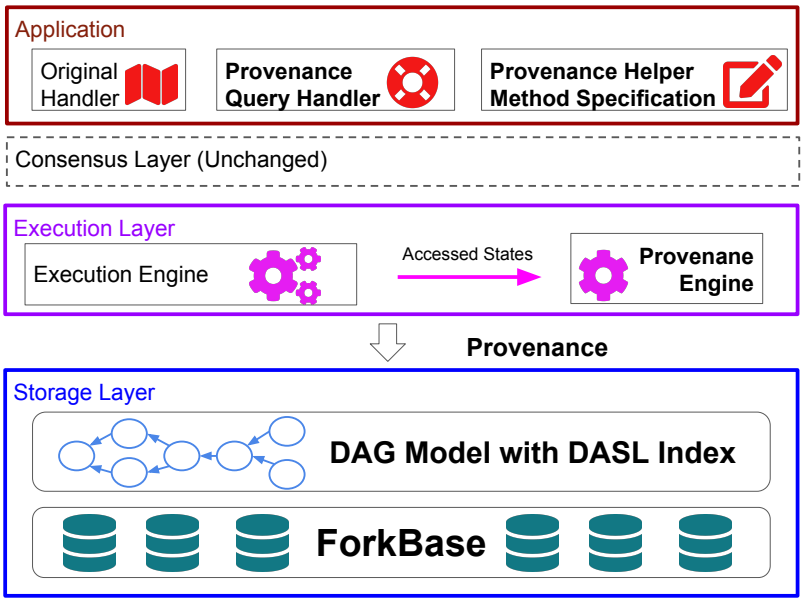


Figure 4.13: The *FabricSharp*'s instrumentation on Fabric to support data provenance.

The original storage layer is replaced with the implementation that supports fine-grained provenance. The original execution layer is instrumented with a provenance capture engine. The application layer contains the new helper method and provenance query APIs. The consensus layer is unchanged.

may empirically estimate the hops (as well as the cost) based on the above-derived theoretical upper bound.

4.6 Implementation

In this section, we present our implementation of *FabricSharp* based on Hyperledger Fabric v2.2. Figure 4.13 highlights our changes in a layer-wise fashion. In particular, we completely replace the storage layer with our implementation of the Merkle DAG and DASL index. This new storage is built on top of ForkBase [163], a state-of-the-art blockchain storage system with support for version tracking. We instrument the original execution engine to record read and write sets during contract execution. At the application layer we add a new helper method and three provenance APIs. The execution engine is modified to invoke the helper method after every successful contract execution.

```
// Update the value for key in a branch with the metadata
//   and return a tamper-evident vid.
// Value can be a primitive like a string, or more complex types like map, list, etc.
vid <- Put(key, branch, value, metadata);
// Retrieve the latest value for a key on a branch
value <- GetLastValue(key, branch);
// Retrieve the value or the metadata for a key based on vid.
value <- GetValue(key, vid);
meta <- GetMetadata(key, vid);
```

Figure 4.15: ForkBase APIs to implement the *FabricSharp*'s storage.

(a) *metadata* is a user provided string that describes the updated value. And *vid* is uniquely determined by the updated value, key, branch and the metedata.

4.6.1 Storage Layer

Instead of implementing the storage layer from scratch, we leverage ForkBase for its support of version tracking. We exploit three properties of ForkBase in *FabricSharp*. The first is the fork semantics, with which the application can specify a *branch* for the update. Given a branch, ForkBase provides access to the latest value. The second property is tamper evidence, in which every update returns a tamper-evident identifier *vid* which captures the entire evolution history of the updated value. A data object in ForkBase is uniquely identified by the key and *vid*. In *FabricSharp*, *vid* is used as the entry hash in the DAG, and as the pointer in DASL. The third property is the rich set of built-in data types including map, list and set. Figure 4.16a lists our utilized ForkBase APIs with their explanations.

Algorithm 3 details our implementation for updating states and computing the global state digest when a block is being committed. The update function is invoked with a new state and a list of dependencies. We first prepare the list of backward pointers by retrieving the latest *vids* of the dependent states (line 5-8). Next, we build the pointers for the DASL index, then retrieve the forward pointers of the previous state (line 9). The metadata from these steps are serialized and stored together with the updated value in ForkBase (line 14-16). The result is a new *vid* for the update, which is appended to the list of forward pointers of every dependent state (line 18-20). *vid* is now the latest version (line 21).

The global states are stored in a map object in ForkBase. To compute the global state digest, we simply update the map object with the new *vids* computed for this

block. The update operation of the map object, which is built as a Merkle tree in ForkBase, returns a digest *latest_vid* which is then included to the block header. This digest provides tamper evidence for the evolution histories of all the states up to the current block. Given a *key*, the query on its latest value can be directly answered by *fb.GetLastValue(key, branch="default")*. For the historical value and dependency, one must locate the corresponding *vid* for that entry. DASL facilitates this query process. Then one shall use *vid* to fetch for the value and metadata with the corresponding ForkBase APIs.

4.6.2 Application and Execution Layer

In *FabricSharp*, users write their smart contracts by implementing the `Chaincode` interface. Given any `Chaincode` implementations, the execution engine triggers the `Init` and `Invoke` method during deployment and invocation respectively. Both methods take as input an instance of `ChaincodeStubInterface` which supplies relevant context, such as access to the ledger states, to the smart contract.

We add the helper method, called `ProvHelper`, to the `Chaincode` interface. This method's signature, and how to write user-defined provenance rules with it, are explained in Chapter 4.3. The execution engine intercepts `PutState` and `GetStates` during execution to record the read set and the write set. It invokes `ProvHelper` when the execution finishes. Then it piggybacks the computed dependency along with the transaction, later to be stored securely when the transaction commits. The three new provenance APIs, namely `Hist`, `Backward` and `Forward`, are added to `ChaincodeStubInterface` and therefore previously-stored provenance is accessible to all contract methods.

4.7 Performance Evaluation

Algorithm 3: Provenance update and digest Computation

```
   /* fb is the shorthand for ForkBase. fb.Map is the built-in map data type
      in Forkbase.                                                          */
 1: fb.Map<id,vid> latest;
 2: String branch = "default";
   // Buffered forward pointers
 3: Map<id, List<vid» forward;
   Input: id, version, value of the updated state
   Input: ids of the dependent states, dep_ids
 4: Function Update(id, version, value, dep_ids):
      /* Backward pointers                                                 */
 5:   List<vid> back_vids;
 6:   for dep_id in dep_ids do
 7:     back_vid = latest[dep_id];
 8:     back_vids.push_back(back_vid);
      /* Forward pointers                                                  */
 9:   forward_vids = forward[id];
10:
      /* Retrieve pointer to last DASL node                                */
11:   last_vid = latest[id];
      /* Refer DASL Append in Algorithm 2                                  */
12:   pre_versions, pre_vids = DaslAppend(version, last_vid);
13:   node = new DaslNode{version, pre_versions, pre_vids} ;
14:   meta = Serialize(back_vids, forward_vids, node) ;
15:
      /* Store the updated value                                           */
16:   new_vid = fb.Put(id, branch, value, meta);
17:
      /* Update forward pointers                                           */
18:   for dep_id in dep_ids do
19:     forward[dep_id].push_back(new_vid);
20:   forward[id].Clear();
21:   latest[id] = new_vid;
22:
   Output: The state digest for the committed block
23: Function ComputeDigest():
24:   latest_vid = fb.Put("state", branch, latest, nil);
25:   return latest_vid;
```

Chapter 5

A Transactional Perspective on Execute-order-validate Blockchains

5.1 Introduction

From Chapter 3, blockchains systems can be classified into *permissionless* (*public*), such as Bitcoin and Ethereum, and *permissioned* (*private*), such as Hyperledger Fabric [9]. In public blockchains, the data and transactional logic are transparent to the public, hence, are subject to private data leakage. Due to their openness, public blockchains use expensive PoW consensus. This, together with the serial transaction execution limit these systems' capacity. Addressing the limitations of public blockchains, Hyperledger Fabric is a private blockchain that supports *concurrent* transactions [9]. A Fabric blockchain requires its members to enroll through a trusted membership service in order to interact with the blockchain. In this paper, we focus on permissioned blockchains as they are more suitable for supporting applications such as supply-chain, healthcare and resource sharing, and in particular, we use Fabric as the underlying blockchain system.

Fabric supports a new transaction execution architecture called execute-order-validate (EOV). In this architecture, a transaction's lifecycle consists of three phases. In the first phase, *execution*, a client sends the transaction to a set of nodes, or peers, specified by an endorsement policy. The transaction is executed by these peers in parallel and its effects in terms of read and written states are recorded. Moreover, transactions from different clients may be parallelized during the execution. In the second phase, *ordering*, a consensus protocol is used to produce a totally ordered sequence of endorsed transactions grouped in blocks. This order is broadcast to all

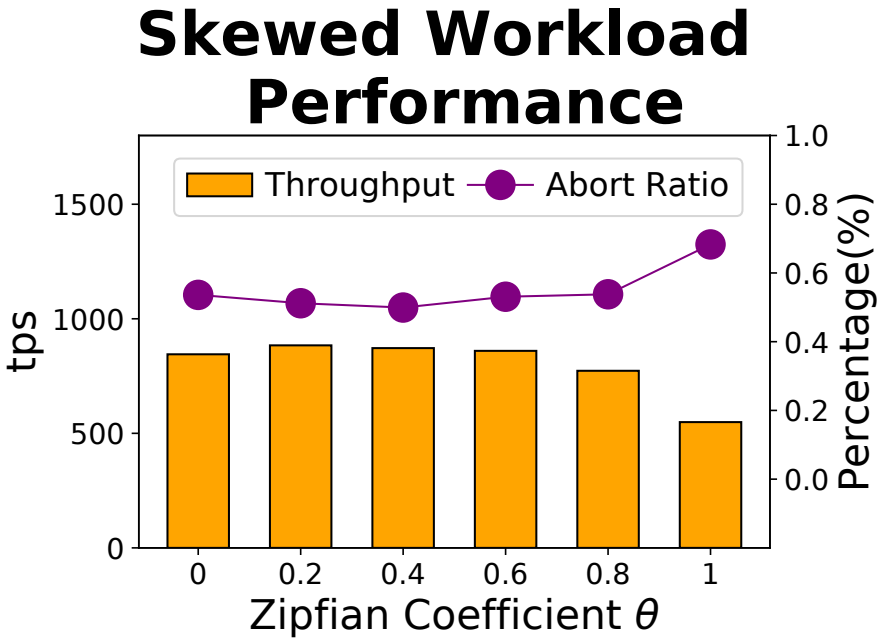


Figure 5.1: Fabric's raw and effective throughput under both no-op transactions and single modification transactions with varying skewness

peers. In the third phase, *validation*, each peer validates the state changes from the endorsed transactions with respect to the endorsement policy and serializability.

The new EOV architecture limits the execution details of a transaction to the endorsing peers to enhance confidentiality and exploit concurrency. But such concurrency comes at the cost of aborting transactions that do not abide serializability. We evaluate the impact of concurrency control in Fabric on the setup described in Chapter 5.5. We measure both the raw and effective peak throughputs under both no-op transactions, with no data access, and update transactions, with varying skewness controlled by the zipfian coefficient. The raw throughput represents the in-ledger transaction rate, while the effective throughput represents committed transactions by excluding the aborted transactions from raw throughput. In Chapter 5.1, a bar reports the raw throughput, while its blue part reports the effective throughput. The raw throughput is constant (xxx tps) despite the workload type and request skewness. But with higher skewness, a larger proportion of transactions are aborted for serializability.

