FEDERAL UNIVERSITY OF CEARÁ

CENTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

ACADEMIC MASTER'S DEGREE IN COMPUTER SCIENCE

RAFAEL AVILAR SÁ

IMPROVING INTEROPERABILITY BETWEEN RELATIONAL AND BLOCKCHAIN-BASED DATABASE SYSTEMS: A MIDDLEWARE APPROACH

FORTALEZA

2024

RAFAEL AVILAR SÁ

IMPROVING INTEROPERABILITY BETWEEN RELATIONAL AND BLOCKCHAIN-BASED DATABASE SYSTEMS: A MIDDLEWARE APPROACH

Dissertation submitted to the Graduate Program in Computer Science of the Center of Science of the Federal University of Ceará, as a partial requirement for obtaining the title of Master in Computer Science. Concentration Area: Computer Science

Advisor: Prof. Dr. Javam de Castro Machado

Co-advisor: Prof. Dr. Leonardo Oliveira Moreira

FORTALEZA

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S115i Sá, Rafael Avilar.
Improving Interoperability between Relational and Blockchain-based Database Systems : a middleware approach / Rafael Avilar Sá. – 2024.
99 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2024.
Orientação: Prof. Dr. Javam de Castro Machado.
Coorientação: Prof. Dr. Leonardo Oliveira Moreira.

1. Databases. 2. Blockchain. 3. Database Interoperability. 4. Data Transformation. 5. Middleware. I. Título.

CDD 005

RAFAEL AVILAR SÁ

IMPROVING INTEROPERABILITY BETWEEN RELATIONAL AND BLOCKCHAIN-BASED DATABASE SYSTEMS: A MIDDLEWARE APPROACH

Dissertation submitted to the Graduate Program in Computer Science of the Center of Science of the Federal University of Ceará, as a partial requirement for obtaining the title of Master in Computer Science. Concentration Area: Computer Science

Approved on: 26/07/2024

EXAMINATION BOARD

Prof. Dr. Javam de Castro Machado (Advisor)
Federal University of Ceará (UFC)

Prof. Dr. Leonardo Oliveira Moreira (Co-advisor)
Federal University of Ceará (UFC)

Prof. Dr. Angelo Roncalli Alencar Brayner
Federal University of Ceará (UFC)

Prof. Dr. Victor Aguiar Evangelista de Farias
Federal University of Ceará (UFC)

Profa. Dra. Carmem Satie Hara
Federal University of Paraná (UFPR)

To my family, for their understanding and support through this journey. If not for them, I would never be here.

ACKNOWLEDGEMENTS

To Prof. Dr. Javam de Castro Machado and Prof. Dr. Leonardo de Oliveira Moreira for advising me during this research and giving full support whenever possible. Their invaluable guidance was paramount to the success of this work.

To the examination board, Prof. Dr. Angelo Roncalli Alencar Brayner, Prof. Dr. Victor Aguiar Evangelista de Farias, and Prof. Dr. Carmem Satie Hara, for the time and effort spent in evaluating this dissertation and providing valuable feedback.

To my colleagues and professors at Laboratório de Sistemas e Bancos de Dados (LSBD) for their friendship, commentary, insight, and support. Our weekly study group meetings were one of my greatest motivators.

To my beloved mother, father, and sister, who were my staunch allies and greatest supporters. My family taught me all of the values that make up the person I am today.

To Prof. Dr. Tobias Rafael Fernandes Neto, coordinator of Laboratório de Sistemas Motrizes (LAMOTRIZ), where this template was developed.

To the PhD candidate in Electrical Engineering, Ednardo Moreira Rodrigues, and his assistant, Alan Batista de Oliveira, an undergraduate student in Electrical Engineering, for adjusting the template used in this work to comply with the standards of the Federal University of Ceará (UFC) library.

To Fundação Cearense de Apoio ao Desenvolvimento Científico e Tecnológico (FUNCAP) and LSBD, for offering funding and crucial resources that enabled the execution of this research.

To Lenovo, for sponsoring my master's research through the Brazilian Informatics Law.

“It simply isn’t an adventure worth telling if there aren’t any dragons.”

(J. R. R. Tolkien)

ABSTRACT

Multi-model architectures enable the querying of data from different sources through a unified interface, providing interoperability among databases. However, support for blockchain is still scarce. Inter-MOON is a new middleware approach that promotes the interoperability of relational databases and blockchain through virtualizing blockchain assets in a relational environment, allowing for the execution of SQL DML commands. Experimental results show that compared to the blockchain, Inter-MOON provides minimal overhead in writing operations or reading operations that retrieve many entries but significant overhead when querying a few entries, trading high performance for powerful querying capabilities.

Keywords: databases; blockchain; database interoperability; data transformation; middleware.

RESUMO

Arquiteturas multi-modelos permitem a consulta de dados de diferentes fontes por meio de uma interface unificada, fornecendo interoperabilidade entre bancos de dados. No entanto, o suporte para blockchain ainda é escasso. Inter-MOON é uma nova abordagem que visa promover a interoperabilidade de sistemas de banco de dados relacionais e blockchain por meio da virtualização de objetos blockchain em um ambiente relacional, permitindo a execução de comandos SQL DML. Resultados experimentais mostram que, em comparação com blockchain, o Inter-MOON proporciona uma sobrecarga mínima em operações de escrita ou operações de leitura que recuperam muitas entradas, mas uma sobrecarga significativa ao consultar poucas entradas, trocando alto desempenho por capacidade de consulta poderosa.

Palavras-chave: bancos de dados; blockchain; interoperabilidade de bancos de dados; transformação de dados; middleware.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BC	Blockchain
BFT	Byzantine Fault Tolerance
BNF	Backus–Naur form
CFT	Crash Fault Tolerance
DApp	Decentralized Application
DApps	Decentralized Applications
DB	Database
DBMS	Database Management System
DCL	Data Control Language
DDL	Data Definition Language
DML	Data Manipulation Language
DPoS	Delegated Proof of Stake
DQL	Data Query Language
DTL	Data Transaction Language
ECDSA	Elliptic Curve Digital Signature Algorithm
EOV	Execute-Order-Validate
EVM	Ethereum Virtual Machine
FK	Foreign Key
HTTP	Hypertext Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
Inter-MOON	**Inter**operable approach to data **M**anagement on relati**O**nal database and bl**O**ckchai**N**
IoT	Internet-of-Things
IPFS	InterPlanetary File System
KVS	Key-Value Store
LCIM	Levels of Conceptual Interoperability Model
LISI	Levels of Information Systems Interoperability
MOON	approach to data **M**anagement on relati**O**nal database and bl**O**ckchai**N**
OLAP	Online Analytical Processing

PBFT	Practical Byzantine Fault Tolerance
PK	Primary Key
PoS	Proof-of-Stake
PoW	Proof-of-Work
RDB	Relational Database
RDBMS	Relational Database Management System
SQL	Structured Query Language
SSI	Serializable Snapshot Isolation
TLS	Transport Layer Security
TPS	Transactions per Second
VM	Virtual Machine

CONTENTS

1 INTRODUCTION

The blockchain, originally conceived as part of the Bitcoin electronic cash system (NAKAMOTO, 2008), allows storing data without a trustworthy third party to create an immutable, irrefutable, and tamper-proof distributed linked list. Blocks containing transactions are linked, creating a chain of cryptographic trust that is replicated across all participating nodes. Blockchains offer strong data integrity and security, crucial for applications requiring tamper-proof records, such as financial transactions, supply chains, and medical records (GUO; YU, 2022). However, blockchains are characteristically slow at writing operations (ZHENG *et al.*, 2018). Additionally, due to the differing design philosophies and lack of standards (TASCA *et al.*, 2017), the data querying capabilities of blockchain are highly dependent on implementation. On the other hand, relational databases excel in performance and offer complex and comprehensive querying via Structured Query Language (SQL), making them ideal for applications that demand fast and efficient data retrieval and manipulation. Therefore, both architectures have different priorities and divergent data models, each presenting unique challenges.

Given the diverse characteristics of data, and the variety of available storage solutions with distinct strengths and weaknesses, enhancing the interoperability of heterogeneous data systems has become imperative (BABCOCK *et al.*, 2002; STONEBRAKER; CETINTEMEL, 2018). Federated databases, multistores, and polystores exemplify this trend. Despite the increasing adoption of blockchain technology (GADEKALLU *et al.*, 2022), enhancing the interoperability of blockchain with other solutions remains a challenge (BELCHIOR *et al.*, 2021; MEYER; MELLO, 2022; MACIEL *et al.*, 2023). Most blockchains are not interoperable with other systems (TASCA *et al.*, 2017) and exhibit a distinct lack of standards, protocols, or drivers that may allow for easier interoperability (YUAN; WANG, 2018; MEYER; MELLO, 2022; MACIEL *et al.*, 2023). (NATHAN *et al.*, 2019).

Interoperability between these systems can harness the strengths of both technologies (NATHAN *et al.*, 2019). In a supply chain scenario, for instance, a system that enables SQL queries across both Blockchain (BC) and Relational Database (RDB) data stores can facilitate comprehensive querying while maintaining consistency. This versatility extends to other applications where interoperability between blockchain and existing relational databases is desirable, including sectors such as healthcare and the Internet-of-Things (IoT) (GUO; YU, 2022). Consider a healthcare system where patient records are stored on a blockchain to ensure integrity, while operational data, such as appointment schedules and billing information, are

stored in a relational database. A doctor needs to access a patient's medical history, verify recent treatments, and schedule a follow-up appointment. Interoperability between the blockchain and the relational database could allow an application to execute a single SQL query to retrieve the patient's complete medical history from the blockchain and the appointment details from the database, ensuring a seamless workflow. If the underlying database or blockchain were to be replaced later, rather than having to rework all the related code, one could simply unplug the old drivers and plug in the newer ones.

The approach to data **M**anagement on relati**O**nal database and bl**O**ckchai**N** (MOON) (MARINHO *et al.*, 2020) aims to serve as a unified entry point for database queries in applications utilizing both blockchain and relational databases. Queries are written in standard SQL, analyzed and mapped by the MOON middleware, and executed in one or both data stores. While MOON simplifies development by eliminating the need for clients to use different query languages and frameworks for each data store, it has limitations. It does not support all Data Query Language (DQL) and Data Manipulation Language (DML) operations, such as DELETE queries or queries containing aggregation or subqueries. Additionally, MOON faces performance issues due to the high latency of multiple database calls and a lack of support for changes to the schema of blockchain entities, all of which heavily hampers interoperability.

This dissertation presents **Inter**operable approach to data **M**anagement on relati**O**nal database and bl**O**ckchai**N** (Inter-MOON), a new approach based on MOON focused on enhancing interoperability between blockchain and relational databases through virtualizing blockchain assets within a relational environment. Inter-MOON supports non-distributed queries containing single SQL statements, allowing for querying any blockchain or relational entity defined in the Inter-MOON entity schema using SQL. Two indexing policies are proposed to enhance interoperability, one table-based and another smart contract-based, depending on the available blockchain features. In summary, the contributions are:

1. The proposal and development of Inter-MOON, a novel approach to interoperability between blockchain and relational databases via the virtualization of blockchain assets in a relational environment, supporting comprehensive SQL DQL and DML grammar. The approach is based on MOON and features extensive improvements and modifications regarding interoperability, such as full support for SELECT queries, brand-new support for DELETE queries, support for schema changes, an overhauled architecture, two indexing policies, support for blockchains with smart contract functionality and improved

performance.

2. An exploration of the interoperability challenges between relational and blockchain databases, including querying, modifying, and deleting blockchain data using SQL queries.
3. The specification of the Inter-MOON smart contract approach, enabling optimized querying of key-value blockchain assets via composite key range queries.

1.1 Publications

Three articles related to this dissertation were submitted, evaluated, and accepted by the scientific community.

1. SÁ, R. A.; MOREIRA, L. O.; MACHADO, J. C. **A modular approach to Hybrid Blockchain-based and Relational Database Architectures**. In: XXI Theses and Dissertations Workshop (WTDBD), 2022, Búzios. Extended Proceedings of the XXXVII Brazilian Database Symposium (SBBD), 2022. p. 154-160.
2. SÁ, R. A.; MOREIRA, L. O.; MACHADO, J. C. **Improving Interoperability between Relational and Blockchain-based Database Systems: A Middleware approach**. In: XXXVIII Brazilian Database Symposium (SBBD), 2023, Belo Horizonte. Proceedings of the XXXVIII Brazilian Database Symposium (SBBD), 2023. p. 115-127. (Candidate for best full paper, runner-up).
3. SÁ, R. A.; MOREIRA, L. O.; MACHADO, J. C. **Inter-MOON: Enhanced Middleware for Interoperability between Relational and Blockchain-based Databases**. Journal of Information and Data Management (JIDM), 2024. (Accepted for publication).

2 THEORETICAL FOUNDATION

This chapter is dedicated to describing background information necessary to understand the context of this work. Focus is given to blockchain and related concepts, as it is a more recent technology and one of the central themes of this work. Interoperability, specifically within the context of blockchain and associated systems, is also discussed, being another main theme of the work.

2.1 Blockchain

In the Bitcoin whitepaper, Nakamoto (2008) describes an electronic payment system based on cryptographic proof instead of trust, supported by a peer-to-peer distributed ledger. Each node in the network contains a local copy of the distributed ledger, and if any single blockchain node is compromised, its information remains available within other nodes. This ledger essentially takes the form of a linked list of blocks that contain transactions validated by the network members. Users create transactions, which are digitally signed and broadcast to the network. They are collected by participating nodes and added to a pool of unconfirmed transactions (the "mempool"). These transactions are selected and assembled into candidate blocks. The blocks are connected by cryptographic hashes, calculated using a Proof-of-Work (PoW) algorithm by the nodes participating in the network. Once a node solves the PoW puzzle and builds a valid block, it is also broadcast to the network for verification and then added to the blockchain once verified. According to Nakamoto, PoW makes it computationally unfeasible for an attacker to tamper with a block without redoing the proof-of-work of the block, and all the blocks after it, ensuring no single entity controls the network entirely and promoting security and data integrity. This architecture eventually came to be called Blockchain.

Blockchain has seen continuous research, with many works attempting to apply the technology to areas outside of digital currencies. These areas include, among others, healthcare, supply chain, information systems, Internet-of-things, databases, security, privacy, and voting (GAMAGE *et al.*, 2020; JAVAID *et al.*, 2021; KRICHEN *et al.*, 2022; GUO; YU, 2022). Other research tackles different approaches to consensus mechanisms, hashing functions, smart contracts, and other properties of blockchain as a data model (GUO; YU, 2022). Some studies also aim to explore mutability in blockchains (POLITOU *et al.*, 2019). Scalability is still an open issue in blockchain research (GAMAGE *et al.*, 2020; ZHOU *et al.*, 2020), as well as a lack of

standards or protocols (YUAN; WANG, 2018; MEYER; MELLO, 2022; MACIEL *et al.*, 2023).

2.1.1 Terminology

The following is a general description of some of the more relevant technical terms related to blockchain architecture and used throughout this work:

Ledger: In a blockchain context, the ledger is the replicated and immutable data structure that tracks all blocks validated by network nodes, maintaining a global state of the system. Encryption and consensus are essential to ensure ledger authenticity, integrity, consistency, and availability (ANTONOPOULOS, 2017).

Asset: Refers to digital objects that represent various forms of data, including digital representations of physical objects or purely digital data such as text, files, tokens, and cryptocurrency. They may be stored either on the blockchain (*on-chain)* or outside of it (*off-chain*), in InterPlanetary File System (IPFS) networks or another data storage solution. If they are stored off-chain, the blockchain will often store a hyperlink to these assets. Users may register, transfer, and own assets, through transactions stored on the blockchain.

Transaction: At its core, a transaction represents an interaction between parties (YAGA *et al.*, 2019). It contains information regarding the state and ownership of assets. This information is stored inside a transaction when a new asset is created or an existing asset is transferred. Each transaction will have a unique ID (identifier), in the form of a hash. Transactions are stored inside blocks after verification and confirmed after the block is validated. The exact validation process is affected by the blockchain implementation and consensus algorithm (eg. PoW blockchains validate blocks through *mining*).

Metadata: Represents contextual information regarding a transaction or block (ZHENG *et al.*, 2017). This information can include details such as the timestamp of the transaction, or the version policy.

2.1.2 Architecture

Fig. 1 shows an example of a blockchain ledger, in which i represents the block number and Tx_n a transaction. As mentioned, the ledger is composed of interconnected blocks, housing a list of transactions each. The maximum number of transactions that a block can contain depends on the block size and the size of each transaction (ZHENG *et al.*, 2017). Each block also includes a header, which connects them, and may also contain some metadata, such as a

timestamp or version. The first block ($i = 0$) of a blockchain network is denominated the *genesis* block. Consequently, it is the fixed common ancestor of all blocks and serves as a secure "root" from which to start building the blockchain (ANTONOPOULOS, 2017).

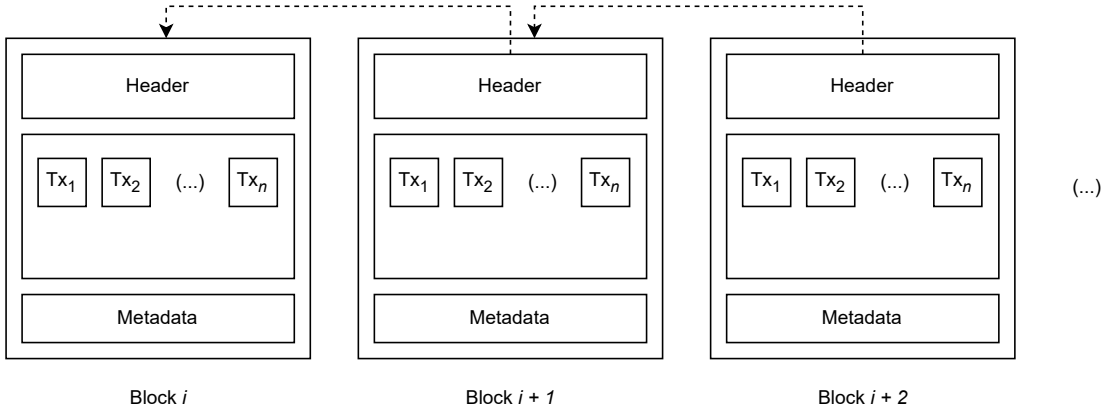

Figure 1 – Blockchain architecture example. Blocks pack transactions and are connected via header.

While the exact structure of a block header depends on the blockchain implementation, there are some identifiable characteristics. The following is a technical description of the elements that commonly make up the header, considering a PoW-based blockchain such as Bitcoin, as per Zheng *et al.* (2017), Yaga *et al.* (2019) and Antonopoulos (2017):

Parent block hash: A 256-bit hash pointer to the previous (*parent*) block that establishes the chain. The genesis block has no value for this property, being the first in the blockchain, thus having no parent.

Merkle Root hash: A pointer to the root of the Merkle Tree built from hashed transactions. See Sec. 2.1.3 for a description of the Merkle Tree.

Timestamp: The time the block was added to the Blockchain in universal time since epoch.

Nonce: In PoW, the nonce (a portmanteau of "number only used once") is the answer to the cryptographic puzzle that must be computed to achieve consensus. It is usually a 4-byte field that starts with a value of 0 and is increased for every hash calculation made.

Target: Like the nonce, it is a particular property of the PoW model, and defines the difficulty of the cryptographic puzzle that must be solved to mine a valid block. If the calculated nonce adds up to a value of less than or equal to the network's difficulty target, the miner's block is added to the blockchain.

The block itself is identified by the block header hash, which is calculated by hashing

the whole block header structure twice using the SHA256 algorithm, also known as double-SHA256 (ANTONOPOULOS, 2017). Note the block header hash is not included within the block's header, only on subsequent blocks or when calculated for verification. Because every block post-genesis contains the hash of the previous block in the chain, if any of the data contained inside of a block changes, the block header hash also changes, breaking the chain of trust. This includes any of the transactions (since the block header also contains the merkle root hash, calculated by hashing each transaction data), as well as other metadata such as the nonce or header timestamp. This property empowers the integrity and immutability of the blockchain.

The structure of each transaction also varies according to the blockchain implementation and application domain. In a financial context such as in cryptocurrency, a transaction is usually a transfer of assets between two users (nodes) in the network (YAGA *et al.*, 2019). Typically, it may include the sender's address (or another relevant identifier) and public key, a digital signature, transaction inputs, and transaction outputs (YAGA *et al.*, 2019). Addresses are strings of digits and characters that identify a user (ANTONOPOULOS, 2017). Transaction inputs represent the assets being transferred. Each input contains the data of the asset as well as the hash of the transaction it originated from. Outputs represent the recipients (YAGA *et al.*, 2019). Transactions must fulfill Atomicity, Consistency, Isolation, Durability (ACID) properties (RAMAKRISHNAN; GEHRKE, 2002).

2.1.3 Cryptography

Regarding cryptography, blockchain employs hash functions, public key cryptography, and merkle trees to realize various tasks and enhance the security and integrity of the network (YAGA *et al.*, 2019).

Hash functions are used to create unique digests (values outputted from hash functions). When hashed using the same hash function, the same input always returns the same result. Any change to the input (e.g., changing a single bit) will result in a completely different output, making them hard to invert and collision-resistant (ROGAWAY; SHRIMPTON, 2004; YAGA *et al.*, 2019). Consequently, blockchain uses hash functions to calculate information such as the block header hash, transaction identifiers, and user addresses (YAGA *et al.*, 2019). The SHA256 algorithm is blockchain's most commonly used hashing function (ANTONOPOULOS, 2017).

Blockchain also uses public key cryptography for verification and authentication (ANTONOPOULOS, 2017; YAGA *et al.*, 2019). Each user owns a pair of private and derived

public keys, which are mathematically connected. The private key must remain secret while the public key can be shared, as the public key alone is not enough to determine the private key. One can encrypt with a private key and then decrypt with the public key and vice versa, enabling trust between unknown parties (YAGA *et al.*, 2019). The private keys are used to generate digital signatures for transactions, which anyone can verify using the respective public key (ZHAI *et al.*, 2019; ANTONOPOULOS, 2017). This ensures that a third party has not tampered with the data of a transaction. The typical digital signature algorithm used in blockchains is the Elliptic Curve Digital Signature Algorithm (ECDSA) (ANTONOPOULOS, 2017).

Additionally, blockchain commonly employs merkle trees to verify the integrity of large-scale data (ANTONOPOULOS, 2017; ZHAI *et al.*, 2019). It is used to summarize all the transactions in a block and produce a digital fingerprint of the transaction set, providing an efficient process for verification of whether a transaction is present in a block. A merkle tree is constructed by recursively hashing pairs of nodes until there is only one hash, called the root, or merkle root (ANTONOPOULOS, 2017). See Fig. 2 for an example of this construction process. The data of each transaction (Tx) is hashed to construct the leaf nodes. Because it is a binary tree, if there are an odd number of transactions to summarize, the last transaction hash is duplicated to create an even number of leaf nodes. To verify if a transaction is included in a block, a node only needs to produce $log2(N)$ 32-byte hashes, constituting an authentication path or merkle path connecting said transaction hash to the root of the tree (ANTONOPOULOS, 2017). The cryptographic hash algorithm used in Bitcoin's merkle trees is also a double-SHA256, same as the block header hash (ZHAI *et al.*, 2019).

2.1.4 Consensus

Because they are distributed and decentralized, blockchain networks use consensus protocols (also called consensus algorithms or mechanisms) such as PoW to agree on what actions should be performed. For example, whether a mined block or transaction is valid and can be added. According to Xiao *et al.* (2020), a consensus protocol is said to be Crash Fault Tolerance (CFT) or Byzantine Fault Tolerance (BFT) if it can tolerate a certain amount of crash or Byzantine faults while remaining functional. Common causes of crash failure include power shutdown, software errors, and denial-of-service attacks. A Byzantine failure, however, is more severe in that the process can act arbitrarily while appearing normal, send contradicting messages to other processes, and sabotage consensus. A BFT protocol is by definition also CFT, and

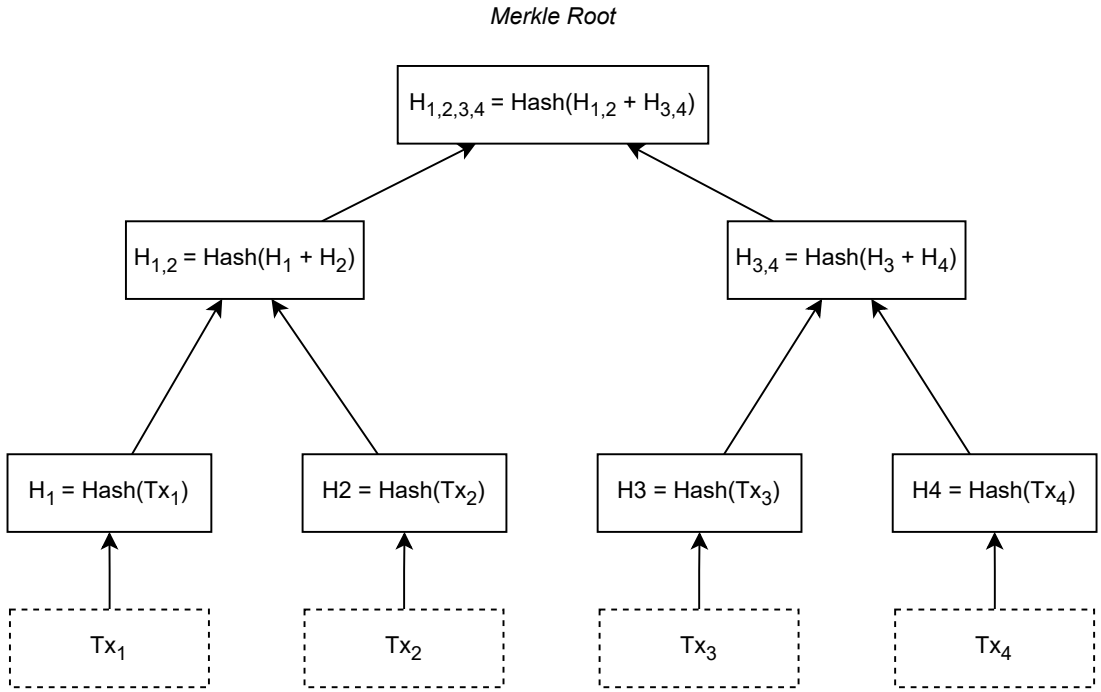

Figure 2 – Representation of the Merkle Tree construction.

additionally abides by four requirements:

Termination: Every non-faulty process eventually decides on an output.

