Enhancing Blockchain Implementations

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Technology (Research)

IN THE

Faculty of Engineering

BY

Hemant Kumar

Department of Computational and Data Sciences (CDS)

Indian Institute of Science

Bangalore – 560 012 (INDIA)

June, 2021

Declaration of Originality

I, **Hemant Kumar**, with SR No. **06-02-02-10-22-18-1-15816** hereby declare that the material presented in the thesis titled

Enhancing Blockchain Implementations

represents original work carried out by me in the **Department of Computational and Data Sciences** at **Indian Institute of Science** during the years **2018-21**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Hemant Kumar

Date: 20/06/2021 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

Mom and Dad

for being there always

Acknowledgements

Firstly, I would like to sincerely thank my supervisor, Professor Jayant R. Haritsa, for giving me an opportunity to work along with him. His immense knowledge, expertise, support, and guidance helped me be in the right direction and cross all barriers throughout the research. I am eternally grateful to him for the care, encouragement, and affection shown for the years I have spent at IISc. Thank you for always being there.

A true friend and colleague, Rakshith Gopalakrishna, has always been a person of appreciation. I am very thankful to him for his support throughout the days.

Also, I am extremely thankful to my Database Systems Lab family for always helping me in times of need. Their efforts, kindness, and support helped me remain motivated and ambitious. I wholeheartedly appreciate their guidance, time, and encouragement.

Further, I would also like to thank my close friends at IISc. They helped me in making my IISc days memorable and kept me motivated all time. For this, I am very grateful to them. Lastly, but importantly, I am eternally thankful to my mom, dad, and brothers for their unconditional love and support. They continuously helped me in building my character, strength and enlightened me to accomplish my dreams. They have always been there in my mind and heart. Thank you for everything.

Abstract

Blockchain technology has taken the computing world by storm over the past decade and has been widely addressed and researched in today's era of Computer Science. Blockchain technology provides the essential semantic properties of immutability, verifiability, authenticity, non-repudiation, and data integrity in a very efficient and reliable manner. Being an emerging technology, blockchain has gained wide-spread use cases, covering finance, health care, government, supply chain, and lots more. With the blockchain logical concepts developed, there has always been a question regarding its implementation method, i.e., whether it should be built fresh bottom-up or developed adopting the extremely-optimized relation database platforms.

In this research, a *non-invasive* methodology of building a blockchain platform on top of relational DBMS is studied - in particular, **Credereum** [23]. Credereum is a permissioned centralized blockchain implementation designed to be run on top of native PostgreSQL, and retains all the essential blockchain properties and functionalities. Further, to help generate the evidence for transaction modification and provenance, Credereum uses an immutable trusted storage repository, referred to as Ethereum [2, 14]. The principal reason to choose the Credereum platform is that it is among the few public-domain permissioned blockchain systems developed on an RDBMS platform. However, while exploring the Credereum platform, two significant issues were examined - basically, server-held malicious activities and performance. We address both these issues in our work, as described in the sequel.

With regard to malicious activities, the server might perform fraudulent actions in block Merkle formation, which remains unacknowledged to the clients. To address this, we propose SecCred, which adds to the functionalities provided by the base Credereum software, and indeed helps overcome the malicious activity in Merkle formation.

SecCred addresses the functionality enhancement proposed to the Credereum software, via the virtue of which every client can verify the correctness of the block formed by the server in an encrypted form. The client can perform the verification once the block gets created and is made public. The SecCred led changes in the hash calculation methodology of the nodes of the Merkle tree. Additionally, improvements to the Credereum held provenance function were made by computing only the desired results.

Now, concerning performance, the blockchain semantics implemented on top of the PostgreSQL via Credereum was certain to have a definite impact on the PostgreSQL performance. To identify this, we checked for its pitfall intensity, and on analysis, considering the native PostgreSQL throughput as the yardstick, a *huge* – about *3 orders-of-magnitude*! dip was witnessed. Even with relaxing the intrinsic overheads, i.e., the heavy-duty blockchain semantics' computational load, the decay was *heavy* – about *2 orders-of-magnitude*! In the research, we examined the primary reasons for this pitfall in performance, and with the help of simple but potent programming (ProgCred) and algorithmic changes (PerfCredA and PerfCredB) to the Credereum software, a substantial performance gain of an order-of-magnitude was achieved.

ProgCred dealing with programming changes can be considered to be the first step towards performance improvements. The CREDEREUM_LONGEST_PREFIX() function, being called about *hundred* times more often than the next most frequent function in Credereum, was converted from SQL to its C equivalent with programming changes. This transition reflected in ProgCred has led to a decrement in the function execution time by a huge factor of 99.93%.

Further, in Credereum, for each transaction fired, a Merkle tree gets constructed by depending upon the previous block in the ledger. With the second phase of performance improvements made in PerfCredA, we propose an independent high-performance design, leading to a substantial rise in performance and also efficiently summarizing the modifications made. With PerfCredA, we achieved a performance gain, but with the dependency removal, changes in provenance verification logic were needed and hence was re-designed.

An alternative and more radical optimization were with the development of the hash computing function, i.e., MODIFICATION_HASH_VALUE(), due which we can altogether drop the Merkle tree formation at the per-transaction level. With this strategy in PerfCredB, an even higher performance gain was witnessed; however, there's a downside to this approach, that it leaks the row counts updated by a transaction. But, since the leakage deals with *quantity* and not *identity*, no serious security implications are expected. With the new methodology involved in PerfCredB for summarizing the transaction modifications, it led to generating the evidence for provenance with certain changes, which indeed motivated us to re-design the provenance verification logic for PerfCredB.

The proposed versions - SecCred, ProgCred, PerfCredA, and PerfCredB handling functionality addition and performance improvements to the base Credereum, retain all the blockchain semantics and properties offered initially by the Credereum. The work also details the provenance handling mechanism and is explained with sample scenarios for each Credereum proposed version. On performance analysis, the SecCred, even with the additional work in the Merkle tree hash computation, was observed to have an approximately similar performance compared to the Credereum. The PerfCredB was found to achieve the highest performance amongst all the proposed versions. Additionally, PerfCredB has a comparatively low provenance response time than the Credereum and the rest proposed versions. Lastly, the thesis concludes by detailing significant points through which more advancements in performance can be researched.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **v**

List of Figures **ix**

List of Tables **xii**

1 Introduction **1**

1.1 Blockchain 1

1.1.1 History 2

1.1.2 Motivation and Capabilities 3

1.1.3 Properties and Types 4

1.2 Related Work 5

1.2.1 Blockchain meets database 5

1.2.2 BlockchainDB 6

1.2.3 BigchainDB 7

1.2.4 ChainifyDB 8

1.2.5 ForkBase 9

1.3 Credereum 10

1.4 Summary 16

2 Credereum **17**

2.1 Introduction 17

2.2 Credereum Internals 18

2.3 Architecture 23

2.4 Blockchain Properties 26
2.5 RC Isolation Level: Safe 27
2.6 Merkle Proof and Minimal Disclosure 28
2.7 Handling Double-spending 31
2.8 Drawbacks 32
2.9 Summary 34

3 Limitations and Proposal **35**
3.1 Introduction 35
3.2 Functional Limitation 36
3.2.1 Merkle Tree Display 36
3.2.1.1 Malicious Attack 37
3.2.1.2 SecCred: *Exhibit Merkle Data* 40
3.2.2 Provenance Space 40
3.2.3 Unique Indexing 41
3.3 Performance Limitation 42
3.3.1 Performance Evaluation 42
3.3.2 Credereum Shortcomings 43
3.3.3 Proposed Modifications 45
3.3.3.1 ProgCred: *Programming Modification* 45
3.3.3.2 PerfCredA: *Algorithmic Modification 1* 46
3.3.3.3 PerfCredB: *Algorithmic Modification 2* 46
3.4 Pointed Queries 46
3.5 Summary 47

4 Functional Improvement **48**
4.1 Introduction 48
4.2 SecCred: *Exhibit Merkle Data* 48
4.2.1 Merkle Tree: Hash Computation 49
4.2.2 TRANSACTION_AND_BLOCK_DATA relation 50
4.2.3 Execution and BMT Proof 51
4.2.4 TMT Display Necessity 53
4.2.5 BMT Sample Results 54
4.2.6 BMT Validity 60
4.3 Provenance Engine 63

4.4 Summary 63

5 Performance Improvement **64**

5.1 Introduction 64

5.2 ProgCred: *Programming Modification* 65

5.2.1 Function Optimization 66

5.3 PerfCredA: *Algorithmic Modification 1* 68

5.3.1 Architecture Design 68

5.3.2 Architecture Validity 72

5.4 PerfCredB: *Algorithmic Modification 2* 73

5.4.1 Architecture Design 73

5.4.2 MHV Algorithm 77

5.5 Performance Evaluation 78

5.6 Summary 79

6 Blockchain Semantics Retention **80**

6.1 Introduction 80

6.2 SecCred: Blockchain Properties 80

6.3 ProgCred: Blockchain Properties 82

6.4 PerfCredA: Blockchain Properties 84

6.5 PerfCredB: Blockchain Properties 85

6.6 Summary 87

7 Provenance Handling **88**

7.1 Introduction 88

7.2 Credereum/SecCred/ProgCred 89

7.2.1 Provenance Explanation 89

7.2.2 Sample Results 91

7.3 PerfCredA 97

7.3.1 Provenance Explanation 98

7.3.2 Sample Results 98

7.4 PerfCredB 105

7.4.1 Provenance Explanation 106

7.4.2 Sample Results 108

7.5 Summary 116

8 Experiment and Analysis 117

8.1 Experimentation Setup and Details . . . 117

8.2 Performance Analysis . . . 117

8.2.1 Varying Row Alteration (Closed System) . . . 118

8.2.2 Varying Row Alteration (Open System) . . . 119

8.2.3 Provenance Analysis . . . 120

8.3 Summary . . . 120

9 Conclusions and Future Work 121

9.1 Conclusions . . . 122

9.2 Future Work . . . 123

Bibliography 126

List of Figures

1.1 From Figure 1 of ‘Blockchain Meets Database’ [28] 6
1.2 From Figure 2 of ‘BlockchainDB - A Shared Database on Blockchains’ [10] . . . 7
1.3 From Figure 3 of ‘BigchainDB: A Scalable Blockchain Database’ [25] 8
1.4 From Figure 4 of ‘ChainifyDB: How to Blockchainify...’ [34] 9
1.5 From Figure 7 of ‘ForkBase: An Efficient Storage Engine for Blockchain...’ [36] . 10
1.6 Sample of CREDEREUM_BLOCK relation 13
1.7 Sample of CREDEREUM_TX_LOG relation 14
1.8 Sample of CREDEREUM_MERKLIX relation 15

2.1 Digitally-signed DBMS 19
2.2 Merkle key presentation 20
2.3 Credereum Merkle hash computation 21
2.4 Merkle proof 22
2.5 Credereum workflow 23
2.6 Credereum Merkle tree 24
2.7 Credereum Merkle proof generation 25
2.8 Credereum block creation 26
2.9 Credereum block BMT 30
2.10 Credereum old Merkle proof 30
2.11 Double-spending problem 32
2.12 Solving double-spending problem using trusted storage 33

3.1 Block 1 and Block 2 BMT 37
3.2 Block 3 BMT 38
3.3 Block B BMT 41
3.4 PostgreSQL v/s Credereum (TPE) 43

4.1 Transaction/Block Merkle tree 50
4.2 Block 1 BMT-Query Output - BMT of block 1 56
4.3 Block 2 BMT-Query Output 57
4.4 Block 3 BMT-Query Output (Part-1) - BMT of block 3 59
4.5 Block 3 BMT-Query Output (Part-2) - Block 2 BMT 60
4.6 Block B BMT 61
4.7 Block B-1 BMT - Hanging node verification 62

5.1 PerfCredA workflow 69
5.2 PerfCredA Merkle proof generation 70
5.3 PerfCredA block creation 71
5.4 PerfCredB workflow 74
5.5 PerfCredB proof generation 75
5.6 PerfCredB block creation 76
5.7 Comparative TPE Analysis 78

7.1 Provenance outcome for transaction T1 of block 1 92
7.2 Provenance outcome for transaction T2 of block 2 93
7.3 Provenance outcome for transaction T3 of block 2 94
7.4 Provenance outcome for transaction T4 of block 3 (Part-1) 95
7.5 Provenance outcome for transaction T4 of block 3 (Part-2) 96
7.6 Provenance outcome for transaction T4 of block 3 (Part-3) 96
7.7 Provenance outcome for transaction T4 of block 3 (Part-4) 97
7.8 PerfCredA provenance outcome for transaction T1 of block 1 100
7.9 PerfCredA provenance outcome for transaction T2 of block 2 101
7.10 PerfCredA provenance outcome for transaction T3 of block 2 102
7.11 PerfCredA provenance outcome for transaction T4 of block 3 (Part-1) 103
7.12 PerfCredA provenance outcome for transaction T4 of block 3 (Part-2) 104
7.13 PerfCredA provenance outcome for transaction T4 of block 3 (Part-3) 104
7.14 PerfCredA provenance outcome for transaction T4 of block 3 (Part-4) 105
7.15 PerfCredB provenance outcome for transaction T1 of block 1 110
7.16 PerfCredB provenance outcome for transaction T2 of block 2 111
7.17 PerfCredB provenance outcome for transaction T3 of block 2 112
7.18 PerfCredB provenance outcome for transaction T4 of block 3 (Part-1) 113
7.19 PerfCredB provenance outcome for transaction T4 of block 3 (Part-2) 114
7.20 PerfCredB provenance outcome for transaction T4 of block 3 (Part-3) 115

LIST OF FIGURES

8.1 TPE Analysis (Open System) 119

List of Tables

1.1 Abbreviations used . . . 11
1.2 Credereum - Relation and attributes . . . 13

4.1 TRANSACTION_AND_BLOCK_DATA schema . . . 51

5.1 Credereum - Functions list . . . 65

8.1 TPE Analysis (Closed System) . . . 118
8.2 Average Provenance Response Time (secs) . . . 120

Chapter 1

Introduction

The chapter introduces Blockchain and **Credereum**, a blockchain platform that runs on top of native PostgreSQL. A brief history of the blockchain and the underlying motivation and capabilities have been explained. The chapter additionally outlines core blockchain properties and types. A brief discussion of the relevant works on the blockchain system design and development, including the underlying architecture, has been made. Finally, the Credereum retaining blockchain properties get introduced.