There are two notable directions attempting to solve this issue. The first is to improve upon Fabric's architecture to enhance its attainable throughput [119,

155]. For example, FastFabric proposes to split a node's functionality to alleviate the bottleneck and achieves the highest throughput among all improvements of Fabric [69]. However, these approaches are implementation-specific and might not generalize well to other blockchains. The second direction is to abstract out the transaction lifecycle to reduce abort rate. For example, Fabric++ [148] uses well-established concurrency techniques from databases to early abort transactions or reorder them to reconcile the potential conflicts.

Our work corresponds to the second direction, as a major attempt to *databasify* blockchains. Here, we take a principled approach to learn from transactional analysis techniques in databases with optimistic concurrency control (OCC) and apply them to enhance transaction processing in blockchains. We formally analyze the behavior of the current implementations of Fabric, and discover that both achieve *Strong Serializability* [16] (as described in Chapter 5.3.2). In fact, these implementations are more stringent than *One-Copy Serializability* (or simply Serializability), as prescribed by the original Fabric protocol [9]. Both systems employ a preventive approach which might over-abort transactions that are still serializable. In contrast, our proposal consists of a novel reordering technique that eliminates unnecessary abort due to in-ledger conflicts, with the serializability guarantee established on our theoretical insights. Our approach does not change Fabric's architecture, therefore it is orthogonal to the aforementioned optimizations, such as FastFabric [69].

In summary, our paper makes the following contributions:

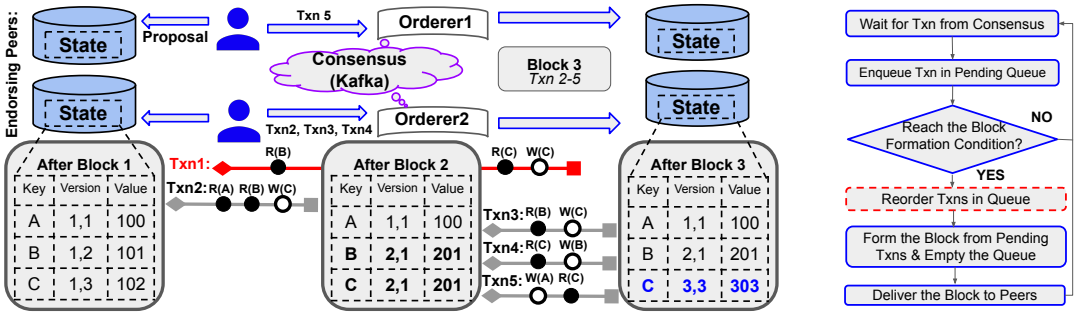
- We theoretically analyze the resemblance of transaction processing in blockchains with EOV architecture and databases with optimistic concurrency control (Chapter 5.3.1). Based on this resemblance, we analyze the transactional behavior of state-of-the-art EOV blockchains, such as Fabric and Fabric++ (Chapter 5.3.2).

- We propose a novel theorem to identify transactions that can never be reordered for serializability (Chapter 5.3.3). Based on this theorem, we propose efficient algorithms to early filter out such transactions (Chapter 5.3.4), with the serializability guarantee for the remaining after reordering. We also discuss the security implications of our proposal (Chapter 5.3.5).

- We implement our proposed algorithms on top of *FabricSharp* and extensively evaluate *FabricSharp* by comparing it with the vanilla Fabric, Fabric++, and two other implementations based on database concurrency control techniques from one standard approach [31] and a recent proposal by Ding et al [49]. The experimental results show that the throughput of *FabricSharp* is more than 25% higher compared to the other systems.

The remaining of this paper is structured as follows. Chapter 5.2 provides background on EOV blockchains and OCC techniques. Our theoretical analysis follows in Chapter 5.3, ending with our reordering algorithm. Chapter 5.4 describes the implementation of our approach. Chapter 5.5 reports our experimental results.

5.2 Background



(a) (b)

Figure 5.2: Background in Execute-order-validate architecture and Fabric++'s instrumentation.

(a) Example of transaction workflow in Fabric. An arrow represents the lifespan of a transaction's execution (simulation), e.g, Txn1 starts its execution immediately after block 1 and finishs its simulation after block 2. (b) Procedures replicated in each Fabric Orderer. Fabric++ introduces a reordering step before the block formation to reduce the transaction abort rate.

5.2.1 EOV architecture in Fabric and Fabric++

Hyperledger Fabric [9] is a state-of-the-art permissioned blockchain that features a modular design based on the EOV architecture. Fabric++ [148] is an optimization of Fabric, which reorders transactions after consensus to reduce the abort rate. A

Table 5.1: The transaction summary in Figure 5.2.

		Txn1	**Txn2**	**Txn3**	**Txn4**	**Txn5**
Readset	Key	B C	A **B**	B	**C**	**C**
	Version	1,2 2,1	1,1 **1,2**	2,1	**2,1**	**2,1**
Writeset	Key	C	C	**C**	B	A
	Value	301	302	**303**	304	305
Commit status	Fabric	**N.A.**	x	✓	x	x
	Fabric++	x	x	x	✓	✓

Staled reads and installed writes are marked in red and blue colors. The symbols ✓, x, **N.A.** respectively indicate committed, aborted, or not-allowed transactions.

Fabric/Fabric++ blockchain is run by a set of authenticated nodes, whose identity is provided by a membership service. A node in this blockchain has one of the following three roles: (i) *client* which submits a transaction proposal for execution, (ii) *peer* which *executes* and *validates* transaction proposals, or (iii) *orderer* which *orders* transactions and batches them in blocks. Transaction order is determined collectively by all orderers in the blockchain based on a consensus protocol.

The state of a blockchain after forming a block is maintained by a versioned key-value store. Each entry in this store is a tuple (`key`, `ver`, `val`), where `key` is a unique name representing the entry, and `ver` and `val` are the entry's latest version and value, respectively. Moreover, `ver` is a pair consisting of the sequence number of the block and the transaction that updated the entry. For example, in Figure 5.2a, the entry $(\texttt{C}, (2, 1), 201)$ in the state after block 2 indicates that the key `C` contains the latest value 201 which was lastly updated by the 1st transaction in block 2.

In Fabric/Fabric++, the workflow of a transaction consists of three phases: execution, ordering, and validation. We elaborate on these phases below, using the example in Figure 5.2a.

Execution. In this phase, clients propose transactions consisting of smart contract invocations to a set of endorsing peers, which are selected by an endorsement policy. Each endorsing peer executes transaction proposals concurrently and speculatively and returns the simulation results together with its endorsement signature. The results contain two value sets called the *readset* and the *writeset* which respectively represent the version dependencies (all keys read along with their version numbers) and the state updates (all keys modified along with their new values) produced by the simulation. For example, the readset and writeset of transactions in

Figure 5.2a are summarized in Table 5.1. Throughout the execution, a transaction holds a read lock on the state database to guarantee that read values are the latest. Transactions that read across blocks, such as `Txn1` in Figure 5.2a, are not allowed in Fabric. In contrast, Fabric++ optimistically removes this lock for more parallelism but aborts transactions that read across blocks. After a client collects enough identical simulation results as required by the endorsement policy, it packages them into a single transaction and submits it to orderers.

Ordering. In this second phase, orderers receive transactions and sequence them into a total order to form a block, as shown in Figure 5.2b. Each orderer may belong to different administrative domains and receive different transaction proposals from various clients. But all orderers rely on a single consensus protocol to establish a common transaction order. Fabric/Fabric++ outsources this consensus service to Kafka. With the consistent transaction stream from the consensus, each orderer employs the same block formation protocol to batch transactions into blocks, and consequently delivers them to peers. A block is formed when the number of pending transactions reach the threshold or a timeout triggers. For example, in Figure 5.2a, Orderer1 receives `Txn5` and Orderer2 receives `Txn2`, `Txn3`, and `Txn4`. They send the transactions to the consensus service and receive the same transaction order. Based on this order, both orderers package the transactions into identical blocks, i.e., block 3 with `Txn2` to `Txn5`, given that the protocol limits the maximum number of transactions per block to 4.

Validation. This phase is executed by each peer after a block has been retrieved from orderers. Transactions in a block are sequentially validated based on the corresponding endorsement policy and transaction serializability. The serializability of a transaction is tested by inspecting the staleness of its readset. The transaction is marked as invalid if it reads a key whose version at the read time is inconsistent (or older) than the latest version. For example, in Figure 5.2a, transaction `Txn2` in block 2 is unserializable since it reads key `B` with version $(1, 2)$ from block 1, which is inconsistent with the latest version $(2, 1)$ in block 2. Suppose that `Txn3` passes the serializability test and updates the version of key `C` from $(2, 1)$ to $(3, 3)$ in block 3. Then, transactions `Txn4` and `Txn5` become invalid, since they both read an inconsistent version of key `C` in block 2. Hence, after this validation phase, only transaction `Txn3` in block 3 is committed, while transactions `Txn2`, `Txn4`, `Txn5` are

aborted. To satisfy the serializability constraint, Fabric++ introduces a reordering step immediately before block formation but after consensus. The reordering is based on the commit order determined by the consensus and the accessed records in the transactions. For example, each orderer in Fabric++ puts `Txn3` behind `Txn4` and `Txn5`. Then, `Txn4` and `Txn5` are committed while `Txn3` is aborted. Hence, Fabric++ commits one more transaction than Fabric.

5.2.2 Optimistic Concurrency Control in Databases

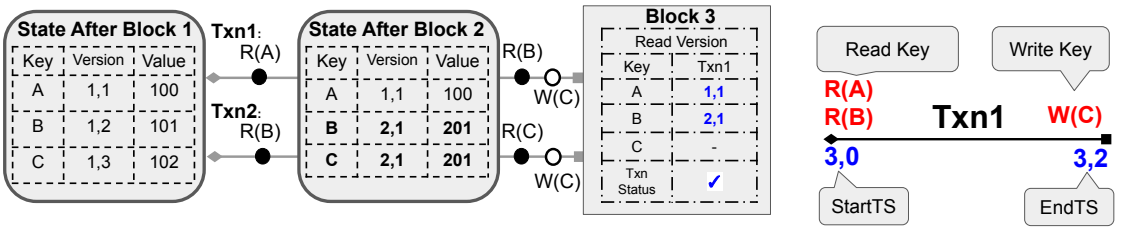
Unlike pessimistic concurrency control, the OCC technique does not hold locks to regulate transactional interference. Instead, each transaction has a unique *start timestamp* assigned to it from a global atomic clock. All queries reflect the state snapshot of the database at the start timestamp, without observing later changes. Each transaction is also assigned an *end timestamp*. Before committing, the database system checks the validity of a transaction based on these two timestamps and the accessed records. OCC can easily achieve *Snapshot Isolation*, which disallows concurrent transactions updating the same key [22]. Considering the fact that Snapshot Isolation suffers anomalies such as Lost Update and Write Skew, a number of attempts have been made to transform Snapshot Isolation to Serializable level [59, 175, 26].

5.3 Theoretical Analysis

In this section, we first describe the resemblance of transaction processing techniques in EOV blockchains and OCC databases. Then, we use the transactional analysis method of OCC databases to reason about the serializability behavior of EOV blockchains, such as Fabric and Fabric++. Finally, we propose a reordering-based concurrency control algorithm for ordering serializable transactions in EOV blockchains, along with the discussion on its security implications.

5.3.1 Resemblance in Transaction Processing

Similar to database systems where the concept *database snapshot* is used to describe a read-only, static view of a database [94], in blockchains, we can define the similar concept of *blockchain snapshot* as follows.



(a) Txn1, which reads across blocks, is snapshot consistent and can be scheduled with serializability. Txn2 is not as its early-read key B is updated before its execution ends.