Agreement: Every non-faulty process eventually decides on the same output, say y.

Validity: If every process begins with the same input, say x, then $y = x$.

Integrity: The defined output y must have been one that was proposed by a non-faulty process.

For a BFT protocol to work, the number of processes N in the network must satisfy $N \geq 3f + 1$, where f is the maximum number of Byzantine faulty processes. This means that to tolerate f Byzantine faults, there must be at least $3f + 1$, processes in total. Specifically for blockchain, BFT or CFT consensus protocols are often used with modifications, such as the separation of consensus and execution, due to the extra cryptographic computations present in blockchain (XIAO *et al.*, 2020). Blockchain networks may also have other particular needs that influence the protocol choice. For example, networks may prioritize either speed or security or prefer a mechanism that allows for staking tokens. Consequently, various types of consensus protocols have been proposed and developed, both for distributed systems and specifically for blockchain use. They can be classified using a variety of attributes, such as finality, scalability, accessibility, and agreement (LASHKARI; MUSILEK, 2021). The following subsections describe some of blockchain's more commonly seen consensus algorithms.

2.1.4.1 Proof-of-Work

PoW is the consensus algorithm used by Bitcoin (NAKAMOTO, 2008). In PoW, nodes compete to solve a complex mathematical puzzle, and the first to solve it broadcasts the solution to the network to verify it (YAGA *et al.*, 2019). The puzzle is simply designed to be a complex computation, so solving it is difficult but validation is easy. The solution arrived at by the first solver is the "proof" they have performed "work". The puzzle-solving process is called *mining*, while the nodes solving the puzzle are called *miners*.

In Bitcoin, the puzzle is to repeatedly calculate the hash digest of the candidate block header by changing the nonce property until the resulting digest is less than a specific target value. Because each attempt hashes the entire block header, it is an overall intensive computation. Over time, the target value is automatically modified so that the network block publication rate remains at around 10 minutes and the puzzle's difficulty remains appropriate, given the network computing power. The complexity of the computation is also purposeful in making it harder for malicious actors to solve the puzzle and take control of the network — considering a large enough network, it would be prohibitively expensive for an attacker to take over, as the required processing power to alter a block and re-mine all subsequent blocks would be sky-high, preventing Sybil attacks (NAKAMOTO, 2008; YAGA *et al.*, 2019). The miner who successfully mines a block is rewarded with newly minted cryptocurrency (block reward) and transaction fees from the transactions included in the block to incentivize further contribution to the network processing power.

Two or more valid blocks may be generated at nearly the same time, leading to the existence of branches, or forks (ZHENG *et al.*, 2018; YAGA *et al.*, 2019). However, it is unlikely that the next block or any thereafter will also be generated simultaneously. Consequently, PoW awaits the next block generation while the forks exist and then considers the longest fork to be the authentic one. The remaining fork is *orphaned* and its transactions, if not present in the now valid blockchain, return to the mempool. Consider Fig. 3 depicting two forks created by simultaneously validated blocks B11 and G11. Miners work on both forks and add the newly generated blocks to one of them. When a new block, say B12, is added after B11, the miners working on fork G11-G12 will switch to B12. The fork G11-G12 is orphaned as B11-B16 becomes the authentic blockchain.

By design, PoW is energy-intensive contentiously. This property has raised concerns about exorbitant energy consumption, environmental impact, and related topics in blockchain

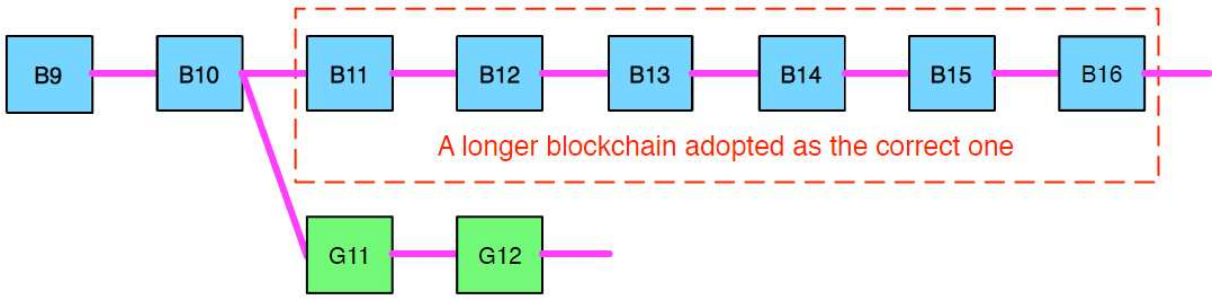

Figure 3 – Example of a scenario where two branches of the replicated ledger exist. Ownership of Zheng *et al.* (2018).

research (VRANKEN, 2017). Moreover, the reward given to successful miners promotes competition. This may lead to decentralization as miners form mining pools, coalitions of allied miners who split rewards (YAGA *et al.*, 2019). Additionally, there is a socioeconomic and environmental impact as miners seek more powerful hardware to keep up with the competition (O'DWYER; MALONE, 2014).

2.1.4.2 Proof-of-Stake

In Proof-of-Stake (PoS), a special group of users of a given blockchain network, often called *validators*, are responsible for creating new blocks based on the amount of cryptocurrency they hold and are willing to "stake" (YAGA *et al.*, 2019). Validators are incentivized to act honestly, as their stakes within the system are at risk. The stake is often an amount of cryptocurrency that the network user has invested into the system. This investment can be done through a variety of means, such as by locking it via a special transaction type, by sending it to a specific address, or holding it within special wallet software (YAGA *et al.*, 2019). PoS is considered a more energy-efficient alternative to PoW — because the validators are few, there is no need for computationally intensive operations (XIAO *et al.*, 2020). In such systems, the reward for block publication is usually the earning of user provided transaction fees (YAGA *et al.*, 2019).

Once staked, currency cannot be spent, and the likelihood of that validator being chosen is tied to the ratio of their stake to the overall blockchain network amount of staked currency. Given this, Xiao *et al.* (2020) identifies four classes of PoS protocols: chain-based PoS, committee-based PoS, BFT-based PoS, and Delegated Proof of Stake (DPoS).

Chain-based PoS inherit many attributes of PoW, with a different block generation algorithm. In chain-based PoS, the validator is chosen at random, based on their stake in the system. So, if a user had 42% of the entire blockchain network stake they would be chosen 42%

of the time; those with 1% would be chosen 1% of the time (YAGA *et al.*, 2019). Because of the inherent unfairness of validators being chosen based solely on investment, there are often other factors at play (ZHENG *et al.*, 2018; YAGA *et al.*, 2019). For example, Peercoin (KING; NADAL, 2012) uses a *coin age* metric for stake valuation, which allows the value of a stake to increase linearly with time since staking. Once it is chosen, the "age" property of a stake is reset, allowing older stakes and larger stakes to both contribute towards the likelihood of a validator being chosen.

In committee-based PoS, a pseudo-randomly organized committee of stakeholders is selected based on their stakes and generate blocks in turns (XIAO *et al.*, 2020). The selected sequence is broadcast so that it is the same for all stakeholders, and those with higher stakes may take up more spots in the sequence. There are variations, such as in Ouroboros (KIAYIAS *et al.*, 2017), where time is divided into fixed periods called epochs, which are further divided into slots. Stakeholders with enough stake can become *electors*, who collectively elect slot leaders (i.e. the committee) for the next epoch. Slot leaders are responsible for generating the block in that slot.

BFT-based PoS essentially incorporates BFT consensus to provide fast and deterministic block finalization (XIAO *et al.*, 2020). There are multiple consensus rounds, and each involves a committee of validators, where one is designated as the *proposer* to suggest a new block. Validators then engage in *pre-voting* and *pre-commit phases* to signal their agreement with proposed blocks. Final block confirmation requires a majority consensus, typically two-thirds or more, ensuring Byzantine fault tolerance and preventing forks. This kind of PoS protocol is based on Practical Byzantine Fault Tolerance (PBFT). See also Sec. 2.1.4.3.

As for DPoS, it is a democratic form of committee-based PoS in that the committee is chosen via public stake delegation (SCHUH; LARIMER, 2017; XIAO *et al.*, 2020). Small stakeholders vote on *delegates*, who are responsible for creating blocks. Unwanted delegates can also be voted out (YAGA *et al.*, 2019; ZHENG *et al.*, 2018). Votes are weighed, so the more stake a stakeholder has, the higher the value of their vote (YAGA *et al.*, 2019). Because it is a fully democratic election, there are external socioeconomic factors involved in voting, as delegates receive daily vote-reward proportional to the votes received (XIAO *et al.*, 2020). The threat of being voted out motivates delegates to act honestly (YAGA *et al.*, 2019).

Naturally, many PoS protocols run the risk of centralization, due to often tying wealth within the system with staking power (XIAO *et al.*, 2020).

2.1.4.3 Practical Byzantine Fault Tolerance

To conclude, PBFT by Castro *et al.* (1999) is one of the first distributed BFT protocols. Tendermint (KWON, 2014) and Hyperledger Fabric (ANDROULAKI *et al.*, 2018) use PBFT as the basis for their consensus mechanisms. PBFT is capable of achieving consensus in the presence of up to 1/3 malicious replicas with a few rounds of exchanging messages (ZHENG *et al.*, 2018). In PBFT blockchains, nodes take turns proposing and validating blocks of transactions. In each round, one node is designated as the primary (*leader*) to propose a block. The remaining nodes act as backups and validate the proposed block through a series of phases: *pre-prepare*, *prepare*, *commit*, and *checkpoint* (MONRAT *et al.*, 2019). During the pre-prepare phase, the primary proposes a block and broadcasts it to all backups. Upon validation, backups send prepared messages to indicate acceptance of the block. Once a backup receives prepared messages from a majority, it sends a commit message, signaling agreement. After receiving commit messages from a threshold number of nodes, the block is considered committed and added to the blockchain.

Tendermint is very similar to PBFT, but also takes on the staking characteristic of PoS: nodes have to stake to become validators, and only validators can propose a block (ZHENG *et al.*, 2018). However, the "weight" of the stake does not influence the "weight" of the vote during the *prepare* phase, only the likelihood of being chosen as the *leader* during a round (XIAO *et al.*, 2020). Additionally, there is an equal-sharing-style incentive mechanism instead of winner-takes-all. For every block height, the block reward is distributed among the block proposer and validators from whom the proposer received *Commit* votes (XIAO *et al.*, 2020). Hyperledger Fabric is also based on PBFT. See Sec. 2.2.

2.1.5 Taxonomy

Blockchain systems are often roughly categorized into three types: public blockchain, private blockchain and consortium blockchain (ZHENG *et al.*, 2017; ANDONI *et al.*, 2019). See Table 1 for an overview of the characteristics of each type. In public blockchains, anyone can join as a new user or node, and all participants can perform all available actions. In private and consortium blockchains, only specific nodes participate in consensus, creating a partially decentralized design. Additionally, in private blockchains, only certain users or organizations are allowed to join the network at all. Generally speaking, the difference between

a private and consortium blockchain is the number or participating organizations — if only one is present, it is private, otherwise, it is a consortium. As such, blockchains can also simply be categorized as permissioned (for private and consortium chains) or permissionless (for public chains) (CASINO *et al.*, 2019). Consensus models in permissioned blockchains are usually faster and less computationally expensive, due to the fewer number of participants with consensus determination rights (YAGA *et al.*, 2019; ANDONI *et al.*, 2019). However, this characteristic may also result in a less tamper-proof system (ZHENG *et al.*, 2017). If a single entity controls who can publish blocks, the users of the blockchain will need to have trust in that entity, leading to a more centralized design (ZHENG *et al.*, 2017; YAGA *et al.*, 2019).

Property	Public Blockchain	Consortium Blockchain	Private Blockchain
Visibility	Public	Public or restricted	Public or restricted
Consensus process	Permissionless	Permissioned	Permissioned
Consensus determination	All nodes	Selected set of nodes	One organization
Decentralization	Full	Partial	Low
Immutability	Nearly impossible to tamper	Could be tampered	Could be tampered
Transaction Output	Low	High	High

Table 1 – Blockchain properties per permission model. Adapted from Zheng *et al.* (2017).

2.1.6 Smart Contracts

Szabo (1996) originally defines smart contracts as computerized transaction protocols that execute the terms of a contract. It is a digital agreement between multiple parties written in code, without a liaison (DELMOLINO *et al.*, 2016). Code in the blockchain comprises the transaction conditions, so a contract is distributed across the network as part of the blockchain, in which peers may join (TOLMACH *et al.*, 2021). Consequently, smart contract code is stored, verified, and executed on a blockchain. They can read or write data to the blockchain ledger automatically, depending on the terms and conditions of the contract or when invoked via client messages or API (WOOD *et al.*, 2014; ANDROULAKI *et al.*, 2018). Besides functions for writing or reading states, the contract may also store contract-scoped constants or generic data. A smart contract may even invoke a function from another if the blockchain allows it. See Fig. 4 for an example of how a smart contract might be written, considering a supposed *Car* entity that may be queried, transferred or updated by external applications via smart contract.

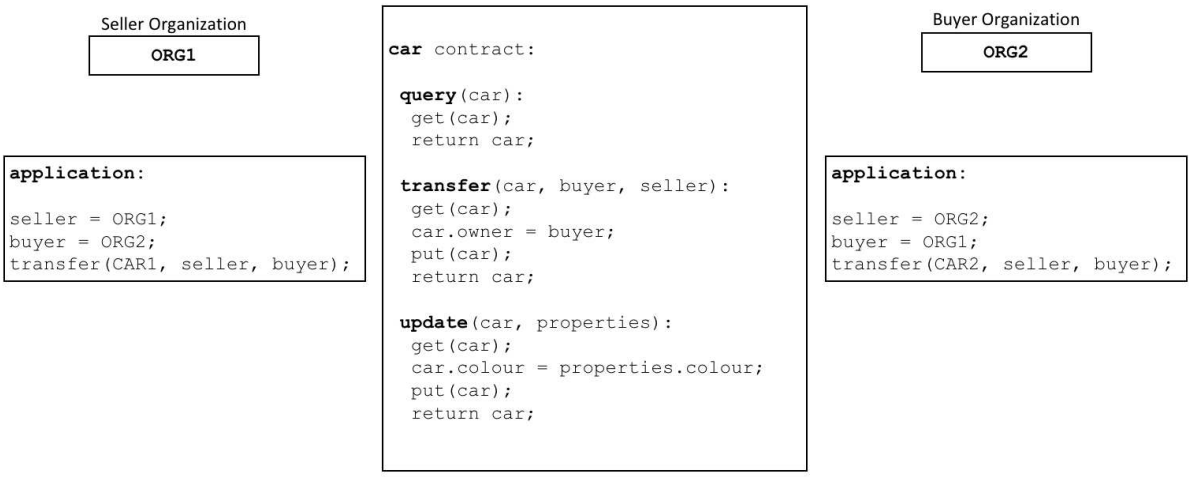

Figure 4 – Example of a smart contract of a supposed *Car* entity [1].

There is no standard specification for smart contracts (TOLMACH *et al.*, 2021), so each blockchain may implement the concept differently. Conceptually, smart contracts are language agnostic. Ethereum (WOOD *et al.*, 2014), for instance, implements smart contracts written in Solidity, a language dedicated to smart contract development. It also associates each contract with a balance and an address within the blockchain. In Hyperledger Fabric (ANDROULAKI *et al.*, 2018), smart contracts are called "chaincodes", can be written in standard programming languages like Go, Java, and JavaScript, and have no such association as in Ethereum.

Smart contracts are associated with the application domain of the blockchain. In a supply chain scenario, smart contracts can be used to keep track of products, automate delivery checks, track sales targets and limits, and verify transportation conditions such as humidity and temperature (MOHANTA *et al.*, 2018; GADEKALLU *et al.*, 2022). In cryptocurrency and finances, such as in Ethereum, they are often implemented to handle transactions between peers, balance checking, or Decentralized Application (DApp) development (BUTERIN *et al.*, 2014; MOHANTA *et al.*, 2018).

2.2 Hyperledger Fabric

Fabric is a modular and extensible open-source system for deploying and operating permissioned blockchains and one of the Hyperledger projects hosted by the Linux Foundation (ANDROULAKI *et al.*, 2018). Unlike most blockchains, it supports the configuration of consensus protocols and runs distributed applications written in general-purpose programming

[1] https://hyperledger-fabric.readthedocs.io/en/latest/smartcontract/smartcontract.html. Accessed: May, 22, 2024.

languages, without systemic dependency on a native cryptocurrency. These characteristics make Fabric suitable for general-purpose blockchain development and research, such as in this work.

See Fig. 5 for a depiction of the Fabric concept of *ledger*. While most blockchains feature the ledger as a distributed sequential log, fabric incorporates the idea of the *world state*, determined by the blockchain component of the ledger. The world state is by default a Key-Value Store (KVS) database that holds the current values of ledger states. The ledger (ie. the world state and the blockchain) is distributed across the network.

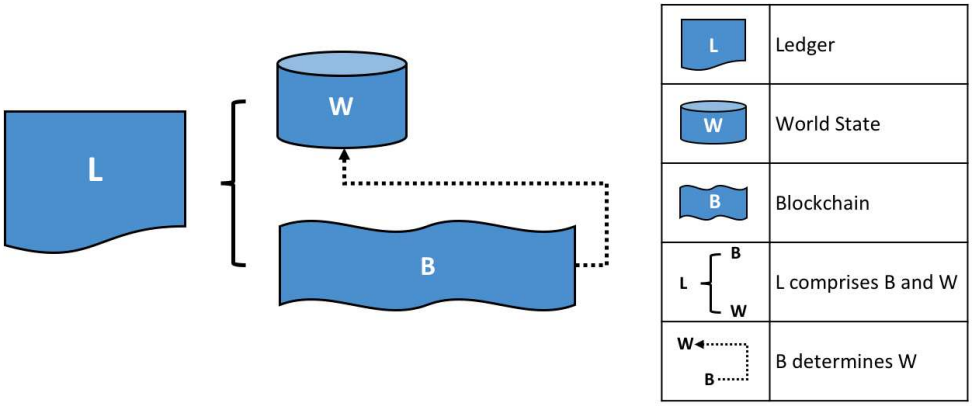

Figure 5 – Fabric divides the concept of the *ledger* into the world state and blockchain [2].

The blockchain, depicted in Fig. 6, is extremely similar to any standard blockchain implementation. Transactions are immutable, collected inside blocks appended to the blockchain upon validation. Each block's header includes the merkle tree root (Sec. 2.1.3), as well as a hash pointer of the previous block header. The transactions represent world state changes and are collected into blocks by the ordering service. The ordering service will be explained further below. Transactions contain some metadata, the before and after values of the updated world state as a *read-write set*, the digital signature of the user who proposed that transaction, the parameters supplied by the user for the world state update, and the list of endorsing organizations who validated that transaction, according to the endorsement policy.

Nodes in a Fabric network can be *peers* or *orderers*. Peers belong to organizations, execute transaction proposals, and contribute towards validating transactions. An endorsement policy dictates which and how many organizations must sign a transaction for it to be considered valid. The rights of each peer (eg. which peers can validate transactions) are defined by a separate

[2] https://hyperledger-fabric.readthedocs.io/en/release-2.5/ledger/ledger.html. Accessed: May, 22, 2024.
[3] See footnote 2

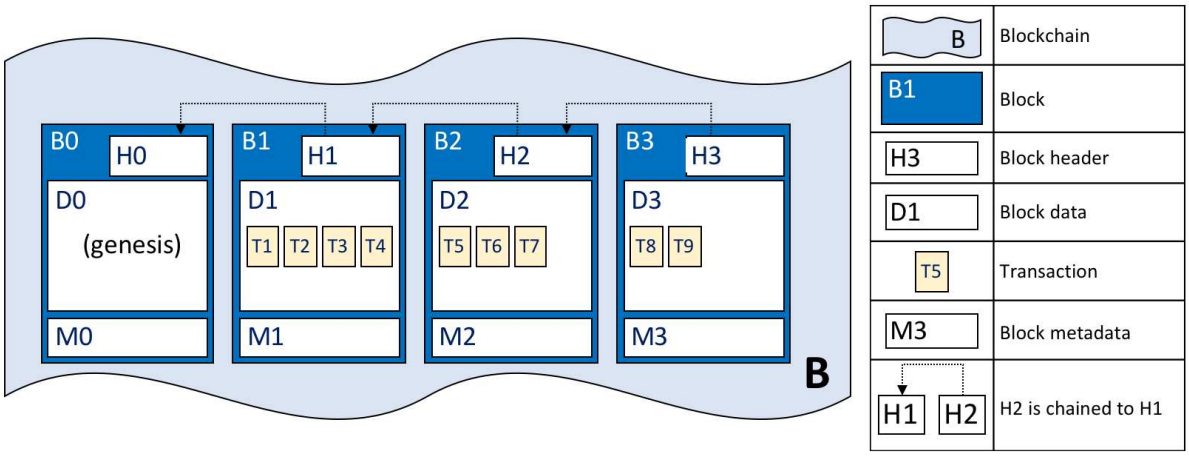

Figure 6 – Fabric architecture example [3].

organization policy. This is the basis of the permissible model of Fabric: nodes must belong to a trusted organization and have their rights decided upon by it, and only trusted organizations validate transactions.

Orderers collectively form the ordering service, which manages consensus. It establishes the order of all transactions in Fabric. Orderers are unaware of the application state and do not participate in the execution nor the validation of transactions. This approach is based on PBFT, but further divides consensus into separate ordering and validation services for better modularity (XIAO *et al.*, 2020). Consensus has three phases: Proposal and Endorsement, Submission and Ordering, and Validation and Commitment. The following is a step-by-step summary of how consensus is achieved in Fabric:

1. **Proposal and Endorsement**
   - 1.1 A client application creates a transaction proposal and sends it to one or more endorsing peers.
   - 1.2 Each endorsing peer simulates the transaction using its current state and the proposed changes. The peer then generates an endorsement, which includes the read/write set of the transaction (what was read from and written to the ledger) and a signature.
2. **Submission and Ordering**
   - 2.1 The client collects the endorsements from the endorsing peers. Once it has enough endorsements to satisfy the endorsement policy, the client assembles the transaction and sends it to the ordering service.
   - 2.2 The ordering service receives transactions from multiple clients, orders them, and

collects them into blocks. The ordering service does not process the content of transactions but focuses on ensuring a deterministic order.

3. **Validation and Commitment**

   3.1 The ordered blocks are then distributed to all committing peers (Committers). Each peer validates the transactions in the block by checking the endorsements against the endorsement policy and ensuring there are no conflicts (e.g., double-spending). Valid transactions are committed to the ledger.

   3.2 After validation, the ledger state is updated with the results of the valid transactions, and the new state is reflected across the network.

Note that the ordering service is pluggable in Fabric. By default, three options are available: a centralized, single-node implementation, used in development and testing; A CFT ordering service running on a cluster, based on Apache Kafka, and another based on Raft.

Finally, Fabric also offers support for smart contracts (See Sec. 2.1.6), called "chaincodes". Chaincodes are agreed upon by organizations to be installed within the blockchain. They are *invoked* by clients to query the world state and create transaction proposals. Every transaction must be created via chaincode.