1.1 Blockchain

Blockchain is a time-stamped collection of immutable and permanent digital records in entities called *Blocks*, where each block is linked using cryptographic standards. Once recorded within a block, the data cannot be changed retroactively without modifying all subsequent future blocks and involves the majority of the network entities' agreement. The transaction details verified contain every single modification being made and get validated by the maximum. The low transaction fee, reduction in transaction waiting time, enhance trust, no 3-rd party involvement, advanced traceability are certain benefits linked to blockchain systems. Blockchain is an exceptionally assuring and innovative technology that possess clients' trust by resisting fraudulent activities and enabling transparency for innumerable use cases in a scalable manner. Furthermore, blockchain has widespread use cases, incorporating asset and real estate management, global trade and enterprise, cryptocurrency and payment handling, real-time IoT OS development, polling mechanism, keeping track of supply chain and logistics, insurance, banking and monetary businesses, media, healthcare, and lots more. By securing the digital connection, blockchain, a revolutionary technology, achieves the unattainable.

1.1.1 History

The evolution of blockchain began with the work published by research scientists Stuart Haber and W. Scott Stornetta in 1991 [15], introducing *timestamp* in generating digital records to resist tampering and avoid backdating. They designed a cryptographically secured system to store the timestamp records in the forms of blocks linked in a chain, and the concept has been used to date in the design of blockchain systems. Additionally, to integrate a list of records within the block, in 1992, researchers incorporated system designs with the Merkle trees [5].

In 2004, computer scientist Hal Finney designed a system named Reusable Proof-of-Work (RPoW) [12]. The system generates an RSA-signed token, which is peer-to-peer transferable and works by intaking a Hashcash based Proof-of-Work (PoW) token, which is non-fungible/cannot be exchanged. The fundamental *double-spending* [1] problem [1] was resolved using RPoW, where the token's right registered on the authentic servers gets retained, allowing clients to verify its fidelity.

With the author/group being pseudonym as Satoshi Nakamoto, a white paper was published in late 2008 [27], which proposed a decentralized peer-to-peer e-cash system and introduced distributed blockchain. The double-spending was handled using the decentralized peer-to-peer network, and the transactions were mined by miners earning rewards through PoW methodologies. With Satoshi Nakamoto, on January 3, 2009, mining the first bitcoin block, and making a reward of 50 bitcoins, led to establishing the public Bitcoin network.

In 2013, computer programmer Vitalik Buterin, apart from cryptocurrency, started forming a new distributed blockchain computing system, named Ethereum [2, 37, 20], featuring and introducing scripts, being named as *Smart Contracts* [3]. Smart contracts are coded protocols that execute automatically when the specified conditions according to contract terms are met. Smart contracts draw other assets like services, certificates, mortgages, and so forth to be shared similarly to cryptocurrency. Furthermore, Ethereum was designed with a focus on providing support to the blockchain applications.

2014, and onward, began an era where blockchain and Bitcoin were sought differently, diverging from the earlier concept that keyed them one. Blockchain 2.0 seeded up looking into applications apart from cryptocurrency, like healthcare, polling, security, supply chains, and more. Ethereum, Ripple [32], and HyperLedger [4] are specific applications developed in the Blockchain 2.0 initiative.

[1]In a digital payment system, double-spending is a potential risk where the same digital currency can get spent multiple times.

1.1.2 Motivation and Capabilities

The immutability and providing trust to the clients in a trustless society form the significant reason for adopting blockchain technology. With earlier methods, deploying a 3-rd party firm and considering it to be trustworthy, the entire set of transactions was being managed between the clients. With the evolution of blockchain technology, possessing a distributed and dense peer-to-peer network required no such middle 3-rd party firm involvement, and it provides clients an assurance/certainty of a better system with enhanced security and concreteness. In the blockchain network, each transaction fired gets verified and mined by a miner, competing with the rest. With the low fraudulent miners' malicious actions, the honest miners can readily reject such activities. Thus, the system provides a *diffused trust* to the clients, where a joint assemblage of miners is believed/trusted.

The secured recording of transactions in immutable blockchain ledgers drives corporations, governments, social sections, institutions, museums, etc., to widespread adoption of blockchain technology. The protocols strengthening blockchain, like the transparency and maintenance of the public distributed ledger, makes it more resilient and robust. The decentralized environment helps overcome traditional issues like single server dependency and failure, faster transaction settlement, security concerns, etc. The blockchain networks being resistant to malevolent and counterfeiting actions lead as the backbone to multi applications, and due flexibility, allows groups to decide the usage as per requirements.

The blockchain network provides provenance proof, which helps acknowledge the modification history/supply-chain to the desirables, making it a significant system functionality. The introduction of smart contracts in blockchain helps form automatic executable legal documents that operate on the specific conditions being met and is competent in deploying large-scale applications with the design necessities. With traditional systems relying on 3-rd party firms, the blockchain can provide an upgraded level of security and accuracy in a P2P network with no intermediate firms. The blockchain networks are capable of providing access control, via which a client can list the others who are allowed to access a particular asset registered on the ledger, using *Access Control List's* (ACLs). In a blockchain, each client possesses a set of *Public*, and *Private* keys, which helps form a *digital signature* of modifications made, and by being anonymous, can prove the ownership of a specific record/asset registered on the ledger. Since the blockchain network in a distributed environment has multiple servers connected, it can handle or is resilient to data damage and loss. Further, the breakdown server can fetch the records from the rest entities connected. These lists a set of system capabilities underlying the blockchain network.

1.1.3 Properties and Types

The following outlines some of the blockchain's core properties:

1. **Immutability**: The blockchain holds an essential property of immutability, which implies anything if it has written in a ledger (or blockchain), it cannot be modified/tampered with. The blockchain attains this property with the help of cryptographic hash functions [16]. Cryptography being mathematical and holding complex computation, aids in resistance to malicious attacks and provides a unique descriptor to every data. With each block linked with the previous in blockchain, any malevolent modifications made within a block will lead the ledger to be re-evaluated and approved by majority network entities.

2. **Decentralized and Distributed Ledger**: The blockchain system operating in a decentralized environment has multiple entities connected locally/globally, and all retain the information about the network. The block ledger remains the same, and for every new block appendage to the ledger, a consensus gets achieved. Because of being decentralized, it is fault-tolerant and does not have to rely on one single entity. The platform's decentralized and distributed design helps form a dense P2P network with lower transaction computation cost and time. Blockchain, being decentralized and due consensus, can handle malicious attacks efficiently.

3. **Transparency**: In a blockchain system, every entity can examine the transactions being fired to the system and respond to their validity. If the transaction is found valid, it is committed and gets used in the block formation. Furthermore, a client's individuality in the network is obscured via complex cryptography and uses its public address for identification in a permissionless blockchain system.

4. **Verification**: In a blockchain network, for every transaction fired, each miner verifies the correctness of the modifications made by the transaction, and if found valid, is accounted for in the block formation. With max honest miners, any malicious actions get detected, and the transaction gets dropped.

5. **Authenticity**: Many blockchain deployments ask the clients to submit a digital signature using its private key to authenticate the transaction. The clients can also be asked to sign the smart contracts to provide evidence of an agreement. Like PBFT [8], consensus algorithms encourage the clients/nodes to submit digital signatures with the current block formation agreement.

With the blockchain system requiring digital signature submission, the non-repudiation and data integrity property gets conserved.

Public, *Private*, and *Hybrid* blockchain networks are the three forms. The public blockchain network is open and accessible with no restrictions, and any individual can join the network, fire transactions, and become a validator. The private blockchain network is more permissioned, where an individual needs to get approval from the network admins. The public blockchain network works in a decentralized environment, while the private blockchain network shows more centrality. The hybrid network possesses features of both centralized and decentralized environments.

1.2 Related Work

Blockchain with underlying RDBMS has been a center of attraction for numerous enterprises dealing with issues related to 3-rd party dependency, seeking faster and low-cost transaction management, holding immutable and auditable records, providing trust, etc. With the blockchain adoption rate spanning worldwide, many countries/organizations invest resources to research and incorporate the technology. This section explains a few blockchain systems working along with databases and briefs of their underlying architecture. In particular, BlockchainDB, BigchainDB, ForkBase, etc., have been addressed.

1.2.1 Blockchain meets database

The research work by IBM, Blockchain meets database [28], demonstrates the designing and deployment of decentralized-replicated relational database systems with blockchain assets. A permissioned blockchain model is considered, with mutually distrustful organizations each operating their own replicated database instance. The paper addresses two approaches, specifically, *order-then-execute* and *execute-and-order-in-parallel*, shown in Figure 1.1 (referred from Figure 1 of [28]). With the transactions committing order being settled before execution in the illustrated order-then-execute approach, the researchers in the execute-and-order-in-parallel procedure proposed methodologies where the transaction execution is made without any assumption/information of the commit order. Further, at the same time, the ordering gets implemented in parallel. During block processing in both approaches, all the transaction details (with commit/abort status) and the block number get automatically stored in the ledger table to sustain recovery from failure. Further, the PostgreSQL database with *abort during the*

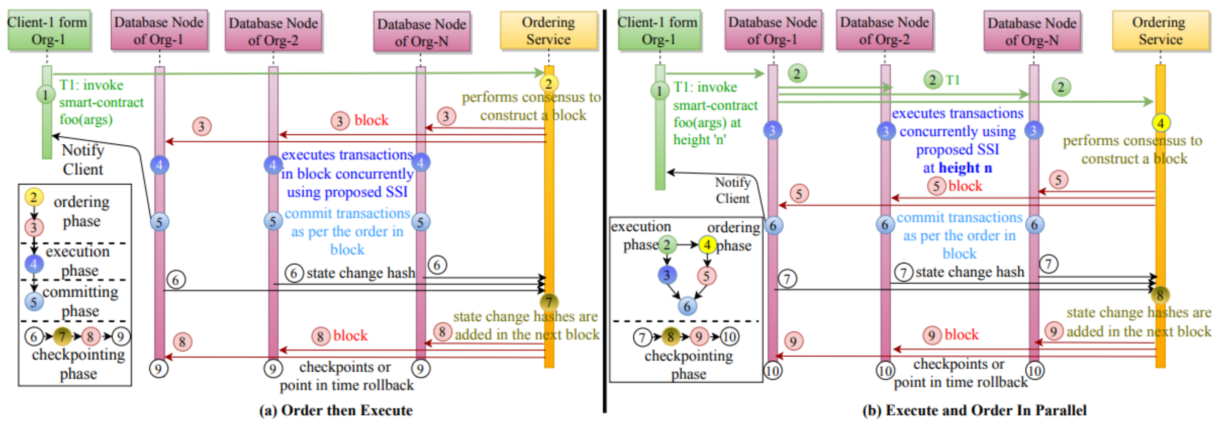

Client-1 form Org-1
Database Node of Org-1
Database Node of Org-2
Database Node of Org-N
Ordering Service
T1: invoke smart-contract foo(args)
performs consensus to construct a block
block
Notify Client
executes transactions in block concurrently using proposed SSI
commit transactions as per the order in block
state change hash
state change hashes are added in the next block
block
checkpoints or point in time rollback
ordering phase
execution phase
committing phase
checkpointing phase
(a) Order then Execute
T1: invoke smart-contract foo(args) at height 'n'
T1
executes transactions concurrently using proposed SSI at height n
block
Notify Client
execution phase
ordering phase
commit transactions as per the order in block
state change hash
state change hashes are added in the next block
block
checkpoints or point in time rollback
checkpointing phase
(b) Execute and Order In Parallel

Figure 1: Proposed transaction flows—*order-then-execute* and *execute-and-order-in-parallel*

Figure 1.1: From Figure 1 of 'Blockchain Meets Database' [28]

commit SSI variant [29] gets used to implement and construct a blockchain relational database management system.

1.2.2 BlockchainDB

The BlockchainDB [10] is designed to run the DB layer on top of the blockchain network, as shown in Figure 1.2 (referred from Figure 2 of [10]). The BlockchainDB to circumvent the consensus overhead doesn't replicate the records with all entities/peers. The relations are partitioned and replicated to a few entities. Each partition is being referred to as a shard and gets implemented as an independent blockchain network. Focusing performance boost, all entities do not store data locally, and on-demand will redirect the request to local/global storage. For modifications made by the client on the BlockchainDB network, an entity can host the database with a replica of at least one shard referred to as *Full Peer* (e.g., *Full Peer A*, *Full Peer B*, *Full Peer C* in Figure 1.2), or those possessing inadequate means can connect with the rest entities to obtain the shard information and are referred as *Thin Peer* (e.g., *Thin Peer D* in Figure 1.2). BlockchainDB, like private blockchains, assumes authenticated entities in the blockchain network and are identifiable.