(b) Concise notation to represent a transaction

Figure 5.4: An example of transactions reading across blocks

Definition 7 (Blockchain snapshot)**.** *A blockchain snapshot is the state of a blockchain after a block is committed. Let M be the sequence number of the committed block, then the corresponding snapshot is denoted as M and is said to have the sequence number $(M{+}1, 0)$* [1]*.*

Definition 8 (Snapshot consistency)**.** *A transaction is snapshot consistent if there exists a blockchain snapshot M from which all the transaction's records are read.*

Transactions in Fabric satisfy snapshot consistency since Fabric uses a lock to ensure the simulation is done against the latest state. Fabric++ optimistically removes the lock but early aborts transactions which read across blocks. Hence, it also satisfies the snapshot consistency. However, eliminating transactions based on cross-block reading might lead to over-aborting snapshot consistent transactions.

Example 1. *In Figure 5.4a, `Txn1` reads key `A` of version $(1, 1)$ in snapshot 1 and key `B` of version $(2, 1)$ in snapshot 2. These versions are the same as the versions of keys `A` and `B` in snapshot 2. Hence, `Txn1` is snapshot consistent with block snapshot 2. In contrast, transaction `Txn2`, which also reads across blocks, does not achieve snapshot consistency because the value of previously read key B changes in block 2.*

Proposition 1. *There exist snapshot-consistent transactions that read across blocks. For such a transaction, its block snapshot is determined by its last read operation.*

[1] We use the two-value tuple with 0 fixed for the second element. This is to facilitate the ordering relations $<$ of sequence numbers of blockchain snapshots and transaction timestamps.

Proof. Txn1 in Figure 5.4a is a witness example. We have described in Example 1 that Txn1 reads across blocks 1 and 2, and it is still consistent with block snapshot 2. □

Proposition 1 shows that a legitimate transaction in an EOV blockchain can read across blocks, if their states are consistent. This makes the EOV blockchain similar to an OCC database, as the latter also reads from consistent states determined by the transaction's start timestamp. We also observe that the blockchain's sequence numbers have similar properties with databases' timestamps, such as atomicity, monotony, total order, and unique mapping to snapshots. Therefore, we define the timestamps of blockchain transactions using their sequence numbers.

Definition 9 (Start timestamp)**.** *The start timestamp of transaction* Txn*, denoted by* StartTs(Txn)*, is the sequence number of its read snapshot.*

Definition 10 (End timestamp)**.** *The end timestamp of transaction* Txn*, denoted by* EndTs(Txn)*, is its sequence number in the block, determined by the consensus.*

For example, in Figure 5.4a, Txn1 has $\mathsf{StartTs}(\mathsf{Txn1}) = (3, 0)$ and $\mathsf{EndTs}(\mathsf{Txn1}) = (3, 1)$, since it lastly reads from block 2 and occupies the first position in block 3. For brevity, in later paragraphs, we use the notation presented in Figure 5.4b to denote a transaction. Moreover, the sequence numbers of transactions' start or end timestamps are lexicographically ordered, e.g., (2,1) < (2,2) = (2,2) < (3,0).

Definition 11 (Concurrent transactions)**.** *Two transactions* Txn1 *and* Txn2 *are said to be concurrent if their executions overlap. To be specific, if* Txn1 *ends earlier than* Txn2 *(i.e.,* $\mathsf{EndTs}(\mathsf{Txn1}) < \mathsf{EndTs}(\mathsf{Txn2})$*), then* Txn2 *must start before* Txn1 *ends (i.e.,* $\mathsf{StartTs}(\mathsf{Txn2}) < \mathsf{EndTs}(\mathsf{Txn1})$*). Otherwise, if* Txn2 *ends earlier than* Txn1 *(i.e.,* $\mathsf{EndTs}(\mathsf{Txn2}) < \mathsf{EndTs}(\mathsf{Txn1})$*), then* Txn1 *must start before* Txn2 *ends (i.e.,* $\mathsf{StartTs}(\mathsf{Txn1}) < \mathsf{EndTs}(\mathsf{Txn2})$*).*

Proposition 2. *Each pair of transactions in the same block are concurrent.*

Proof. Suppose two transactions Txn1 and Txn2 are committed in the same block M at position p and q, respectively, where $p < q$. Since the latest block that Txn2 can read from is $M-1$, we have that: $\mathsf{StartTs}(\mathsf{Txn2}) \leq (M, 0) < \mathsf{EndTs}(\mathsf{Txn1}) = (M, p) < \mathsf{EndTs}(\mathsf{Txn2}) = (M, q)$. Hence, Txn1 and Txn2 are concurrent. □

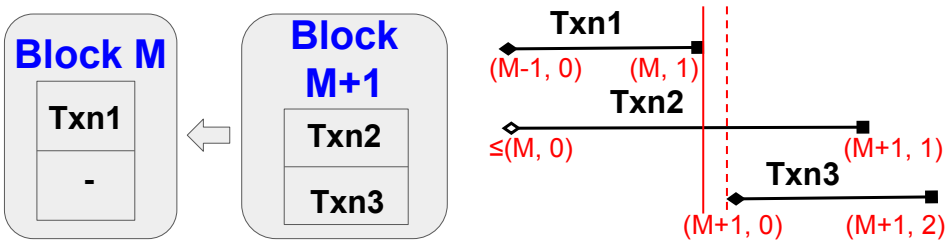


Figure 5.5: Concurrency of transactions within and across blocks

`Txn2` and `Txn3` are in the same block and concurrent. `Txn1` and `Txn2` are in different blocks, but they are still concurrent. `Txn1` and `Txn3` are not concurrent.

Proposition 3. *The reverse of Proposition 2 is not true: there are concurrent transactions not belonging to the same block.*

Proof. We present a witness example in Figure 5.5, where transactions `Txn1` and `Txn2` respectively belong to block M and $M{+}1$. However, `Txn2` reads from a block earlier than M. Hence, we have: $\mathsf{StartTs}(\texttt{Txn2}) \leq (M, 0) < \mathsf{EndTs}(\texttt{Txn1}) = (M, 1) < \mathsf{EndTs}(\texttt{Txn2}) = (M{+}1, 1)$. Therefore, `Txn1` and `Txn2` are concurrent. □

From the above two propositions, concurrency does not only occur between transactions within the same block. Fabric++ fails to consider dependencies among transactions across blocks. Hence, its reordering effect is limited.

5.3.2 Serializability Analysis

Figure 5.7 shows all six scenarios of canonical transaction dependency (or conflict) between snapshot transactions, as described by [59]. Among them, three dependencies, namely *n-ww*, *n-wr*, and *n-rw* are between non-concurrent transactions. The other three dependencies, namely *c-ww*, *c-rw*, and *anti-rw* are between concurrent transactions. According to the conflict serializability theorem in [167], the effect of a serializable transaction schedule is equivalent to any serialized transaction history that respects dependency order. Note that the dependency graph of the serializable transaction schedule must be acyclic.

Definition 12 (Strong Serializability)**.** *A schedule of transactions is Strong Serializable if its effect is equivalent to the serialized history, which conforms to the transactions' commit order determined by their end timestamps.*

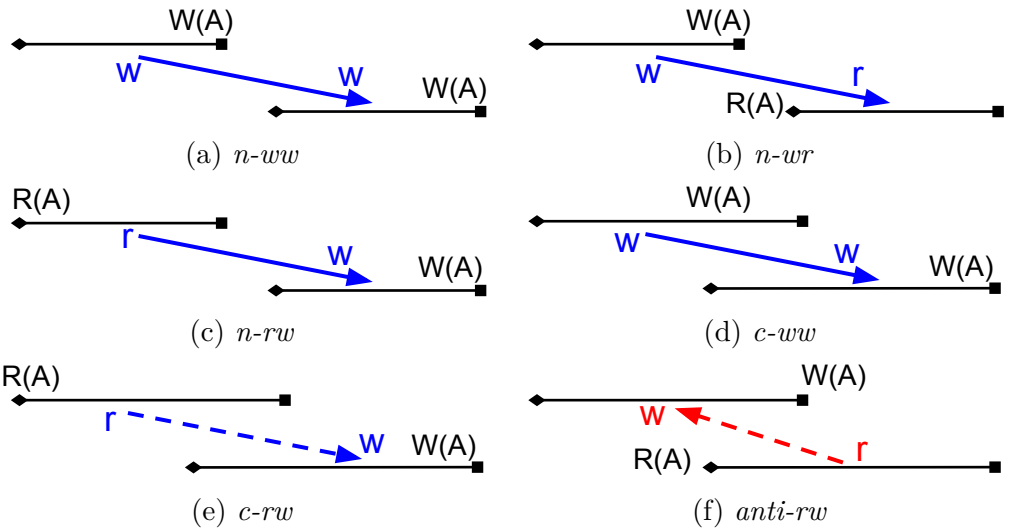


Figure 5.7: Six canonical dependencies between snapshot transactions.
(a), (b), and (c) are non-concurrent and (d), (e), and (f) are concurrent.

Theorem 1. *A schedule of transactions without anti-rw achieves Strong Serializability.*

Proof. We first prove that any transaction schedule without *anti-rw* achieves Serializability. By contradiction, suppose that such a transaction schedule does not achieve Serializability. Then, in the schedule there must be a subset of transactions with a dependency cycle, in which the last committed transaction is denoted by `Txn`. Then `Txn` must exhibit an *anti-rw* dependency because *anti-rw* is the only one among all six dependencies that relates later transactions to earlier ones. But this contradicts our assumption. Hence, the transaction schedule is serializable. Next, we prove that it also achieves Strong Serializability. Since the order of the five remaining dependencies is consistent to their commit order, the serialized history that respects the commit order also respects the dependency order. According to the conflict serializability theorem in [167], this serialized transaction history has the equivalent effect of the serializable schedule. Hence, the transaction schedule is Strong Serializable. □

We remark that Fabric/Fabric++ do not allow *anti-rw* between two transactions because the latter transaction would read an old version of the updated key, hence, it must be aborted. Based on Theorem 1, transactions in Fabric/Fabric++ satisfy

Strong Serializability, which is more stringent than Serializability [9]. This opens up the opportunity to reduce the transaction abort rate.

5.3.3 Reorderability Analysis

Under Serializability instead of Strong Serializability, we formally analyze the reorderability of transactions in EOV blockchains. We focus on determining a serializable schedule by switching the commit order of pending transactions.

Lemma 1. *In blockchains, reordering can only happen between concurrent transactions.*

Proof. Assume transaction reordering occurs between two non-concurrent transactions. These transactions are committed in different blocks, due to the contra-positive of Proposition 2. Switching their order means changing a previously committed block, which is impossible in blockchains due to their immutability. □

Lemma 2. *A transaction does not change its concurrency relationship with respect to others after reordering.*

Proof. Assume the next block's sequence number is M. For any pending transaction Txn, we have: $\mathsf{StartTs}(\mathsf{Txn}) \leq (M, 0) < \mathsf{EndTs}(\mathsf{Txn})$. Other transactions are classified into three cases. (i) For any non-concurrent transaction $\mathsf{Txn1}$, we have: $\mathsf{EndTs}(\mathsf{Txn1}) < \mathsf{StartTs}(\mathsf{Txn})$. Since reordering does not affect $\mathsf{StartTs}(\mathsf{Txn})$, the non-concurrency between Txn and $\mathsf{Txn1}$ still holds. (ii) For any concurrent transaction $\mathsf{Txn2}$ committed earlier than block M, we have: $\mathsf{StartTs}(\mathsf{Txn}) < \mathsf{EndTs}(\mathsf{Txn2}) < (M, 0) < \mathsf{EndTs}(\mathsf{Txn})$. Since reordering cannot move the commit time of Txn before $(M, 0)$, $\mathsf{Txn2}$ and Txn remain concurrent. (iii) For any pending transaction $\mathsf{Txn3}$, we have either $\mathsf{StartTs}(\mathsf{Txn}) < (M, 0) < \mathsf{EndTs}(\mathsf{Txn3}) < \mathsf{EndTs}(\mathsf{Txn})$, or $\mathsf{StartTs}(\mathsf{Txn3}) < (M, 0) < \mathsf{EndTs}(\mathsf{Txn}) < \mathsf{EndTs}(\mathsf{Txn3})$. Hence, Txn and $\mathsf{Txn3}$ remain concurrent after reordering. □

The above Lemma 1 ensures that reordering does not impact non-concurrent transactions and their dependencies. Lemma 2 ensures that non-concurrent transactions are not introduced by reordering. Therefore, we restrict our analysis to concurrent dependencies.