2.3 Interoperability

In computing, interoperability can be defined as the ability of two or more software components to cooperate despite differences in language, interface, and execution platform (WEGNER, 1996). Research regarding interoperability often tackles the concept within specific domains or discusses theoretical contributions. Additionally, it is commonly described using contextual layers, or levels (HASSELBRING, 2000; MACIEL *et al.*, 2023). In truth, there are various interoperability models or frameworks, such as Levels of Information Systems Interoperability (LISI) (KASUNIC; ANDERSON, 2004) and Levels of Conceptual Interoperability Model (LCIM) (TOLK *et al.*, 2007). Maciel *et al.* (2023) breaks down different understandings of interoperability over the years as essentially three "levels": technical (referring to the standardization of the interfaces between hardware and software), syntactic (interoperable systems sharing a common understanding of the structure and syntax of transmitted messages), and semantic (sharing an understanding of the meaning of the exchanged messages, enabling the combination of data for meaningful processing). This definition is corroborated by Mello *et al.* (2022), detailing semantic interoperability as using domain concepts, context knowledge, and

formal data representation to enable meaningful exchange of information.

Specifically relating to blockchain and blockchain-based solutions, according to Xu *et al.* (2020), the problem of interoperability is commonly tackled in one of three contexts: interoperability between different blockchains, interoperability between Decentralized Applications (DApps) using the same blockchains, and interoperability between blockchain and other systems, such as is the case in this work, with Relational Database Management System (RDBMS). Tasca *et al.* (2017) identifies three layouts for interoperability within a blockchain system:

Implicit Interoperability: when the smart contracts that specify conditions under which a particular transaction (or event) is to take place can be written in a Turing complete blockchain script language. In this context, implicitly any kind of condition can be specified, even those involving specific status in other systems, theoretically allowing interaction from a blockchain solution to any Application Programming Interface (API) tool or interface.

Explicit Interoperability: when the script language is not Turing complete or the system has specific tools implemented that enable interoperability with the real world. The given example is Bitcoin with Counterparty[4], a platform built on top of Bitcoin that extends its ecosystem with smart contract support.

No Interoperability: A blockchain without any kind of possibility to interact with other systems. The given example is Bitcoin in the absence of external solutions, which has no interoperability implemented. According to Tasca *et al.* (2017), this is the case for most existing blockchain-based systems whose script language is not Turing complete.

The interoperability challenge between blockchains is commonly manifested in the lack of proper standardization (YUAN; WANG, 2018; MEYER; MELLO, 2022; MACIEL *et al.*, 2023). Most blockchains are not interoperable and are built according to differing design philosophies (TASCA *et al.*, 2017). Without standards, protocols, or specifications, different ledgers cannot directly communicate with each other or outside systems, requiring specialized solutions and compromise between participating parties (MISTRY *et al.*, 2020). Ontologies and conceptual models can be helpful tools in enhancing semantic interoperability (TASCA *et al.*, 2017). Standards and protocols have also been employed to tackle syntactic and technical interoperability (MEYER; MELLO, 2022). For technical interoperability in cross-blockchain scenarios, adapters are also often employed to enable communication (MEYER; MELLO, 2022).

[4] https://www.counterparty.io/. Accessed: June 25, 2024.

The approach presented in this work is a middleware, thus consisting of a tool primarily geared towards enhancing the technical interoperability of blockchain and relational databases. By decreasing the dependence of applications on a particular system, middleware increases the ease of moving applications to new systems and decreases dependence on systems that might fall out of favor (KASUNIC; ANDERSON, 2004). There are also certain aspects relating to syntactic and semantic interoperability. For example, the presented approach applies a pseudo-relational schema (Sec. 4.2.1) to blockchain entities, a characteristic of both semantic and syntactic interoperability. Additionally, the mapper (Sec. 4.1.2) also plays a role in enhancing communication between the blockchain and relational data stores via data transformations, relating to syntactic interoperability. The smart contract specification in Sec. 4.2.2.2 also constitutes efforts toward enhancing technical interoperability. Sec. 4.1.9 describes the requirements a blockchain must fulfill to be supported by the presented approach.

3 RELATED WORK

This chapter explores how this work relates to the state of the art. Related works are within the context of Interoperability of blockchain or blockchain-based data storage solutions to relational database systems; The usage of SQL in blockchain data exploration or querying; Blockchain-based database systems; Database-based blockchain systems. The search for papers from 2019 to 2023 was carried out in the following repositories: ACM Digital Library, IEEE Xplore Digital Library, Proceedings of the Brazilian Symposium on Databases, and Google Scholar. The main keywords used were “blockchain database”, “blockchain relational interoperability”, “blockchain sql”, and “blockchain database middleware”. To conclude, Sec. 3.8 shows a comparison overview between this work and related works.

3.1 Yue *et al.* (2019)

Yue *et al.* (2019) discusses the authors’ experience using SQL databases for blockchain analysis, elaborates upon the characteristics of the Bitcoin blockchain that make it an interesting database case, and examines the relative merits of three different methods for storing and querying Bitcoin data through relational databases: (1) a local SQL database, (2) the cloud, and (3) third-party web-based interfaces.

In (1), a tool called Abe[1] is used to generate a PostgreSQL database from a local Bitcoin blockchain node. The tool directly reads the block files from the file system and generates a series of tables and views on a SQL database according to the data. The work does not describe the exact structure of each generated table but explains that the database stores information related to blocks and transactions using 17 tables and 4 views (Table 2).

	(1) Abe	(2) BigQuery	(3) bcsql
Tables	17	2	13
Views	4	0	0
Stored Derived Columns	5	0	15

Table 2 – Overview of generated databases of each technique. Adapted from Yue *et al.* (2019).

The second approach (2) uses Google BigQuery[2], a cloud-based enterprise data warehouse platform for real-time data analysis using SQL. BigQuery’s extensions to SQL allow columns to store records and structures. Structures can be expanded into tables by using the

[1] https://github.com/bitcoin-abe/bitcoin-abe. Accessed: June 25, 2024.
[2] https://cloud.google.com/bigquery/. Accessed: June 25, 2024.

UNNEST function, which can then be used like derived tables in JOIN and SELECT clauses. BigQuery also offers a public Bitcoin dataset which, due to the BigQuery characteristics described above, contains only two tables: blocks and transactions (See Table 2). Internal structures are stored in columns. For example, the many TXIn (transaction inputs) and TXOut (transaction outputs) of a transaction are stored in the 'inputs' and 'outputs' columns of the transactions table, respectively.

The third approach (3) uses another tool, *bcsql* (*blockchainsql.io*)[3]. This tool presents a web interface that allows the submission of SQL statements to query its proprietary SQL Bitcoin database. It uses Microsoft SQL Server and has 13 tables (Table 2), with many stored derived columns to improve performance.

To conclude, the paper discusses the merits of each solution regarding database education and design, using derived columns as an example. Stored derived columns have to be recomputed whenever there are changes. For write-intensive databases, this can degrade the performance significantly. Thus, most DBMS compute the values at query runtime rather than physically store them in derived columns by default. However, the authors argue that append-only scenarios, such as with the Bitcoin blockchain, do not have this problem, as stored derived columns will never be recomputed.

While the full discussion is irrelevant to this work, the paper highlights a desire to use a database to store and query blockchain data. Using SQL, it is possible to execute extensible and complex querying operations with a familiar language, a point both this work and Yue *et al.* (2019) agree upon. Conceptually, approaches (1) and (2), which store blockchain data in a local database and cloud-based platform respectively, are applicable within the context of Inter-MOON. However, the scenarios presented in both works are different. Yue *et al.* (2019) assumes a fixed scenario of data analysis and exploration of Bitcoin transactions. On the other hand, this work tackles a generalist scenario of blockchain as a data store. The structure of a Bitcoin transaction is always the same, as it is always within the context of Bitcoin as a cryptocurrency. However, in this work, we cannot assume that every transaction will have the same structure (schema), as it will depend on the specific use case, such as supply chain, e-health, or provenance.

[3] https://blockchainsql.io. Accessed: June 25, 2024.

3.2 Nathan *et al.* (2019)

Nathan *et al.* (2019) proposes a "blockchain relational database" by developing a blockchain layer on top of the PostgreSQL relational database. This blockchain layer represents a permissioned blockchain system, whereupon each participating organization can contribute clients, database peers, and ordering service nodes to the network. The key components of the system are described:

Clients: Each organization has an administrator responsible for onboarding users. Clients and administrators possess digitally signed certificates for authentication and access control. They can submit transactions and receive updates through a secure communication channel.

Database Peer Nodes: Organizations can run database nodes in the network. These nodes communicate securely using Transport Layer Security (TLS) and maintain individual ledger replicas. They also independently execute smart contracts as stored procedures and validate/commit transaction blocks.

Ordering Service: To ensure agreement on transaction ordering among potentially untrusted nodes, a pluggable consensus service is implemented. This allows leveraging existing CFT or BFT consensus algorithms. The ordering service consists of consensus nodes owned by different organizations, each with its digital certificate. Consensus generates a block of transactions that is broadcast atomically to all database nodes.

To achieve this, two approaches are studied: The first approach, denominated *order-then-execute* (Fig. 7), orders all the transactions through a consensus service, and then nodes execute them concurrently. In the second approach, *execute-order-in-parallel* (Fig. 8), execution happens on nodes without prior knowledge of ordering, which occurs in parallel through a consensus service.

Both proposed approaches are implemented on top of PostgreSQL. The implementation comprises the addition of several new components to the PostgreSQL architecture, and the modification of some existing components. Additions include a communication middleware to handle communication with other nodes and the orderer and a block processor responsible for executing the commit phase of a block. A *pgLedger* system catalog table maintains information about each transaction and aids recovery and provenance queries; Another table denominated *pgCerts* stores the cryptographic credentials of blockchain users. The implementation also adds support to a new type of read-only provenance query, that can access all committed rows in tables for audit.

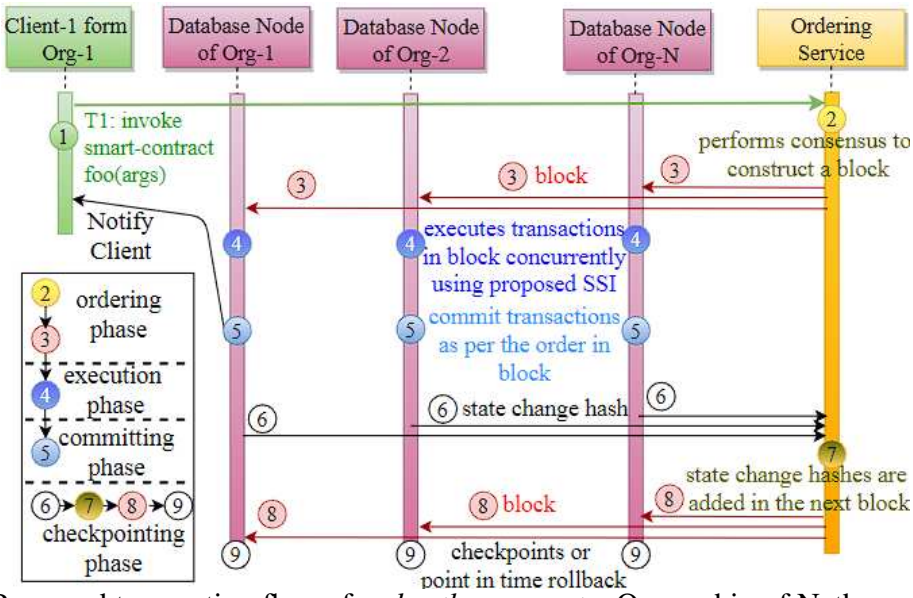

Figure 7 – Proposed transaction flow of *order-then-execute*. Ownership of Nathan *et al.* (2019).

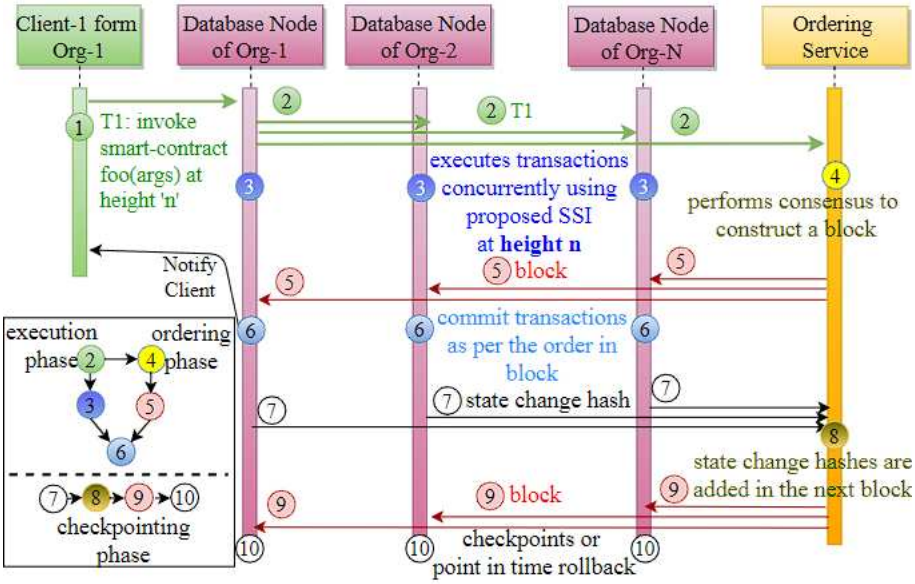

Figure 8 – Proposed transaction flow of *execute-order-in-parallel*. Ownership of Nathan *et al.* (2019).

The application interface and PL/SQL procedures were modified to support blockchain transactions and provenance queries, enforce deterministic behavior, and restrict certain functions and statements to ensure consistent transaction processing. The row visibility logic was modified to use block height, preventing transactions from seeing rows affected by concurrent transactions and enforcing index-based read access to avoid phantom or stale reads. Finally, the Serializable Snapshot Isolation (SSI) mechanism was modified to incorporate block-aware abort rules during commit and manage write-write dependencies, ensuring that only one transaction can write to a row to prevent lost updates.

Given these modifications, the approach supports the execution of a custom SQL

CREATE FUNCTION statement to create a new smart contract, CREATE OR REPLACE FUNCTION to update an existing smart contract, or DROP FUNCTION to delete an existing smart contract. In a table, each row has two additional header elements, “xmin” and “xmax”, which are the IDs of the transactions that created and deleted the row, respectively. Every update to a row is a DELETE followed by an INSERT (both in the table and index). Deleted rows are flagged by setting “xmax” instead of being truly deleted.

In summary, Nathan *et al.* (2019) details extensive modifications to the PostgreSQL architecture to add a blockchain layer on top of the relational database, which naturally sets it apart from Inter-MOON. Inter-MOON does not interfere with the architecture of either system but instead creates a multi-modal architecture by acting as a middleware, enabling independent systems to work in tandem via SQL. One similarity to Inter-MOON is the support for SQL DELETE. While the specifics are different, conceptually, both implement a “soft-delete” mechanism to delete blockchain data. On the other hand, Nathan *et al.* (2019) supports the manipulation of smart contracts via stored procedures within the modified PostgreSQL, while Inter-MOON details a specification for the development of smart contracts in a language-agnostic manner.

3.3 Zhu *et al.* (2020)

Zhu *et al.* (2020) proposes SEBDB (Semantics Empowered BlockChain DataBase), a permissioned blockchain database that aims to incorporate relational data semantics into the blockchain. It achieves this by associating each transaction with a tuple representing a table in a relational database schema. Additionally, SEBDB employs a language similar to SQL for communication, making it familiar to users with database experience. To optimize data storage and retrieval, SEBDB utilizes a separate off-chain RDB for storing data not directly residing on the blockchain network. The approach enables joins between on-chain (blockchain) data and off-chain (RDB) data, a capability the authors named “on-off join.” The system architecture is depicted in Fig. 9, and consists of five layers:

API Layer: Provides APIs, access control, and smart contracts. Access control verifies user permissions and smart contracts use an SQL-like language to define applications and access data.

Query Engine Layer: Parses, optimizes, and executes SQL-like queries.

Storage Layer: Manages data storage and indexing (indexes, Merkle tree, cache). It also

handles searching, scanning, and inserting block data.

Consensus Layer: Uses a "plug-in" approach to allow users to choose different consensus protocols (in the paper, Kafka[4] and PBFT are specified) based on their needs.

Network Layer: Leverages the Gossip protocol for communication, commonly used for failure detection, membership management, and block propagation in distributed systems and blockchains.

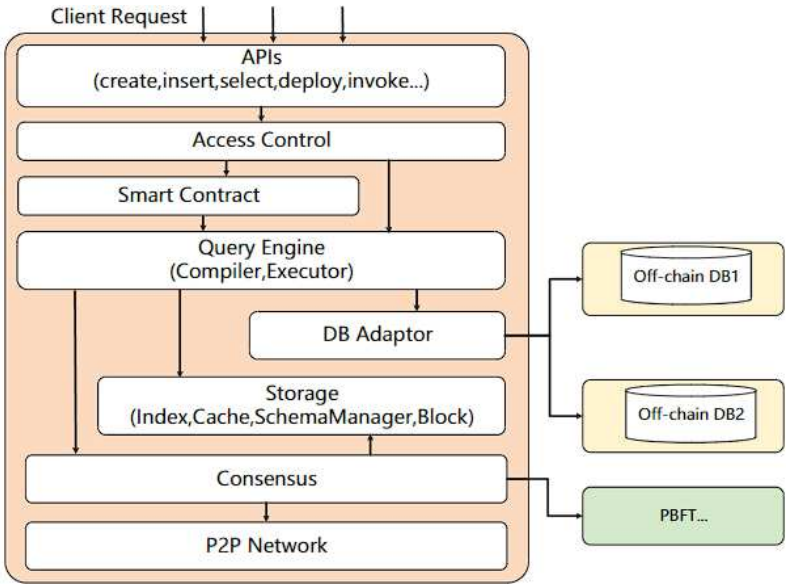

Figure 9 – SEBDB architecture. Ownership of Zhu *et al.* (2020).

SEBDB uses file-based indexing for blockchain data, acknowledging the inefficiency of sequentially scanning individual blocks within a chain that may contain multiple tables. The indexing approach is meant to optimize three key operations: The first uses a Block-level B+-tree Index to facilitate rapid retrieval of a specific block based on its block ID, transaction ID, or timestamp. The second uses a Table-level Bitmap Index to expedite finding tuples belonging to the same transaction type. The third is a Layered Index for efficient retrieval of transactions satisfying certain conditions. Finally, SEBDB allows the execution of CREATE, INSERT, and SELECT commands to create a table, send a new transaction, and get query results respectively. Additionally, it proposes the TRACE command to track provenance within its network.

There are several differences between SEBDB and Inter-MOON. The most important one is the distinct goal of each approach: SEBDB presents a SQL interface for blockchain operations, and Inter-MOON aims to execute SQL operations on blockchain data. Inter-MOON adopts a middleware approach to support existing blockchain and relational solutions, while

[4] https://kafka.apache.org/. Accessed: June 25, 2024.

SEBDB is, itself, a blockchain database — it proposes a new blockchain that is integrated alongside a relational database, using Kafka and Tendermint[5] for consensus and a SQL-like language for querying.

In both approaches, each table represents an entity. However, in SEBDB rows represent blockchain transactions, while in Inter-MOON they represent assets. One similarity is that they both allow join of blockchain and relational data. However, SEBDB does not support operations such as SQL UPDATE or DELETE to blockchain entities, unlike Inter-MOON. SEBDB allows the creation of the schema for a new blockchain entity through the SQL CREATE TABLE command, while Inter-MOON allows this through a schema configuration file (Sec. 4.1.7). However, in Inter-MOON the schema can be modified, while in SEBDB it is immutable. In summary, querying in SEBDB is limited by the constraints of blockchain as a data model, while Inter-MOON offers an alternative approach that bypasses some of the limitations of immutability by defining a mutable Asset meta-object, composed of a series of immutable transactions. For more information, see Sec. 4.2.1.

3.4 Marinho *et al.* (2020)

Marinho *et al.* (2020) introduces MOON. It is a middleware approach for enabling the execution of SQL queries to both blockchain and relational entities. The proposed architecture of MOON is depicted in Fig. 10 and consists of:

MOON Client: Stores the blockchain and database credentials.

Communication: Receives queries from the MOON Client and forwards them to the Scheduler.

Scheduler: Manages an ordered list of incoming transactions and forwards them to the SQL Client or Blockchain Client.

SQL Analyzer: Uses pattern matching to extract information from received SQL, such as the operation type and list of involved entities.

Mapper: Maps blockchain data to and from tuples, to be manipulated via the Temp Data Manager. The Schema Manager is also used in this task.

Index Manager: Retrieves and stores blockchain index entries.

Temp Data Manager: Manages the creation and population of temporary tables used to house blockchain data.

Schema Manager: Manages the blockchain entity schema information.

[5] https://tendermint.com/. Accessed: June 25, 2024.

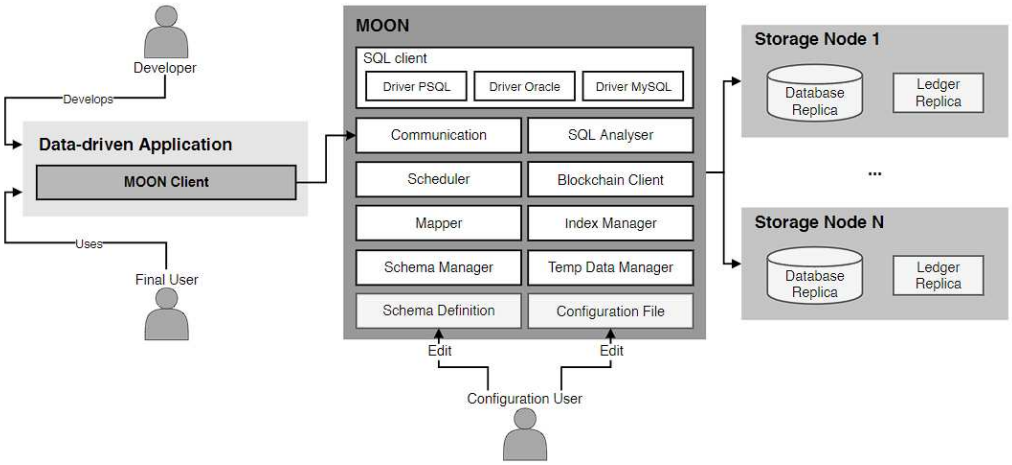

Figure 10 – MOON architecture. Ownership of Marinho *et al.* (2020).

At its core, MOON is similar to the approach proposed in this work, denominated Inter-MOON. Both works take a middleware approach and propose a similar core architecture. Both works also use temporary tables to store blockchain data. However, MOON shows limitations regarding interoperability, which are tackled in this work with a new proposal.

First, there is a limitation in query syntax. MOON lacks support for DELETE and multi-valued INSERT operations, as well as only offering limited capabilities for subqueries, aggregations, and sorting operations. In SELECT queries, for the sake of optimization, MOON will only create temporary tables if the received SELECT contains a JOIN clause including a blockchain and relational entity. Otherwise, the operation is processed with data in memory and other keywords are ignored. Consequently, queries such as `SELECT * FROM` X `ORDER BY` X`.date`, where X is a blockchain entity with a *date* attribute, will ignore the ORDER BY clause, as the data is not virtualized. Other examples include keywords such as LIMIT and OFFSET, GROUP BY, and HAVING, none of which need a JOIN clause to be present. In contrast, Inter-MOON supports all of the previously mentioned operations, including providing comprehensive support for subqueries, joins, and aggregations in SELECT operations, by always virtualizing blockchain data.

Second, regarding blockchain asset retrieval. Both MOON and Inter-MOON retrieve assets from the blockchain to insert into virtual tables. However, MOON does not optimize this retrieval, opting instead for a naive sequential search that retrieves all assets of a given entity one by one. Consequently, every request involving blockchain entities will retrieve potentially thousands of blockchain assets, even if the query predicate limits the search to a select few assets. Inter-MOON optimizes this by analyzing the query predicates to look for conditionals regarding the defined Primary Key (PK) of a blockchain entity, and only fetching the assets referenced in

the conditional if so (See Sec. 4.2.2). Otherwise, all assets are retrieved to complete the query. Additionally, rather than a sequential search, Inter-MOON retrieves them in bulk.

There is a limitation concerning the schema of blockchain entities. MOON does not consider the scenario of an evolving schema, and the impact of such an event in querying and the construction of temporary tables. The approach proposed in this work (Sec. 4.2.1) allows for the application of a mutable schema for blockchain entities, and accounts for missing attributes of old data.

The proposed architecture itself is also different. First, Inter-MOON has no equivalent to the MOON Client module, and the information stored within (blockchain and database credentials) is instead kept in a configuration file inside the middleware. Applications are expected to simply send queries to the middleware via Hyper Text Transfer Protocol Secure (HTTPS), and the middleware will take care of the proper authentication of an application. There is also no equivalent to the Scheduler module either — The blockchain and RDB contain their concurrency managers, and there is no need to add a third ordering service on top. Queries are executed as they are received, and Inter-MOON creates different tasks to asynchronously execute each request from a client. Additionally, connection pools are now used to preserve resources. New in the architecture is also a logging mechanism to aid in recovery in case of failure, the Logger (Sec. 4.1.6). To conclude, this work extends support to blockchains with smart contract functionality and includes a specification to aid in the development of smart contract functions to query blockchain data (Sec. 4.2.2.2) within the constraints of Inter-MOON.