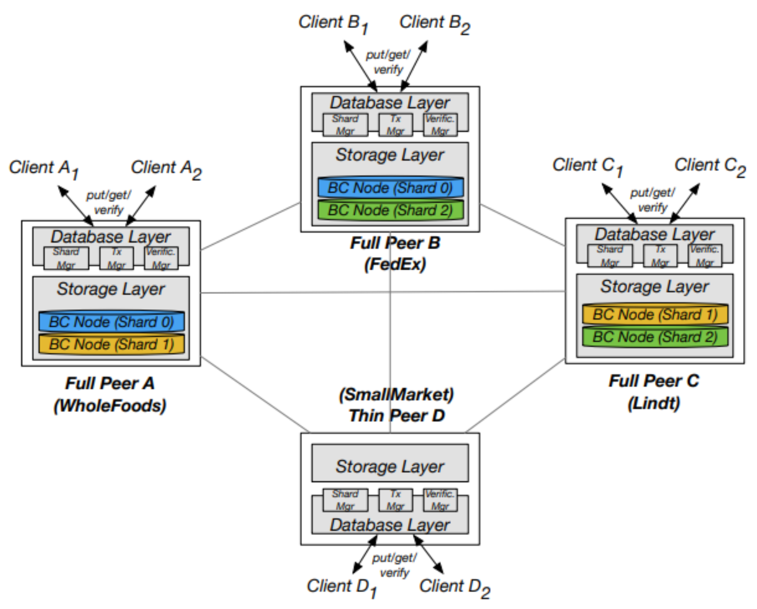


Figure 2: A typical *BlockchainDB* Network

Figure 1.2: From Figure 2 of 'BlockchainDB - A Shared Database on Blockchains' [10]

1.2.3 BigchainDB

BigchainDB [6, 25], with distributed database and traditional blockchain concepts, try to gain high throughput, huge capacity, lower latency, immutability, strong querying, and ample authority providence. BigchainDB system has two underlying distributed database components, named S and C, as shown in Figure 1.3 (referred from Figure 3 of [25]). The S and C are connected via *BigchainDB Consensus Algorithm* (BCA) and made to run on each signing entity. To retain consistency, an internal *Paxos-like Consensus Algorithm* gets run on each database. With the transactions allocated, each signing node moves the unordered transaction set from S to an ordered list, creates a block referring to the parent block, and places it in the ordered blocks' list at C. Each signing node votes for a valid/invalid block, and with the maximum valid votes, the block moves from undecided state to valid state, else is termed invalid, and the voting concludes. Each block B has an ID, timestamp, transactions, and polling details. With requirement for database possessing Paxos or its descendant, ElasticSearch [11], Cassandra [7], Riak [24], MongoDB [26], Redis [30], RethinkDB [31], and HBase [17], were selected for implementation. With strong consistency assurance [13] and auto change alerts [18], the developers built the first BigchainDB version on top of RethinkDB.

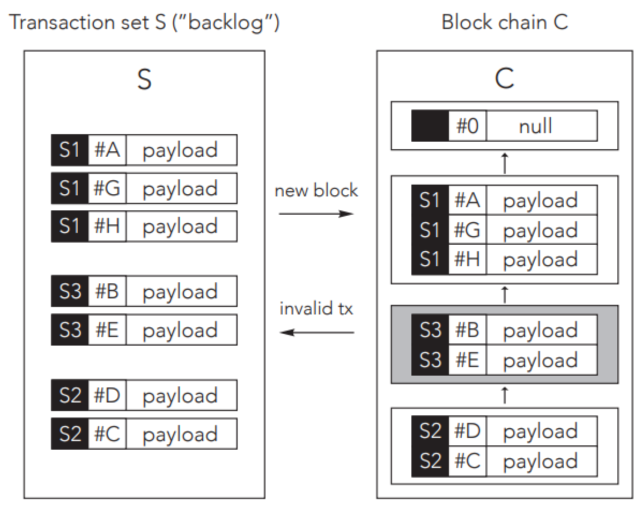


Figure 3: Architecture of BigchainDB system. There are two big data distributed databases: a Transaction Set **S** (left) to take in and assign incoming transactions, and a Blockchain **C** (right) holding ordered transactions that are "etched into stone". The signing nodes running the BigchainDB Consensus Algorithm update **S**, **C**, and the transactions (txs) between them.

Figure 1.3: From Figure 3 of 'BigchainDB: A Scalable Blockchain Database' [25]

1.2.4 ChainifyDB

ChainifyDB [9, 34, 33] introduces the Whatever-LedgerConsensus (WLC) processing model, which helps establish a permissioned blockchain layer on top of the heterogeneous database systems. Figure 1.4 illustrates the execution workflow of ChainifyDB (referred from Figure 4 of [34]). The order-subphase and execute-subphase define the two subphases of the W-phase. The transactions batch gets globally ordered and grouped together within a block in the order-subphase, while the execution of the authentic block transactions in the local database gets handled in the execute-subphase. Further, a consensus gets achieved through a lightweight voting mechanism by all the participating organizations in the ledger consensus phase. Each organization verifies its LedgerBlockHash with the consenting LedgerBlockHash (having occurrence of at least predefined constant c times), and if it matches, the organization commits and appends the block to the local ledger. Else, the organization enters a recovery phase. ChainifyDB uses an 'Optimized Partial Replay from a (Logical) Snapshot' recovery mechanism.

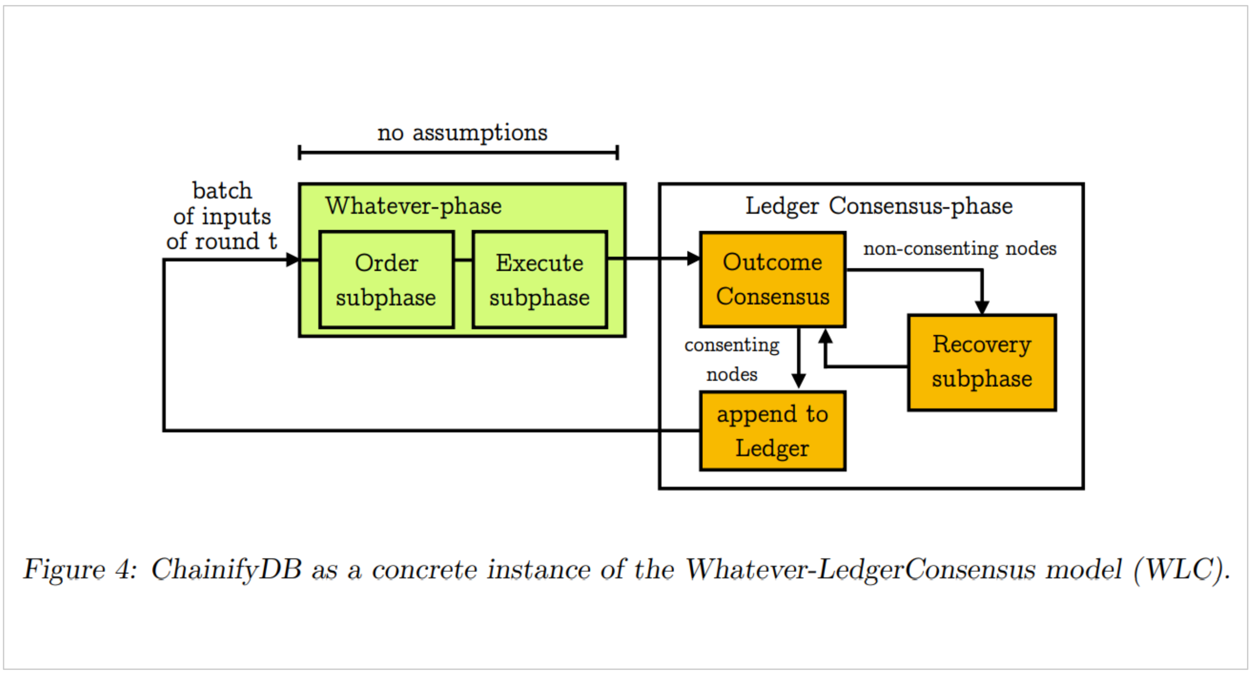


Figure 4: ChainifyDB as a concrete instance of the Whatever-LedgerConsensus model (WLC).

Figure 1.4: From Figure 4 of ‘ChainifyDB: How to Blockchainify...’ [34]

1.2.5 ForkBase

ForkBase [36] deals with designing and developing a storage engine that focuses on immense performance and lessens expansion efforts for blockchain and forkable applications. With duplicate entry discovery and aid for effective queries, ForkBase is intended to be space-efficient. The fork semantics and multi-version tamper-evident data types possessed by ForkBase promote blockchain framework creation and application development. The data types retained helps form complex blockchain designs with low development costs and eliminates integrity concerns. Figure 1.5 (referred from Figure 7 of [36]), shows the architecture involved in ForkBase with major components as master, dispatcher, servlet, and chunk storage. Master maintains information generated during run time, and the dispatcher receives and sends the request to the particular servlet. The access controller sub-module in a servlet before execution confirms approval, and branch heads get saved in the branch table. With concealed internal data design, the object manipulations get supervised by the object manager. The data chunks are obtained via chunk storage and are available to remote servlets.

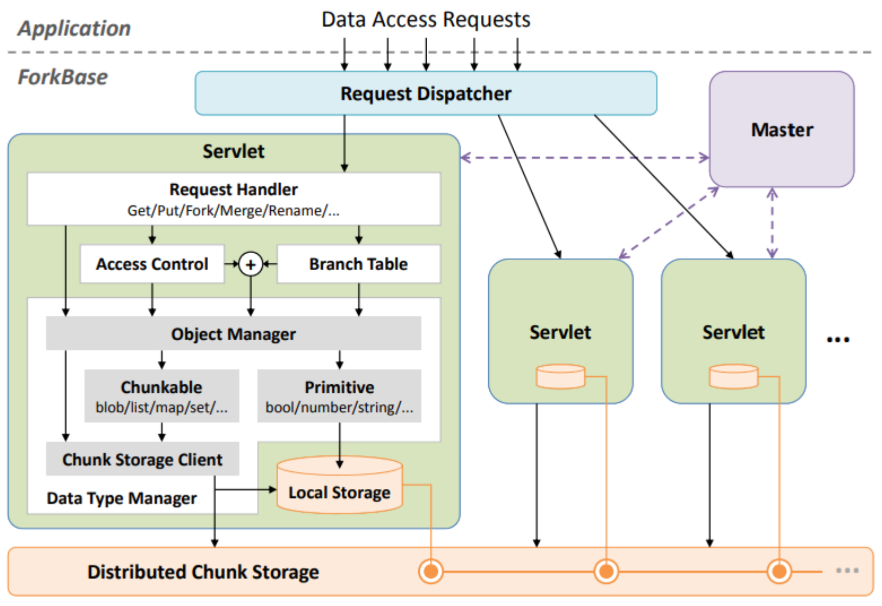


Figure 7: Architecture of a *ForkBase* cluster.

Figure 1.5: From Figure 7 of 'ForkBase: An Efficient Storage Engine for Blockchain...' [36]

1.3 Credereum

Credereum [19] is a public-domain software that layers blockchain semantics on native PostgreSQL. Specifically, Credereum is intended for delivering cryptographically verifiable consensus and provenance in a permissioned centralized setting, using an Ethereum smart contract as an immutable trusted storage repository. Credereum possesses a wide range of features, ranging from Merkle tree formation to the usage of trusted storage, that helps provide resistance to malevolent actions and makes the system concrete. Immutability, verification, authentication, and non-repudiation are a set of blockchain properties that Credereum retains. Our motivation for choosing Credereum is that it is amongst a few public-domain permissioned blockchain implementations on a relational database platform.

However, Credereum having enriching features and blockchain properties is currently centralized, unlike most systems having a decentralized environment. Despite being centralized, with the involvement of trusted storage and retention of blockchain semantics, it guarantees the users the data values, which can be easily verified. A system with a decentralized environment would have performed such activities via consensus.

In Credereum, for each transaction a Merkle tree, namely, *Transaction Merkle Tree* (TMT)

BMT	Block Merkle Tree
TMT	Transaction Merkle Tree
RHV	Root Hash Value
DS	Digital Signature
PK	Public Key
MHV	MODIFICATION-HASH-VALUE

Table 1.1: Abbreviations used

gets created. Further, for each node of TMT, a hash is computed. The hash of the root node of the Merkle tree is referred to as *Root Hash Value* (RHV). The TMT RHV summarizes the modifications made by the transaction. Additionally, in Credereum, at block-level, a Merkle tree is created, referred to as *Block Merkle Tree* (BMT). The BMT RHV computed summarizes the entire modification made by the transactions within a block. Further, Credereum supports provenance, by which the client can check for the modification history of a row/set-of-rows from the initial block 0 (known as the *Genesis* block).

Table 1.1 list the set of abbreviations used in this document. Now, before moving ahead, we define the following terms:

1. `key`: In Credereum, the nodes of a Merkle tree (TMT/BMT) are represented using bit-strings. These bit-strings associated with a node are referred to as `key`/`node-key`. Here, `key` and `node-key` are synonymous and used interchangeably throughout this document.

2. `node-value`: The leaf nodes of TMT and BMT hold data values associated with the row of a relation. These data values related to the node of the Merkle tree are represented as `node-value`. Internal nodes of TMT and BMT, doesn't have any `node-value`.

3. `node-hash`: With Merkle tree (TMT/BMT) creation, the hash of each node is computed. The hash value of the node is referred to as `node-hash`. Further, part 3 of section 2.2, defines the `node-hash` calculation methodology.

4. `transaction-hash`: For each transaction in Credereum, a transaction-level hash is computed, which considers the associated TMT RHV, previous block BMT RHV, the public key, and the digital signature of the client. This transaction-level hash is represented as `transaction-hash`.

5. `block-hash`: For each block in Credereum, a block-level hash is computed, which considers the associated BMT RHV, previous block `block-hash` and all associated transaction's `transaction-hash`. This block-level hash is represented as `block-hash`. Further, the

accounting of the previous block `block-hash` in the current block `block-hash` helps establish the ledger chain. The `block-hash` computed gets archived to the trusted storage.

Now, Credereum running on a centralized environment does possess certain limitations (section 2.8), but despite that, as mentioned, it retains the following set of blockchain properties:

1. **Immutability**: In Credereum, at periodic intervals, a block process arrives, forms a block from the committed transactions set within the block period, and computes `block-hash`. The computed `block-hash` gets stored in the trusted storage, which cannot be modified once fed. Credereum provides an immutability property, considering that even a single bit modification within a block, at transaction-level or block-level, will alter the `block-hash` and the newly obtained `block-hash` will differ from the one stored in the immutable trusted storage.

2. **Verification**: In Credereum, for every transaction fired by the client, proof for the old and new data values of the modified relation rows are shown to the client. The client verifies the evidence generated by the server and checks whether it is valid or invalid. The client can also check the modification history for a queried row from the *Genesis* block using provenance. Credereum thus retains verification property.