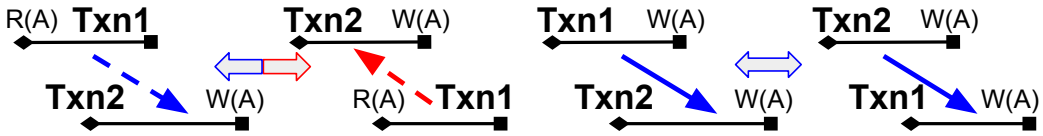


Figure 5.9: The implication of reordering to concurrent dependencies

Dependency order preserves between *c-rw*, *anti-rw* but not *c-ww* when switching commit order

We describe the dependency order of concurrent transactions using the two lemmas below.

Lemma 3. *If two transactions `Txn1` and `Txn2` exhibit c-rw or anti-rw dependency, switching their commit order does not affect their dependency order.*

Proof. When `Txn1` and `Txn2` exhibit *c-rw* (or *anti-rw*) dependency, if we switch their commit order, they will exhibit *anti-rw* (or *c-rw*) dependency, as illustrated in the left side of Figure 5.9. Consequently, in both two cases, their dependency order remains the same, i.e., `Txn1` reads a key which will be written later by `Txn2`. □

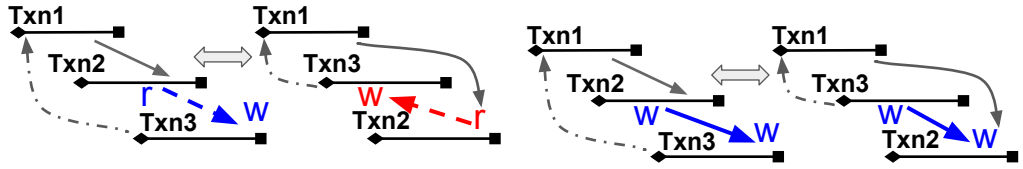
Lemma 4. *If two transactions `Txn1` and `Txn2` exhibit c-ww dependency, switching their commit order flips their dependency order.*

Proof. When `Txn1` and `Txn2` exhibit *c-ww* dependency, `Txn1` writes to a key which will be over-written by `Txn2`. If their commit order is switched, then `Txn2` and `Txn1` will exhibit *c-ww* dependency, as illustrated in the right side of Figure 5.9. Now, `Txn2` writes to a key which will be over-written by `Txn1`. Consequently, the dependency order of `Txn1` and `Txn2` is flipped. □

Finally, we present a theorem on reordering transactions containing a dependency cycle. This theorem is utilized in Chapter 5.3.4 to design our novel fine-grained concurrency control algorithm.

Theorem 2. *A transaction schedule cannot be reordered to be serializable if there exists a cycle with no c-ww dependencies involving pending transactions.*

Proof. We classify the dependencies in the cycle into two categories. The first category includes those involving at least one committed transaction in the dependency. Due to the immutability of blockchains, reordering does not impact these



(a) An unreorderable schedule without *c-ww*

(b) A reorderable schedule with *c-ww*

Figure 5.11: Transaction schedule reorderability

dependencies, because the relative commit order of two transactions is fixed. The second category includes all dependencies between a pair of pending transactions. For each dependency, its corresponding transactions must be concurrent, otherwise, the preceding transaction would be committed. Due to the fact that the pending transactions are concurrent and the absence of *c-ww*, the order switching can only happen between conflicting transactions with *c-rw* or *anti-rw*. Their dependency order preserves despite being reordered (Lemma 3). Hence, the cyclic schedule remains unserializable, as shown in Figure 5.11a. □

However, a transaction schedule can be reordered to be serializable if there exists a cycle with one *c-ww* conflict between pending transactions. Due to Lemma 4, their dependency order can be flipped. We present this scenario in Figure 5.11b, where a cyclic schedule formed by `Txn1`, `Txn2` and `Txn3` becomes serializable by switching the commit order of `Txn2` and `Txn3`, which exhibit *c-ww* dependency.

5.3.4 Fine-grained Concurrency Control

Theorem 2 states that a cyclic transaction schedule without *c-ww* among pending transactions can never be serializable despite reordering. Based on this insight, we formulate the following three steps for fine-grained concurrency control in EOV blockchains.

- For a new transaction, we first consider all dependencies, except *c-ww*, among all pending transactions (including the new transaction). Then, we directly drop the new transaction if there is a dependency cycle.
- On block formation, we retrieve the pending transaction order that respects all the computed dependencies.

- Finally, we restore *c-ww* dependencies on pending transactions based on the retrieved schedule.

Note that *c-ww* dependency restoration is still necessary, as future unserializable transactions may encounter a cycle with a *c-ww* dependency which involves committed transactions. But both their commit and dependency order are already fixed. Hence, the dependency graph remains acyclic after the restoration.

We outline our fine-grained concurrency control in Algorithms 4, 5, and 6, while the implementation details are presented in Chapter 5.4. We use the notation $A \cup= B$ to represent the self-assignment with union $A := A \cup B$. Here, we argue that the topological sort in Algorithm 6 always has a solution since the transaction dependency graph G is guaranteed to be acyclic by Algorithm 5. Even the sub-graph containing only the pending transactions, P, is a directed acyclic graph and, hence, must have a topological order.

Compared to the reordering algorithm in Fabric++, ours is more fine-grained because the unserializable transactions are aborted before ordering and the remaining transactions are guaranteed to be serializable without being aborted. Our reordering is no longer limited to a block's scope. ./b Another notable difference is that we determine the block snapshot at the start of the simulation, while Fabric and Fabric++ determine it based on the last read operation. We allow block commit during the contract simulation for more parallelism, but this may introduce stale snapshots when previously read records are updated by committed transactions during the simulation.

Algorithm 4: Contract simulation

Input: Contract invocation context.
Output: *readset*, *writeset* are simulation results,
b is the number of the block simulated on.

1: $b :=$ fetch the number of the last block;
2: $readset, writeset :=$ simulate the contract invocation on Block b snapshot;
// Chapter 5.4.2

Algorithm 5: On the arrival of a transaction

Data: G is the transaction dependency graph with nodes U and edges V, and P is the pending transaction set.
Input: t is the transaction identifier, b is the number of the block simulated on, and *readkeys*, *writekeys* are accessed keys during simulation.
Output: *reorderable* property of t.

1: *dep* := Compute t's dependency except *c-ww* among P based on $G, b, readkeys, writekeys$; // Chapter 5.4.3
2: *reorderable* := *true* if no cycle is detected in G with respect to *dep*, or $false$ otherwise; // Chapter 5.4.4
3: **if** *reorderable* **then**
4: $P \cup= \{t\}$;
5: $G.U \cup= \{t\}$;
6: $G.V \cup= dep$;

Algorithm 6: On the formation of a block

Data: G is the transaction dependency graph, and P is the pending transaction set.
Output: s is the commit order of pending transactions.

1: s := Topologically sort P based on reachability in G;
2: ww := Compute *c-ww* among P with s;
3: $G.V \cup= ww$; // Chapter 5.4.5
4: $P := \emptyset$

5.3.5 Security Analysis

Our reordering algorithm serves as a part of the ordering process and needs to be replicated on each honest orderer to form the ledger after the consensus service has established the transaction order. We assume the safety and liveness of the original consensus service under its security model, either crash-failure or byzantine-failure. We now discuss whether both properties preserve after our reordering.

Safety. In the original Fabric design, there are four safety properties: *agreement, hash chain integrity, no skipping*, and *no creation* [9]. These properties require honest orderers to sequentially deliver consistent, untampered blocks in a ledger. We claim that our approach preserves *hash chain integrity* and *no skipping* as we do not change the block formation procedure. Next, *no creation* holds because we do not introduce new transactions. Lastly, we achieve *agreement* because we fully replicate the reordering on each orderer. Moreover, we do not introduce non-determinism

which may lead to execution bifurcation. As long as honest orderers perform the reordering individually from a consistent transaction stream, they shall produce identical ledgers.

Liveness. Fabric defines liveness in terms of the *validity* property, which mandates all broadcasted transactions to be included in the ledger. Our algorithm may compromise this liveness property as aborted transactions are excluded from the ledger. However, we propose the following approach to prevent abusive usage. To be specific, in the consensus protocol, the transaction order is tentatively proposed by a leader node. When this order is accepted by the other nodes, it becomes the input of our reordering approach. Hence, the order is controlled by the leader, which may hinge on the publicly available reordering algorithm to maliciously defer certain transactions. Suppose the malicious leader detects an undesirable transaction `TxnT` which reads and writes a record against the state snapshot of block N. The leader, using both a proxy peer and a proxy client, can immediately prepare another transaction `TxnT'` which reads and writes the same record against block N. Next, the leader places `TxnT'` ahead of `TxnT` during ordering. The other orderers, unaware of this manipulation, may accept this ordering. Assuming `TxnT'` passes the reorderability test in Algorithm 5, each honest orderer will abort `TxnT`. It is because these two transactions form an unreorderable cyclic schedule, namely `TxnT'` depends on `TxnT` with *c-rw* and `TxnT` on `TxnT'` with *anti-rw*. The crux of the mitigation is to hide the transaction's details, such as accessed records, before the transaction order is established. For example, we allow clients to send only the transaction hash to the orderers. Moreover, clients have incentives to do so to avoid the above manipulation. After the sequence of a transaction hash is decided, its details are then disclosed to orderers for reordering. We remark that this approach also defers malicious clients from exploiting the reordering by mutating the transaction contents. It is because clients have already made a security commitment by publishing the transaction hash.

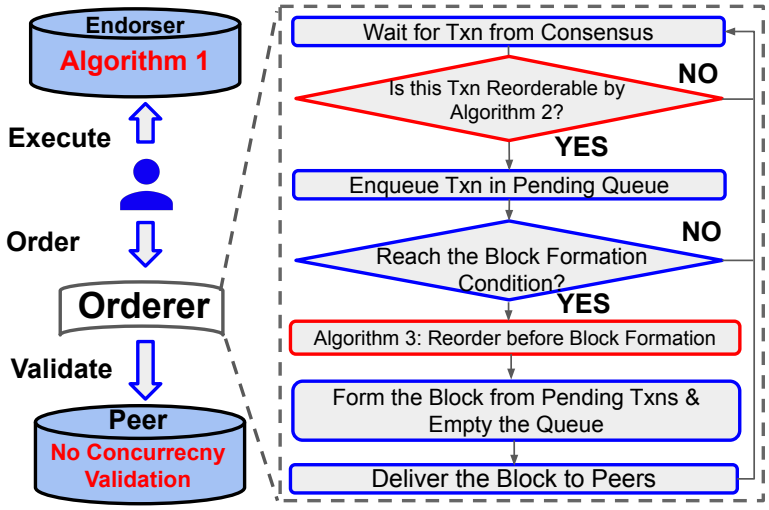


Figure 5.12: The integration of our concurrency control on EOV blockchains

5.4 Implementation

5.4.1 Overview

We illustrate in Figure 5.12 the integration of our proposed fine-grained concurrency control on EOV blockchains. For simplicity, we only show a single orderer and a single peer in the EOV pipeline, but the algorithms are replicated on each node. While the majority of our implementation is done in orderers, Algorithm 4 is integrated in the peers for snapshot-consistent transaction execution during the endorsement phase. In the ordering phase, we employ Algorithm 5 to test the reorderability of an incoming transaction after the consensus decides its commit order. Algorithm 6 performs the abort-free reordering immediately before pending transactions are batched into a block. We remark that Algorithm 5 and Algorithm 6 are far from implementation-friendly to system developers, as both employ an abstract dependency graph. In light of this, we present the details of designing the dependency graph and efficient operations on it.