Finally, regarding the indexing of blockchain entities. MOON proposes indexing blockchain assets using a simplified version of the table-based indexing policy proposed in this work (Sec. 4.2.2.1). Unlike Inter-MOON, index retrievals in MOON are not optimized. Given the indexing functions proposed in this work (See Sec. 5), it is the equivalent of MOON always using the worst-case scenario considered by this work. This work also introduces a smart contract-based indexing policy with optimized querying functions based on composite-key blockchain asset keys (Sec. 4.2.2.2), extending the scope of supported blockchains and taking advantage of more modern blockchain capabilities.

3.5 Schuhknecht *et al.* (2021)

Schuhknecht *et al.* (2021) proposes to “chainify” existing DBMSs by installing a lightweight permissioned blockchain layer on top, coined chainifyDB. To deal with potentially

contrasting black-box DBMSs that users may have in operation, chainifyDB introduces Whatever-Voting (WV), a new processing model centered around blockchain technology that can be integrated into existing heterogeneous database setups. The architecture of chainifyDB is depicted in Fig. 11. A pluggable ordering service connects each node. Every node is composed of the DBMS, the local ledger and 3 main components that make up the chainifyDB layer:

Chainify Server: This component handles the initial reception of SQL transaction proposals from clients. It is responsible for early abort checks and authenticating the client based on predefined policies.

Execution Server: This server takes the ordered transactions and executes them in parallel, generating a digest in the form of a *LedgerBlock*.

Commit Server: This component performs the agreement round for the *LedgerBlocks* according to the agreement policy, ensuring consistency across the distributed ledgers of all participating organizations.

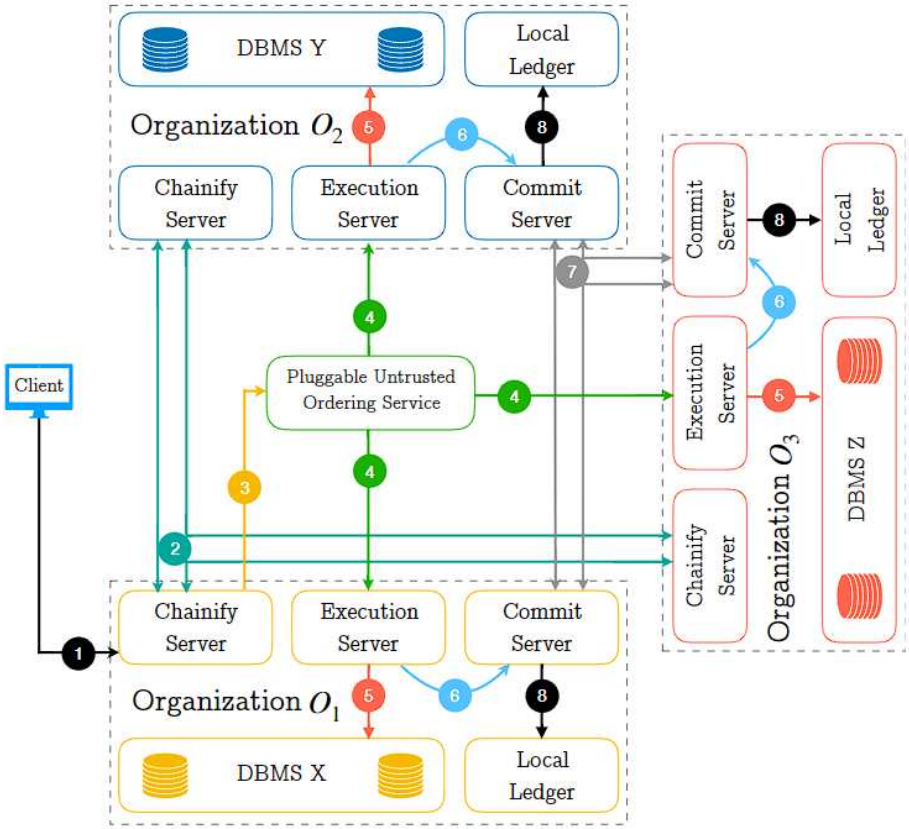

Figure 11 – Architecture of chainifyDB. Ownership of Schuhknecht *et al.* (2021).

ChainifyDB treats SQL transactions as proposals that need to be agreed upon by all participating organizations. A client sends a transaction proposal to the Chainify Server, which includes the original SQL transaction string and metadata. The server performs an early

abort check by authenticating the client and verifying the proposal against predefined policies. After the proposal passes the early abort checks, it is forwarded to other organizations' Chainify Servers for agreement. Once all organizations agree, the proposal is sent to the ordering service that queues the transactions. The ordered transactions are then grouped into transaction blocks, which are sent to the execution server.

The Execution Server constructs a dependency graph to identify and handle conflicts between transactions. It performs a semantic analysis of each SQL transaction to determine the read and write sets, allowing for conflict-free parallel execution. Transactions are executed in stages, where each stage contains mini-batches of transactions that can be executed in parallel without conflicts. The Execution Server generates a *LedgerBlock* as a digest of the executed transactions and forwards it to the Commit Server. Commit Servers across organizations perform an agreement round to finalize the *LedgerBlock*. Upon agreement, the *LedgerBlock* is appended to the local ledgers of all organizations, emulating an immutable, replicated blockchain ledger. This approach denominated the Whatever-Voting model, allows chainifyDB to categorically handle both DML and DDL transactions.

Like some of the other works described in this section, Schuhknecht *et al.* (2021) proposes a blockchain layer on top of existing Database Management System (DBMS). This layer, denominated chainifyDB, executes SQL queries as blockchain operations. Inter-MOON virtualizes blockchain entities, enabling SQL queries to be run directly on blockchain data by treating blockchain entities as relational tables. Also, chainifyDB does not specifically mention indexing policies, although one can hypothesize from the described approach that DBMS indexes may be created via SQL much like other operations, and will be independently created within each DBMS in the network upon agreement. However, because each node may have an unknown black box DBMS, all of which may be different from one another, there is an unaddressed limitation regarding accepted SQL syntax. Inter-MOON proposes both table-based and smart contract-based indexing to improve query performance, and the accepted SQL keywords are described at length. Finally, the blockchain ledger present in each node is meant for audit and record-keeping. Therefore, it will not be queried, simply written to. In Inter-MOON, the ledger is queried to prevent desynchronization.

3.6 Han *et al.* (2023)

Han *et al.* (2023) proposes a system that addresses the challenge of efficient data retrieval in Ethereum, a permissionless blockchain platform, by adding a relational database within the blockchain node, thus allowing SQL SELECT queries to fetch Ethereum smart contract data.

Fig. 12 shows the architecture of the proposal. The Service Interface Layer forwards transactions to the Transaction Layer, which classifies them as either smart contract transactions or regular transactions. Smart contract transactions are processed and validated by the Ethereum Virtual Machine (EVM) through the Smart Contract Manager. Regular transactions are verified by the Transaction Manager by checking the sender's balance. Valid transactions of either kind are stored in the *mempool* as pending — the *mempool* is an Ethereum node's mechanism for storing information on unconfirmed transactions.

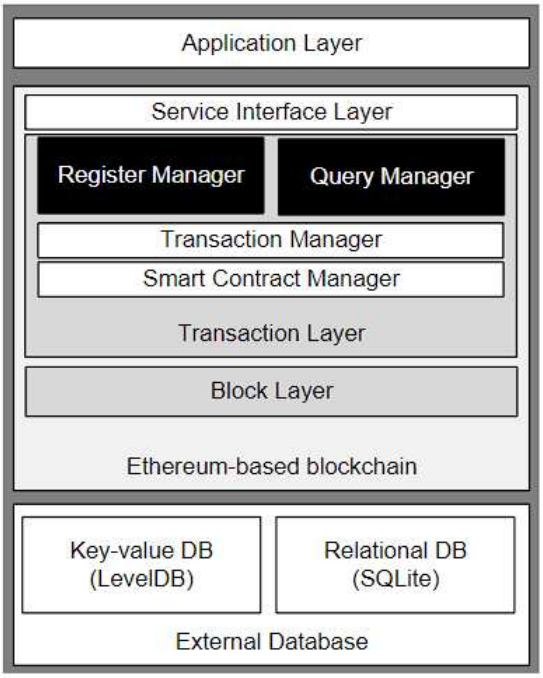

Figure 12 – Architecture of the system proposed by Han *et al.* (2023). Ownership of Han *et al.* (2023).

The Register Manager registers smart contracts and wallet addresses, allowing them to be tracked and stored in the embedded RDB. This process involves activating APIs such as *registerContractAddress* and *registerWalletAddress*, which store the addresses in SQLite. Once registered, the results of transactions associated with these smart contracts and wallet addresses are stored in the RDB (SQLite) by the Block Layer. During block generation, the Block Layer

aggregates pending transactions from the *mempool*, performing a two-level check to determine whether transactions are part of the registered smart contracts or wallet addresses. Transactions not associated with registered entities are stored solely in LevelDB. Undesired registrations can also be removed via APIs like *removeContractAddress* and *removeWalletAddress*. When a registered entity is removed, all associated data is deleted from SQLite.

The Query Manager handles SQL queries on the embedded RDB. It supports APIs like *getData*, which fetch data from both smart contracts and regular transactions stored in SQLite. To ensure data integrity, the Query Manager permits only SELECT queries, filtering out any write queries (INSERT, UPDATE, DELETE). The proposed system uses *quorum*[6] to enable the retrieval of range and conditional data in both smart contracts and regular transactions. See Fig. 13. When a user initiates the registration of a smart contract address via the *registerContractAddress* API, the register manager creates a corresponding table in SQLite to store transaction outcomes and records the address in the database. For regular transactions, the wallet address is stored in a predefined table in SQLite without creating a new table.

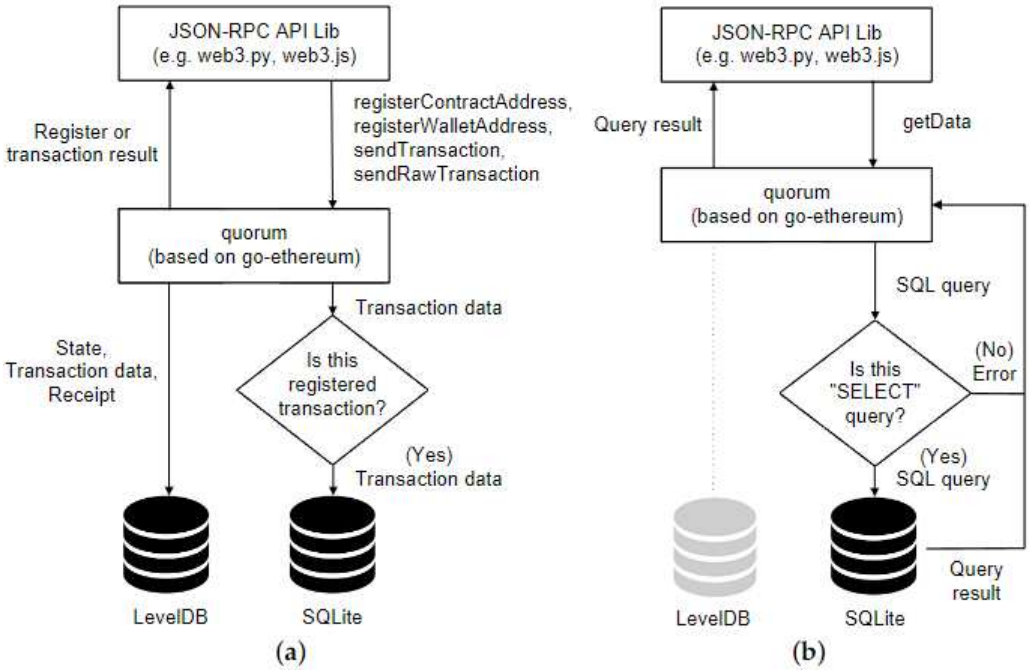

Figure 13 – Overview of the process for registration (a) and querying (b). Ownership of Han *et al.* (2023).

Upon processing transactions initiated by *sendTransaction* and *sendRawTransaction*, the block layer checks if each transaction should be stored in SQLite for range queries, based on their registration status. Transactions that need to be tracked are stored in both SQLite

[6] https://github.com/Consensys/quorum. Accessed: June 25, 2024.

and LevelDB, while others are stored only in LevelDB. For smart contract transactions, data processed through the EVM is inserted into SQLite, while for regular transactions, details are stored using the source and destination addresses and the amount. For retrieving regular transactions over a specific period or between specific users, the user can initiate queries with the relevant wallet addresses and conditions. The query manager checks the transaction type and searches the regular transaction table in SQLite to respond to the queries.

The system proposed by Han *et al.* (2023) contains certain limitations, some of which are a result of design choices. First, it only supports read queries (SQL SELECT) to registered smart contracts or wallet addresses. Secondly, it only supports Ethereum. Thirdly, registration is done via API, meaning there are two interfaces for interaction, API and SQL. These limitations mean that the system is only applicable in cryptocurrency scenarios using Ethereum, unlike Inter-MOON, which aims to be generally applicable in permissible blockchain scenarios. Therefore, there is no concept of schema or entity — Transaction data is replicated into relational tables for querying, but there is no differentiation beyond either a registration or transfer of crypto assets. Inter-MOON differentiates assets by defining a relational schema to blockchain entities, allowing the execution of complex SQL predicates based on asset properties. In the proposed system, query predicates are either based on the timestamp, addresses, or transfer amount.

3.7 Wang *et al.* (2023)

Wang *et al.* (2023) proposes aChain, a blockchain framework that integrates SQL-based Online Analytical Processing (OLAP) capabilities directly into its architecture. The system is designed to handle complex SQL queries, including SELECT, INSERT, UPDATE, and DELETE operations, on blockchain data.

The architecture of aChain, depicted in Fig. 14, includes a transaction processing mechanism that supports SQL commands. When a SQL query is issued, it is parsed and validated before being simulated on a local database instance to generate a transaction proposal. This proposal includes a *readset*, which records the data read during the transaction, and a *writeset*, which captures the data to be written. The transaction is then ordered using an Execute-Order-Validate (EOV) consensus mechanism, ensuring that all nodes agree on the transaction sequence before it is executed and validated across the network.

The block structure is adapted to accommodate SQL-based transactions. Each block

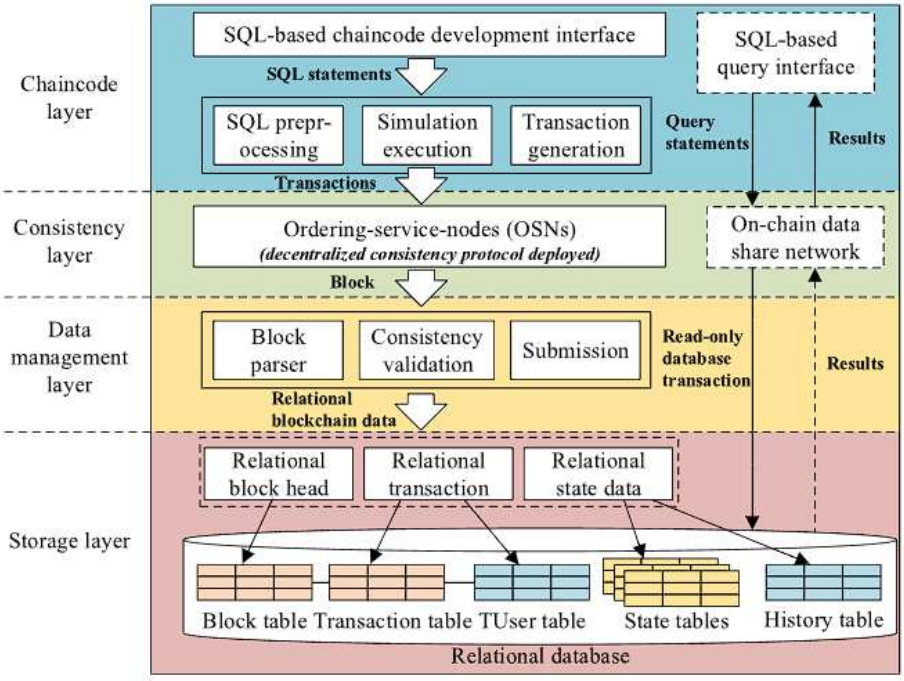

Figure 14 – Architecture of aChain. Ownership of Wang *et al.* (2023).

contains a header with standard blockchain components such as the block hash, previous block hash, Merkle root, timestamp, nonce, and block number. The block body includes a list of transactions, each encapsulating a SQL operation along with metadata. This design allows aChain to maintain the history of all transactions. Blocks and transactions are also stored within tables, for querying.

The framework handles the entity schema by organizing on-chain data into a relational *init* table. SQL CREATE, ALTER, and DROP are used to add, update, and delete entries from this table, respectively. Consequently, this model supports schema evolution, allowing changes to the database schema to be recorded as transactions on the blockchain. Each row in the *init* table represents a schema modification transaction. One limitation is that this *init* table cannot be queried using SELECT, only modified using the above commands.

SELECT commands are converted into a query that retrieves the necessary data from the relational database and generates *readsets*. INSERT statements are transformed into DQL statements that query the table to find any existing rows with attributes matching those specified in the INSERT's VALUES clause. The entire INSERT statement is also recorded in an operational log (Ops). During the simulation phase, the converted DQL statements are executed to obtain query results, which are then used to generate *writesets* and ensure that new data rows are correctly added to the blockchain's state.

An UPDATE statement is handled by converting it into a DQL statement that selects

all attributes from the table specified in the UPDATE's WHERE clause. The SET clause of the UPDATE statement is recorded in Ops. The query results from these DQL statements are used to apply the updates, with the updated attributes being recorded in the *writeset*. This process ensures that all changes are tracked and versioned within the blockchain. DELETE commands are similarly converted into DQL statements that query the table for rows matching the conditions specified in the DELETE's WHERE clause. The operation is recorded in Ops with an empty value (null) to signify the deletion. The results of the DQL queries are used to identify the rows to be deleted, and these deletions are then recorded in the *writeset*. This method ensures that deletions are processed in a way that maintains the integrity and versioning of the blockchain data.

aChain shows similar characteristics to this work. Once again, the biggest difference is in approach — it is a proposal for a new blockchain integrating a RDB, rather than a middleware that avoids directly interfering with the consensus and architecture of existing blockchains like Inter-MOON. This is an important distinction, as aChain will replace an existing blockchain setup rather than work with it like Inter-MOON. Consequently, aChain also uses a custom block and transaction structure. Like Inter-MOON, aChain contains an entity schema. However, unlike Inter-MOON, schema changes are recorded in the blockchain alongside the assets themselves. The approach for UPDATE and DELETE are essentially the same as in this work: add a new version of the asset with updated attributes, and mark the asset as deleted, respectively.

3.8 Comparison Review

Table 3 shows a summarized comparison of the approach proposed in this work and related works. The type of approach is categorized as either "BC-based" (works that propose blockchain features on top of a database), "DB-based" (works that propose database features on top of a blockchain), and "Middleware" (works that propose a middleware allowing interaction between existing blockchains and databases). Each work is also categorized based on which SQL DQL and DML commands are supported, namely SELECT for DQL and INSERT, UPDATE, and DELETE for DML. Partial support means the syntax is limited in a significant way. Data partitioning and schema manipulation are also considered, and whether each work allows SQL JOIN operations with on-chain and off-chain data via the proposed interface.

Nathan *et al.* (2019) adds a blockchain layer on top of PostgreSQL, meaning there is no data partitioning nor replication — all data becomes blockchain-relational data. Zhu *et*

Work	Approach	DQL	DML	Schema manipulation	Data partitioning	JOIN on & off-chain data
Nathan *et al.* (2019)	BC-based	Yes	Yes	Yes	No	No
Zhu *et al.* (2020)	DB-based	Yes	Partial	Partial	No	Yes
Marinho *et al.* (2020)	Middleware	Partial	Partial	Partial	Yes	Yes
Schuhknecht *et al.* (2021)	BC-based	Yes	Yes	Yes	No	No
Han *et al.* (2023)	BC-based	Partial	No	No	No	No
Wang *et al.* (2023)	DB-based	Yes	Yes	Yes	No	No
Inter-MOON (This work)	Middleware	Yes	Yes	Yes	Yes	Yes

Table 3 – Comparison table of related works.

al. (2020), which introduces SEBDB, only partially supports DML. It allows INSERT, but not UPDATE or DELETE. Additionally, while it explicitly supports joining with off-chain data, there is no partitioning within the proposed API. There is also no way to modify an existing schema (ie. No support for the ALTER command or any alternatives). Schuhknecht *et al.* (2021) proposes chainifyDB, which allows for DQL, DML, and DDL. However, there is no data partitioning or off-chain join: relational data is replicated into a blockchain ledger for audit. In aChain, proposed by Wang *et al.* (2023), there is no partitioning or off-chain join either. The underlying relational store is transformed into a blockchain.

Marinho *et al.* (2020) and Han *et al.* (2023) offer only partial support for DQL queries, as discussed in sections 3.4 and 3.6 respectively. Marinho *et al.* (2020) also offers no support for DELETE, and while new entities can be added to the schema, they cannot be modified. For example, adding a column to an entity after the schema definition is impossible. Han *et al.* (2023) neither approaches the schema concept nor supports any blockchain other than Ethereum. An API is used to register data, not SQL, so an SQL-like DML query language is unavailable.

Inter-MOON finds a niche in the scenario of an application wishing to partition data into an existing blockchain solution as well as a relational database. Other referenced works either propose a new blockchain-based or database-based solution, in which all data has to be fully moved, rather than partitioned. In the case of most BC-based solutions, they wholeheartedly change the basic functionality of a database. Consequently, existing relational data becomes blockchain data, without partitioning, only full replication. Additionally, they do not consider interoperability with off-chain data.

4 METHODOLOGY

This chapter details the Inter-MOON proposal, including an overview of the proposed architecture (Sec. 4.1) and a detailed description of the approach.

Blockchains have no standard query language, and directly mapping SQL queries to equivalent API calls would depend highly on the quality and grammatical completeness of the underlying querying API of each blockchain. Consequently, the approach proposed in this work called Inter-MOON, was conceptualized as a middleware that enables the execution of SQL DQL and DML queries to blockchain and relational entities simultaneously by mapping blockchain entities to the relational data model, inside of tables. Once mapped, blockchain entities will exist in the relational space alongside other relational entities and may be queried via SQL. This mapping is supported by an indexing mechanism (Sec. 4.2.2) which allows the middleware to track assets. To implement this mechanism, two policies are proposed: a table-based policy (Sec. 4.2.2.1) and a smart contract-based policy (Sec. 4.2.2.2). While the smart contract-based policy is the most optimal, not every blockchain offers smart contract functionality. In such scenarios, the table-based policy is still applicable.

Read operations, such as SELECT, can be organically executed once this mapping technique, called virtualization (Sec. 4.2), is finished. Write operations, however, warrant a more intricate approach, since there is also the need to persist the data into the blockchain (Sec. 4.3). INSERT is executed by skipping virtualization, mapping the given SQL to a blockchain *Put* operation and activating the indexing mechanism (Sec. 4.3.2). A versioning strategy (Sec. 4.2.1) is utilized to support the execution of UPDATE (Sec. 4.3.3). Updates are simulated via virtualization and mapped to a blockchain *Put* using the newer version of each asset. The indexing mechanism is then activated to update the indexes. DELETE is executed similarly to UPDATE. A soft-delete mechanism is applied to prevent the retrieval of deleted data in exchange for maintaining immutability (Sec. 4.3.4). This approach allows for any blockchain implementation that fulfills some basic requirements to be used (See Sec. 4.1.9 for more details).

4.1 Middleware Architecture

The Inter-MOON architecture (Fig. 15) is composed of three major components: the Inter-MOON middleware, the RDB, and the BC. Any RDB or BC that fulfills the requirements specified in Sec. 4.1.9 can be supported.

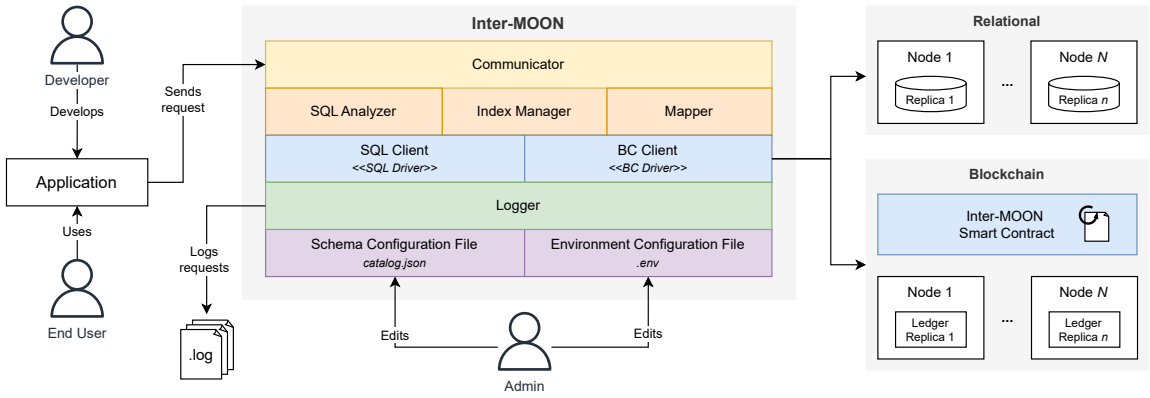

Figure 15 – Overview of the Inter-MOON architecture.