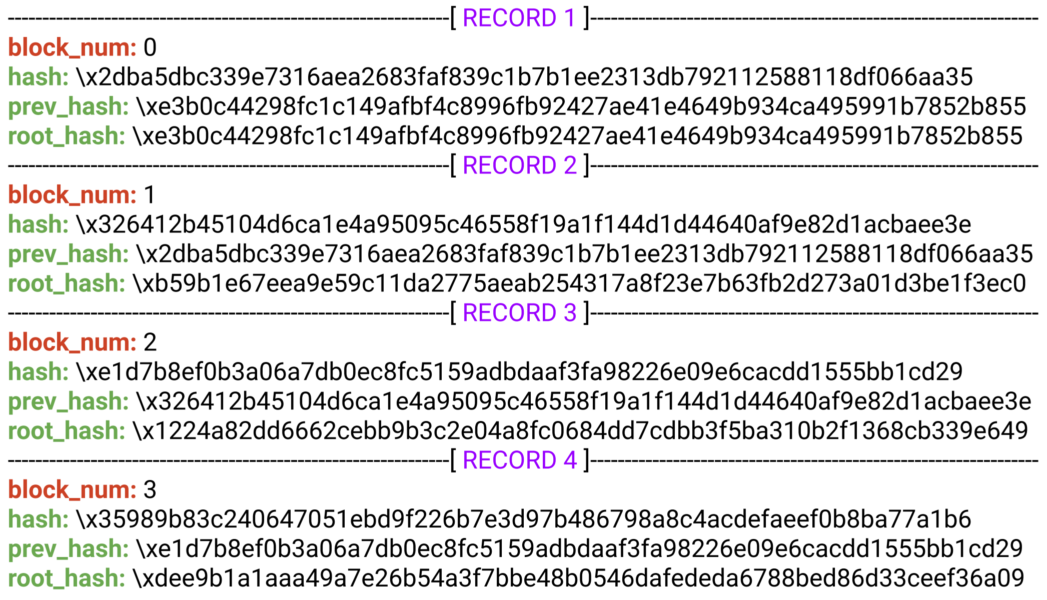
3. **Authentication**: The clients in Credereum, after verification of proofs and modifications, need to digitally sign the transaction and submit it to the server. The digital signature gets verified by the server, and if found valid, the server commits the transaction. Hence, Credereum retains an authentication property.

4. **Non-repudiation**: In Credereum, since each client submits the digital signature to the server after verifying the modifications, the client can never make false claims or deny the authorship of the changes being made by him. Thus, the non-repudiation property is maintained by Credereum.

Presently, many blockchain systems, along with Credereum, uses SHA-256 cryptographic hash function [35, 16], which retains the following properties:

1. SHA-256 is deterministic, which means for a given message M, the same hash H gets always generated.

2. SHA-256 is irreversible, which means from a given hash H, the message M cannot be derived.



```
-------------------------------------------------------------[ RECORD 1 ]-------------------------------------------------------------
block_num: 0
hash: \x2dba5dbc339e7316aea2683faf839c1b7b1ee2313db792112588118df066aa35
prev_hash: \xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
root_hash: \xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
-------------------------------------------------------------[ RECORD 2 ]-------------------------------------------------------------
block_num: 1
hash: \x326412b45104d6ca1e4a95095c46558f19a1f144d1d44640af9e82d1acbaee3e
prev_hash: \x2dba5dbc339e7316aea2683faf839c1b7b1ee2313db792112588118df066aa35
root_hash: \xb59b1e67eea9e59c11da2775aeab254317a8f23e7b63fb2d273a01d3be1f3ec0
-------------------------------------------------------------[ RECORD 3 ]-------------------------------------------------------------
block_num: 2
hash: \xe1d7b8ef0b3a06a7db0ec8fc5159adbdaaf3fa98226e09e6cacdd1555bb1cd29
prev_hash: \x326412b45104d6ca1e4a95095c46558f19a1f144d1d44640af9e82d1acbaee3e
root_hash: \x1224a82dd6662cebb9b3c2e04a8fc0684dd7cdbb3f5ba310b2f1368cb339e649
-------------------------------------------------------------[ RECORD 4 ]-------------------------------------------------------------
block_num: 3
hash: \x35989b83c240647051ebd9f226b7e3d97b486798a8c4acdefaeef0b8ba77a1b6
prev_hash: \xe1d7b8ef0b3a06a7db0ec8fc5159adbdaaf3fa98226e09e6cacdd1555bb1cd29
root_hash: \xdee9b1a1aaa49a7e26b54a3f7bbe48b0546dafededa6788bed86d33ceef36a09
```

Figure 1.6: Sample of CREDEREUM_BLOCK relation

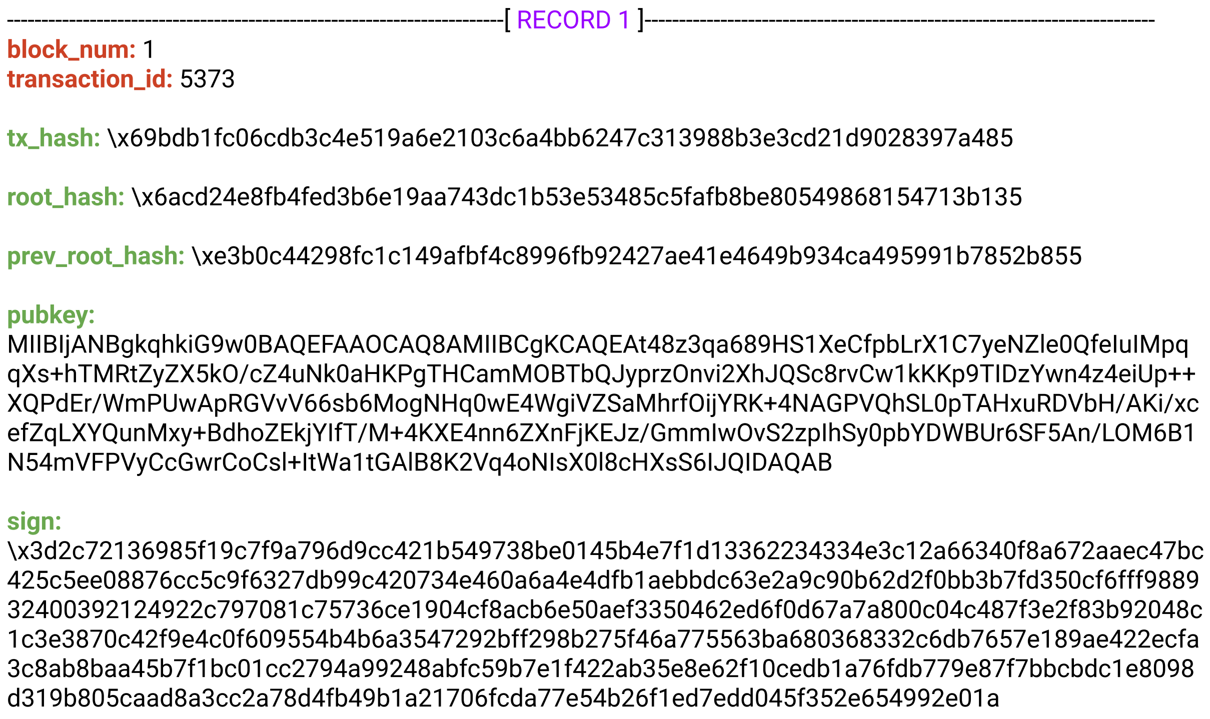
3. SHA-256 is collision-resistant, which means for messages M1 and M2, the respective hashes H1 and H2 have negligible probability for being the same.

4. SHA-256 is very sensitive, which means even a single bit change will modify the entire hash, and the newly obtained hash remains uncorrelated to the previous old hash.

5. SHA-256 is quick, which means the computational time is low and hence is immediately calculated.

Relation	Attributes
CREDEREUM_BLOCK	block_num, hash, prev_hash, root_hash
CREDEREUM_TX_LOG	block_num, transaction_id, tx_hash, root_hash, prev_root_hash, pubkey, sign
CREDEREUM_MERKLIX	key, block_num, transaction_id, children, leaf, hash, value

Table 1.2: Credereum - Relation and attributes



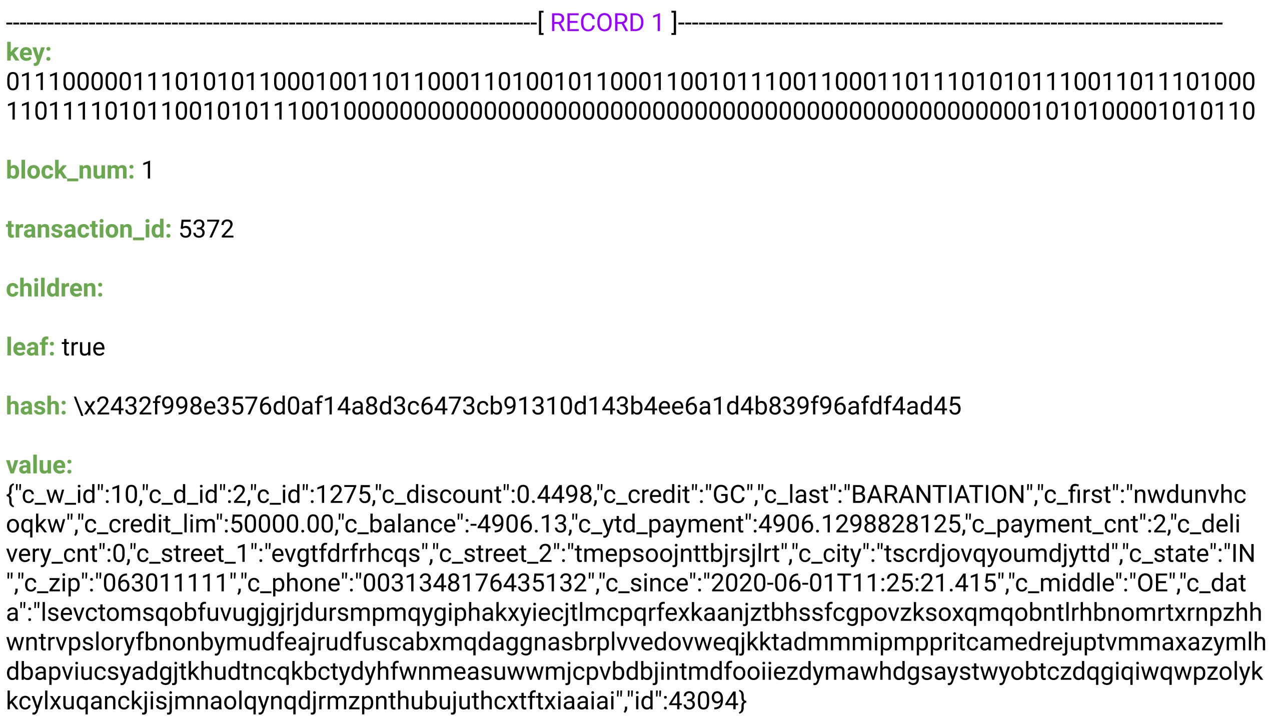
```
---------------------------------------------------------------------[ RECORD 1 ]---------------------------------------------------------------------
block_num: 1
transaction_id: 5373

tx_hash: \x69bdb1fc06cdb3c4e519a6e2103c6a4bb6247c313988b3e3cd21d9028397a485

root_hash: \x6acd24e8fb4fed3b6e19aa743dc1b53e53485c5fafb8be80549868154713b135

prev_root_hash: \xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

pubkey:
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAt48z3qa689HS1XeCfpbLrX1C7yeNZle0QfeIuIMpq
qXs+hTMRtZyZX5kO/cZ4uNk0aHKPgTHCamMOBTbQJyprzOnvi2XhJQSc8rvCw1kKKp9TIDzYwn4z4eiUp++
XQPdEr/WmPUwApRGVvV66sb6MogNHq0wE4WgiVZSaMhrfOijYRK+4NAGPVQhSL0pTAHxuRDVbH/AKi/xc
efZqLXYQunMxy+BdhoZEkjYIfT/M+4KXE4nn6ZXnFjKEJz/GmmIwOvS2zpIhSy0pbYDWBUr6SF5An/LOM6B1
N54mVFPVyCcGwrCoCsl+ItWa1tGAlB8K2Vq4oNIsX0l8cHXsS6IJQIDAQAB

sign:
\x3d2c72136985f19c7f9a796d9cc421b549738be0145b4e7f1d13362234334e3c12a66340f8a672aaec47bc
425c5ee08876cc5c9f6327db99c420734e460a6a4e4dfb1aebbdc63e2a9c90b62d2f0bb3b7fd350cf6fff9889
32400392124922c797081c75736ce1904cf8acb6e50aef3350462ed6f0d67a7a800c04c487f3e2f83b92048c
1c3e3870c42f9e4c0f609554b4b6a3547292bff298b275f46a775563ba680368332c6db7657e189ae422ecfa
3c8ab8baa45b7f1bc01cc2794a99248abfc59b7e1f422ab35e8e62f10cedb1a76fdb779e87f7bbcbdc1e8098
d319b805caad8a3cc2a78d4fb49b1a21706fcda77e54b26f1ed7edd045f352e654992e01a
```

Figure 1.7: Sample of CREDEREUM_TX_LOG relation

Further, the Credereum keeps the blockchain records in three relations, shown in Table 1.2. Exploring the details stored in these relations, from Table 1.2, we have:

1. The CREDEREUM_BLOCK table stores the per block information whenever a block gets created. A snapshot of the details present in the CREDEREUM_BLOCK table is shown in Figure 1.6. The attributes present in CREDEREUM_BLOCK relation include block_num, hash, prev_hash and root_hash, with block_num as primary key. Here, the current block's block number is stored in block_num, the current block `block-hash` is stored in hash, the previous block `block-hash` is stored in prev_hash and current block BMT RHV is stored in root_hash.