Even though our actual implementation extends on *FabricSharp*, our below explanations are based upon the original LevelDB-powered Fabric for clarity. We then dedicate Chapter 5.4.7 to discuss specific techniques for *FabricSharp* , i.e., how to make use of ForkBase storage and make it compatible with the existing provenance support.

5.4.2 Snapshot Read

We first describe the snapshot mechanism used by Algorithm 4. We rely on the storage snapshot mechanism to ensure each contract invocation is simulated against a consistent state. Specifically, after a block is committed, we create a storage snapshot and associate it with the block number. Each transaction, before its simulation, must acquire the number of the latest block, as shown in Algorithm 4. Staled snapshots without any simulation are periodically pruned. This design allows more parallelism across contract simulation in the Execution phase and block commit in the Validation phase. In contrast, vanilla Fabric uses a read-write lock to coordinate these two phases.

5.4.3 Dependency Resolution

To compute the dependency graph in Algorithm 5, we introduce two multi-versioned storages in the orderers to identify committed transactions. These storages are implemented in LevelDB and represent *CommittedWriteTxns* (CW) and *CommittedReadTxns* (CR), respectively. Each key of CW consists of the concatenation of the record key and the commit sequence of the transaction updating the value. For example, if `Txn1` with commit sequence $(3, 2)$ writes to key `A`, CW has an entry $\{$`A_3_2` : `Txn1`$\}$. Similarly, each key of CR consists of the concatenation of the record key and the commit sequence of the transaction reading that key's latest value. For instance, the entry $\{$`A_4_1` : `Txn7`$\}$ indicates that `Txn7` is the first transaction in block 4 which reads the latest value of key `A`. In both CW and CR, we place the record key prior to the commit sequence to efficiently support point query and range query. For example, the query $CW.Before(key, seq)$ returns the last committed transaction updating key with the commit sequence earlier than seq. Similarly, $CW.Last(key)$ returns the last committed transaction updating key. For the range query, $CW[key][seq :]$ returns all committed transactions from seq onward that update key.

We maintain two in-memory indices, $PendingWriteTxns$ (PW) and $PendingReadTxns$ (PR), to respectively store the keys for the write and read sets of pending transactions. Consider a new transaction txn that starts at $startTS$ with read keys R and write keys W. All the dependencies of transaction txn are computed as follows.

$$
\begin{aligned}
\textit{anti-rw}(txn) &= \bigcup_{r \in R} CW[r][startTS:] \cup PW[r] \\
rw(txn) &= \bigcup_{w \in W} CR[w] \cup PR[w] \\
\textit{n-wr}(txn) &= \bigcup_{r \in R} CW.Before(r, startTS) \\
ww(txn) &= \bigcup_{w \in W} CW.Last(w)
\end{aligned}
$$

Note that we ignore *ww* dependencies between pending transactions and do not differentiate whether *ww* and *rw* are concurrent or not. This is because non-concurrent transaction may be part of a cycle. We then compute the predecessor transactions of txn as $ww(txn) \cup \textit{n-wr}(txn) \cup rw(txn)$, and successor transactions as $\textit{anti-rw}(txn)$.

5.4.4 Cycle Detection

We now discuss how we represent the dependency graph G to detect cycles and achieve serializability. We face two design choices. On the one hand, we could maintain only the immediate linkage information for each transaction and then perform graph traversal for cycle detection. On the other hand, we could maintain the entire reachability information among each pair of transactions. But the latter approach shifts the overhead from computation to space consumption. We achieve a sweet spot by maintaining the immediate successors of a transaction ($txn.succ$) and represent all transactions that can reach txn with a bloom filter, referred to as $txn.anti\_reachable$. Cycle detection becomes straightforward by testing $p.anti\_reachable(s)$ for each pair (p, s) consisting of a predecessor and a successor of txn.

We use bloom filters because they are memory efficient and can perform fast union. Union is extensively used to update the reachability information for each transaction, as shown in Algorithm 7. Since a bloom filter internally relies on a bit vector, the set union can be fast computed via the bitwise OR operation. However, bloom filters are known to report false positives [24]. If the filters report such false positives for a pair of adjacent transactions to txn, we preventively abort txn. If they report negative for all pairs, then txn does not belong to any cycle in G.

Algorithm 7 entails the relatively expensive traversal of all reachable transactions from txn. However, this cost is bearable, since the traversal is unnecessary when

anti-rw(txn) is empty. This is often the case under non-skewed workloads. Moreover, we reduce the cost of traversal by pruning the dependency graph, as described in Chapter 5.4.6.

Algorithm 7: Reachability update for transaction txn

Data: G is the transaction dependency graph
Input: M is the number of next block to be committed, $pred$ is txn's immediate predecessor transactions, and $succ$ is txn's immediate successor transactions.

1: $txn.anti\_reachable := \emptyset$;
2: **for** p in $pred$ **do**
3: $\quad p.succ \cup= \{txn\}$;
4: $\quad txn.anti\_reachable \cup= p.anti\_reachable$;
5: **for** s reachabale from $succ$ in G **do**
6: $\quad s.anti\_reachable \cup= txn.anti\_reachable$;
7: $\quad s.age := M$;

Algorithm 7 is the constant growth of the *anti_reachable* filter. In practice, we observe that the false positive rate of a single bloom filter grows to an intolerable ratio. To address this issue, we use two bloom filters with relay. Each transaction is associated with one bloom filter capturing transactions committed after block M and another bloom filter capturing transactions after block N. Suppose block C is the earliest block which contains a committed transaction in G. We maintain $M < C < N$ and use the first bloom filter for testing reachability. Whenever C grows to $M < N < C$, the first bloom filter is emptied and it starts to collect transactions from the current block. We then use the second filter for testing reachability. In this manner, we restrict the number of transactions represented by a bloom filter within a certain block range so that the false positive rate remains acceptable. For safety, honest orderers must use the same M and N for exact replication.

5.4.5 Dependency Restoration

Next, we present our method to install *ww* dependencies into the dependency graph G based on the derived commit sequence, which is a topological order of the pending transactions P according to the reachability in G. One prominent issue is that the reachability of a transaction may be affected by multiple *ww* dependencies from various updated keys. But we want the reachability modification to take place

within a single iteration for efficiency. Algorithm 8 outlines the major steps of the restoration of *ww* dependencies. We further explain this algorithm using the example in Figure 5.13.

Algorithm 8: Restoration of *ww* within pending transactions based on the computed commit sequence

Data: G is transaction dependency graph.
Input: seq is committed sequence of pending transactions, PW is the index that associates updated keys with pending transactions.

1: $head\_txns := \emptyset$;
2: **for** $(key, txns)$ in PW **do**
3: Sort $txns$ based on the relative order in seq;
4: $(txn1, txn2) :=$ the first pair in $txns$ such that $txn1 \notin txn2.anti\_reachable$;
5: $txn2.anti\_reachable \cup= txn1.anti\_reachable$;
6: $head\_txns \cup= \{txn2\}$;
7: **for** txn in the topologically-ordered iteration of all txns reachable from $head\_txns$ **do**
8: **for** t in $txn.succ$ **do**
9: $t.anti\_reachable \cup= txn.anti\_reachable$;

For each *key* to be updated by pending transactions (PW), we topologically sort its associated transactions and select the first pair that is not yet connected in the reachability filter. In such a pair, the second transaction can be reached from all the predecessors of the first transaction. There can be a scenario where transactions in a pair are already connected in the reachability filter, which makes the restoration redundant. For example, this happens with `Txn0` and `Txn3` in Figure 5.13. For transactions that are not yet connected, we need to update their successors. To do this efficiently, we keep the transactions in a set ($head\_txns$) and update their successors based on the topological order. Thereby, we avoid updating the information multiple times during the iteration in line 2. For example, `Txn8` in Figure 5.13 is reachable through the update of both key `A` and `B`. Using our algorithm, the reachability information is updated once.

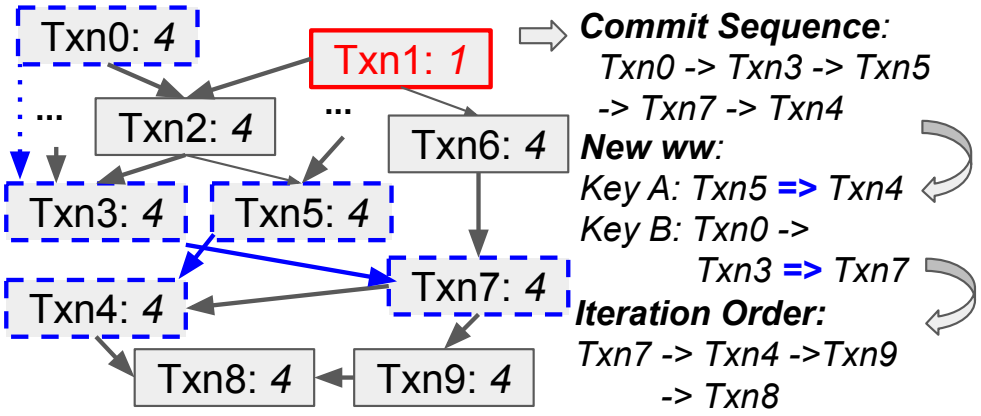


Figure 5.13: An example of a dependency graph with new *c-ww* dependencies

Blue dashed border indicates pending transactions with their commit sequence. Blue solid line indicates new *ww* dependencies and the topologically-sorted iteration order. We do not consider the *ww* dependency between Txn0 and Txn3 (marked with blue dotted line), as it is implicit. Txn1 in red is subject to pruning due to staleness. The transaction age is in italic.

5.4.6 Dependency Graph Pruning

Since graph G can grow quickly, we prune transactions that either (i) are simulated against very old snapshots or (ii) cannot affect pending transactions. For the first case, we introduce a parameter called max_span to limit the block span[2] for a transaction. If the number of the next block is M, we compute the *snapshot threshold* as $H = M - max\_span$. Any transaction simulated against block H or earlier is aborted. For the second case, we define the *age* of a transaction txn to be the sequence number of the last committed block containing at least one transaction reachable from txn in G. When the snapshot threshold is greater than txn's age, future transactions cannot be concurrent with any transaction that can reach txn. In this case, the *anti-rw* dependency will not happen, and this rules out any unserializable schedule containing txn. Therefore, txn can be safely pruned from G. We facilitate the pruning by arranging all transactions in G into a priority queue weighted by age. For new transaction to be committed in block M, we increase the age of the transactions reachable from it to M during the traversal in Algorithm 7 (line 7). For security, all orderers must use the same value for max_span.

[2]If a transaction is simulated against block M and committed in block $M+1$, its block span is 1.