In the proposed architecture, the Communicator demonstrates interoperability at the application level, allowing Inter-MOON to receive and answer requests from clients. For the middleware level, the SQL and Blockchain Clients allow the interaction with data stores. The Mapper, Schema, SQL Analyzer, and Index Manager modules work in tandem to extract information, join data, and map SQL requests to the blockchain data model. Three different users are considered: The Developer, who develops an application that sends requests to Inter-MOON over a network; The End User, who uses said application; And the Admin, who has internal access to the middleware. Note that the Developer and Admin belong to the same organization. The middleware is not meant to be a centralized platform, but instead a middleware placed between an individual application and its data stores.

The middleware is further divided into several modules with separate functions, explained further below.

4.1.1 Communicator

The Communicator is the front-facing service that may receive Hypertext Transfer Protocol (HTTP) requests sent to Inter-MOON. Once received, they are queued and forwarded to a worker pool to be executed. The middleware asynchronously executes every received request.

4.1.2 SQL Analyzer and Mapper

The SQL Analyzer extracts information, such as the operation type, entities, and predicates from requests using a *tokenizer* mechanism. Each SQL statement is divided into groups of tokens, wherein each token is classified based on its purpose and role within the syntax of standard SQL. The token types include Keyword, Identifier, Function, Clause, Operator,

and Wildcard. If any subqueries are found, the mechanism will use recursion to analyze them separately. This mechanism ensures that Inter-MOON can extract the information it needs in a semantically aware manner, even when subqueries or aggregation are present.

Consider the SQL example depicted in Fig. 16. We can extract the operation type by observing the first keyword in the query, and the list of involved entities by fetching any identifiers after a FROM or JOIN keyword that is not a subquery. If a subquery is found, recursion is employed to analyze the subquery in a similar fashion.

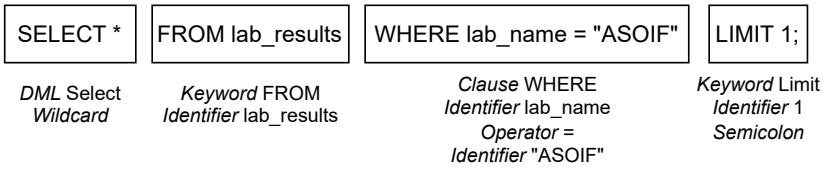

Figure 16 – Inter-MOON SQL Analyzer tokenizer example.

Conversely, the Mapper component will hold functions that can map SQL data to and from blockchain-readable formats using data transformations. Relational tuples are converted into common data structures such as dictionaries, lists or arrays. See Fig. 17 for an example.

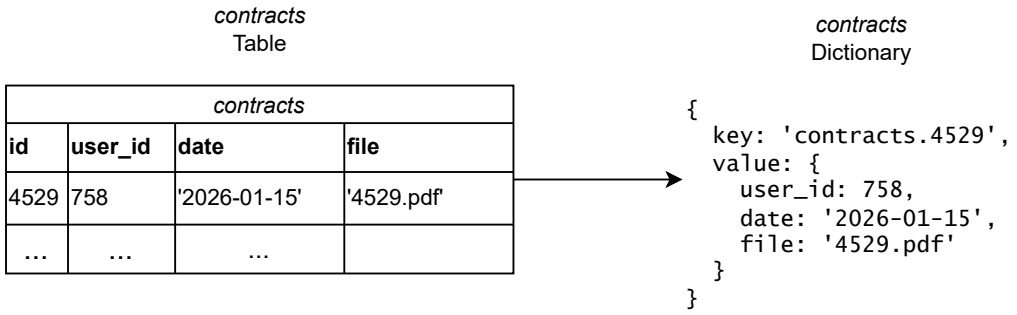

Figure 17 – Example of a mapping operation converting a SQL tuple into a dictionary.

4.1.3 Index Manager

The Index Manager is used by the table-based indexing policy to store and track blockchain asset keys for querying. See Section 4.2 for more information regarding this strategy.

4.1.4 SQL and Blockchain Clients

The SQL and blockchain clients simply represent services meant to communicate with the relational and blockchain stores, respectively. Each client has a generic driver that implements adapters, allowing for generalized support as long as adapters are developed (Sec. 4.1.9). Smart contract support is possible using the same strategy, by having the driver invoke smart contract functions via API. The granularity level of each store is left untouched: For the relational store, it is of tuple level, for blockchain it is, usually, database or collection level, depending on implementation. Connection pools are used to save resources.

4.1.5 Inter-MOON Smart Contract

The Inter-MOON smart contract represents a smart contract developed for usage within the Inter-MOON environment and allows optimal querying of data via function invocations when used alongside the smart contract indexing policy. See Sec. 4.2.2 for details regarding indexing and Sec. 4.2.2.2 for the smart contract indexing policy.

4.1.6 Logger

The Logger is another component used by the table-based indexing policy. It uses the relational database's own continuous archiving and recovery features to create regular logs of lookup table data after the execution of write operations. Consequently, in the event of failure in the relational database, data loss is minimized, and restoration to any desired point in time is facilitated, empowering data integrity. This design avoids introducing additional logging overhead while harnessing the data protection and recovery features inherent to the underlying relational database system.

4.1.7 Schema and Environmental Configuration Files

The schema is a configuration file that defines the relational schema applied to blockchain entities by Inter-MOON during virtualization (Sec. 4.2.1). The environment configuration is a simple file where information like ports, database access, and other environment variables used by Inter-MOON can be configured.

4.1.8 Technical Details

This architecture aims not to interfere with consensus or the inner workings of the blockchain component and to maintain integrity and compatibility. Inter-MOON preserves the autonomy of each Database (DB). The middleware will mainly receive, process, and send or reroute requests to each data store as necessary to fulfill queries.

While the blockchain-relational mapping processes result in some CPU-bound operations, the middleware is still mainly I/O bound, and the bulk of the CPU work will be handled by the blockchain network itself or the relational database. Consequently, performance, scalability, and concurrency are ultimately dependent on the data stores' capabilities.

4.1.9 Supported Databases

Inter-MOON can support any relational database through adapters that implement a common interface with exposed functions for establishing connections, creating and using cursors, and fetching rows of data. A generic database driver object is used instead of any specific driver, and different databases can be supported by switching the internal adapter accessed by this generic driver (Fig. 18).

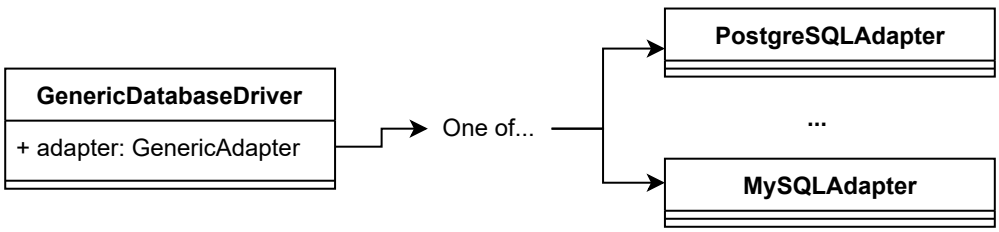

Figure 18 – Simple rendition of the generic database driver.

Listing 1 illustrates a basic pseudo-code implementation of this approach. Connection settings can be obtained from the configuration file, as explained in Section 4.1.7. This structure promotes maintainability, decoupling, and database support.

Código-fonte 1 – The Generic Database Driver basic structure. It holds an adapter object which represents the driver of a database engine.

```
1 import psycopg2 # PostgreSQL
2
```

```
class GenericDatabaseDriver:
    def __init__(self, adapter):
        self.adapter = adapter

    def connect(self, *args, **kwargs):
        return self.adapter.connect(*args, **kwargs)

driver = GenericDatabaseDriver(psycopg2)
with driver.connect("config.cfg") as conn:
    conn.execute("SELECT * FROM users;")
```

As for blockchain, a similar approach can be used, with a Generic Blockchain Driver and corresponding adapters to connect to the blockchain and execute API commands. Consequently, the blockchain must offer some API for reading and sending data to the network. At a minimum, functions equivalent to $Get(key)$, $GetList(keys)$, and $Put(key, value)$ are required to, respectively, fetch an asset, fetch a list of assets, and store an asset.

Provided the outlined requirements are met and an appropriate indexing policy (table-based or smart contract-based) is applied, blockchains with and without smart contract functionality are supported by Inter-MOON. Considering not every blockchain implements the same features, each policy has particular characteristics, detailed in the dedicated sections within this work (sections 4.2.2.1 and 4.2.2.2 respectively for each policy). With the previous considerations in mind, despite the lack of a unified data model for blockchains being a current research issue (MEYER; MELLO, 2022; YUAN; WANG, 2018), some degree of general support is still feasible.

4.2 Virtualization of Blockchain entities

The core of the Inter-MOON approach is the virtualization of blockchain entities in the relational model. Essentially, a modified relational schema is defined and applied to blockchain entities (Sec. 4.2.1). Entities are tracked by an indexing mechanism, of which two different policies are proposed for interoperability: one based on lookup tables, and another based on smart contracts (Sec. 4.2.2). When Inter-MOON receives a SQL query, the blockchain entities needed to complete the query are fetched from the blockchain and inserted into temporary

tables inside the relational database. Finally, the query can be executed (Sec. 4.3). Consequently, indexing is necessary to allow the middleware to track, read, and write entities to the blockchain network. Without indexing, the middleware will have no way of knowing which entities need to be retrieved to execute the virtualization. Conversely, it also means that Inter-MOON will only track indexed entities (See Sec. 6.1 for details regarding this limitation).

4.2.1 Blockchain-relational Mapping

First, it is necessary to describe how Inter-MOON perceives blockchain entities. Blockchain entities are considered generic business objects. Each instance of an entity, denominated asset, is a key-value tuple $< key, value >$. *key* is a composite key string following the format $entityName.id$. $entityName$ is the entity identifier — in the relational model, this would be the table name. *id* will uniquely identify the asset within the $entityName$ namespace, and together $entityName.id$ will uniquely identify the asset in the world state. *value* is the asset data in bytecode.

Consider the asset *key* equivalent to the relational PK. Both will uniquely identify a piece of data. Note that this *key* differs from the cryptographic hash generated via consensus and assigned to the transaction. The transaction ID identifies the transaction itself, and the asset *key* identifies the asset, which is stored inside of a transaction.

To account for immutability, assets are considered historical records connected by *key*. Every version of an asset will share the same *key* but exist in different transactions. The latest version is tracked by Inter-MOON by one of the proposed indexing policies (Sec. 4.2.2). This notion enables Inter-MOON to execute UPDATE (Sec. 4.3.3) and DELETE (Sec. 4.3.4) operations to BC entities. UPDATE creates a new version of the asset, and DELETE marks the asset as deleted — consider it a soft delete. Earlier versions will still exist and may be accessed externally using tools provided by the blockchain itself. This approach helps align the contrasting structure of blockchain architecture, in which assets are immutable, to historical records where each record by itself is immutable but the historical series as a whole represents a mutable meta asset.

See Fig. 19 for an example of this strategy. Suppose there is an asset of composite key $contracts$.758, whose property *value* is updated. A second version of the asset will be generated, say asset v2, connected to the original asset, now denominated asset v1, via key. Each version of the asset is contained inside different transactions, but the common key historically

connects them.

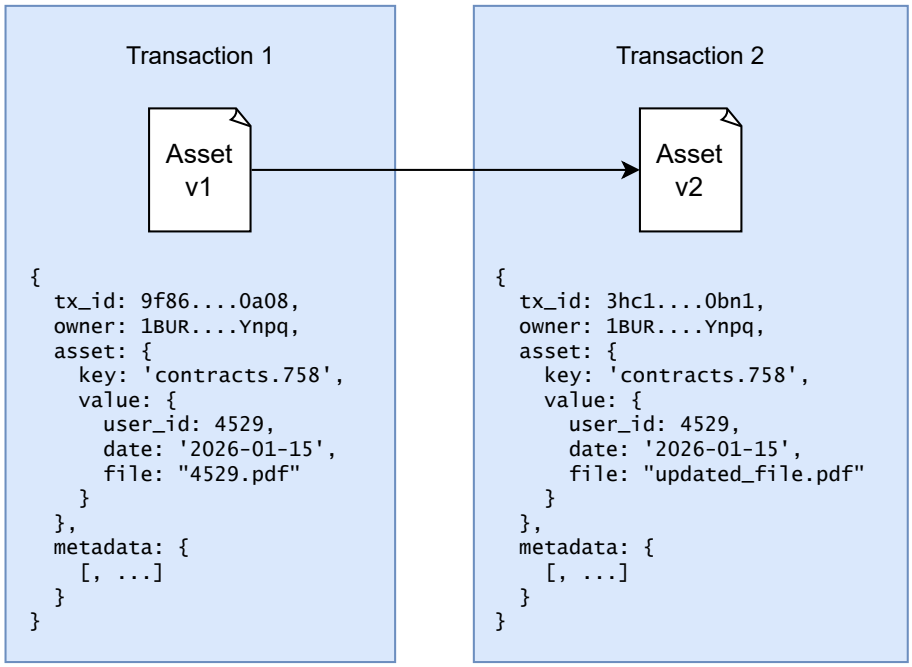

Figure 19 – Example of the asset versioning strategy. Asset v1 and v2 are connected via *key*, but have different transaction IDs.

Given this definition, a relational schema is applied to instruct the RDB on how each blockchain entity's relational mapping shall be executed. Fig. 20 illustrates an example for a supposed *Contracts* entity. The *entity* property defines the *entityName* identifier. *primary_key* defines the name of the attribute whose value will be used for the blockchain *id*. Together, they form the blockchain composite *key* described earlier (*entityName.id*). Additionally, other attributes, as well as any foreign keys, will also be defined here. Attributes other than the primary key become properties of the *value* component of the key-value structure.

During virtualization, the mapper (Sec. 4.1.2) checks the schema when building tables, and creates each attribute column as defined. For the sake of efficiency in table creation, PK and Foreign Key (FK) constraints will not be created, only the columns themselves. Temporary tables are used to avoid potential concurrency or synchronization issues arising from the mixed usage of read and write operations in queries. If blockchain data were to simply be replicated in the RDB, it would be liable to third-party tampering or modification, given the middleware approach. However, performance is undeniably a concern if tables are constantly being created and populated with every request. The viability of this approach is measured in Sec. 5.2, and Fig. 21 below shows an overview of this technique.

<*Contracts*>
Entity Schema
catalog.json

```
{
 entity: 'contracts',
 attributes: [
  {name: 'id', type: 'integer'},
  {name: 'user_id', type: 'integer'},
  {name: 'date', type: 'datetime'},
  {name: 'file', type: 'string'},
 ],
 primary_key: ['id'],
 foreign_key: [{
  name: 'user_id',
  ref: {name: 'users', attr: 'id'}
 }],
 source: 'blockchain'
}
```

Figure 20 – Blockchain relational schema example for a *Contract* entity.

The schema is mutable. Over time, attributes can be added or removed. To preserve blockchain immutability, existing assets of that entity will not be affected by changes, but newly added ones will use the updated schema. If desired, older assets can be updated with an UPDATE operation (Sec. 4.3.3) to use the latest version of the schema. Consequently, attributes are considered NULLABLE by default, as there is a need to account for missing attributes from older assets.

For example, suppose a schema change introduces a new *address* string attribute to the *Contracts* schema. Older *contracts*, created before this addition, will use a NULL value instead for the *address* column when virtualized.

Considering the proposed architecture (Fig. 15), this schema must be defined by the *Admin* user before virtualization can occur, as Inter-MOON will only track indexed entities, and indexing can only occur if the schema is defined. Consider this schema the cornerstone of the virtualization approach presented in this work.

4.2.2 Blockchain Indexing and Data Fetching

As mentioned, virtualization is supported by a key concept, the indexing mechanism. It enables the Inter-MOON middleware to track blockchain assets by indexing asset keys using a policy. The mechanism is activated after a successful INSERT operation on a blockchain entity. Assets are indexed using composite keys matching the *entityName.id* definition proposed earlier (Sec. 4.2.1) for blockchain asset key-value tuples. During virtualization, it is used so

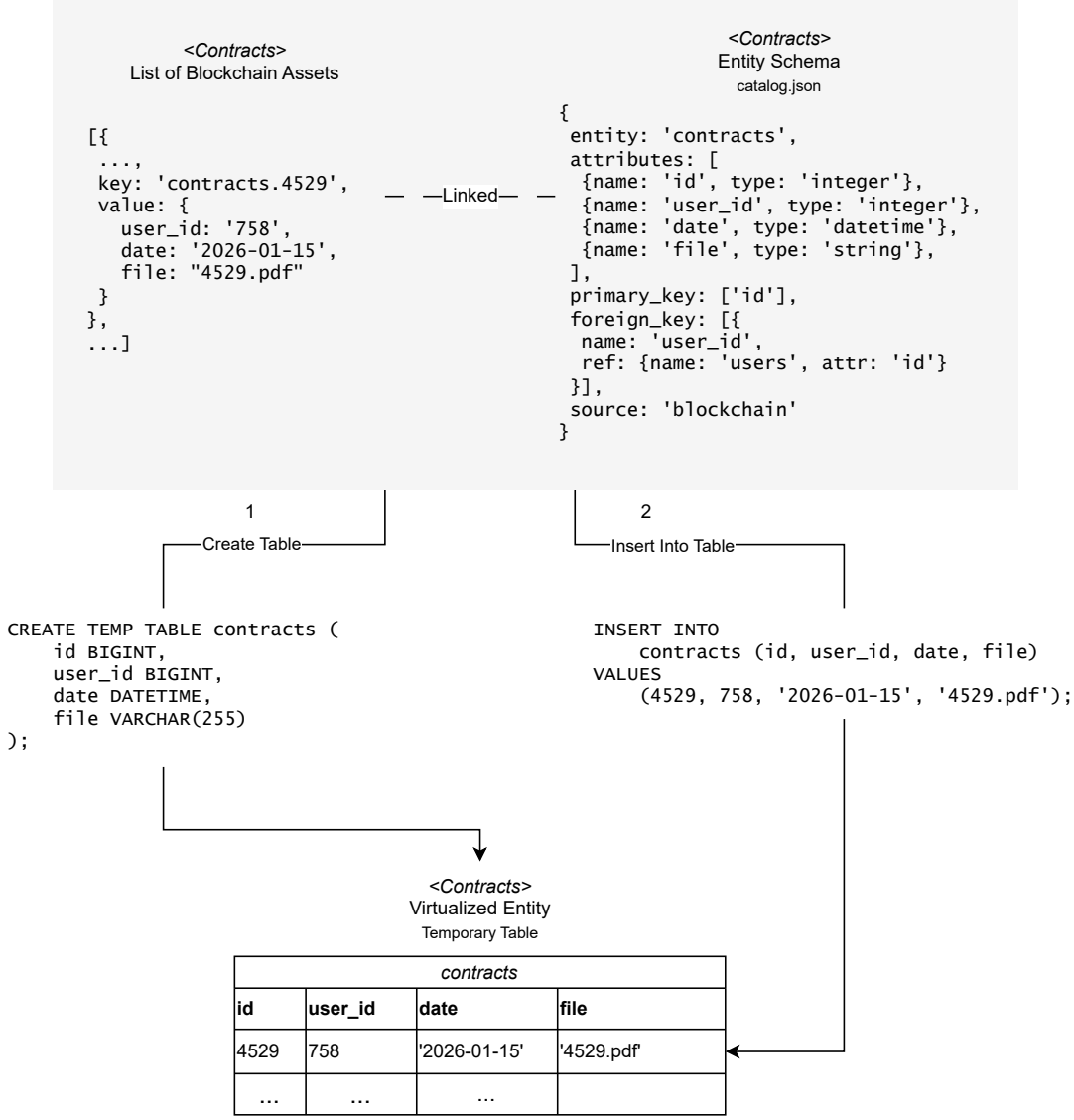

Figure 21 – Inter-MOON proposal for the virtualization of a *Contracts* blockchain entity.

that the middleware can know which assets need to be retrieved from the blockchain. Naturally, a limitation of this mechanism is that blockchain entities will not be tracked by Inter-MOON unless the key is also indexed using Inter-MOON.

Regardless of policy, the indexing mechanism comprises write and read functions. Table 4 below describes each function. Write (W) functions to create or update indexed asset keys. W_1 creates the index, W_2 updates it, and W_3 marks it as deleted. Read (R) functions optimally retrieve keys based on predicates concerning the blockchain *entityName* and *id*.

The worst-case scenario is represented by R_2, in which all keys of X are returned. It is used for SQL queries that do not contain predicates, or whose predicates do not reference *id*. R_1 optimizes for scenarios where a single key is given. R_3 is for membership operations over a

Function	Description
W_1	Insert key with $entityName = X$ and $id = Y$
W_2	Update key where $entityName = X$ and $id = Y$
W_3	Mark one key as deleted where $entityName = X$ and $id = Y$
R_1	Select one index where $entityName = X$ and $id = Y$
R_2	Select all keys where $entityName = X$
R_3	Select all keys where $entityName = X$ and $id \in \{Y_1, Y_2 ... Y_n\}$
R_4	Select all keys where $entityName = X$ and $Y_i \geq id \geq Y_j$

Table 4 – Functions executed by the indexing mechanism.

set of keys. R_4 optimizes for range comparisons.

To implement this mechanism, two policies are proposed: a table-based policy (Sec. 4.2.2.1) and a smart contract-based policy (Sec. 4.2.2.2). The table-based policy uses the relational database to implement indexing and the smart contract-based policy uses a smart contract instead. The proposed policies will implement each function described in Table 4 within their platform. As is the case for the schema (Sec. 4.2.1), the *Admin* user must preemptively choose a policy and either create the relational table or write the smart contract to enable virtualization.

4.2.2.1 Table-based Indexing

The table-based indexing policy tracks blockchain entities by storing every blockchain asset key in a lookup table in the relational database. The lookup table stores the *entityName* and *id* components that make up the blockchain asset composite *key*. The transaction ID (*tx_id*) and a version parameter are also stored for record-keeping. The transaction ID denotes which transaction holds the latest version of an asset. An *is_deleted* flag is used to mark deletion. A composite index on the tuple $< entity, id >$ provides optimized lookup times, allowing the table to efficiently index the asset *key*. Table 5 illustrates an example.

entity	id	tx_id	version	is_deleted
'contracts'	1	'a23c...f137'	0	FALSE
'contracts'	2	'taa4...vd19'	0	FALSE
'files'	1	'cs3c...fd3c'	1	FALSE
...	...	...	...	...

Table 5 – Example of a lookup table for use with the table-based indexing policy.

As for implementation requirements, this policy assumes that the blockchain API may only expose the functions nominated in Sec. 4.1.9. These are: *Get*, *Put* and *GetList*.

In the proposed architecture, the Index Manager (Sec. 4.1.3) is responsible for

interacting with this table. When indexed, the keys can be retrieved and then used to query the blockchain for the asset data. For example, suppose X is defined within the schema as a blockchain entity with *id* as the primary key. Then, Inter-MOON receives the following query: `SELECT * FROM` X `WHERE` X`.id > 5000`. The Index Manager queries the lookup table for the keys matching the predicate. Then, the resulting collection of keys is used by the blockchain client to fetch the data of the matching assets. Finally, they are virtualized according to the schema.

Naturally, in this policy, the functions described in Table 4 become SQL queries (See Table 6). The read (R) functions query the lookup table to return a collection of matching keys, which are fed into a *GetList* call during data fetching to optimally fetch only the assets matching the predicate. Of note is that R_1 and R_3 do not need to be implemented in this policy. R_1 would be used to retrieve a single key from the lookup table. However, this operation can be easily translated into a simple *Get* blockchain API call using the given *key*. The same is true of R_3: It is essentially a *GetList* call using the provided list of keys. If the blockchain API only allows querying via transaction ID, then *tx_id* can be returned instead of the *key* in all R queries and used as parameters for *Get* and *GetList*.

Function	SQL
W_1	`INSERT INTO blockchain_index (entity, id, tx_id)` `VALUES (`X`, `Y`, `Z`)`
W_2	`UPDATE blockchain_index` `SET version = version + 1, tx_id =` Z `WHERE entity =` X `AND id =` Y

W_3	`UPDATE blockchain_index`
	`SET is_deleted = TRUE`
	`WHERE entity =` X `AND id =` Y

R_2	`SELECT entity, id FROM blockchain_index`
	`WHERE entity =` X
	`AND is_deleted = FALSE`

R_4	`SELECT entity, id FROM blockchain_index`
	`WHERE entity =` X `AND id >=` Y_i `AND id <=` Y_j
	`AND is_deleted = FALSE`

Table 6 – Table-based policy implementation of each indexing function.