2. The CREDEREUM_TX_LOG table stores the per transaction information whenever a transaction is committed. Figure 1.7 shows the sample details present in the CREDEREUM_TX_LOG table. The attributes present in CREDEREUM_TX_LOG relation include block_num, transaction_id, tx_hash, root_hash, prev_root_hash, pubkey and sign. Here, block_num and transaction_id pair forms the primary key. Now, the current block's block number is stored in block_num, the current transaction's transaction-id is stored in transaction_id, the current transaction `transaction-hash` is stored in tx_hash, the current transaction TMT RHV



```
-------------------------------------------------------------------------[ RECORD 1 ]-------------------------------------------------------------------------
key:
01110000011101010110001001101100011010010110001100101110011000110111010101110011011101000
1101111010110010101110010000000000000000000000000000000000000000000000001010100001010110

block_num: 1

transaction_id: 5372

children:

leaf: true

hash: \x2432f998e3576d0af14a8d3c6473cb91310d143b4ee6a1d4b839f96afdf4ad45

value:
{"c_w_id":10,"c_d_id":2,"c_id":1275,"c_discount":0.4498,"c_credit":"GC","c_last":"BARANTIATION","c_first":"nwdunvhc
oqkw","c_credit_lim":50000.00,"c_balance":-4906.13,"c_ytd_payment":4906.1298828125,"c_payment_cnt":2,"c_deli
very_cnt":0,"c_street_1":"evgtfdrfrhcqs","c_street_2":"tmepsoojnttbjrsjlrt","c_city":"tscrdjovqyoumdjyttd","c_state":"IN
","c_zip":"063011111","c_phone":"0031348176435132","c_since":"2020-06-01T11:25:21.415","c_middle":"OE","c_dat
a":"lsevctomsqobfuvugjgjrjdursmpmqygiphakxyiecjtlmcpqrfexkaanjztbhssfcgpovzksoxqmqobntlrhbnomrtxrnpzhh
wntrvpsloryfbnonbymudfeajrudfuscabxmqdaggnasbrplvvedovweqjkktadmmmipmppritcamedrejuptvmmaxazymlh
dbapviucsyadgjtkhudtncqkbctydyhfwnmeasuwwmjcpvbdbjintmdfooiiezdymawhdgsaystwyobtczdqgiqiwqwpzolyk
kcylxuqanckjisjmnaolqynqdjrmzpnthubujuthcxtftxiaaiai","id":43094}
```

Figure 1.8: Sample of CREDEREUM_MERKLIX relation

is stored in root_hash, the previous block BMT RHV is stored in prev_root_hash, while the public key and the digital signature submitted by the client is stored in pubkey and sign, respectively.

3. The CREDEREUM_MERKLIX table stores each of the transactions and blocks, TMT and BMT details, respectively. The sample details stored in the CREDEREUM_MERKLIX table, is shown in the Figure 1.8. The attributes present in CREDEREUM_MERKLIX table are key, block_num, transaction_id, children, leaf, hash and value. Here, the `key/node-key` of a Merkle tee node is stored in key, the current block's block number is stored in block_num, the current transaction's transaction-id is stored in transaction_id, the `node-key` of children's of a Merkle tree node is stored in children, indicates whether or not a node is a leaf node by leaf, while the `node-value` and `node-hash` associated with a Merkle tree node is stored in value and hash, respectively. For a BMT, the transaction_id is NULL. Further, from above, as mentioned, the `node-key`, `node-value` and `node-hash` associated with a Merkle tree node is kept in the CREDEREUM_MERKLIX table.

1.4 Summary

The blockchain network has miners connected in a decentralized and distributed environment through P2P links, responsible for accumulating transactions, and at intervals end, creating blocks. For every block created, a reward gets generated to the miner for the PoW. The blockchain core properties include immutability, distributed and decentralized ledger, verification, authentication, data integrity, and non-repudiation. Ethereum, and the HyperLedger, were the systems designed under the Blockchain 2.0 initiative. Also, Ethereum led to the evolution of the smart contract, an automatic executable legal document. Credereum, being a blockchain platform, has enhanced features like TMT and BMT formation, trusted storage usage, and more. Credereum application areas include educational institutions, private organizations, assets management, secured voting mechanism, healthcare management, logistics monitoring, and lots more. Despite possessing high-grade qualities, Credereum is currently centralized and carries certain traditional and blockchain-based limitations.

Chapter 2

Credereum

The chapter introduces and explains the internals and the architecture sustained by the Credereum. The internals covering `key` generation, Merkle tree formation, and Merkle hash calculation has been explained. With being a blockchain implementation, the chapter highlights and defines the blockchain properties retained by Credereum. The chapter also discusses the minimal-disclosure feature of Credereum in providing proofs and explains handling of the double-spending problem using trusted storage. Credereum drawbacks are studied and gets described in later sections.

2.1 Introduction

Blockchain is a transaction-based system that provides a mathematical promise of guaranteeing security, anonymity, and immutability to the data. The essential properties of immutability, verifiability and authenticity can be applied to relational database systems. Credereum [19, 23] is a private, centralized blockchain implementation that provides features of non-repudiation, enhanced security, and data provenance to the traditional RDBMS PostgreSQL. Importantly, pg_credereum [23] is a PostgreSQL extension being used to implement Credereum over Postgres and adds a cryptographically verifiable audit capability to the PostgreSQL databases.

Whenever a client fires a transaction to a server in a typical client-server DBMS, the server doesn't provide proof for the generated output data. Indeed for data integrity and authenticity, the client blindly relies upon the server. Even with the server's audit features, the server administrator can forge the audit data and show incorrect results. Digital signatures for authentication allow blockchain users to track down their state from the origin with the correctness guarantee. With the stated blockchain functionality, Credereum endeavors to incorporate these into the client-server relational database systems. Further, Credereum generates proof of correctness when a transaction fired by the client results in database modification. The client verifies the proof generated and checks the correctness of the modified data. In Credereum individual actions are authenticated by their respective digital signatures, and a client can trace the history of all the records without inherently trusting the server. The client, after verifying the proof and modifications, needs to sign the transaction digitally.

Credereum uses immutable trusted storage, facilitating any data fed into the trusted storage becomes impossible to modify or alter retroactively. The trusted storage provides confidence and assurance to the client for stored data non-alteration and stabilization. The pg_credereum uses an Ethereum smart contract as trusted storage. Credereum being a blockchain application, also has to overcome the familiar double-spending problem. A situation may arise where multiple forks of the database are maintained by the malevolent database administrator and return output from different forks to different users, producing fraudulent results. Credereum creates a block digest and is uploaded to the trusted storage periodically with public read access to tackle this situation.

In Credereum, the transactions fired by multiple clients to PostgreSQL processes parallelly under Read Committed (RC) isolation level. Hence, two transactions modifying distinct rows can parallelly access the lock and process in Postgres and the Credereum engine. However, if two transactions A and B, want to alter the same rows, only one transaction, say A, is granted the lock, and the other transaction B waits for the lock release. Transaction A first process in the Postgres engine, and then in the Credereum engine. Once transaction A commits/aborts, the lock gets released, and the waiting transaction B starts processing.

2.2 Credereum Internals

The following are the underlying internal logic retained within Credereum:

1. **Digitally-signed DBMS**: Whenever a client/user fires a transaction to the Credereum, a Merkle proof is generated by the server. The user verifies the modification and evidence, and if found correct, digitally signs the transaction. The server verifies the digital signature, and if found valid, commits the transaction. A commit acknowledgment further gets sent as a receipt to the client. Figure 2.1 shows multiple users signing their respective transactions and receiving an acknowledgment. With the digital signature submission, the client agrees to the current database state, resulting from its modifications.

2. **Merkle Tree Formation**: Each row of a relation in the database gets represented by a unique differentiator, say `key-id`, defined as the concatenation of relation name (varying

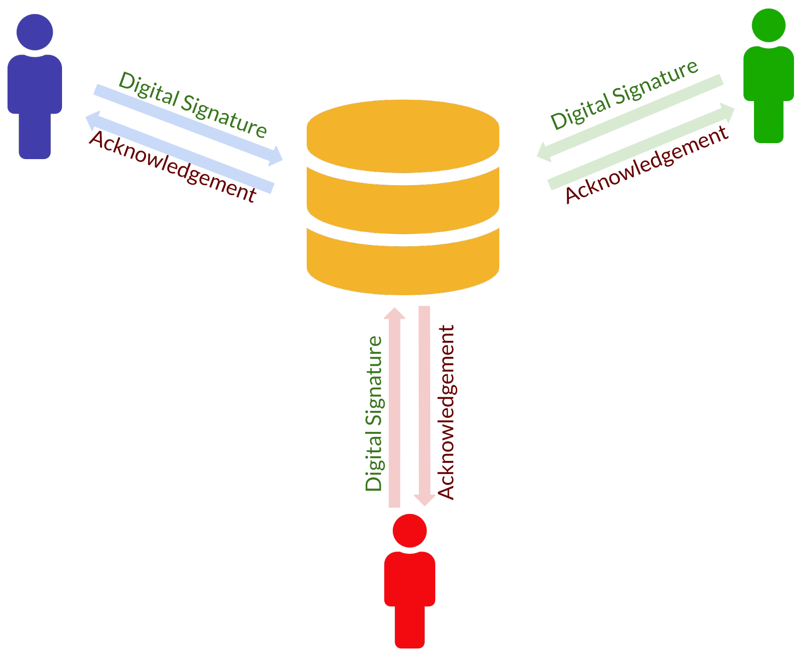


Figure 2.1: Digitally-signed DBMS

length) and 64-bit row identifier. A limitation of having at max 2^{64} rows within a relation, set as a drawback to the Credereum. A set of rows modified by the transaction fired by the client results in the generation of a group of `key-id` values. These `key-id` values are used to form a transaction/block Merkle tree based on the previous BMT structure. Figure 2.2 shows a Merkle tree developed using rows of Table 1, 2, 3, and 4. The Merkle tree leaf nodes (green nodes) represent the modified rows. Therefore, in general, for a leaf node of the Merkle tree, the `key/node-key` is the `key-id`. Now, regarding Table 1, three rows are shown being changed, i.e., Row 1, Row 2, and Row 3. Let's say Row 1, Row 2, and Row 3 of Table 1 have the following `key-id` representations:

(a) Row 1: ...0000000000000000000000000000000000000000000000000000000000000011

(b) Row 2: ...0000000000000000000000000000000000000000000000000000000000001010

(c) Row 3: ...0000000000000000000000000000000000000000000000000000000000001011

For the above, the initial bits representing Table 1 being the same; only the 64-bit row identifier of each Row 1, Row 2, and Row 3 is displayed. Row 1 differentiates Row 2 and Row 3 early at the 4th right bit, hence forming a level below compared to Row 2 and

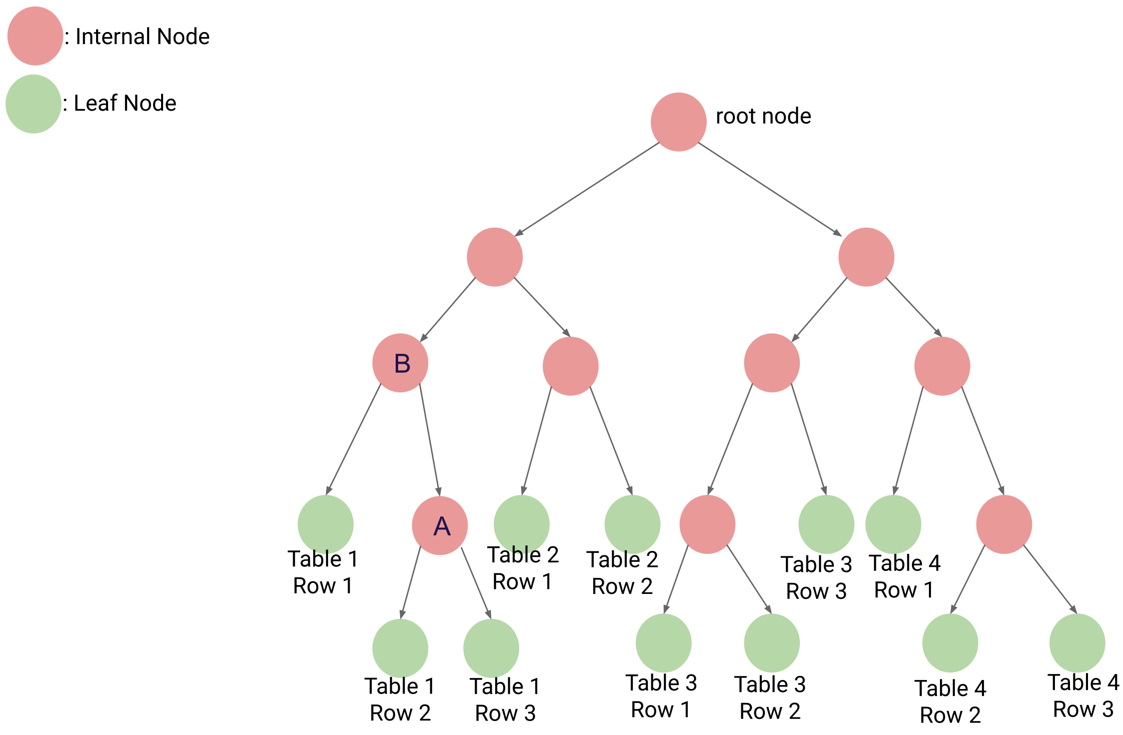


Figure 2.2: Merkle `key` presentation

Row 3 in the Merkle tree formation. Row 2 and Row 3 having more common bits get differentiated at the last right bit. Node A, being the parent node of Row 2 and Row 3, has `key` as bit-strings common to Row 2 and Row 3 `key-ids`. Node B, being the parent node of Row 1 and node A, has `key` as bit-strings common to Row 1 and node A `keys`.