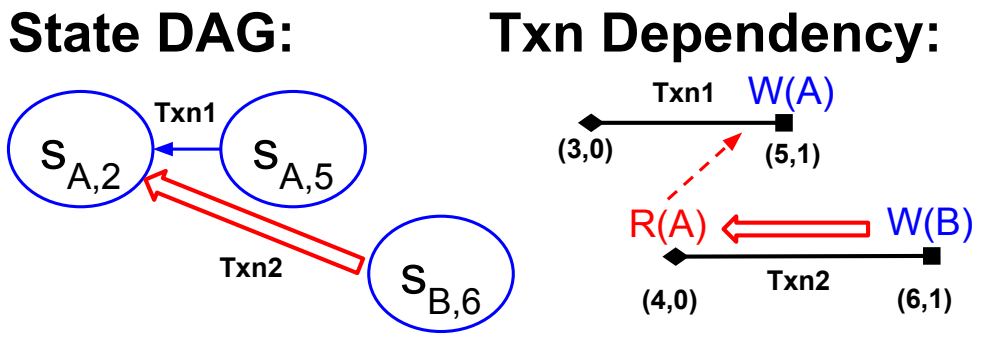


Figure 5.15: An example that leads to the incorrect forward query

5.4.7 *FabricSharp* Specifics

When comparing Figure 4.13 and 5.12, one may observe that the instrumentation for concurrency control is most orthogonal to the provenance support. It is because the former mostly works on orderer nodes (or the consensus layer), which the latter leaves unchanged. In addition, the historical query powered by DASL in Chapter 4.5 can greatly facilitate the snapshot-based simulation in Algorithm 4.

Our concurrency control relaxes from the Strong Serializability in Fabric to the Serializability in *FabricSharp*. In other words, transactions committed later can now depend on earlier ones with *anti-rw*. But this may lead to an incorrect forward query. Figure 5.15 illustrates a concrete case. It juxtaposes both the state DAG, which draws the provenance dependency between state entries, and the transactional graph, which draws the dependencies between transactions. In particular, `Txn1` simulates on block 2, updates key `A` and commits as the first transaction in block 5. Hence, it creates a new entry $S_{A,5}$ with the version 5. `Txn2`, with the simulation on block 3, reads a staled entry of `A`, updates `B` and commits in block 6. Most importantly, `Txn2` relates `A` as one of `B`'s provenance dependencies. Hence the forward query on $S_{A,3}$ shall return $S_{B,6}$. In Chapter 4.4.3, we assume once creating $S_{A,5}$, the entries dependent on $S_{A,3}$ becomes permanent. So we dump all the information of these entries to $S_{A,5}$. However, due to the *anti-rw*, `Txn2` creates the dependent entry $S_{B,6}$ later than $S_{A,5}$. To address this issue, we directly drop `Txn2`. In all, *FabricSharp* will abort a transaction with *anti-rw* dependencies to earlier committed transactions and the conflicted keys in its captured provenance. Notably, the abort only applies when `Txn1` is already committed in earlier blocks. If `Txn1` and `Txn2` are both pending, our

reordering procedure guarantees to place `Txn2` ahead and avoid the above scenario.

5.5 Experiments

Chapter 6

Conclusion and Future Directions

6.1 Conclusion

This thesis focuses on the design and optimization on permissioned blockchains with database techniques. In light of the growing demand on blockchains as emerging transaction platforms, our mission is to identify their bottlenecks and pain points for the improvement while preserving the security. This leads to a series of three works, the first as a behavior study, the second as a utility enhancement, and the last as the performance speedup. Throughout, we adopt the modularized methodology and ground all our implementation into *FabricSharp*.

Firstly, we presented a comprehensive dichotomy between blockchains and distributed databases, viewing them as two different types of transactional distributed systems. We proposed a taxonomy consisting of four design dimensions: replication, concurrency, storage, and sharding. Using this taxonomy, we discussed how both system types make different design choices driven by their high-level goals (e.g., security for blockchains, and performance for databases). We then performed a quantitative performance comparison using five different systems covering a large area of the design space. Our results illustrated the effects of different design choices to the overall performance. Our work provides the first framework to explore future database-blockchain design fusions.

In second work, we showed how to build a fine-grained, secure and efficient provenance system on top of blockchains. We implemented our techniques into *FabricSharp*. The system efficiently captures provenance information during runtime and stores it in secure storage. It exposes simple APIs to smart contracts, which enables a new class of provenance-dependent blockchain applications. Provenance queries

are efficient in *FabricSharp*, thanks to a novel skip list index. We benchmarked it against several baselines. The results show the benefits of *FabricSharp* in supporting rich, provenance-dependent applications. We demonstrate that provenance queries are efficient and that the system incurs small storage and performance overhead.

Last but not the least, we proposed a novel solution to efficiently reduce the transaction abort rate in execute-order-validate blockchains by applying transactional analysis from optimistic-concurrency-control databases. We first draw theoretical parallelism between both blockchains and databases. Then, we introduced a fine-grained concurrency control method and implemented it in *FabricSharp* based on Fabric v2.2, respectively. Our experimental analysis shows that *FabricSharp* outperforms other blockchain systems, including the vanilla Fabric, and Fabric++. Unlike databases that achieve high throughput, the blockchains' limited throughput due to factors related to security opens up opportunities for precise transaction management.

6.2 Future Directions

6.2.1 Blockchain Interoperability

While a growing number of blockchains proliferate, most of them operate in silos, with poor synchronization and coordination. Such fragmentation of the landscape not only results into a waste of resources and data isolation, but it also runs counter to the very essence of the Internet, openness and freedom. Even though we observe a number of research works with the special emphasis on the across-ledger token swap, their scope is mostly restricted to the cryptocurrency domain [74, 136, 180]. For wider applications, the community of blockchains should look forward to some more generic standards, just like TCP/IP to the Internet. Promisingly, we notice that Interledger Protocol has taken on the initial attempt [79]. And we expect more will follow in the future.

6.2.2 Declarative Language for Smart Contracts

Even though we demonstrate the provenance support on blockchains in Chapter 4, it still follows an imperative approach. To be specific, users are required to explicitly

program *how-to-do*, instead of implicitly declaring *what-to-do* in the smart contract. The high-level declarative language can not only allow users to work on a high abstraction level and save development efforts. It also opens up a vast room for the common optimization. We haven't observed any progress along with this direction. But according to the development roadmap of the database over the decades, we believe that an easy-to-use and intuitive contract scheme is essential for the mass adoption of blockchains.

6.2.3 Blockchain-like Verifiable Databases

The impact of blockchains comes from its revolutionary decentralization. But in reality, their byzantine tolerant consensus proves to be an overkill for most applications. In light of this, there are a growing number of secure databases, which, unlike blockchains, completely eliminate the decentralized setup [10, 184]. Despite the single point, these databases still support verifiability on the state storage. Some vendors simulate a ledger-structure to expose the data provenance with the integrity guarantee [6]. We believe such blockchain-like verifiable databases already satisfy the majority of business requirements, in which blockchains would prove redundant.

6.2.4 Federated Learning on Blockchains

Considering their common decentralized nature of the federated learning and blockchains, it is not hard to image a number of literatures that pair both hot topics together [103, 88, 13]. The researchers have attempted to rely on blockchains to consolidate data from mutual distrusted users and collectively train for a shared model. In their design, a blockchain serves a trust-building platform to regulate on data ownership and the model copyright. But the challenge is that data privacy may be at odds with the blockchain transparency. And it remains a open topic how to fairly allocate the model ownership according to the heterogeneous data sources and use blockchains to coordinate this process. In the AI-driven future with the immense adoption of Internet-of-Things, we expect more such interdisciplinary proposals between blockchains and machine learning.

Bibliography

[1] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Efficient synchronous byzantine consensus", *arXiv preprint arXiv:1704.02397*, 2017.

[2] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin, "Revisiting fast practical byzantine fault tolerance", *arXiv preprint arXiv:1712.01367*, 2017.

[3] I. Abraham, G. Gueta, D. Malkhi, and J.-P. Martin, "Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma", *arXiv preprint arXiv:1801.10022*, 2018.

[4] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication", in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 106–118.

[5] S. Akoush, R. Sohan, and A. Hopper, "Hadoopprov: Towards provenance as a first class citizen in mapreduce." In *TaPP*, 2013.

[6] "Amazon quantum ledger database", `https://aws.amazon.com/qldb/`, Accessed: 2020-09-2.

[7] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems", in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2019, pp. 1337–1347.

[8] Y. Amoussou-Guenou, A. Del Pozzo, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Dissecting tendermint", in *International Conference on Networked Systems*, Springer, 2019, pp. 166–182.

[9] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, *et al.*, "Hyperledger fabric:

A distributed operating system for permissioned blockchains", in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[10] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy, "Concerto: A high concurrency key-value store with integrity", in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 251–266.

[11] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)", in *International conference on principles of security and trust*, Springer, 2017, pp. 164–186.

[12] M. Atzori, "Blockchain technology and decentralized governance: Is the state still necessary?" *Available at SSRN 2709713*, 2015.

[13] S. Awan, F. Li, B. Luo, and M. Liu, "Poster: A reliable and accountable privacy-preserving federated learning framework using the blockchain", in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2561–2563.

[14] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management", in *2016 2nd International Conference on Open and Big Data (OBD)*, IEEE, 2016, pp. 25–30.

[15] X. Bai, Z. Cheng, Z. Duan, and K. Hu, "Formal modeling and verification of smart contracts", in *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, 2018, pp. 322–326.

[16] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations", *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 181–192, 2013.

[17] A. Baliga, I. Subhod, P. Kamat, and S. Chatterjee, "Performance evaluation of the quorum blockchain platform", *arXiv preprint arXiv:1809.03421*, 2018.

[18] M. Bartoletti and L. Pompianu, "An analysis of bitcoin op_return metadata", in *International Conference on Financial Cryptography and Data Security*, Springer, 2017, pp. 218–230.

[19] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform", *arXiv preprint arXiv:1708.03778*, 2017.

[20] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, "State machine replication in the libra blockchain", *The Libra Assn., Tech. Rep*, 2019.

[21] I. Bentov, R. Pass, and E. Shi, "Snow white: Provably secure proofs of stake." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 919, 2016.

[22] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels", *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[23] bitcoinwiki, "Erc20", `https://en.bitcoinwiki.org/wiki/ERC20`, [Online; accessed 08-September-2020], 2020.

[24] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[25] G. Blossey, J. Eisenhardt, and G. Hahn, "Blockchain technology in supply chain management: An application perspective", in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[26] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete, "One-copy serializability with snapshot isolation under the hood", in *IEEE 27th International Conference on Data Engineering*, 2011, pp. 625–636.

[27] J. Brown-Cohen, A. Narayanan, A. Psomas, and S. M. Weinberg, "Formal barriers to longest-chain proof-of-stake protocols", in *Proceedings of the 2019 ACM Conference on Economics and Computation*, 2019, pp. 459–473.

[28] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains", PhD thesis, 2016.

[29] P. Buneman, A. Chapman, and J. Cheney, "Provenance management in curated databases", in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ACM, 2006, pp. 539–550.

[30] C. Cachin, S. Schubert, and M. Vukolić, "Non-determinism in byzantine fault-tolerant replication", *arXiv preprint arXiv:1603.07351*, 2016.

[31] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases", in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, 2008, pp. 729–738.

[32] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses", *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[33] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi, "On security analysis of proof-of-elapsed-time (poet)", in *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Springer, 2017, pp. 282–297.

[34] S. Chen, J. Zhang, R. Shi, J. Yan, and Q. Ke, "A comparative testing on performance of blockchain and relational database: Foundation for applying smart technology into current business systems", in *International Conference on Distributed, Ambient, and Pervasive Interactions*, Springer, 2018, pp. 21–34.