If the SQL received by Inter-MOON also contains pagination parameters (ie. LIMIT and OFFSET), they are used to further optimize asset querying by limiting the number of keys

returned by the *R* functions. For example, in the following query: `SELECT * FROM X LIMIT 10`. R_2 would be used, as the SQL contains no predicates on *id*. If hundreds of thousands of keys are indexed, this would be highly inefficient. However, the pagination parameters are inherited by R_2, allowing it to return only the first 10 keys.

4.2.2.2 Smart Contract-based Indexing

The smart contract-based indexing policy uses a smart contract to store and track blockchain entities by directly reading and writing assets into the state database with custom smart contract functions. Given the proposed Inter-MOON architecture (Fig. 15), the blockchain driver interfaces with the smart contract to execute commands, and smart contract functions execute the querying and indexing operations as written.

Usually, smart contracts are modeled after business objects. This means that separate smart contracts would be created for every blockchain entity. However, this policy assumes a generic Inter-MOON smart contract that can interact with any blockchain entity recorded in the Inter-MOON schema (Sec. 4.2.1). This approach helps prevent redundancy — with several smart contracts, any functions in this specification will have to be rewritten in each contract. The approach is based on the Fabric (ANDROULAKI *et al.*, 2018) implementation of smart contracts (See also Sec. 2.2) due to its open-source nature and non-dependency on cryptocurrency, and assumes that:

- The smart contract offers functions that can write ($Put(key, value)$), read ($Get(key)$) and delete ($Del(key)$) state to the underlying state database.
- The smart contract allows the creation and querying of composite keys and the state database has its indexing mechanism for keys.
- *Get* returns the value of a key. The ledger state can be iterated using *Get* range queries.
- *Put* updates an existing state using a key or writes a new state if the key doesn't exist.
- *Del* deletes a state, preventing retrieval. The deleted data still exists inside of the blockchain. The blockchain has its record-keeping mechanism to preserve the history of a key, and the smart contract API has its interface to retrieve historical data.

These assumptions cover the extent of the Inter-MOON approach to blockchain entity life-cycle, as described in Sec. 4.2.1. The scalability of this approach depends on the smart contract implementation and its underlying state database.

Each function defined in Table 4 of Sec. 4.2.2 describing the indexing mechanism

becomes a smart contract function in this policy. The algorithms for each function will be described in the following subsections using pseudo-code.

4.2.2.2.1 W_1 — *PutState*

See Algorithm 1 for $PutState(k, v)$, which implements W_1. *entityName* is the name of the blockchain entity, *id* is the identifier (the attribute defined as the PK in the schema, see Sec. 4.2.1) and *v* is the *value* in bytecode. The function will build the composite key and then call the state database $Put(key, value)$ to write the state to the ledger.

Algoritmo 1: *PutState*

Data: *entityName*, *id*, *v*
Result: asset *A* if successful, nil otherwise.

```
1 k ← entityName + id
2 A ← Put(k, v)
```

4.2.2.2.2 W_2 — *UpdateState*

The smart contract algorithm for W_2 is the same as W_1, since $Put(k, v)$ updates the value of a *key* if the *key* already exists in the state database. The older value associated with the key remains inside the blockchain but is inaccessible within the Inter-MOON interface.

4.2.2.2.3 W_3 — *DeleteState*

See Algorithm 2 for $DeleteState(k)$, which implements W_3. The function will build the composite *key* and then call the state database $Get(k)$ function to query the ledger using the *key*. If the asset exists, it is marked as deleted and the *key* is returned.

Algoritmo 2: *DeleteState*

Data: *entityName*, *id*
Result: *k* of the deleted asset if successful, nil otherwise.

```
1 k ← entityName + id
2 a ← Get(k)
3 if a exists then
4 |   Del(a.key)
5 end
```

4.2.2.2.4 R_1 — *GetStateByKey*

$GetStateByKey(entityName, id)$ implements R_1. See Algorithm 3. It simply builds the composite *key* and then calls the state database $Get(k)$ function to retrieve the state from the ledger.

Algoritmo 3: *GetStateByKey*

Data: *entityName*, *id*
Result: $< key, value >$ tuple.
1 $k \leftarrow entityName + id$
2 $A \leftarrow Get(k)$

4.2.2.2.5 R_2 — *GetStateByEntityName*

The $GetStateByEntityName(entityName, N, k_b)$ function implements R_2. See Algorithm 4 below. N is maximum number of items, and k_b is a bookmark. These parameters are used for pagination, similar to LIMIT and OFFSET in SQL. It begins by fetching the state of the bookmark if given, to denote the starting position for the iteration. If not, iterate through the ledger. In each state, it checks whether each key matches the requirement. If so, the asset (a) is added to the return set. The returning set is limited by N. Note that the efficiency of this procedure will depend on the blockchain state database. For example, KVS using indexed composite keys in sorted order will allow efficient iteration.

Algoritmo 4: *GetStateByEntityName*

Data: $entityName$, N, k_b

Result: set of assets $A = \{a_1, a_2 ... a_n\}$

```
1  if k_b = ∅ then
2  |   L ← ledger
3  else
4  |   L ← Get(k_b) // If a bookmark is given
5  end
6  A = {}
7  n ← N
8  while L has next do
9  |   a ← next
10 |   a_k ← a.key
11 |   if a_k.entityName = entityName then
12 |   |   if n > 0 then
13 |   |   |   A ← A ∪ a
14 |   |   |   n ← n − 1
15 |   |   end
16 |   end
17 end
```

4.2.2.2.6 R_3 — *GetStateByKeyList*

$GetStateByKeyList(entityName, S_{id}, N, k_b)$ implements R_3. It retrieves a set of states using a set of keys (S_{id}). See Algorithm 5. It uses the same algorithm as *GetStateByEntityName* (Algorithm 4) but with an additional set membership comparison (line 12).

4.2.2.2.7 R_4 — *GetStateByKeyRange*

$GetStateByKeyRange(entityName, id_i, id_j, N, k_b)$ implements R_4, which retrieves assets based on a range comparison. See Algorithm 6. Just like *GetStateByKeyList* (Algorithm 5) proposed for R_3, it also uses the same base algorithm as *GetStateByEntityName* but with the addition of the range comparison on top. The two parameters id_i and id_j comprise the range limits and are inclusive.

Algoritmo 5: *GetStateByKeyList*

Data: $entityName$, S_{id}, N, k_b
Result: set of assets $A = \{a_1, a_2 ... a_n\}$

if $k_b = \emptyset$ **then**
  $L \leftarrow ledger$
else
  $L \leftarrow Get(k_b)$ `// If a bookmark is given`
end
$A = \{\}$
$n \leftarrow N$
while L *has next* **do**
  $a \leftarrow next$
  $a_k \leftarrow a.key$
  if $a_k.entityName = entityName$ **then**
    if $n > 0$ *and* $a_k.id \in S_{id}$ **then**
      $A \leftarrow A \cup a$
      $n \leftarrow n - 1$
    end
  end
end

Algoritmo 6: *GetStateByKeyRange*

Data: $entityName$, id_i, id_j, N, k_b
Result: set of assets $A = \{a_1, a_2 ... a_n\}$

if $k_b = \emptyset$ **then**
  $L \leftarrow ledger$
else
  $L \leftarrow Get(k_b)$ `// If a bookmark is given`
end
$A = \{\}$
$n \leftarrow N$
while L *has next* **do**
  $a \leftarrow next$
  $a_k \leftarrow a.key$
  if $a_k.entityName = entityName$ **then**
    if $n > 0$ **then**
      if $a_k.id \geq id_i$ *and* $a_k.id \leq id_j$ **then**
        $A \leftarrow A \cup a$
        $n \leftarrow n - 1$
      end
    end
  end

4.3 Query Mapping

This section describes the approach taken to execute each type of SQL operation supported by Inter-MOON, including the supported syntax.

Inter-MOON supports queries written in standard SQL that exclusively contain a single DML or DQL statement, such as SELECT, INSERT, UPDATE, or DELETE. The query must start with one of these keywords. Support for Data Definition Language (DDL) commands, distributed queries, and SQL transactions is beyond the scope of this work.

See the flowchart in Fig. 22 for a general overview of how Inter-MOON executes queries. Assuming a valid query is received, the SQL Analyzer (Sec. 4.1.2) extracts the type of operation and the list of involved entities from the query (step 1 in Fig. 22). Then, the list of entities is checked against the blockchain schema (Sec. 4.2.1) to reveal which, if any, are blockchain entities (step 2 in Fig. 22).

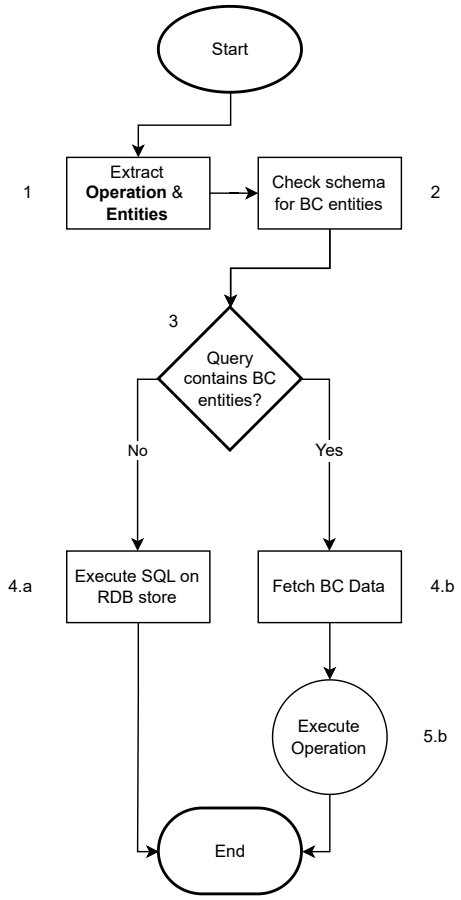

Figure 22 – Flowchart showing a simplified view of the Inter-MOON querying mechanism. The "Execute Operation" bubble will result in a different set of actions depending on the operation type.

Queries are expected to fall into one of the following scenarios: (1) SELECT,

INSERT, UPDATE, or DELETE with only relational entities, (2) SELECT, INSERT, UPDATE, or DELETE with only blockchain entities and (3) SELECT with JOIN using both blockchain and relational entities.

For scenario (1), Inter-MOON simply forwards the query to the RDB and sends back the response. For scenario (2), there are separate approaches depending on the type of SQL statement, explained further below. The approach for scenario (3) is the same as (2) for SELECT. In Fig. 22, step 4.a represents scenario (1), and step *5.b* represents scenarios (2) and (3). Note that for scenarios (2) and (3), before the query can be executed, the indexing mechanism uses one of the policies to fetch blockchain data in step *4.b* of Fig. 22 to enable virtualization. Virtualization is then executed in step *5.b*, except for INSERT operations. Inter-MOON does not virtualize assets during INSERT, as it is unnecessary to conclude the operation.

In the following subsections, each type of statement in scenario (2) will be described in detail.

4.3.1 SELECT

In SQL, SELECT queries are read operations that return tuples matching the query criteria. They can be quite complex and involve numerous operations, like aggregations, joins, and sub-selects, among others.

Inter-MOON completes SELECT queries by simply executing the proposed virtualization technique (Sec. 4.2) and then executing the received SQL query. After virtualization, blockchain entities will exist inside temporary tables alongside other relational entities, categorically allowing the execution of any kind of SQL SELECT query.

Code 2 is a Backus–Naur form (BNF)-like representation of the syntax supported by Inter-MOON in SELECT operations:

Código-fonte 2 – Supported syntax for SQL SELECT.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...]
       ]
```

```
6     [ HAVING condition ]
7     [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ]
        select ]
8     [ ORDER BY expression [ ASC | DESC | USING operator ] [
        NULLS { FIRST | LAST } ] [, ...] ]
9     [ LIMIT { count | ALL } ]
10    [ OFFSET start [ ROW | ROWS ] ]
11    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } {
        ONLY | WITH TIES } ]
```

Where `from_item` can be one of (Code 3):

Código-fonte 3 – Supported syntax for `from_item` element.

```
1     [ ONLY ] table_name [ * ] [ [ AS ] alias [ (
        column_alias [, ...] ) ] ]
2                 [ TABLESAMPLE sampling_method ( argument [,
                    ...] ) [ REPEATABLE ( seed ) ] ]
3     [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [,
        ...] ) ]
4     with_query_name [ [ AS ] alias [ ( column_alias [, ...]
        ) ] ]
5     [ LATERAL ] function_name ( [ argument [, ...] ] )
6                 [ WITH ORDINALITY ] [ [ AS ] alias [ (
                    column_alias [, ...] ) ] ]
7     [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS
        ] alias ( column_definition [, ...] )
8     [ LATERAL ] function_name ( [ argument [, ...] ] ) AS (
        column_definition [, ...] )
9     [ LATERAL ] ROWS FROM( function_name ( [ argument [,
        ...] ] ) [ AS ( column_definition [, ...] ) ] [,
        ...] )
10                [ WITH ORDINALITY ] [ [ AS ] alias [ (
```

```
                    column_alias [, ...] ) ] ]
    from_item join_type from_item { ON join_condition |
       USING ( join_column [, ...] ) [ AS join_using_alias
       ] }
    from_item NATURAL join_type from_item
    from_item CROSS JOIN from_item
```

And `grouping_element` can be one of (Code 4):

Código-fonte 4 – Supported syntax for the `grouping_element` element.

```
    ( )
    expression
    ( expression [, ...] )
    ROLLUP ( { expression | ( expression [, ...] ) } [,
       ...] )
```

The flowchart depicted in Fig. 23 illustrates the proposal for SELECT. (1) The necessary assets are fetched from the blockchain and virtualized. Virtualization incurs the activation of the indexing mechanism to fetch the relevant assets. In the flowchart, this step is omitted for clarity. (2) Then, the SELECT query is executed.

4.3.2 INSERT

On SQL, INSERT statements add one or more tuples into a table. The target table, column names, and attribute values for each given column can be defined in the query. For example, the query `INSERT INTO users (id, name, email) VALUES (57, 'John', 'john@mail.com')` will add a new tuple to the *users* table and assign it the combination of values *57*, *'John'* and *'john@example.com'* to the *id*, *name*, and *email* columns, respectively.

To execute INSERT on blockchain entities, Inter-MOON considers the relational INSERT to be functionally equivalent to the blockchain $Put(key, value)$ function (See also Sec. 4.1.9), with multi-valued INSERT equivalent to multiple consecutive *Put* calls. Fig. 24 shows the proposed mapping. The entity name is extracted from the table name identifier. The blockchain

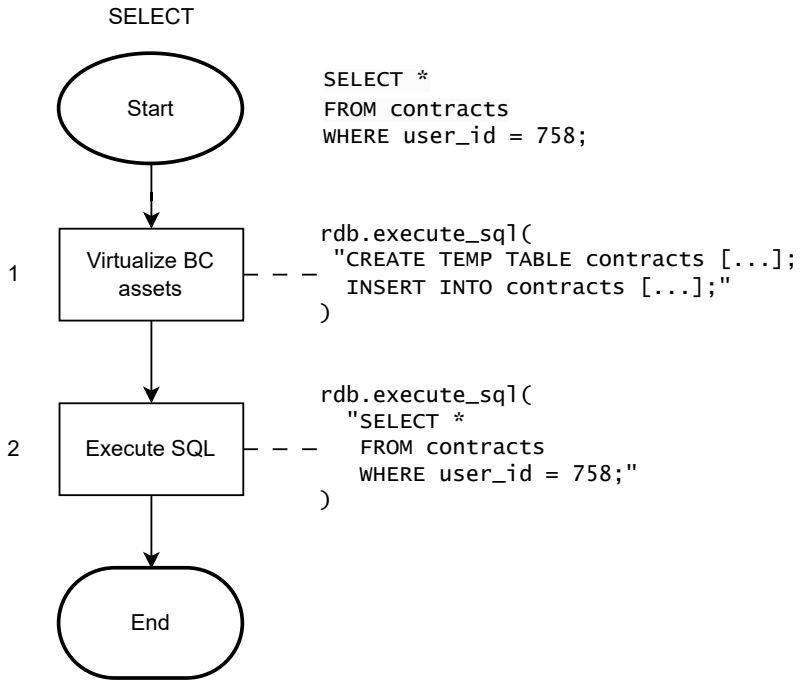

Figure 23 – Blockchain SELECT in Inter-MOON

key is constructed from the primary key and table name identifier (the *entityName*). The columns and values are extracted and transformed into tuples, except for the primary key, and comprise the *value*. The final result is a blockchain $< key, value >$ tuple.

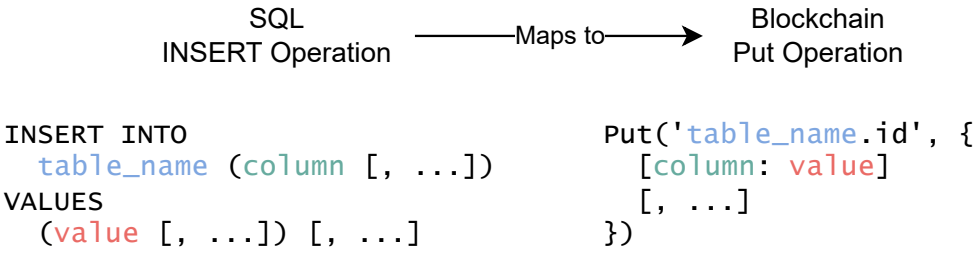

Figure 24 – SQL INSERT to Blockchain mapping.

As an aside, whenever an asset needs to be added to a blockchain, there is a need for extra information in the form of the asset owner's public and private keys to assign ownership and verify the transaction. There are two common scenarios for asset ownership: The asset owner is the application (or organization), and the asset owner is the application client. In this work, the approach is based on the first scenario. This is handled by the blockchain itself, in which the Inter-MOON middleware will be configured as part of the organization.

Inter-MOON supports INSERT statements written in the following format (Code 5):

Código-fonte 5 – Supported syntax for SQL INSERT.

```
1 INSERT INTO table_name [ AS alias ] [ ( column_name [, ...]
      ) ]
2     { VALUES ( { expression | DEFAULT } [, ...] ) [, ...] }
```

Fig. 25 shows an overview of the whole INSERT procedure with a *Contracts* entity. Note that virtualization is unnecessary in INSERT operations. If the blockchain *Put* is successful (step 2), the generated *key* is indexed according to the specified policy (step 3).

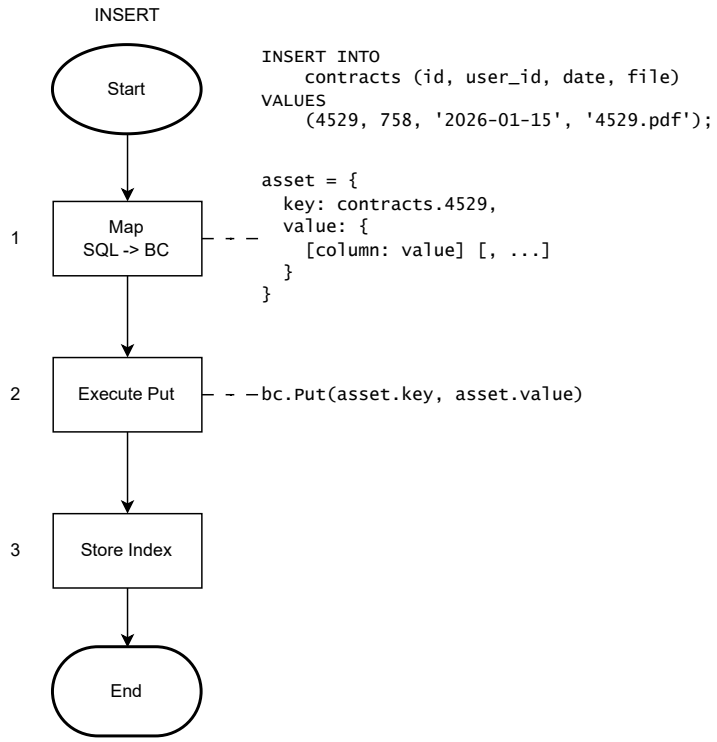

Figure 25 – Blockchain INSERT in Inter-MOON.

4.3.3 UPDATE

In SQL, the UPDATE statement is used to modify existing records in a table. One must specify the target table, and then the SET clause can be used to define the new values for the desired columns. Additionally, a WHERE clause can be used to specify which rows should be updated based on certain conditions.

In blockchain, each asset is unique, immutable, and may hold some arbitrary data.

Therefore, updating an asset is not normally allowed. Inter-MOON allows assets to be updated by viewing them as historical records, where each version of an asset represents a specific state of the asset meta-object. Inter-MOON creates a new version of the asset with an updated *value*. The mapping operation (Fig. 26) itself is another $Put(key, value)$, similar to INSERT. Both versions of the asset exist inside the blockchain within different transactions linked by their *key*.

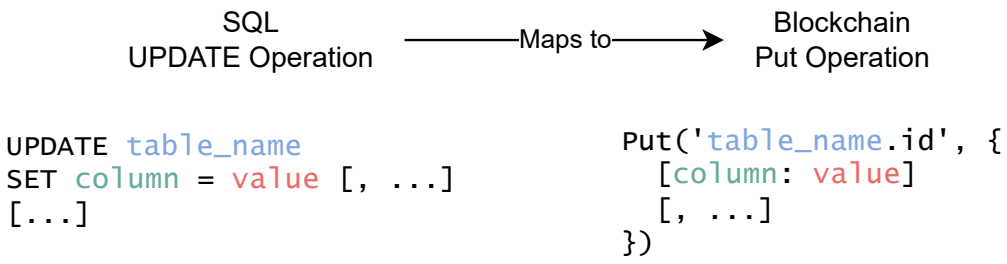

Figure 26 – SQL UPDATE to Blockchain mapping.

With this mapping in mind, Inter-MOON supports UPDATE statements written in this syntax (Code 6):

Código-fonte 6 – Supported syntax for SQL UPDATE.

```
1 UPDATE table_name [ * ] [ [ AS ] alias ]
2     SET { column_name = { expression | DEFAULT } } [, ...]
3     [ FROM from_item [, ...] ]
4     [ WHERE condition ]
```

Fig. 27 shows an overview of the Inter-MOON UPDATE procedure. (1) The relevant assets are virtualized. Then, (2) the PK of each asset that must be updated is selected. This is done by executing a SELECT operation that inherits the original UPDATE operation's predicates. (3) The UPDATE operation is executed, which updates the values of the virtualized assets. Finally, (4) the virtualized assets are extracted from the temporary table, packed into $< key, value >$ tuples, and stored on the blockchain using *Put*. If the operation is successful, the indexing mechanism is activated to update the asset versions.

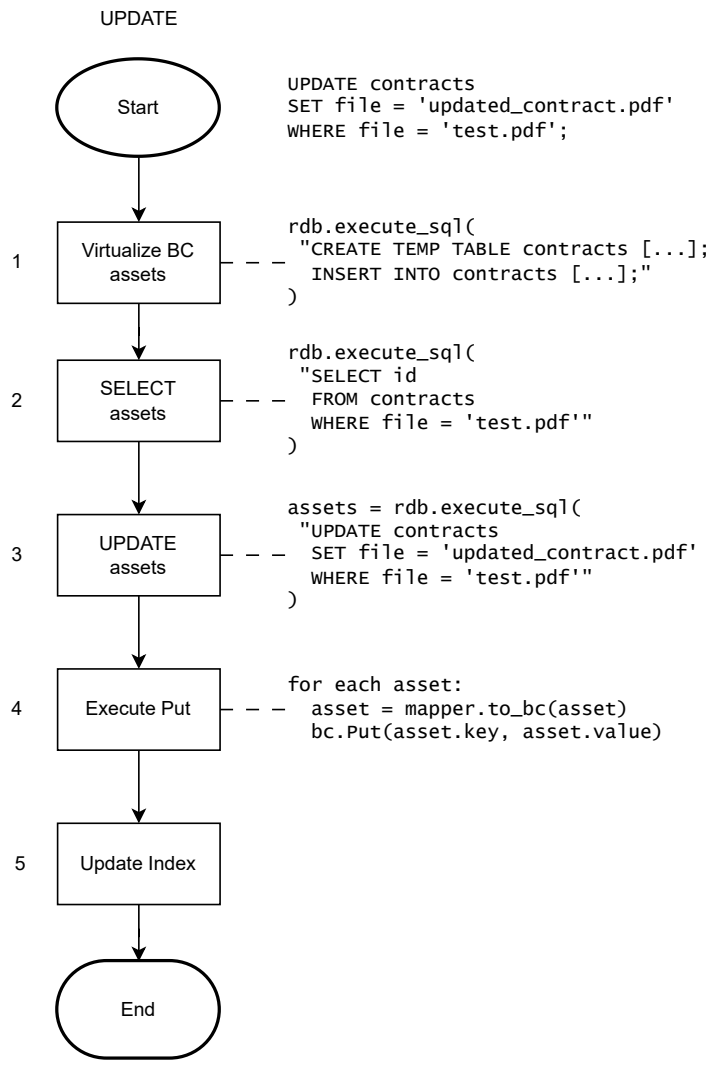

Figure 27 – Blockchain UPDATE in Inter-MOON

4.3.4 DELETE

A DELETE operation on SQL will delete tuples from a table. Predicates and other keywords can be used to specify which tuples should be deleted, as well as how related objects and constraints should behave, for example, in the presence of FKs referencing the deleted tuple. Once deleted, the tuples cease to exist and cannot be retrieved or referenced.

In blockchain, data cannot be deleted without affecting the integrity of the whole chain. Deleting a transaction will change the cryptographic hash of the block containing it and all subsequent blocks, breaking consensus. The same goes for the assets stored inside the transactions. Consequently, blockchain does not provide any means to delete or alter data from

mined blocks without interfering with the inner workings of the consensus algorithm.

However, due to the indexing mechanism (Sec. 4.2.2) proposed in this work, Inter-MOON offers a viable alternative. The approach is a soft-delete mechanism in which the asset is marked as deleted. This approach ensures that the blockchain consensus and integrity are maintained while replicating the result of a DELETE SQL operation within the Inter-MOON interface: preventing data retrieval and referencing of the deleted data. When viewed as a historical record, the asset is considered deleted because its latest version is unreachable. However, previous versions are still accessible from outside the Inter-MOON interface, and exist within the blockchain, as previously established. The mapping follows the structure depicted in Fig. 28. In blockchain, it is a *Del* operation that marks the specified asset and *key* as deleted.

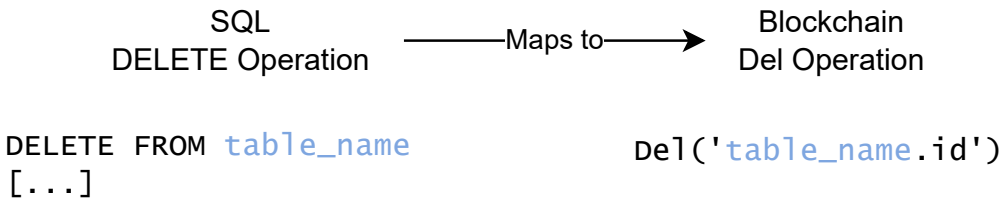

Figure 28 – SQL DELETE to Blockchain mapping.

Inter-MOON supports DELETE statements written in the following format (Code 7):

Código-fonte 7 – Supported syntax for SQL DELETE.