3. **Merkle Tree Hash Calculation**: In Credereum, for every transaction fired, a TMT gets developed. At unit intervals, the block process begins and creates a BMT by accounting for each modification, being performed by all the committed transactions within the block period. For each TMT and BMT, the Merkle tree RHV gets calculated. The TMT RHV and BMT RHV are then used in the `transaction-hash` and `block-hash` value computation, respectively. The leaf nodes of the TMT and BMT have `node-value` data associated with them. Figure 2.3 shows a Merkle tree with leaf nodes (green nodes) and internal nodes (red nodes). Each leaf node's `node-hash` is computed by concatenating the row `key-id` (i.e., `key`) and associated `node-value` and further passing to the SHA-256 hash function. Each internal node's `node-hash` gets found by concatenating the child's `key` value and the returned `node-hash`'s of each child and forwarding to the SHA-256 hash function. With each node's hashes stored, the Merkle tree's root node `node-hash`

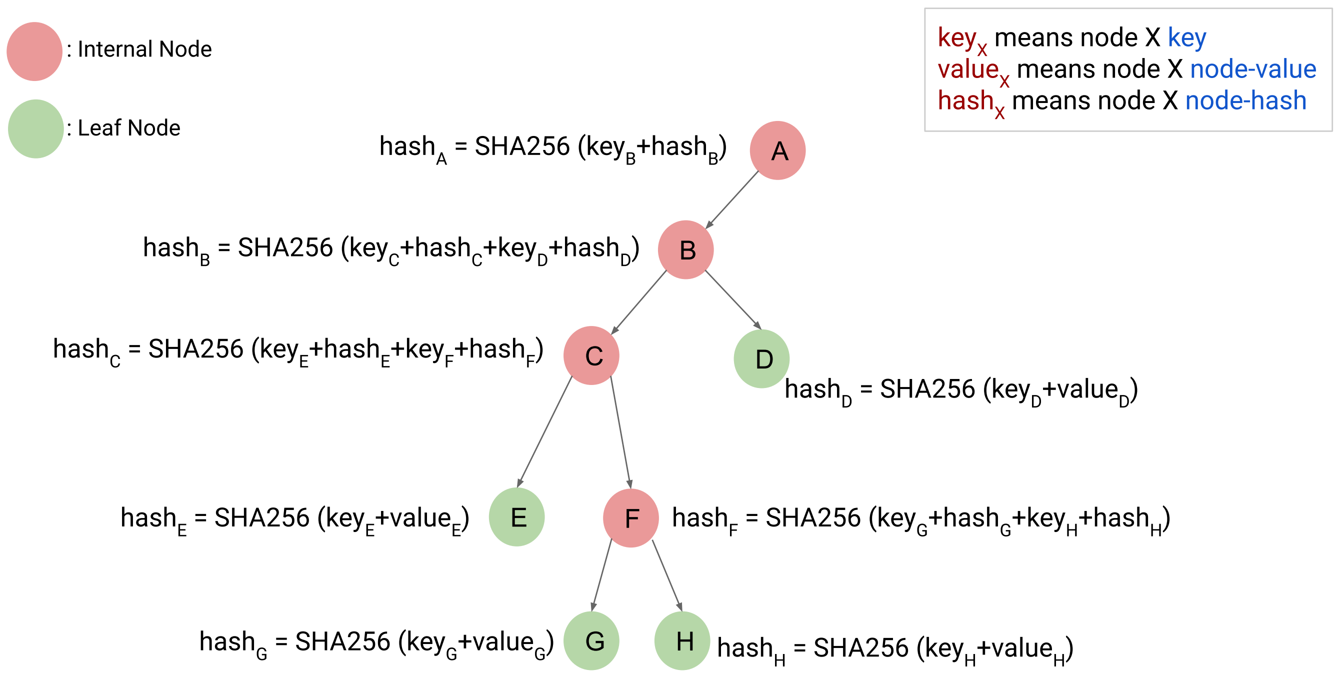

: Internal Node
: Leaf Node
key$_X$ means node X key
value$_X$ means node X node-value
hash$_X$ means node X node-hash
hash$_A$ = SHA256 (key$_B$+hash$_B$)
A
hash$_B$ = SHA256 (key$_C$+hash$_C$+key$_D$+hash$_D$)
B
hash$_C$ = SHA256 (key$_E$+hash$_E$+key$_F$+hash$_F$)
C
D
hash$_D$ = SHA256 (key$_D$+value$_D$)
hash$_E$ = SHA256 (key$_E$+value$_E$)
E
F
hash$_F$ = SHA256 (key$_G$+hash$_G$+key$_H$+hash$_H$)
hash$_G$ = SHA256 (key$_G$+value$_G$)
G
H
hash$_H$ = SHA256 (key$_H$+value$_H$)

Figure 2.3: Credereum Merkle hash computation

summarizes the complete database contents.

4. **Merkle Proof Generation**: When a user fires a transaction, it results in a set of updated/inserted/deleted rows. For each row modification, the server needs to provide proof for the old and new value of the modified row; hence comes the Merkle proof concept. For the provenance proof requests, the server needs to show all the modifications made to a particular row from the *Genesis* block along with proof. The Merkle proof provides the validity and evidence of one specific value represented by a Merkle tree's leaf node without revealing the entire Merkle tree's contents. The Merkle proof gets formed by considering the sub-tree of the Merkle tree that contains the leaf nodes to be proven (focused leaf node), the path from the leaves to the root node, and the supporting nodes (with sibling's aide, help calculate and verify parent node `node-hash`). Figure 2.4 shows a Merkle tree with focused leaf nodes (grey nodes), path nodes (purple nodes), and the supporting nodes (blue nodes). Only for the focused leaf nodes, the `key-id` (i.e., `key`) and associated `node-value` get displayed, and for the rest nodes, only their `key` and `node-hash` values are shown. Using the provided data, the user can compute the leaf node `node-hash` and recursively Merkle tree RHV using `node-hash`'s of supporting nodes

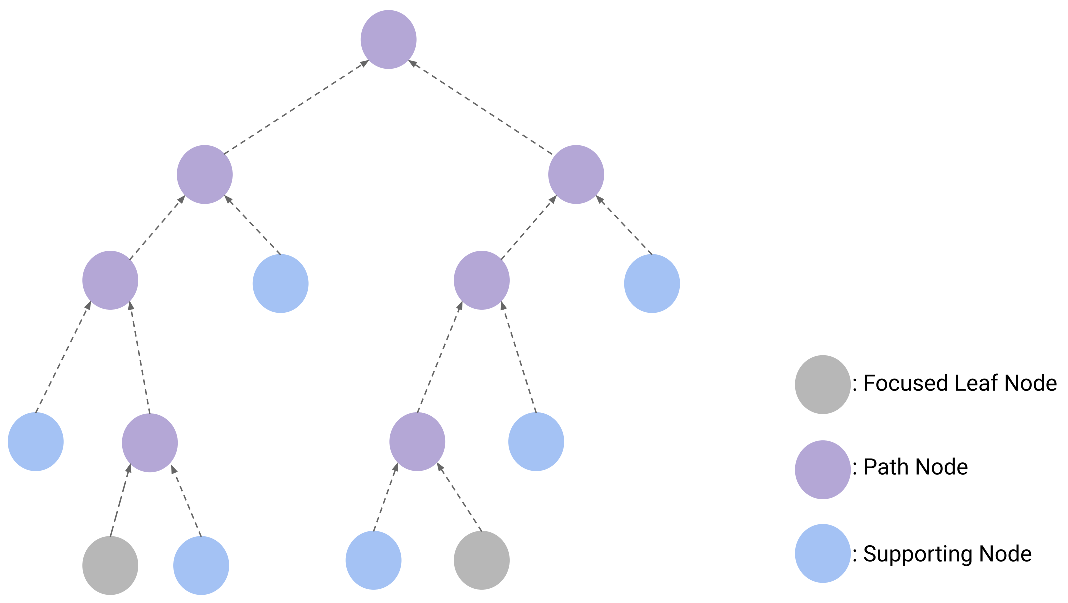


Figure 2.4: Merkle proof

and verifying with the path nodes. The Merkle tree RHV generated gets used to compute `block-hash` value, which the user can further confirm from the trusted storage. If the match occurs, then the leaf node's `node-value` shown can be trusted to be valid.

5. **Trusted Storage**: In Credereum, whenever a block forms, a `block-hash` gets calculated. This `block-hash` summarizes each information present within a block. If any fraudulent event happens within a block, then the `block-hash` value changes. In Credereum, the `block-hash` value calculated is archived to the trusted storage, which clients can read and rely upon. With being immutable, the trusted storage assures clients with any deletion/modification to the record/digest put forward to it. Due to this cause, any retroactive/malevolent database contents alteration can be easily identified, thanks to the trustworthy storage. Further, the application of trusted storage in Credereum helps solve the double-spending problem. Credereum uses Ethereum smart contract as the trusted storage to store all the `block-hash` digest values.

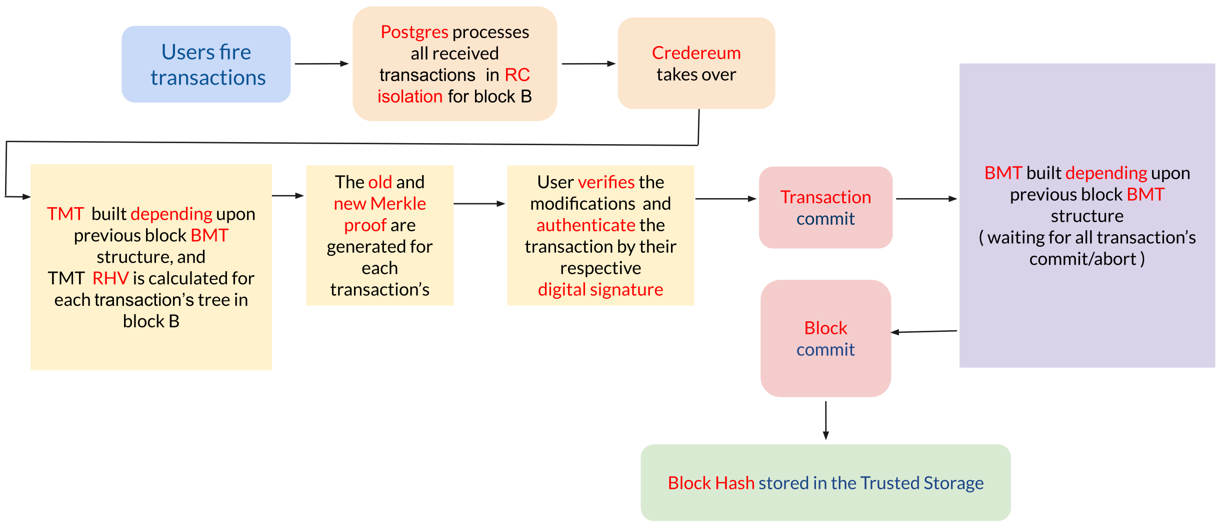

Users fire transactions
Postgres processes all received transactions in RC isolation for block B
Credereum takes over
TMT built depending upon previous block BMT structure, and TMT RHV is calculated for each transaction's tree in block B
The old and new Merkle proof are generated for each transaction's
User verifies the modifications and authenticate the transaction by their respective digital signature
Transaction commit
BMT built depending upon previous block BMT structure (waiting for all transaction's commit/abort)
Block commit
Block Hash stored in the Trusted Storage

Figure 2.5: Credereum workflow

2.3 Architecture

In Credereum, whenever a client fires a transaction X, the transaction first processes in the Postgres engine with the Read Committed isolation level. Figure 2.5 shows that after Postgres processing, the control gets handed over to the Credereum engine. In the Credereum engine, the transaction X process first builds its TMT depending upon the previous block BMT structure, and using recursion, the TMT RHV gets calculated.

For example, using Figure 2.6, we display the TMT formation process to make the notions concrete. Dissecting transaction X fired by the client in the current block (B), the transaction constructs a TMT (Figure 2.6, part 'b') referring to the existing BMT structure of the previous block (Figure 2.6, part 'a'). The TMT leaf nodes [D, I, J] represent rows updated [D, I] and inserted [J] by transaction X in the database (green nodes), whereas the remaining leaves, called 'hanging nodes' (yellow nodes), are virtual and shows the connections to the unmodified parts of the database via linkages to the previous BMT. The current block TMT hanging nodes, i.e., node 1, node 2, node 3, node 4, node 5, and node 6, are the nodes of the previous block BMT, i.e., node 1, node 'C', node 'E', node 4, node 'H', and node 6, respectively. Additionally,

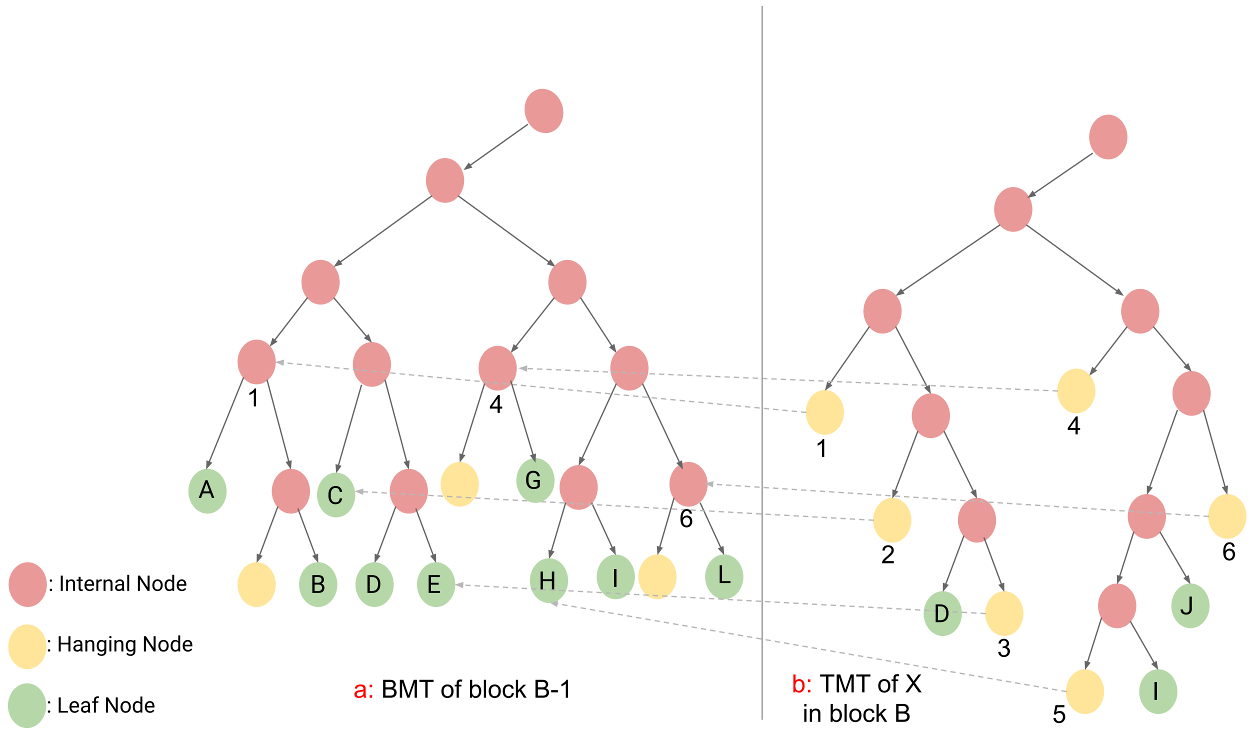


Figure 2.6: Credereum Merkle tree

from Figure 2.6, the dashed arrow helps explain the previous block BMT node, portrayed as a hanging node in TMT. The TMT's leaf nodes, along with the hanging nodes, are incorporated in the RHV calculation of the TMT.