[35] W. Chen, T. Zhang, Z. Chen, Z. Zheng, and Y. Lu, "Traveling the token world: A graph analysis of ethereum erc20 token ecosystem", in *Proceedings of The Web Conference 2020*, 2020, pp. 1411–1421.

[36] J. Cheney, L. Chiticariu, W.-C. Tan, *et al.*, "Provenance in databases: Why, how, and where", *Foundations and Trends® in Databases*, vol. 1, no. 4, pp. 379–474, 2009.

[37] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "Dbnotes: A post-it system for relational databases based on provenance", in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ACM, 2005, pp. 942–944.

[38] H. Cho, "Asic-resistance of multi-hash proof-of-work mechanisms for blockchain consensus protocols", *IEEE Access*, vol. 6, pp. 66 210–66 222, 2018.

[39] M. J. M. Chowdhury, A. Colman, M. A. Kabir, J. Han, and P. Sarda, "Blockchain versus database: A critical analysis", in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And*

Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), IEEE, 2018, pp. 1348–1353.

[40] A. Churyumov, "Byteball: A decentralized system for storage and transfer of value", *URL https://byteball. org/Byteball. pdf*, 2016.

[41] L. Cocco, A. Pinna, and M. Marchesi, "Banking on blockchain: Costs savings thanks to the blockchain technology", *Future internet*, vol. 9, no. 3, p. 25, 2017.

[42] N. T. Courtois and L. Bahack, "On subversive miner strategies and block withholding attack in bitcoin digital currency", *arXiv preprint arXiv:1402.1718*, 2014.

[43] G. Danezis and S. Meiklejohn, "Centrally banked cryptocurrencies", *arXiv preprint arXiv:1505.06895*, 2015.

[44] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding", in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 123–140.

[45] B. David, P. Gaži, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2018, pp. 66–98.

[46] C. Decker, J. Seidel, and R. Wattenhofer, "Bitcoin meets strong consistency", in *Proceedings of the 17th International Conference on Distributed Computing and Networking*, 2016, pp. 1–10.

[47] N. Diallo, W. Shi, L. Xu, Z. Gao, L. Chen, Y. Lu, N. Shah, L. Carranco, T.-C. Le, A. B. Surez, *et al.*, "Egov-dao: A better government using blockchain based decentralized autonomous organization", in *2018 International Conference on eDemocracy & eGovernment (ICEDEG)*, IEEE, 2018, pp. 166–171.

[48] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts", *Distributed Computing*, pp. 1–17, 2019.

[49] B. Ding, L. Kot, and J. Gehrke, "Improving optimistic concurrency control through transaction batching and operation reordering", *Proceedings of the VLDB Endowment*, vol. 12, no. 2, pp. 169–182, 2018.

[50] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, "Untangling blockchain: A data processing view of blockchain systems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 7, pp. 1366–1385, 2018.

[51] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains", in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1085–1100.

[52] M. Divya and N. B. Biradar, "Iota-next generation block chain", *International journal of engineering and computer science*, vol. 7, no. 04, pp. 23 823–23 826, 2018.

[53] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical", in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[54] S. Ducasse, H. Rocha, S. Bragagnolo, M. Denker, and C. Francomme, "Open-source tool suite for smart contract analysis", *Blockchain and Web 3.0: Social, Economic, and Technological Challenges*, 2019.

[55] I. Eyal, "The miner's dilemma", in *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 89–103.

[56] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol", in *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*, 2016, pp. 45–59.

[57] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable", in *International conference on financial cryptography and data security*, Springer, 2014, pp. 436–454.

[58] "Fabricsharp", `https://github.com/ooibc88/FabricSharp`, Accessed: 2020-09-2.

[59] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable", *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 2, pp. 492–528, 2005.

[60] Q. Feng, D. He, S. Zeadally, M. K. Khan, and N. Kumar, "A survey on privacy protection in blockchain system", *Journal of Network and Computer Applications*, vol. 126, pp. 45–58, 2019.

[61] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process", *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[62] M. Fleder, M. S. Kester, and S. Pillai, "Bitcoin transaction graph analysis", *arXiv preprint arXiv:1502.01657*, 2015.

[63] A. Gaihre, Y. Luo, and H. Liu, "Do bitcoin users really care about anonymity? an analysis of the bitcoin transaction graph", in *2018 IEEE International Conference on Big Data (Big Data)*, IEEE, 2018, pp. 1198–1207.

[64] J. Garay and A. Kiayias, "Sok: A consensus taxonomy in the blockchain era", in *Cryptographers' Track at the RSA Conference*, Springer, 2020, pp. 284–318.

[65] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2015, pp. 281–310.

[66] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains", in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.

[67] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies", in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 51–68.

[68] C. Gorenflo, L. Golab, and S. Keshav, "Xox fabric: A hybrid approach to blockchain transaction execution", in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2020, pp. 1–9.

[69] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second", in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2019, pp. 455–463.

[70] A. Greaves and B. Au, "Using the bitcoin transaction graph to predict the price of bitcoin",, 2015.

[71] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: A scalable and decentralized trust infrastructure", in *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, IEEE, 2019, pp. 568–580.

[72] Y. Guo and C. Liang, "Blockchain application and outlook in the banking industry", *Financial Innovation*, vol. 2, no. 1, p. 24, 2016.

[73] M. Hearn, "Corda: A distributed ledger", *Corda Technical White Paper*, vol. 2016, 2016.

[74] M. Herlihy, "Atomic cross-chain swaps", in *Proceedings of the 2018 ACM symposium on principles of distributed computing*, 2018, pp. 245–254.

[75] M. Herlihy, "Blockchains from a distributed computing perspective", *Communications of the ACM*, vol. 62, no. 2, pp. 78–85, 2019.

[76] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "Blockchaindb: A shared database on blockchains", *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1597–1609, 2019.

[77] S. Huh, S. Cho, and S. Kim, "Managing iot devices using blockchain platform", in *2017 19th international conference on advanced communication technology (ICACT)*, IEEE, 2017, pp. 464–467.

[78] "Hyperledger fabric", `https://github.com/hyperledger/fabric`.

[79] "Interledger", `https://interledger.org/`, Accessed: 2020-09-02.

[80] Z. István, A. Sorniotti, and M. Vukolić, "Streamchain: Do blockchains need blocks?" In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2018, pp. 1–6.

[81] J. Al-Jaroodi and N. Mohamed, "Blockchain in industries: A survey", *IEEE Access*, vol. 7, pp. 36 500–36 515, 2019.

[82] Y.-w. Jeng, Y.-c. Hsieh, and J.-L. Wu, "Step-by-step guidelines for making smart contract smarter", in *2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE, 2019, pp. 25–32.

[83] A. P. Joshi, M. Han, and Y. Wang, "A survey on security and privacy issues of blockchain technology", *Mathematical foundations of computing*, vol. 1, no. 2, p. 121, 2018.

[84] M. A. Khan and K. Salah, "Iot security: Review, blockchain solutions, and open challenges", *Future Generation Computer Systems*, vol. 82, pp. 395–411, 2018.

[85] A. Kiayias and G. Panagiotakos, "Speed-security tradeoffs in blockchain protocols." *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1019, 2015.

[86] A. Kiayias and G. Panagiotakos, "On trees, chains and fast transactions in the blockchain", in *International Conference on Cryptology and Information Security in Latin America*, Springer, 2017, pp. 327–351.

[87] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol", in *Annual International Cryptology Conference*, Springer, 2017, pp. 357–388.

[88] H. Kim, J. Park, M. Bennis, and S.-L. Kim, "Blockchained on-device federated learning", *IEEE Communications Letters*, vol. 24, no. 6, pp. 1279–1283, 2019.

[89] S. King, "Primecoin: Cryptocurrency with prime number proof-of-work", *July 7th*, vol. 1, no. 6, 2013.

[90] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake", *self-published paper, August*, vol. 19, p. 1, 2012.

[91] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing", in *25th usenix security symposium (usenix security 16)*, 2016, pp. 279–296.

[92] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding", in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 583–598.

[93] K. Korpela, J. Hallikas, and T. Dahlberg, "Digital supply chain transformation toward blockchain integration", in *proceedings of the 50th Hawaii international conference on system sciences*, 2017.

[94] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control", *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[95] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem", in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 203–226.

[96] C. LeMahieu, "Nano: A feeless distributed cryptocurrency network", *Nano [Online resource]. URL: https://nano. org/en/whitepaper (date of access: 24.03. 2018)*, 2018.

[97] C. Li, P. Li, D. Zhou, W. Xu, F. Long, and A. Yao, "Scaling nakamoto consensus to thousands of transactions per second", *arXiv preprint arXiv:1805.03870*, 2018.

[98] W. Li, S. Andreina, J.-M. Bohli, and G. Karame, "Securing proof-of-stake blockchain protocols", in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, Springer, 2017, pp. 297–315.

[99] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems", *Future Generation Computer Systems*, vol. 107, pp. 841–853, 2020.

[100] S. Y. Lim, P. T. Fotsing, A. Almasri, O. Musa, M. L. M. Kiah, T. F. Ang, and R. Ismail, "Blockchain technology the identity management and authentication service disruptor: A survey", *International Journal on Advanced Science, Engineering and Information Technology*, vol. 8, no. 4-2, pp. 1735–1745, 2018.

[101] I.-C. Lin and T.-C. Liao, "A survey of blockchain security issues and challenges." *IJ Network Security*, vol. 19, no. 5, pp. 653–659, 2017.

[102] Y. Lootsma, "From fintech to regtech: The possible use of blockchain for kyc", *Fintech To Regtech Using block chain*, 2017.

[103] Y. Lu, X. Huang, Y. Dai, S. Maharjan, and Y. Zhang, "Blockchain and federated learning for privacy-preserved data sharing in industrial iot", *IEEE Transactions on Industrial Informatics*, vol. 16, no. 6, pp. 4177–4186, 2019.

[104] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter", in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[105] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 17–30.

[106] L. Luu, R. Saha, I. Parameshwaran, P. Saxena, and A. Hobor, "On power splitting games in distributed computation: The case of bitcoin pooled mining", in *2015 IEEE 28th Computer Security Foundations Symposium*, IEEE, 2015, pp. 397–411.

[107] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer", in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 706–719.

[108] D. D. F. Maesa, P. Mori, and L. Ricci, "Blockchain based access control", in *IFIP international conference on distributed applications and interoperable systems*, Springer, 2017, pp. 206–220.

[109] D. Malkhi, K. Nayak, and L. Ren, "Flexible byzantine fault tolerance", in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1041–1053.

[110] J.-P. Martin and L. Alvisi, "Fast byzantine consensus", *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[111] "Medilot: Transforming healthcare for all", `https://medilot.com/`, Accessed: 2020-09-2.

[112] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz, "Permacoin: Repurposing bitcoin work for data preservation", in *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 475–490.

[113] A. Miller and J. J. LaViola Jr, "Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin", *University of Central Florida Tech. Report CS-TR-14-01 (accessed 5 June 2019) https://socrates1024. s3. amazonaws. com/consensus. pdf*, 2014.

[114] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 31–42.

[115] B. K. Mohanta, D. Jena, S. S. Panda, and S. Sobhanayak, "Blockchain technology: A survey on applications and security privacy challenges", *Internet of Things*, vol. 8, p. 100 107, 2019.

[116] M. B. Mollah, J. Zhao, D. Niyato, K.-Y. Lam, X. Zhang, A. M. Ghias, L. H. Koh, and L. Yang, "Blockchain for future smart grid: A comprehensive survey", *IEEE Internet of Things Journal*, 2020.

[117] M. Moser, "Anonymity of bitcoin transactions",, 2013.