```
1 DELETE FROM table_name [ [ AS ] alias ]
2     [ WHERE condition ]
```

Fig. 29 shows a general flowchart of the Inter-MOON DELETE procedure. Steps (1) and (2) are the same as in the UPDATE procedure (Sec. 4.3.3). The assets are virtualized and a SELECT operation that inherits the predicates from the original query is executed to extract the PK of the soon-to-be-deleted assets. In (3), the *Del* operation is executed to mark the matching assets as deleted. Then, the index is updated.

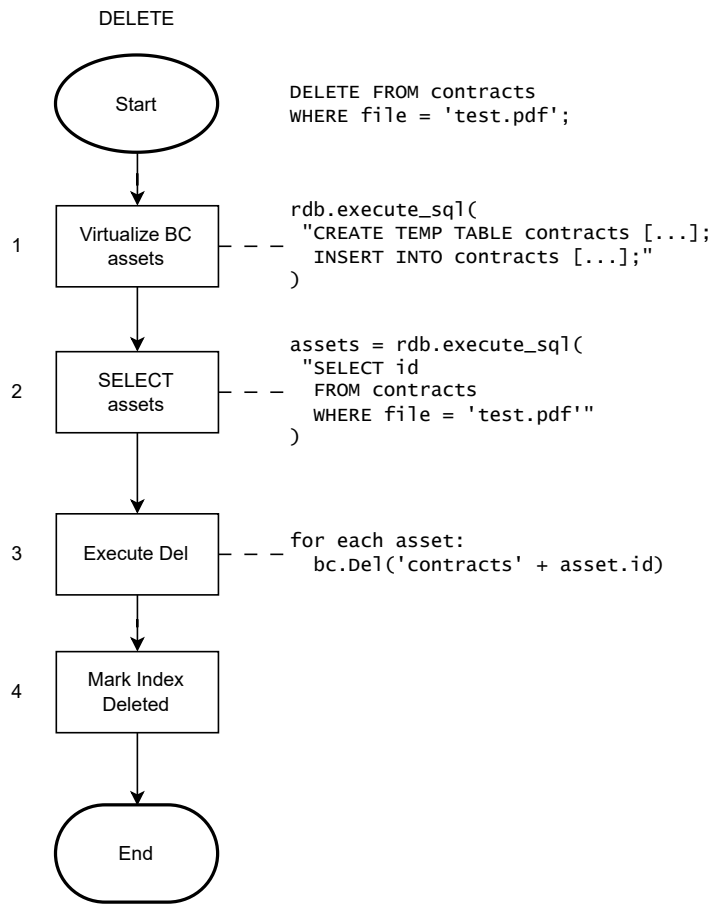

Figure 29 – Blockchain DELETE in Inter-MOON

5 EVALUATION AND RESULTS DISCUSSION

This chapter details the experimental methodology for Inter-MOON. Throughout this research, two prototypical versions of the Inter-MOON middleware were developed, one using the table-based indexing policy for the first experiment (Sec. 5.1) and another using the smart contract-based policy for the second (Sec. 5.2).

5.1 Comparing the performance of MOON & Inter-MOON

Among the differences between Inter-MOON and MOON, there are performance improvements. Therefore, an experiment was conducted to illustrate how effective these improvements have been. The Inter-MOON middleware prototype used for comparison was developed with Python 3.6.9 and the same blockchain network as MOON, BigchainDB. Since BigchainDB offers no support for smart contracts, the indexing policy was table-based, detailed in Sec. 4.2.2.2.

To keep comparisons fair, this experiment mirrored the experiment executed in the MOON paper (MARINHO *et al.*, 2020). Response speed was the metric, calculated using the full round-trip time taken from the moment the client sends the request to when it receives a response, same as in the MOON experiment. The data consisted of a synthetic dataset containing two entities, *Patients* (stored on the RDB) and *Lab Results* (stored on the blockchain), as per Fig. 30. The testing environment was built using a series of Ubuntu 18.04.6 Virtual Machine (VM)s running on a local network (Fig. 31). Table 7 describes each VM in detail. VM-1 contained instances of both MOON and Inter-MOON, only one of which was running at any time. PostgreSQL 9.6 was used for the SQL database in VM-2, while BigchainDB 2.2 was used for the blockchain nodes in the network. Lastly, a machine running Ubuntu 22.04, 4 GB of RAM, and an Intel i5-4300 2.60 GHz CPU was used to simulate the client.

Lab Results (100 rows)	
`varchar`	uid
`integer`	patient_id
`varchar`	content_base64
`date`	datetime
`varchar`	lab_name
`integer`	lab_site
`integer`	expired

Patients (100 rows)	
`integer`	id
`varchar`	name
`varchar`	email
`varchar`	phone
`date`	birth_date

Figure 30 – Entity schema used for the first experiment.

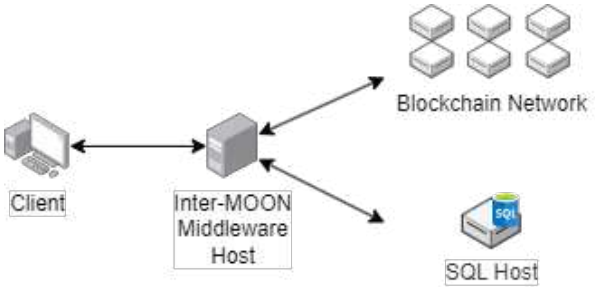

Figure 31 – Testing environment.

Name	Role	RAM	Disk Read & Write Speed
VM-1	Middleware host	4 GB	7.5 GB/sec & 0.8 GB/sec
VM-2	SQL database host	2 GB	6 GB/sec & 0.8 GB/sec
VM-3 … VM-8	Blockchain network nodes	1 GB/each	5 GB/sec & 0.4 GB/sec

Table 7 – Summary of the virtual machines used in the first experiment.

A set of four queries (Table 8) was executed on each tool. Inter-MOON was expected to provide significantly improved response speeds in queries involving many entities (Q2 and Q3) while maintaining similar, but still slightly faster speeds, in other kinds.

Query	SQL
Q1	`INSERT INTO lab_results (<...columns>) VALUES (<...values>);`
Q2	`SELECT * FROM lab_results;`
Q3	`SELECT * FROM lab_results JOIN patients ON lab_results.patient_id = patients.id;`
Q4	`UPDATE lab_results SET expired = 1 WHERE uid = <uid>;`

Table 8 – Set of queries used in the first experiment.

Results (Fig. 32) show that Inter-MOON was generally much faster. In Q1, the results were in the same ballpark. In Q2 and Q3, they were about 10 times higher. In Q4, there was an improvement of about 5.5 times, instead. UPDATE operations, which is the case for Q4, are more computationally expensive and latency-inducing, as they involve several trips to both database and blockchain systems to read, update, and write the updated information. Regardless, Inter-MOON managed to maintain sub-second response speeds, while MOON demonstrated very high latency (more than two seconds) on Q4.

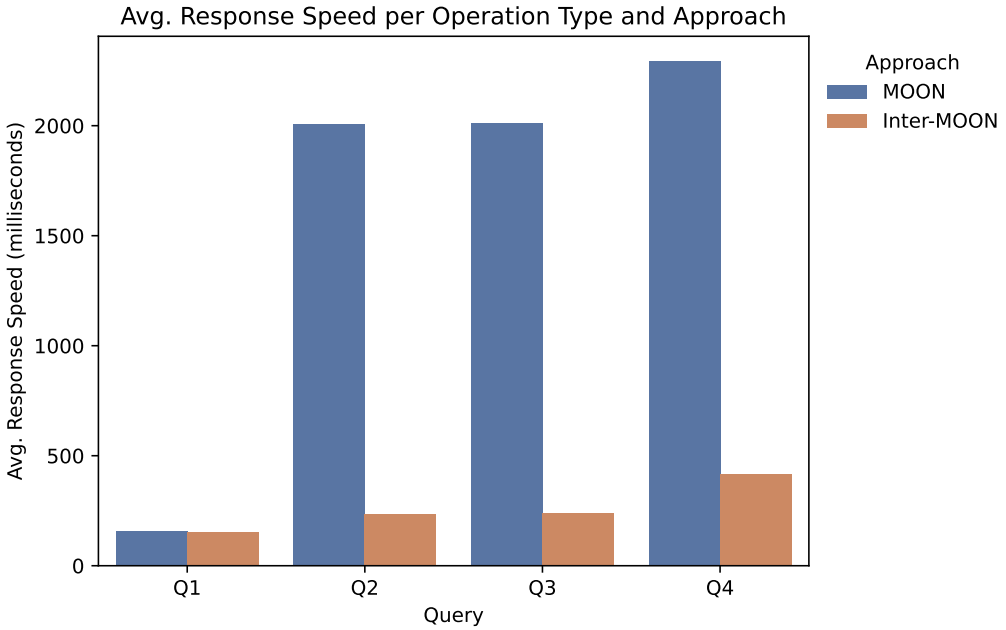

Figure 32 – Graphical comparison of the Avg. Response Speed of 100 query executions between MOON and Inter-MOON.

5.2 Evaluating the overhead introduced by Inter-MOON over a Blockchain-only approach

The purpose of this experiment is to evaluate the performance overhead introduced by the Inter-MOON middleware when compared to the base performance produced by an approach using only blockchain.

Hyperledger Fabric was chosen as the baseline blockchain, as it is a commonly used open-source blockchain solution and offers adequate smart contract functionality. Consequently, a second version of the Inter-MOON middleware was developed, using Python 3.10.12, as well as a Fabric smart contract, written in Go under the specifications laid out in Sec. 4.2.2.2. Naturally, the Inter-MOON and Fabric-only approaches used the same smart contract.

The experiment was conducted on a host machine running Ubuntu 22.04 with an 8-core 3.60GHz Intel i3-10100F CPU and 16 GB of RAM. The environment used docker containers for the Inter-MOON middleware, PostgreSQL 9.6, and a container each for a Hyperledger Fabric network of five peers and one orderer, for a total of six nodes. Fabric allows the choice of LevelDB or CouchDB as the state database, and LevelDB was chosen, as it is a KVS matching the specification. Solo was used for the ordering service plugin, as it is recommended for development and when the network only has one orderer node. This configuration represents the default settings for a Fabric network.

The data used in the experiment was synthetic. Three entities were created: a

relational entity *Users* to represent user data, and two blockchain entities, *Contracts* and *User Files*, to represent data related to users stored in the blockchain for record-keeping purposes.

The methodology was as follows: queries were continuously sent over a five minute (300 seconds) runtime. Query duration and the timestamp were recorded. The chosen metric was throughput, calculated as the total number of transaction requests (the workload) completed over the runtime, measured in Transactions per Second (TPS). Note that, in Fabric, each query is a smart contract function invocation sent via API, while in Inter-MOON, queries are written using SQL. Comparisons made in this experiment between the Inter-MOON and Fabric-only approach are limited to only the querying functions specified in Sec. 4.2.2.2, as Fabric is only capable of executing queries that are defined within smart contract functions.

The experiment was divided into three scenarios. With N being the number of records being retrieved or inserted by a query: (1) queries where $N = 1$, (2) queries where $N = 1$ featuring concurrent connections, (3) queries where $N \in \{1000, 5000, 10000\}$. (1) and (3) represent common use cases of a database and (2) a stress test scenario. For completeness, an extra fourth scenario was considered: (4) in which Inter-MOON executes a series of common SQL select operations unachievable by the Fabric alone, such as aggregation and join.

5.2.1 Scenario 1

To start, the first experimental scenario uses simple write and read queries where $N = 1$. Table 9 shows each query in the Fabric smart contract invocation and SQL syntax.

Query	Fabric	SQL
Q1: Select	`invoke("GetStateByKey", "user_files", "$key")`	`SELECT * FROM user_files WHERE id = $key`
Q2: Insert	`invoke("PutState", "user_files", "$key", '"user_id": 1000, "file": "file_user_$key.pdf"')`	`INSERT INTO user_files (id, user_id, file) VALUES ($key, 1000, 'file_user_$key.pdf')`
Q3: Update	`invoke("UpdateState", "user_files", "$key", '"user_id": 1000, "file": "updated_file.pdf"')`	`UPDATE user_files SET file = 'updated_file.pdf' WHERE id = $key`
Q4: Delete	`invoke("DeleteState", "user_files", "$key")`	`DELETE FROM user_files WHERE id = $key`

Table 9 – Queries used in the first scenario.

Table 10 presents the performance results for each approach. The Fabric-only approach exhibits considerably higher throughput across all queries, particularly for Q1. For Q2, Q3, and Q4, which are write operations, Fabric demonstrates consistent performance, while Inter-MOON shows slower execution for Q3 and Q4. Specifically, Inter-MOON is approximately 2.5 times slower for Q1, 1.7 times slower for Q2, 2.3 times slower for Q3, and 2.8 times slower for Q4 compared to Fabric.

In Inter-MOON, each update or deletion operation to blockchain entities involves multiple rounds of reads and writes, as well as DDL operations to set up and populate virtual tables, making these operations significantly more costly than inserts. Consequently, for operations that involve the creation or retrieval of a single asset, the overhead introduced by Inter-MOON in a Fabric-only approach is generally high. This performance is considered reasonable given the virtualization approach employed by Inter-MOON. In Q2, which represents a single INSERT operation, despite the lack of virtualization by Inter-MOON, there is still quite a difference in performance. This is due to the combined network latency of the indexing operations executed by Inter-MOON for each asset.

Approach	Throughput (tps)			
	Q1	Q2	Q3	Q4
Fabric	371.540000	156.763333	158.963333	153.650000
Inter-MOON	141.983333	89.970000	67.033333	53.253333

Table 10 – Table of the results of the first scenario.

5.2.2 Scenario 2

The next scenario deals with how each approach tackles concurrent connections. See Fig. 33. Two parameters were present: the query and the number of concurrent connections. Consider the queries to be the same as shown in Table 9 for Select (Q1) and Insert (Q2).

Both approaches manage to maintain consistent throughput. The Fabric-only approach demonstrates much higher throughput than Inter-MOON when selecting a single record. Inter-MOON struggles to keep up, once again for the same reason as in the first scenario. However, both are comparable when inserting a record with concurrent connections. On insertion, the blockchain network must achieve consensus between peers to pack data into a transaction and commit it to the ledger, a more complex process than a simple query. Given this, and considering the higher processing load required to handle a flood of requests from concurrent connections, it

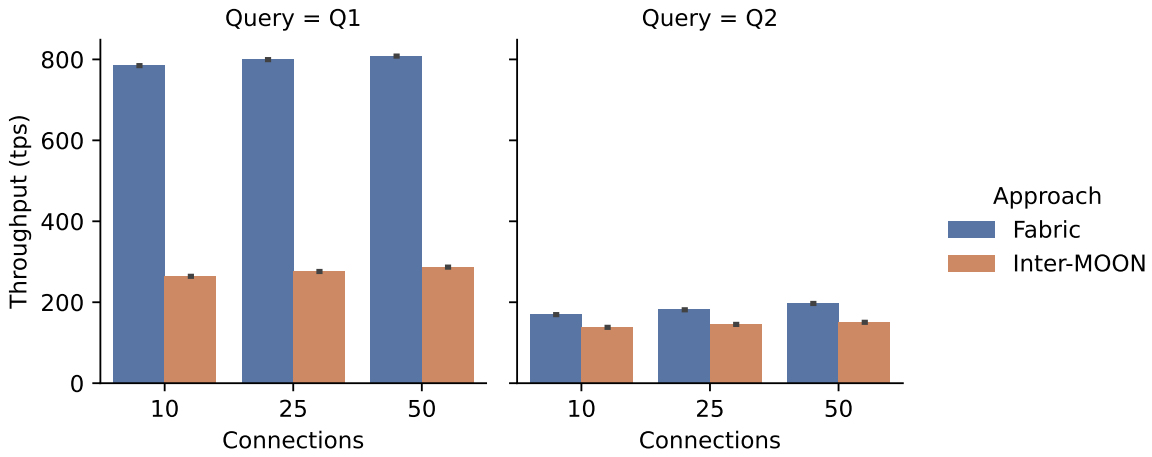

Figure 33 – Graph of the results of the second scenario.

is understandable that the concurrency manager of the blockchain will struggle. This prevents Fabric-only from reaching the same throughput in Q2 as in Q1 in this scenario, diminishing the gap between Fabric-only and Inter-MOON approaches.

5.2.3 Scenario 3

This scenario evaluates how each approach behaves when dealing with queries that retrieve multiple records. See Table 11 for the description of each query.

Query	Fabric	SQL
Q5	`invoke("GetStateByEntityName", "user_files", "N", "")`	`SELECT * FROM user_files`

Table 11 – Query used in the third scenario. The number of assets present in the ledger is always N, hence why the SQL query fetches all N even without predicates.

It is clear (See Fig. 34) that throughput is heavily impacted when compared to the first scenario (Sec. 5.2.1), from 150-370 TPS to less than 50 even when $N = 1000$, a relatively small amount of data. While there is still an initial gap in throughput between Fabric-only and Inter-MOON, it quickly diminishes as N increases. With $N \geq 5000$, the processing power required to iterate the ledger and pack every necessary asset into a response heavily impacts the blockchain throughput, rendering the overhead introduced by Inter-MOON minimal.

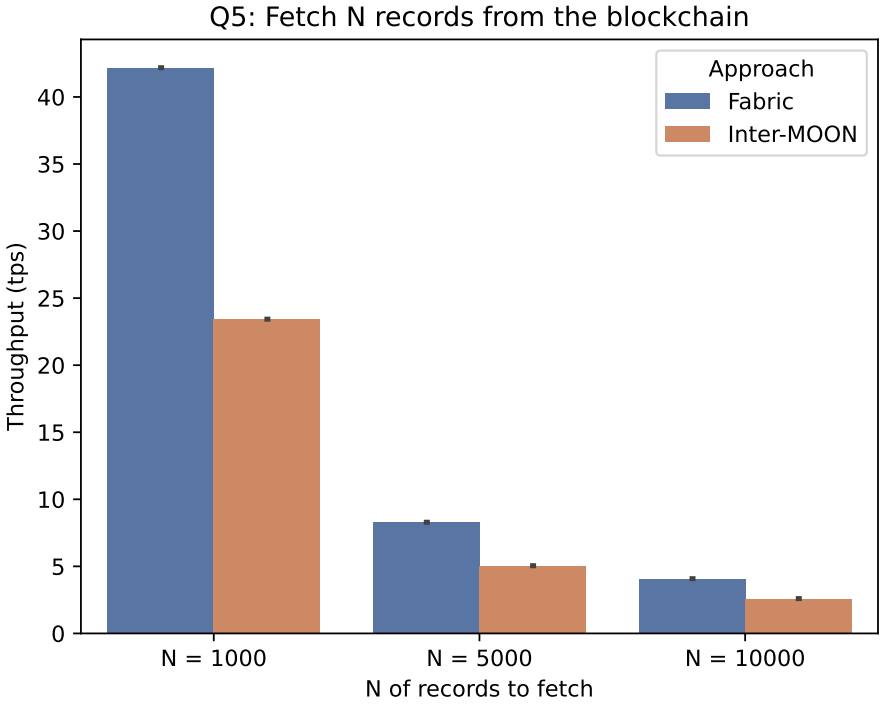

Figure 34 – Graph of the results of the third scenario.

5.2.4 Scenario 4

This scenario tackles Inter-MOON's performance when handling queries that Fabric alone cannot fulfill. There are many such queries, but four were chosen, described in Table 12. They are common SQL querying operations, such as subqueries, join, and aggregation. In this scenario, consider $N = 1000$. While the overhead cannot be fairly measured, given that there is no equivalent Fabric-only query, throughput can still be compared by using the values given in the third scenario (Sec. 5.2.3) for $N = 1000$ as a baseline, since each operation will still retrieve 1000 records. However, note that in Q7, Q8, and Q9, which deal with multiple entities, N is split evenly between each entity. In other words, in Q7 and Q9, $N = 500$ for *Users* and *User Files* respectively. And in Q8, $N = 500$ for *User Files* and *Contracts*.

It can be observed (Fig. 35) that throughput is comparable in all queries, except for Q7 and Q9. In them, throughput is higher because only 500 records were fetched from the blockchain, while the remaining 500 originated from the RDB, which is faster. It can be surmised that while the SQL query itself can have an impact on throughput, the biggest factor in this experiment is still the amount of data being fetched from the BC.

Query	SQL
Q6: Aggregation	`SELECT COUNT(*) FROM contracts GROUP BY expiration_date`
Q7: Join (BC + RDB)	`SELECT users.id, user_files.file, users.name, users.email FROM user_files JOIN users ON user_files.user_id = users.id`
Q8: Join (BC + BC)	`SELECT contracts.id, contracts.contract, contracts.expiration_date, user_files.user_id, user_files.file FROM contracts JOIN user_files ON contracts.file_id = user_files.id`
Q9: Subquery	`SELECT * FROM user_files WHERE user_files.user_id IN (SELECT id FROM users WHERE EXTRACT(YEAR FROM users.birthdate) > 1990)`

Table 12 – Queries used in the fourth scenario.

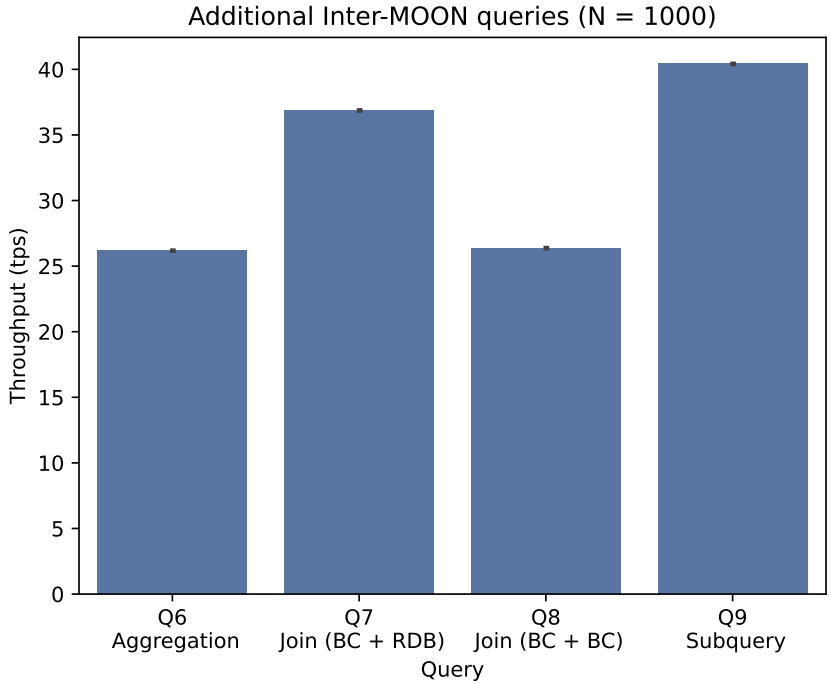

Figure 35 – Graph of the results of the fourth scenario.

In summary, this experiment demonstrates that, based on the testing methodology employed, the overhead produced by Inter-MOON is substantial for read queries involving a low volume of data ($N < 1000$), but becomes minimal otherwise. The overhead is especially pronounced for operations dealing with a single data point ($N = 1$). Conversely, write operations exhibit a significantly smaller performance gap, and in concurrent scenarios, Inter-MOON was able to scale throughput similarly to Hyperledger Fabric.

The analysis indicates that the volume of data retrieved from the blockchain is

the primary determinant of throughput. Inter-MOON's performance is constrained by its reliance on the underlying data stores' performance and scalability. Notably, blockchains can be slower compared to relational databases, particularly in write scenarios. Also note that, in Hyperledger Fabric, the blockchain's performance improves with an increasing number of nodes (ANDROULAKI *et al.*, 2018). This experiment used six nodes (five peers and one orderer), and as such the showcased throughput for both Inter-MOON and Fabric are likely to change in environments possessing better hardware and with a higher node count.

If the blockchain throughput is high, then Inter-MOON is limited mainly by the virtualization approach, which entails the RBD creating and populating temporary tables with blockchain data, a costly process, indicating a likely avenue for optimization. Therefore, when used to query blockchain assets, Inter-MOON might not be the best approach for very simple or light read operations when compared to using the native Fabric smart contract invocations, but shows promise in other scenarios, trading lowered throughput for better generality and more powerful querying capabilities offered by SQL.

When the blockchain throughput is high, Inter-MOON's performance is mainly limited by its virtualization approach. This involves creating and populating temporary tables with blockchain data, a process that is resource-intensive. Potential optimizations are discussed briefly in the limitations (Sec. 6.1) and future works (Sec. 6.2) sections of this dissertation. Therefore, while Inter-MOON may not be the most efficient for very simple or lightweight read operations compared to native Fabric smart contract invocations, it shows considerable promise in other scenarios. It offers a trade-off between reduced throughput in the described scenarios and enhanced generality and querying capabilities through the usage of SQL.

6 CONCLUSION AND FUTURE WORK

This work showcased Inter-MOON, an approach to enhancing interoperability between relational databases and blockchains.

Inter-MOON's approach to interoperability is achieved by virtualizing blockchain entities in the relational data model. Virtualization is characterized as the temporary materialization of blockchain assets into the relational environment (Sec. 4.2). A relational schema is defined and attributed to blockchain entities, enabling the differentiation and specification of blockchain data as if they were a relational object (Sec. 4.2.1). This schema is mutable and may change over time. While virtualization is computationally expensive, some measures were taken to optimize data retrieval and improve throughput, namely indexing and bulk data fetching (Sec. 4.2.2). Virtualization is powered by an indexing mechanism that tracks blockchain assets, of which two policies are proposed: a table-based policy (Sec. 4.2.2.1) and a smart contract-based policy (Sec. 4.2.2.2). The first saves asset keys in relational tables and can be used even by blockchains without smart contract support. The second stores asset keys into a state database via custom smart contract functions and offers optimized querying, but necessitates the development of a smart contract. A specification is described to expedite and facilitate smart contract development.

Virtualization enables the organic execution of SQL SELECT queries (Sec. 4.3.1). INSERT is executed by simply mapping the received query into blockchain data, sending it through the blockchain driver for appending and storing the key and generated transaction ID (Sec. 4.3.2). UPDATE is completed by virtualizing the necessary entities, translating the UPDATE query into a SELECT query, executing the UPDATE, and then extracting and sending the updated data for appending (Sec. 4.3.3). Updated assets represent an updated version of an existing asset. Both versions will exist within the blockchain, connected via key, and Inter-MOON will always use the latest version. DELETE queries are executed by "soft-deleting" the asset, marking the key as deleted to prevent retrieval (Sec. 4.3.4). In this manner, all SQL DQL and DML queries can be executed by Inter-MOON.

Two experiments were executed: one using the table-based policy to compare Inter-MOON against MOON (Sec. 5.1), and another using the smart contract-based policy to observe the overhead introduced by Inter-MOON in a Hyperledger Fabric blockchain setup (Sec. 5.2). For the first experiment, results demonstrated that Inter-MOON provides significantly better response times than MOON in common SQL operations. Improvements in this category ranged from slightly faster up to 10 times as fast. Along with increased performance, more kinds of SQL

operations, such as subqueries, aggregations, and SQL functions are now fully supported. The second experiment shows that the overhead introduced by Inter-MOON compared to a baseline Fabric-only approach can be large in scenarios with low data volumes, but much less significant otherwise. In write queries or queries retrieving large volumes of data ($N >= 5000$, where N is the number of records being retrieved), overhead was considered negligible.

To conclude, Inter-MOON is a novel application that explores a middleware approach to the problem of interoperability of blockchain and relational databases, an area with ongoing research. Inter-MOON enables the execution of SQL queries to blockchain data as if they were natively relational data, using mapping techniques to emulate the effects of these queries in the blockchain domain. Inter-MOON was observed to show significant improvements over MOON, regarding performance, accepted SQL grammar, and blockchain interoperability. Additionally, overhead is minimal in large data volume scenarios and write queries.