For modifications made by transaction X, an old Merkle proof and new Merkle proof get generated for the client. Figure 2.7 shows the old Merkle proof (left tree) and new Merkle proof (right tree), say generated by the server for transaction X. The old Merkle proof presents the data values of a row before modification with evidence and is verifiable. The new Merkle proof shows the latest data values after changes made by transaction X. The left tree contains modified leaf nodes (grey nodes), path nodes from leaf to root nodes (purple nodes), and supporting nodes (blue nodes). Similarly, the right tree contains new altered leaf nodes (green nodes), altered internal nodes (red nodes), and supporting nodes (blue nodes). Conceptually, the supporting nodes comprise internal nodes, leaf nodes, and the hanging nodes of a Merkle tree. Now, except for modified leaf nodes (grey nodes) and new altered leaf nodes (green nodes), for each node, only the `node-key`, and `node-hash` values are shown, which helps compute the RHV. While for the modified leaf nodes (grey nodes) and new altered leaf nodes (green nodes), `key-id` (i.e., `node-key`) and associated `node-value` are shown. Hence, the user can check

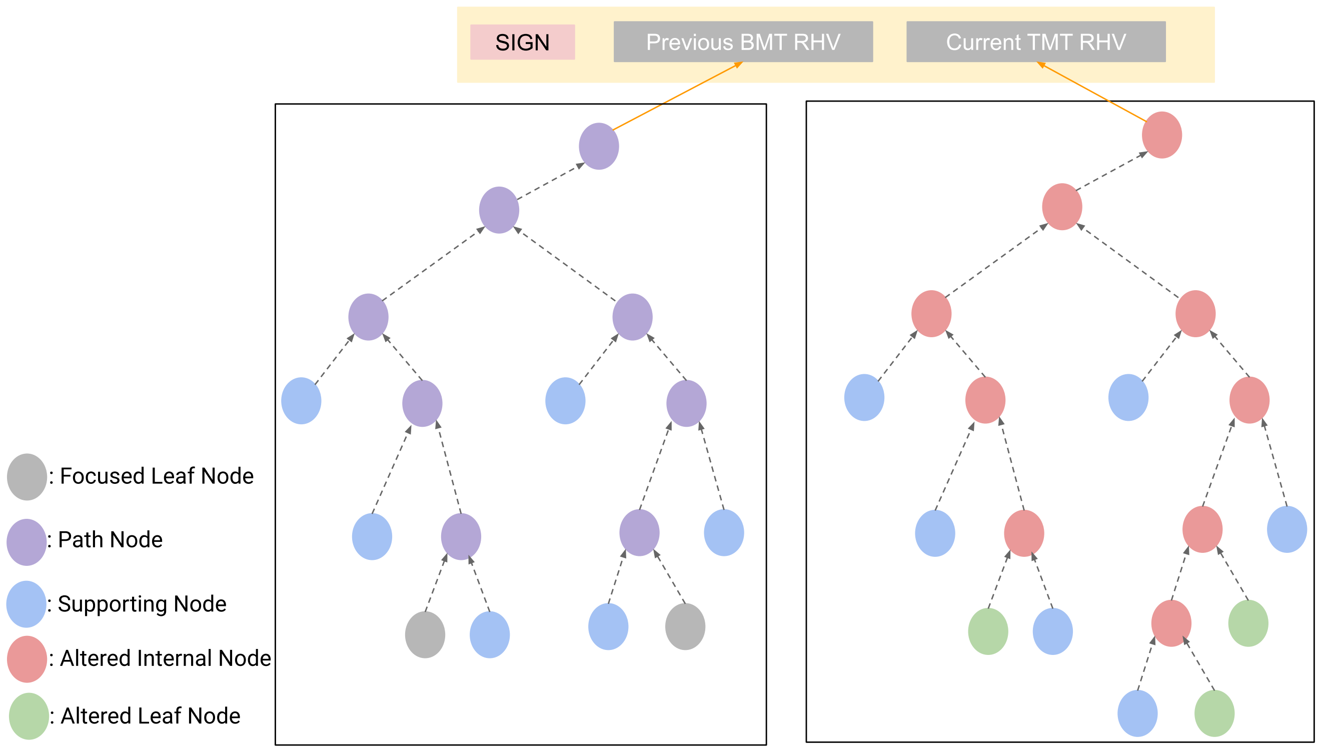


Figure 2.7: Credereum Merkle proof generation

whether the transaction fired by him resulted in changes from the modified leaf node to the respective new altered leaf node.

Further, the user can compute the `block-hash` value by calculating the Merkle tree RHV of old Merkle proof and compare it with the trusted storage. The client, after modification's verification, authenticates the transaction by signing with its digital signature. The digital signature is made by concatenating the old Merkle proof (left tree) RHV and new Merkle proof (right tree) RHV and signing it using the private key. The server verifies the signature, and if valid, computes the `transaction-hash` and commits the transaction, else abort.

In Figure 2.8, we can see that the `transaction-hash` is computed concatenating the previous block BMT RHV, current TMT RHV, public key, and the digital signature. At unit intervals, the block formation process waits for all the transactions within the block period to commit. The block process then begins collecting all the rows modified by all the transactions within the block. A BMT is developed, depending upon the previous block BMT structure, and the current block BMT RHV gets calculated. Further, as presented in the Figure 2.8, the `block-hash` value is computed considering the previous block `block-hash` value, all constituent transaction's `transaction-hash` value, and the current block BMT RHV. The

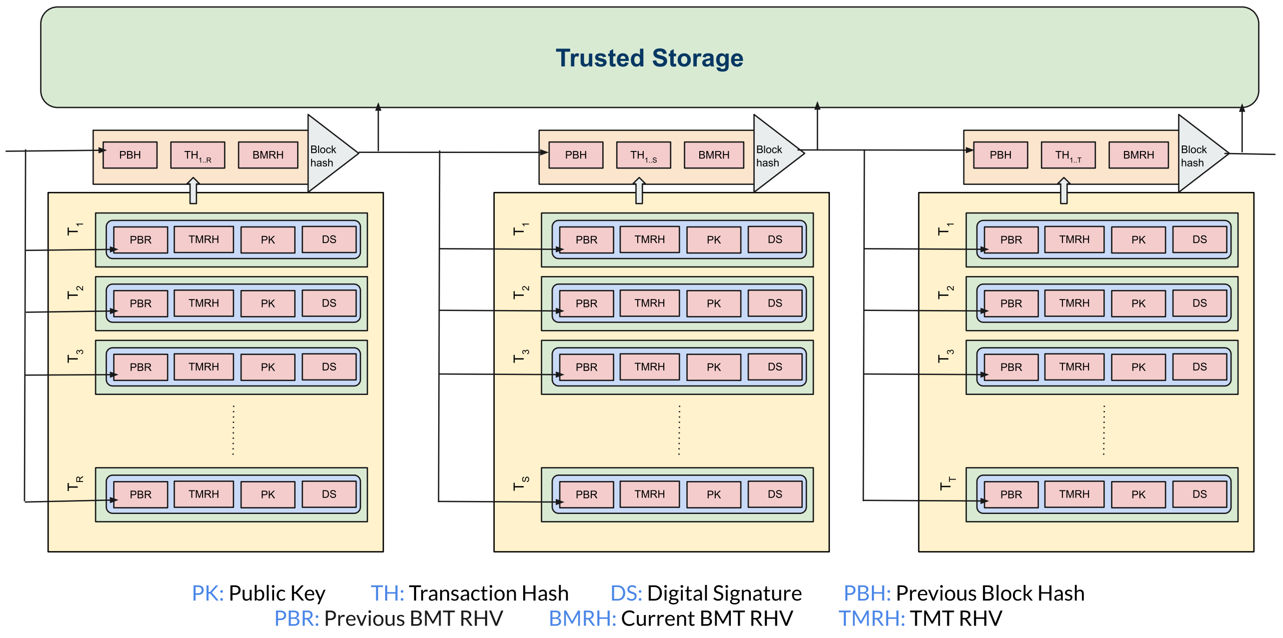


Figure 2.8: Credereum block creation

inclusion of the previous block `block-hash` value in the current block `block-hash` value helps establish the ledger chain. The current `block-hash` value calculated gets archived to the trusted storage.

2.4 Blockchain Properties

The following are a set of blockchain properties owned by Credereum:

1. **Immutability**: In Credereum, whenever a fraudulent person succeeds in modifying transaction T details of block B, then the TMT RHV of transaction T will change. The transaction T TMT RHV is accounted in transaction T `transaction-hash` value calculation. Hence, with any transaction T changes, the TMT RHV and transaction T `transaction-hash` value gets changed. Further `transaction-hash` value of transaction T is accounted for block B `block-hash` calculation. Thus, with any changes in transaction T `transaction-hash` value, the block B `block-hash` value will vary. Now, the block B+1 `block-hash` value accounts for the previous block B `block-hash` value. Hence, with changes in block B `block-hash` value, the `block-hash` value of block B+1

will change. Similarly, all the forward blocks' `block-hash` value needs to be re-calculated to validate the ledger. Let's say the fraudulent person succeeds in validating the chain by re-calculating the `block-hash` digest of all the forward blocks. However, the newly calculated `block-hash` digest of block B and all the successive blocks will differ from the stored original `block-hash` digest at the immutable trusted storage. Hence, any retroactive modification of the database contents by a fraudulent person can be easily detected. Thus, Credereum ensures and possesses the immutability property.

2. **Verification**: In Credereum, whenever a client fires a transaction to the Postgres engine, an old data value proof and the associated new data value proof is being shown to the client. The previous block BMT is acknowledged for the old data value proof, and for the new data value proof, the current transaction TMT gets displayed. Credereum also facilitates the provenance query, through which the client can check the modifications of a row/set-of-rows from the *Genesis* block. The client verifies the above proofs and checks whether the fired transaction's transmutation is correct or not. If the change made is valid, the client validates the transaction by signing with his digital signature. Thus, Credereum maintains the verification property.

3. **Authentication**: The transaction fired by a client in Credereum needs to be authenticated, using the client's digital signature to commit the transaction. After verification of old and new data proof, the client digitally signs the value obtained by concatenating old data proof RHV and the new data proof RHV, using its private key, and submits it to the server. The server verifies the digital signature, and if valid, commits the transaction; else, it aborts the transaction. Hence, Credereum holds the authentication property.

4. **Non-repudiation**: In Credereum, a client submits his digital signature to authenticate the transaction. Upon successfully verifying the signature, the submitted digital signature is accounted for calculating the `transaction-hash` value of transaction T and is stored. Finally, the transaction commits. Hence, a situation can never occur where a client can deny the authorship or the validity of the transaction fired by him. Therefore, Credereum retains the non-repudiation property.

2.5 RC Isolation Level: Safe

In Read Committed (RC) mode, if multiple transactions attempt to modify the same row of a table simultaneously, only one transaction acquires the lock, and the rest transactions wait for the lock access. The waiting transactions get lock access sequentially, each after lock release.

In Read Committed mode, whenever a transaction gets processed, in Credereum, an old Merkle proof and new Merkle proof are generated to the client for verification. With the help of old data of modified rows, a client can check whether the fired transaction results in modified rows' new state. If some issue occurs, the client can always abort the transaction after verifying the old and new Merkle proof and will not sign the transaction. Hence, the control lies in the client's hand to specify whether the transaction fired by him has been correctly processed or not. If not, then the client can always abort the transaction and fire a new one. Else, the client can digitally sign the transaction and submit the digital signature to the server. The server verifies the digital signature, and if found valid, commits the transaction. Hence, the Credereum property of showing old and new data proof to the client prevents anomalies, if any, occur through the Read Committed isolation level.

2.6 Merkle Proof and Minimal Disclosure

In Credereum, whenever a client fires a transaction, a Merkle proof is generated, namely old Merkle proof and new Merkle proof, providing evidence for the old data values and the new modified data values, respectively. However, while generating the old Merkle proof, the client is only shown the details corresponding to the modifications made by his transaction. A client can't see the details of all the other committed transactions used in the Merkle tree formation or the ones stored in the database. Hence, other clients' whole data set is not shown in general while generating the old Merkle proof.

Let's say the client fires a transaction T, and on successful commit will be considered in block B's block formation. The transaction T, say it modifies rows

$$r_1, r_2, r_3, r_4, ..., r_m$$

and a subset of these rows say

$$r_1, r_2, r_3$$

were being modified recently at block B-k, where 1 $\leq$ k $\leq$ B-1. Let the complete set of rows being modified by all the transactions at block B-k be

$$r_1, r_2, r_3, r_1^{'}, r_2^{'}, r_3^{'}, r_4^{'}, ..., r_n^{'}$$

rows. Hence, while generating the old data proof for transaction T of block B, only the modified row values

$$r_1, r_2, r_3$$

of block B-k are shown to the client. The rest row values

$$r_1^{'}, r_2^{'}, r_3^{'}, r_4^{'}, ..., r_n^{'}$$

modified at block B-k is kept hidden, and if required (supporting node), only the `node-hash` value is shown. Thus, for the old Merkle proof, the output shown constitutes the `key-id` (i.e., `node-key`) and `node-value` of the leaf nodes of BMT corresponding to rows

$$r_1, r_2, r_3$$

and only the `node-key` and `node-hash` values for the path nodes (leading to the root node) and the supporting nodes. Hence, for generating the old Merkle proof, only a subset of modified row values of block B-k is shown, not the complete set. Hence minimal disclosure of other client's data is being made for generating the old Merkle proof.