[118] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", Manubot, Tech. Rep., 2019.

[119] Q. Nasir, I. A. Qasse, M. Abu Talib, and A. B. Nassif, "Performance analysis of hyperledger fabric platforms", *Security and Communication Networks*, vol. 2018, 2018.

[120] K. Nayak, S. Kumar, A. Miller, and E. Shi, "Stubborn mining: Generalizing selfish mining and combining with an eclipse attack", in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2016, pp. 305–320.

[121] C. T. Nguyen, D. T. Hoang, D. N. Nguyen, D. Niyato, H. T. Nguyen, and E. Dutkiewicz, "Proof-of-stake consensus mechanisms for future blockchain networks: Fundamentals, applications and opportunities", *IEEE Access*, vol. 7, pp. 85 727–85 745, 2019.

[122] G.-T. Nguyen and K. Kim, "A survey about consensus algorithms used in blockchain." *Journal of Information processing systems*, vol. 14, no. 1, 2018.

[123] Q. K. Nguyen, "Blockchain-a financial technology for future sustainable development", in *2016 3rd International Conference on Green Technology and Sustainable Development (GTSD)*, IEEE, 2016, pp. 51–54.

[124] O. Novo, "Blockchain meets iot: An architecture for scalable access management in iot", *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1184–1195, 2018.

[125] M. Ober, S. Katzenbeisser, and K. Hamacher, "Structure and anonymity of the bitcoin transaction graph", *Future internet*, vol. 5, no. 2, pp. 237–250, 2013.

[126] R. M. Parizi, A. Dehghantanha, *et al.*, "Smart contract programming languages on blockchains: An empirical evaluation of usability and security", in *International Conference on Blockchain*, Springer, 2018, pp. 75–91.

[127] H. Park, R. Ikeda, and J. Widom, "Ramp: A system for capturing and tracing provenance in mapreduce workflows",, 2011.

[128] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song, "Falcondb: Blockchain-based collaborative database", in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 637–652.

[129] "Performance benchmarking results for corda", `https://docs.corda.net/docs/corda-enterprise/4.5/node/performance-results.html`, [Online; accessed 01-September-2020], 2020.

[130] G. W. Peters and E. Panayi, "Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money", in *Banking beyond banks and money*, Springer, 2016, pp. 239–278.

[131] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, "Performance analysis of private blockchain platforms in varying workloads", in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2017, pp. 1–6.

[132] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey", *arXiv preprint arXiv:1908.08605*, 2019.

[133] F. Psallidas and E. Wu, "Smoke: Fine-grained lineage at interactive speed", *PVLDB*, vol. 11, no. 6, pp. 719–732, 2018.

[134] D. Reijsbergen and T. T. A. Dinh, "On exploiting transaction concurrency to speed up blockchains", 2020. arXiv: `2003.06128 [cs.DC]`.

[135] L. Ren, "Analysis of nakamoto consensus." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 943, 2019.

[136] P. Robinson, D. Hyland-Wood, R. Saltini, S. Johnson, and J. Brainard, "Atomic crosschain transactions for ethereum private sidechains", *arXiv preprint arXiv:1904.12079*, 2019.

[137] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability", *arXiv preprint arXiv:1906.08936*, 2019.

[138] D. Ron and A. Shamir, "Quantitative analysis of the full bitcoin transaction graph", in *International Conference on Financial Cryptography and Data Security*, Springer, 2013, pp. 6–24.

[139] S. Rouhani and R. Deters, "Performance analysis of ethereum transactions in private blockchain", in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, IEEE, 2017, pp. 70–74.

[140] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, D. Loghin, B. C. Ooi, and M. Zhang, "Blockchains and distributed databases: A twin study", *arXiv preprint arXiv:1910.01310*, 2019.

[141] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen, "Blockchain technology and its relationships to sustainable supply chain management", *International Journal of Production Research*, vol. 57, no. 7, pp. 2117–2135, 2019.

[142] R. Saltini and D. Hyland-Wood, "Correctness analysis of ibft", *arXiv preprint arXiv:1901.07160*, 2019.

[143] F. Santos and V. Kostakis, "The dao: A million dollar lesson in blockchain governance", *School of Business and Governance, Ragnar Nurkse Department of Innovation and Governance*, 2018.

[144] A. Sapirshtein, Y. Sompolinsky, and A. Zohar, "Optimal selfish mining strategies in bitcoin", in *International Conference on Financial Cryptography and Data Security*, Springer, 2016, pp. 515–532.

[145] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in ethereum smart contracts", *arXiv preprint arXiv:1901.01376*, 2019.

[146] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial", *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[147] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts", in *International Conference on Financial Cryptography and Data Security*, Springer, 2017, pp. 478–493.

[148] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: The case of hyperledger fabric", in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 105–122.

[149] A. Shoker, "Sustainable blockchain through proof of exercise", in *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, IEEE, 2017, pp. 1–9.

[150] S. Somin, G. Gordon, and Y. Altshuler, "Network analysis of erc20 tokens trading on ethereum blockchain", in *International Conference on Complex Systems*, Springer, 2018, pp. 439–450.

[151] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin", in *International Conference on Financial Cryptography and Data Security*, Springer, 2015, pp. 507–527.

[152] G. Srivastava, A. D. Dwivedi, and R. Singh, "Phantom protocol as the new crypto-democracy", in *IFIP International Conference on Computer Information Systems and Industrial Management*, Springer, 2018, pp. 499–509.

[153] StackExchange, "Explanation of what an opreturn transaction looks like", `https://bitcoin.stackexchange.com/questions/29554/explanation-of-what-an-op-return-transaction-looks-like`, [Online; accessed 01-September-2020], 2020.

[154] A. Tapscott and D. Tapscott, "How blockchain is changing finance", *Harvard Business Review*, vol. 1, no. 9, 2017.

[155] P. Thakkar, S. Nathan, and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform", in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, 2018, pp. 264–276.

[156] F. Tian, "An agri-food supply chain traceability system for china based on rfid & blockchain technology", in *2016 13th international conference on service systems and service management (ICSSSM)*, IEEE, 2016, pp. 1–6.

[157] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts", in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[158] J. Tinguely, "Benchmarking of distributed ledger technology",, 2019.

[159] D. Tse, B. Zhang, Y. Yang, C. Cheng, and H. Mu, "Blockchain application in food supply information security", in *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, IEEE, 2017, pp. 1357–1361.

[160] F. Victor and B. K. Lüders, "Measuring ethereum-based erc20 token networks", in *International Conference on Financial Cryptography and Data Security*, Springer, 2019, pp. 113–129.

[161] G. Wang, Z. J. Shi, M. Nixon, and S. Han, "Sok: Sharding on blockchain", in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 41–61.

[162] J. Wang, D. Crawl, S. Purawat, M. Nguyen, and I. Altintas, "Big data provenance: Challenges, state of the art and opportunities", in *Big Data (Big Data), 2015 IEEE International Conference on*, IEEE, 2015, pp. 2509–2516.

[163] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan, "Forkbase: An efficient storage engine for blockchain and forkable applications", *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1137–1150, 2018, ISSN: 2150-8097.

[164] W. Wang, D. T. Hoang, P. Hu, Z. Xiong, D. Niyato, P. Wang, Y. Wen, and D. I. Kim, "A survey on consensus mechanisms and mining strategy management in blockchain networks", *IEEE Access*, vol. 7, pp. 22 328–22 370, 2019.

[165] X. Wang, X. Zha, W. Ni, R. P. Liu, Y. J. Guo, X. Niu, and K. Zheng, "Survey on blockchain for internet of things", *Computer Communications*, vol. 136, pp. 10–29, 2019.

[166] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, and C. E. Leiserson, "Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics", *arXiv preprint arXiv:1908.02591*, 2019.

[167] G. Weikum and G. Vossen, "Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery", Elsevier, 2001.

[168] Wikipedia, "List of cryptocurrencies — Wikipedia, the free encyclopedia", `http://en.wikipedia.org/w/index.php?title=List%20of%20cryptocurrencies&oldid=975908929`, [Online; accessed 01-September-2020], 2020.

[169] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger", *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[170] M. Wu, K. Wang, X. Cai, S. Guo, M. Guo, and C. Rong, "A comprehensive survey of blockchain: From theory to iot applications and beyond", *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8114–8154, 2019.

[171] Y. Wu, "An e-voting system based on blockchain and ring signature", *Master. University of Birmingham*, 2017.

[172] K. Wüst and A. Gervais, "Do you need a blockchain?" In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, IEEE, 2018, pp. 45–54.

[173] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks", *IEEE Communications Surveys and Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020.

[174] B. Xu, D. Luthra, Z. Cole, and N. Blakely, "Eos: An architectural, performance, and economic analysis", *Retrieved June*, vol. 11, p. 2019, 2018.

[175] M. Yabandeh and D. Gómez Ferro, "A critique of snapshot isolation", in *Proceedings of the 7th ACM european conference on Computer Systems*, ACM, 2012, pp. 155–168.

[176] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview", *arXiv preprint arXiv:1906.11078*, 2019.

[177] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness", in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[178] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, and R. P. Liu, "Survey: Sharding in blockchains", *IEEE Access*, vol. 8, pp. 14 155–14 181, 2020.

[179] H. Yu, I. Nikolić, R. Hou, and P. Saxena, "Ohie: Blockchain scaling made simple", in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 90–105.

[180] V. Zakhary, D. Agrawal, and A. E. Abbadi, "Atomic commitment across blockchains", *arXiv preprint arXiv:1905.02847*, 2019.

[181] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding", in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 931–948.

[182] A. R. Zamanov, V. A. Erokhin, and P. S. Fedotov, "Asic-resistant hash functions", in *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, IEEE, 2018, pp. 394–396.

[183] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, "Sok: Communication across distributed ledgers.", 2019.

[184] M. Zhang, Z. Xie, C. Yue, and Z. Zhong, "Spitz: A verifiable database system", *arXiv preprint arXiv:2008.09268*, 2020.

[185] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey", *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[186] P. Zhuang, T. Zamir, and H. Liang, "Blockchain for cyber security in smart grid: A comprehensive survey", *IEEE Transactions on Industrial Informatics*, 2020.

Publications during PhD Study

[1] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains", in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 543–557.

[2] P. Ruan, T. T. Anh Dinh, Q. Lin, M. Zhang, G. Chen, and B. Chin Ooi, "Revealing every story of data in blockchain systems", *ACM SIGMOD Record*, vol. 49, no. 1, pp. 70–77, 2020.

[3] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, D. Loghin, B. C. Ooi, and M. Zhang, "Blockchains and distributed databases: A twin study", *arXiv preprint arXiv:1910.01310*, 2019.

[4] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang, "Fine-grained, secure and efficient data provenance on blockchain systems", *Proceedings of the VLDB Endowment*, vol. 12, no. 9, pp. 975–988, 2019.

[5] Q. Lin, K. Yang, T. T. A. Dinh, Q. Cai, G. Chen, B. C. Ooi, P. Ruan, S. Wang, Z. Xie, M. Zhang, *et al.*, "Forkbase: Immutable, tamper-evident storage substrate for branchable applications", in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, IEEE, 2020, pp. 1718–1721.

[6] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan, "Forkbase: An efficient storage engine for blockchain and forkable applications", *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1137–1150, 2018, ISSN: 2150-8097.

[7] A. Dinh, J. Wang, S. Wang, G. Chen, W.-N. Chin, Q. Lin, B. C. Ooi, P. Ruan, K.-L. Tan, Z. Xie, *et al.*, "Ustore: A distributed storage with rich semantics", *arXiv preprint arXiv:1702.02799*, 2017.