The following list of contributions, described in the introduction (Sec. 1), are present throughout this work:

1. Proposal and development of Inter-MOON, a novel approach to interoperability between blockchain and relational databases via virtualization of blockchain assets in a relational environment, allowing for natively comprehensive SQL DQL and DML grammar support.
2. Exploration of interoperability of relational and blockchain databases, and challenges like how to query, modify, or delete blockchain data using SQL queries.
3. Specification of the Inter-MOON smart contract approach, enabling optimized querying of key-value blockchain assets via composite key range queries.

Considering item 1, Sec. 4 describes the Inter-MOON proposal and Sec. 4.1 the architecture. Regarding item 2, discussions related to interoperability are explored in sections 2.3 and 4.2, while the challenges in querying, modifying, and deleting blockchain data are explored in section 4.3. Item 3, regarding the smart contract specification, is discussed in Sec. 4.2.2.2, as part of the proposed indexing policies.

6.1 Limitations

The Inter-MOON approach presents several limitations, discussed in this section.

6.1.1 Scalability

The scalability challenges still plaguing blockchain technology present obstacles. These scalability issues affect the overall performance and interoperability of the system. While they can be circumvented by using high-throughput scalable blockchains such as Hyperledger Fabric, if a less performant blockchain is used with Inter-MOON, the scalability and throughput of the whole system will be impacted. Some measures are taken to improve throughput, such as optimized indexing and querying, but the system is still limited by the scalability and concurrency mechanisms of the underlying data stores, especially the blockchain.

If the blockchain throughput is high, the main limitation in this regard becomes virtualization. Relational databases are very efficient, however, the proposed virtualization technique is expensive. Continuously fetching volumes of blockchain data and creating and populating tables with them can be quite costly for the relational database. This is why the overhead introduced by Inter-MOON is significant when dealing with queries that retrieve low volumes of data (See Sec. 5.2). In such queries, the main bottleneck is virtualization, rather than the actual data fetching procedure. As the number of fetched records increases, the bottleneck moves from virtualization to data fetching.

6.1.2 Amount of supported data stores

The Inter-MOON architecture was proposed with two data stores in mind: a relational database and a blockchain network. However, certain use cases may require data to be partitioned into multiple stores. For instance, a more secure RDB for critical user data (eg. payment methods, personal information), a less secure RDB for miscellaneous user data (eg. user preferences), and a blockchain for purchases. In such a scenario, Inter-MOON cannot be used fully, as the proposed architecture does not take into account additional data stores.

However, using multiple data stores is not unfeasible. The schema (Sec. 4.2.1) may be used to also denote the physical location of where each entity shall be stored, and this information can be referenced during data fetching.

6.1.3 Unsupported SQL commands

Inter-MOON does not support SQL DDL, Data Control Language (DCL) or Data Transaction Language (DTL) commands. Queries are expected to contain a single DQL or DML

statement such as SELECT, INSERT, UPDATE, or DELETE.

6.1.4 Schema Evolution

Inter-MOON allows for a mutable schema and the table creation algorithm accounts for missing attributes and attributes with changed data types. However, renaming attributes is not supported, and schema changes are not tracked. Sec. 6.2 shows a possible avenue for overcoming this limitation, inspired by the work of Wang *et al.* (2023).

6.1.5 Blockchain Asset Awareness

Inter-MOON is only aware of blockchain assets tracked by its indexing mechanism. If using the table-based policy, only assets created within the Inter-MOON interface will be tracked (Sec. 4.2.2.1). Assets created externally, or previous to the adoption of the approach, will be ignored unless manually indexed. If using the smart contract-based policy, they are indexed as long as the smart contract is still developed per the specifications (Sec. 4.2.2.2) and invoked to generate each asset, as the smart contract itself will be responsible for indexing.

6.2 Future Works

There are still many possibilities for future works regarding the interoperability of relational and blockchain systems. The approach presented in this work contains certain limitations, detailed in Sec. 6.1, which could be addressed. But beyond these limitations, there are still other unaddressed challenges.

This work does not support SQL DDL statements, such as CREATE/ALTER/DROP TABLE made to blockchain entities. Some of the cited related works, namely Zhu *et al.* (2020) and Wang *et al.* (2023), include support for these statements in a limited fashion. Among them, Wang *et al.* (2023) shows the greatest support, as it allows the usage of all three commands (CREATE, ALTER, and DROP) to define the structure (schema) of blockchain entities. Furthermore, schema changes are recorded in the blockchain, a plus for interoperability. A similar approach could be adopted by future works to track the schema evolution of blockchain entities over time. Assets can be associated with the current version of the schema upon generation, and those versions may be fetched and referenced by the algorithm to make older assets compatible with newer schema.

Additionally, Nathan *et al.* (2019) allows the creation of smart contracts via stored procedure syntax. CREATE OR REPLACE FUNCTION is used to create or update a smart contract, and DROP FUNCTION to delete it. Future works could research a similar approach in the context of middleware. For a middleware, there is a unique challenge in translating the PL/SQL or PL/pgSQL stored procedure syntax into each supported blockchain platform's respective smart contract language (eg. Solidity for Ethereum, or Java/JavaScript/Go for Fabric).

However, there are still features of SQL not covered by any of the mentioned works, such as DTL and DCL, that could become topics of future research. DTL encapsulates commands such as BEGIN TRANSACTION, COMMIT, and ROLLBACK, representing atomic transactions in SQL. To enable such commands within the context of blockchain, for example, a transaction could be initiated and kept in memory after a BEGIN TRANSACTION command while the consensus mechanism is accessed to verify and simulate the operations in order. Once verified, a COMMIT command could execute the list of operations. ROLLBACK could be used to discard the transaction altogether. Once committed, the list of operations executed by the transaction could also be stored in the blockchain to enhance integrity. However, it would be necessary first to establish a framework that allows the execution of these commands without hurting consensus.

As for DCL, it describes data access control commands within SQL, such as GRANT, REVOKE, and DENY. Permissioned blockchains often aggregate peers into organizations and use policies to define network rules. For instance, a blockchain may use a policy to define the amount of peers needed to verify a transaction, or which organizations can interact with which smart contracts. Therefore, future works could define a framework that maps SQL DCL operations into blockchain policy updates, possibly via extending the SQL grammar with policy-specific commands. For example, a custom CREATE POLICY keyword to add a new policy, and UPDATE POLICY to update it.

Future works may also tackle the challenge of compliance, standardization, and integration of blockchain and relational databases. For example, the definition of formal specifications or protocols detailing interactions between both data models. A formal API specification could be a step forward to interoperability, establishing a common framework of blockchain functionality and allowing any blockchain API developed using it to follow similar conventions, facilitating interoperability.

As for future works considering the proposed Inter-MOON architecture, the virtualization approach used in this work implements temporary tables, which significantly impact

performance, as noted in Sec. 5.2. Consequently, future works wishing to explore the idea further could look into optimizing virtualization, possibly via caching or a recycling algorithm. Alternatively, a synchronization algorithm could allow for full data replication with conflict resolution and improved throughput. However, Inter-MOON assumes that the RDB will also store other natively relational data, so storage and scalability would be undoubtedly affected by an endlessly growing blockchain ledger fully and consistently replicated in the RDB.

REFERENCES

ANDONI, M.; ROBU, V.; FLYNN, D.; ABRAM, S.; GEACH, D.; JENKINS, D.; MCCALLUM, P.; PEACOCK, A. Blockchain technology in the energy sector: A systematic review of challenges and opportunities. **Renewable and sustainable energy reviews**, Elsevier, v. 100, p. 143–174, 2019.

ANDROULAKI, E.; BARGER, A.; BORTNIKOV, V.; CACHIN, C.; CHRISTIDIS, K.; CARO, A. D.; ENYEART, D.; FERRIS, C.; LAVENTMAN, G.; MANEVICH, Y. *et al.* Hyperledger fabric: a distributed operating system for permissioned blockchains. In: **Proceedings of the thirteenth EuroSys conference**. [S.l.: s.n.], 2018. p. 1–15.

ANTONOPOULOS, A. M. **Mastering Bitcoin: Programming the open blockchain**. [S.l.]: " O'Reilly Media, Inc.", 2017.

BABCOCK, B.; BABU, S.; DATAR, M.; MOTWANI, R.; WIDOM, J. Models and issues in data stream systems. In: **Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems**. New York, NY, USA: Association for Computing Machinery, 2002. (PODS '02), p. 1–16. ISBN 1581135076. Available at: https://doi.org/10.1145/543613.543615.

BELCHIOR, R.; VASCONCELOS, A.; GUERREIRO, S.; CORREIA, M. A survey on blockchain interoperability: Past, present, and future trends. **Acm Computing Surveys (CSUR)**, ACM New York, NY, v. 54, n. 8, p. 1–41, 2021.

BUTERIN, V. *et al.* A next-generation smart contract and decentralized application platform. **white paper**, v. 3, n. 37, p. 2–1, 2014.

CASINO, F.; DASAKLIS, T. K.; PATSAKIS, C. A systematic literature review of blockchain-based applications: Current status, classification and open issues. **Telematics and informatics**, Elsevier, v. 36, p. 55–81, 2019.

CASTRO, M.; LISKOV, B. *et al.* Practical byzantine fault tolerance. In: **OsDI**. [S.l.: s.n.], 1999. v. 99, n. 1999, p. 173–186.

DELMOLINO, K.; ARNETT, M.; KOSBA, A.; MILLER, A.; SHI, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In: SPRINGER. **Financial Cryptography and Data Security: FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers 20**. [S.l.], 2016. p. 79–94.

GADEKALLU, T. R.; HUYNH-THE, T.; WANG, W.; YENDURI, G.; RANAWEERA, P.; PHAM, Q.-V.; COSTA, D. B. da; LIYANAGE, M. Blockchain for the metaverse: A review. **arXiv preprint arXiv:2203.09738**, 2022.

GAMAGE, H.; WEERASINGHE, H.; DIAS, N. A survey on blockchain technology concepts, applications, and issues. **SN Computer Science**, Springer, v. 1, p. 1–15, 2020.

GUO, H.; YU, X. A survey on blockchain technology and its security. **Blockchain: Research and Applications**, v. 3, n. 2, p. 100067, 2022. ISSN 2096-7209. Available at: https://www.sciencedirect.com/science/article/pii/S2096720922000070.

HAN, J.; SEO, Y.; LEE, S.; KIM, S.; SON, Y. Design and implementation of enabling sql–query processing for ethereum-based blockchain systems. **Electronics**, MDPI, v. 12, n. 20, p. 4317, 2023.

HASSELBRING, W. Information system integration. **Communications of the ACM**, ACM New York, NY, USA, v. 43, n. 6, p. 32–38, 2000.

JAVAID, M.; HALEEM, A.; Pratap Singh, R.; KHAN, S.; SUMAN, R. Blockchain technology applications for industry 4.0: A literature-based review. **Blockchain: Research and Applications**, v. 2, n. 4, p. 100027, 2021. ISSN 2096-7209. Available at: https://www.sciencedirect.com/science/article/pii/S2096720921000221.

KASUNIC, M.; ANDERSON, W. Measuring systems interoperability: Challenges and opportunities. **Software engineering measurement and analysis initiative**, 2004.

KIAYIAS, A.; RUSSELL, A.; DAVID, B.; OLIYNYKOV, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In: SPRINGER. **Annual international cryptology conference**. [S.l.], 2017. p. 357–388.

KING, S.; NADAL, S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. **self-published paper, August**, v. 19, n. 1, 2012.

KRICHEN, M.; AMMI, M.; MIHOUB, A.; ALMUTIQ, M. Blockchain for modern applications: A survey. **Sensors**, v. 22, n. 14, 2022. ISSN 1424-8220. Available at: https://www.mdpi.com/1424-8220/22/14/5274.

KWON, J. Tendermint: Consensus without mining. **Draft v. 0.6, fall**, v. 1, n. 11, p. 1–11, 2014.

LASHKARI, B.; MUSILEK, P. A comprehensive review of blockchain consensus mechanisms. **IEEE Access**, IEEE, v. 9, p. 43620–43652, 2021.

MACIEL, R. S.; VALLE, P. H.; SANTOS, K. S.; NAKAGAWA, E. Y. Systems interoperability types: A tertiary study. **arXiv preprint arXiv:2310.19999**, 2023.

MARINHO, S. C.; FILHO, J. S. C.; MOREIRA, L. O.; MACHADO, J. C. Using a hybrid approach to data management in relational database and blockchain: A case study on the e-health domain. In: IEEE. **2020 IEEE International Conference on Software Architecture Companion (ICSA-C)**. [S.l.], 2020. p. 114–121.

MELLO, B. H. de; RIGO, S. J.; COSTA, C. A. da; RIGHI, R. da R.; DONIDA, B.; BEZ, M. R.; SCHUNKE, L. C. Semantic interoperability in health records standards: a systematic literature review. **Health and technology**, Springer, v. 12, n. 2, p. 255–272, 2022.

MEYER, J. V.; MELLO, R. dos S. An analysis of data modelling for blockchain. In: SPRINGER. **Information Integration and Web Intelligence: 24th International Conference, iiWAS 2022, Virtual Event, November 28–30, 2022, Proceedings**. [S.l.], 2022. p. 31–44.

MISTRY, I.; TANWAR, S.; TYAGI, S.; KUMAR, N. Blockchain for 5g-enabled iot for industrial automation: A systematic review, solutions, and challenges. **Mechanical systems and signal processing**, Elsevier, v. 135, p. 106382, 2020.

MOHANTA, B. K.; PANDA, S. S.; JENA, D. An overview of smart contract and use cases in blockchain technology. In: IEEE. **2018 9th international conference on computing, communication and networking technologies (ICCCNT)**. [S.l.], 2018. p. 1–4.

MONRAT, A. A.; SCHELÉN, O.; ANDERSSON, K. A survey of blockchain from the perspectives of applications, challenges, and opportunities. **Ieee Access**, IEEE, v. 7, p. 117134–117151, 2019.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. **Decentralized Business Review**, p. 21260, 2008.

NATHAN, S.; GOVINDARAJAN, C.; SARAF, A.; SETHI, M.; JAYACHANDRAN, P. Blockchain meets database: Design and implementation of a blockchain relational database. **arXiv preprint arXiv:1903.01919**, 2019.

O'DWYER, K. J.; MALONE, D. Bitcoin mining and its energy footprint. IET, 2014.

POLITOU, E.; CASINO, F.; ALEPIS, E.; PATSAKIS, C. Blockchain mutability: Challenges and proposed solutions. **IEEE Transactions on Emerging Topics in Computing**, IEEE, v. 9, n. 4, p. 1972–1986, 2019.

RAMAKRISHNAN, R.; GEHRKE, J. **Database management systems**. [S.l.]: McGraw-Hill, Inc., 2002.

ROGAWAY, P.; SHRIMPTON, T. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In: SPRINGER. **Fast Software Encryption: 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004. Revised Papers 11**. [S.l.], 2004. p. 371–388.

SCHUH, F.; LARIMER, D. Bitshares 2.0: general overview. **accessed June-2017.[Online]. Available: http://docs. bitshares. org/downloads/bitshares-general. pdf**, 2017.

SCHUHKNECHT, F. M.; SHARMA, A.; DITTRICH, J.; AGRAWAL, D. chainifydb: How to get rid of your blockchain and use your dbms instead. In: **CIDR**. [S.l.: s.n.], 2021.

STONEBRAKER, M.; CETINTEMEL, U. "one size fits all": An idea whose time has come and gone. In: ____. **Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker**. Association for Computing Machinery and Morgan & Claypool, 2018. p. 441–462. ISBN 9781947487192. Available at: https://doi.org/10.1145/3226595.3226636.

SZABO, N. Smart contracts: building blocks for digital markets. **EXTROPY: The Journal of Transhumanist Thought,(16)**, v. 18, n. 2, p. 28, 1996.

TASCA, P.; THANABALASINGHAM, T.; TESSONE, C. J. Ontology of blockchain technologies. principles of identification and classification. **SSRN Electronic Journal**, v. 10, 2017.

TOLK, A.; DIALLO, S. Y.; TURNITSA, C. D. Applying the levels of conceptual interoperability model in support of integratability, interoperability, and composability for system-of-systems engineering. **Journal of Systems, Cybernetics, and Informatics**, v. 5, n. 5, 2007.

TOLMACH, P.; LI, Y.; LIN, S.-W.; LIU, Y.; LI, Z. A survey of smart contract formal specification and verification. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 54, n. 7, p. 1–38, 2021.

VRANKEN, H. Sustainability of bitcoin and blockchains. **Current Opinion in Environmental Sustainability**, v. 28, p. 1–9, 2017. ISSN 1877-3435. Sustainability governance. Available at: https://www.sciencedirect.com/science/article/pii/S1877343517300015.

WANG, Y.; PENG, Y.; LIU, X.; YING, Z.; CUI, J.; NIU, D.; XIA, X. achain: A sql-empowered analytical blockchain as a database. **IEEE Transactions on Computers**, IEEE, 2023.

WEGNER, P. Interoperability. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 28, n. 1, p. 285–287, 1996.

WOOD, G. *et al.* Ethereum: A secure decentralised generalised transaction ledger. **Ethereum project yellow paper**, v. 151, n. 2014, p. 1–32, 2014.

XIAO, Y.; ZHANG, N.; LOU, W.; HOU, Y. T. A survey of distributed consensus protocols for blockchain networks. **IEEE Communications Surveys & Tutorials**, IEEE, v. 22, n. 2, p. 1432–1465, 2020.

XU, Q.; CHEN, X.; LI, S.; ZHANG, H.; BABAR, M. A.; TRAN, N. K. Blockchain-based solutions for iot: A tertiary study. In: IEEE. **2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)**. [S.l.], 2020. p. 124–131.

YAGA, D.; MELL, P.; ROBY, N.; SCARFONE, K. Blockchain technology overview. **arXiv preprint arXiv:1906.11078**, 2019.

YUAN, Y.; WANG, F.-Y. Blockchain and cryptocurrencies: Model, techniques, and applications. **IEEE Transactions on Systems, Man, and Cybernetics: Systems**, v. 48, n. 9, p. 1421–1428, 2018.

YUE, K.-B.; CHANDRASEKAR, K.; GULLAPALLI, H. Storing and querying blockchain using sql databases. **Information Systems Education Journal**, v. 17, n. 4, p. 24, 2019.

ZHAI, S.; YANG, Y.; LI, J.; QIU, C.; ZHAO, J. Research on the application of cryptography on the blockchain. In: IOP PUBLISHING. **Journal of Physics: Conference Series**. [S.l.], 2019. v. 1168, p. 032077.

ZHENG, Z.; XIE, S.; DAI, H.; CHEN, X.; WANG, H. An overview of blockchain technology: Architecture, consensus, and future trends. In: IEEE. **2017 IEEE international congress on big data (BigData congress)**. [S.l.], 2017. p. 557–564.

ZHENG, Z.; XIE, S.; DAI, H.-N.; CHEN, X.; WANG, H. Blockchain challenges and opportunities: A survey. **International Journal of Web and Grid Services**, Inderscience Publishers (IEL), v. 14, n. 4, p. 352–375, 2018.

ZHOU, Q.; HUANG, H.; ZHENG, Z.; BIAN, J. Solutions to scalability of blockchain: A survey. **Ieee Access**, IEEE, v. 8, p. 16440–16455, 2020.

ZHU, Y.; ZHANG, Z.; JIN, C.; ZHOU, A.; QIN, G.; YANG, Y. Towards rich qery blockchain database. In: **Proceedings of the 29th ACM International Conference on Information & Knowledge Management**. [S.l.: s.n.], 2020. p. 3497–3500.