Suppose the transaction T of block B fired by the client modifies `key-id`'s [A, D, F, H]. Let's say the `key-id`'s [A, D] was last modified in block B-x (k=x, x≠1), and `key-id`'s [F, H] was last modified in block B-1 (k=1). The section 'a' and section 'b' of Figure 2.9 shows the BMT of block B-x (k=x, x≠1) and BMT of block B-1 (k=1), with the entire set of modifications made within the blocks. The BMT of block B-x shows `key-ids` [A, B, C, D, E] being modified in block B-x and displays the hanging nodes. The BMT of block B-1 shows `key-ids` [F, G, H] being modified in block B-1 and displays the hanging nodes. Node 1 of BMT of block B-1 being hanging node links to node 1 of BMT of block B-x.

The transaction T of block B, since modifies rows recently being modified at block B-x and block B-1, the entire BMT entries of block B-x and block B-1 are not shown to generate the old Merkle proof. Figure 2.10 details the old Merkle proof being shown to the client for the fired transaction T at block B. Figure 2.10 shows that for only `key-ids` [A, D, F, H] the entire details (green nodes) containing associated `node-value` are shown. For the rest, internal nodes, and the supporting nodes, the `node-key` and `node-hash` gets shown.

Hence, a minimal disclosure of data values gets made in generating the old Merkle proof. To check the old Merkle proof's correctness, the user can compute the Merkle tree RHV and block B-1 `block-hash` value. The `block-hash` value calculated can be matched with the database and the trusted storage.

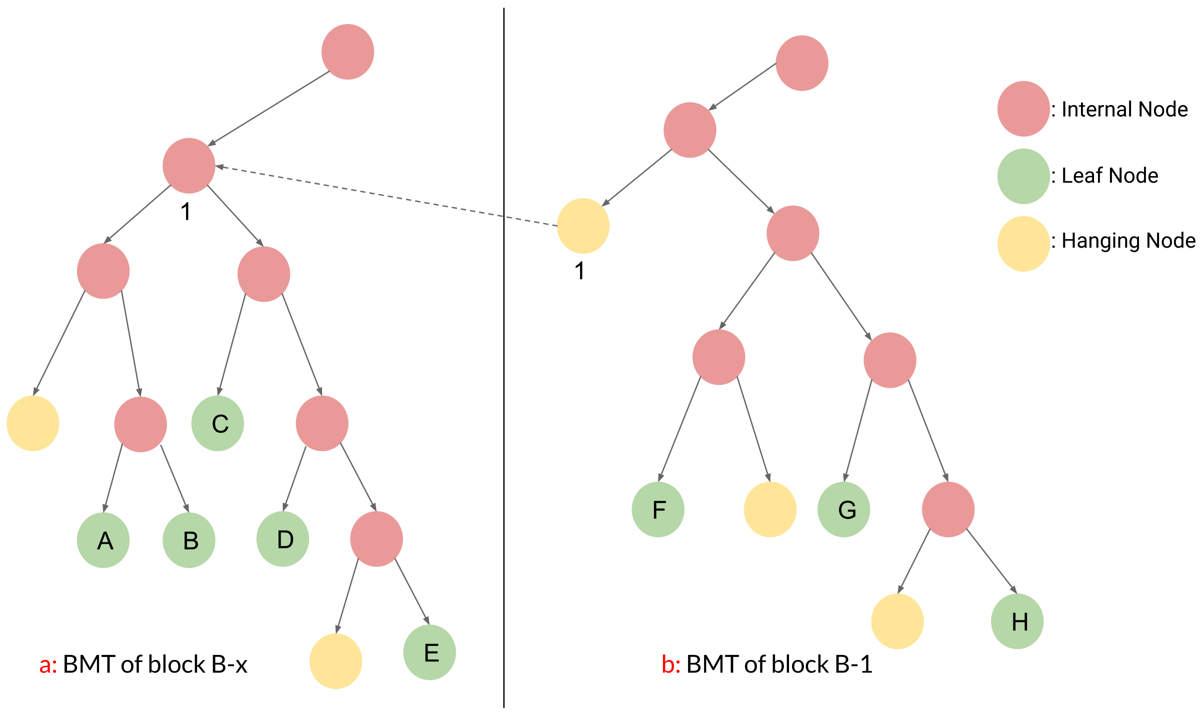


Figure 2.9: Credereum block BMT

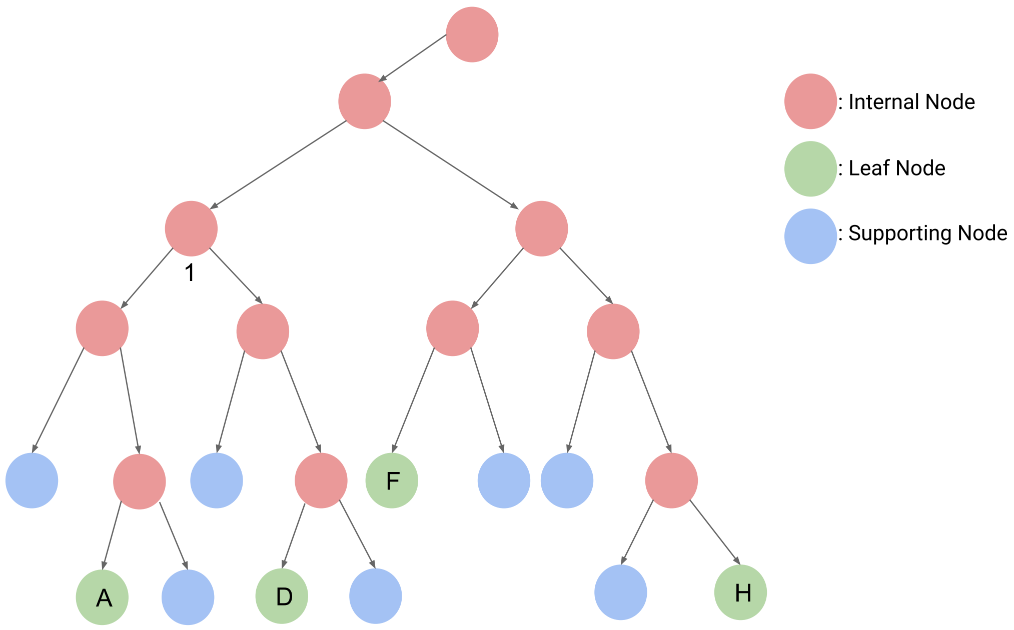


Figure 2.10: Credereum old Merkle proof

For the new Merkle proof, the TMT of transaction T gets displayed to the client. The leaf nodes of TMT of transaction T show the new altered values. The client verifies whether the transaction fired by him results in modifying old values to the current values. If the modification is found correct, the client signs a digital signature and forwards it to the server.

Considering provenance queries, let's say transaction T' of block B-k modified the rows

$$r_1, r_2, r_3, r_1^{'}, r_2^{'}, r_3^{'}, r_4^{'}, ..., r_x^{'}$$

where x≤n. Hence, while generating the provenance proof for rows

$$r_1, r_2, r_3$$

modified by transaction T of block B, only the relevant portion of TMT of transaction T' of block B-k is displayed for verification purposes. The output shown constitutes the `key-id` (i.e., `node-key`) bit-strings and `node-value` of the leaf nodes of transaction T' TMT corresponding to the rows

$$r_1, r_2, r_3$$

and, only the `node-key` bit-strings and `node-hash` value for the path nodes (leading to the root node) and supporting nodes. Hence, while generating the provenance proof, the common modified row-set respective `node-value` is revealed, and the disjoint rest row-set respective `node-value` is kept hidden. The same scenario occurs while generating BMT of block B-k for showing the provenance proof. The `key-id` (i.e., `node-key`) bit-strings and `node-value` associated are shown only for the common modified row-set (leaf nodes), while for the path nodes, and supporting nodes, `node-key` bit-strings and `node-hash` values get shown. Hence, minimal disclosure of other client's data gets made while generating provenance proof. The provenance handling by Credereum has been explained in Chapter 7.

2.7 Handling Double-spending

In Credereum, as shown in Figure 2.11, a situation may occur where withholding multiple database forks, the database server/administrator may return responses to different queries fired by the clients from different forks. For prevention from such activities, Credereum creates a cryptographic `block-hash` digest per block as a representation of the modified database. Further, the immutable trusted storage repositories the computed `block-hash` digest at peri-

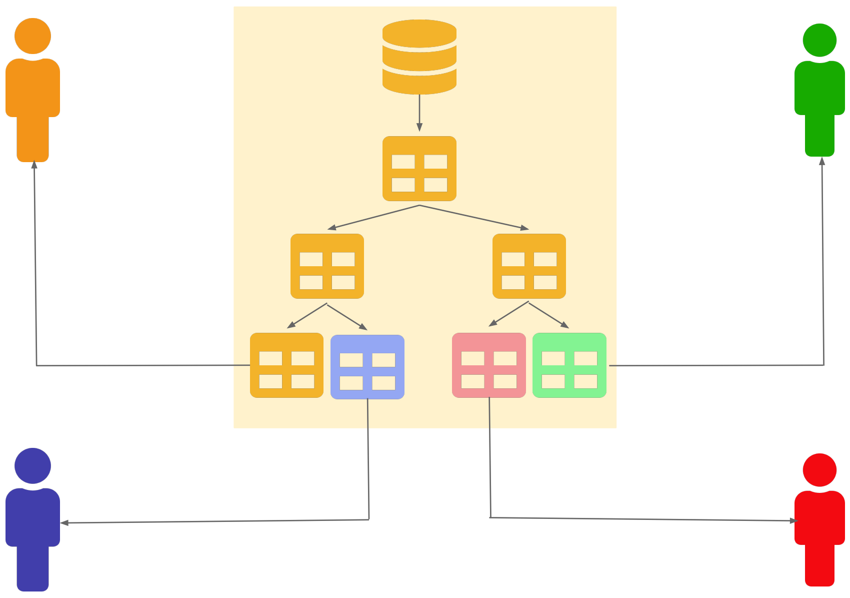


Figure 2.11: Double-spending problem

odic intervals. With feeding done to the trusted storage, the client can remain assured of the non-modification to the stored digest.

In Credereum, whenever a client fires a transaction T for block B to the Postgres engine, an old Merkle proof and new Merkle proof is generated by the server to the client, as shown in Figure 2.12. The old Merkle proof is built based upon the previous block B-1 BMT. The validity of the old Merkle proof, shown by the server, can be obtained by calculating the block B-1 `block-hash` digest using the displayed block B-1 BMT RHV. The client can compare the computed `block-hash` digest with the stored digest at the trusted storage. If a match occurs, then the old Merkle proof shown by the server to the client is valid, and hence client can further do the verification, and authentication step, as shown in the Figure 2.12. Thus, the immutable property of the trusted storage helps solve the double-spending problem in Credereum.

2.8 Drawbacks

Credereum, a permissioned blockchain platform, provides essential properties of immutability, verification, authentication, and non-repudiation. However, Credereum, being centralized, has following downsides:

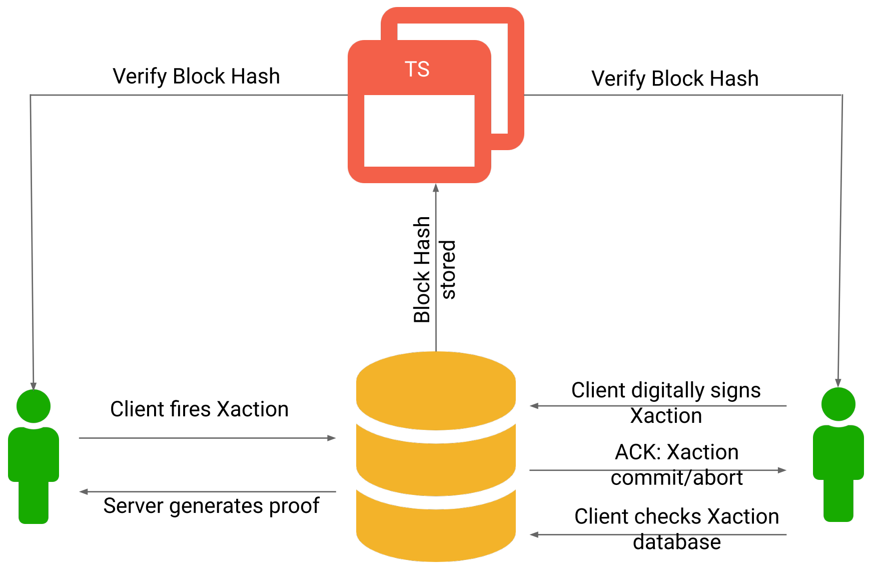


Figure 2.12: Solving double-spending problem using trusted storage

1. If the client fires a transaction in the blockchain network, the transaction gets verified by all the servers connected in a decentralized environment. The servers confirm whether the transaction is correctly processed or not and if found valid, is included in the blockchain ledger. However, Credereum currently being centralized whenever a client fires a transaction to the server, the server, after database modification, needs to prove the old values and the new values. The client has to verify the proof and check for the authentication of the old data values shown. The server generating evidence and the client-end verification can consume significant time.

2. In a blockchain network, whenever a block gets constructed, multiple servers come to a consensus on whether the block formed is valid/invalid. If the block formed is agreed by maximum, then the block is added to the ledger and is made public. However, in Credereum, whenever the server forms a block by creating the BMT and calculating BMT RHV and `block-hash` value, the BMT details are kept hidden from the clients. Hence, the client relies on the server for the correctness of the current block BMT formation. Due to this cause, the server can perform malicious activities (section 3.2.1.1) that will remain unnoticed to the clients.

3. Transparency being a key factor for the blockchain network, the transactions fired are viewable to each node present in the system. Every modification made in the blockchain network is noticeable, providing more resistance to malicious activities, and makes system concrete. Comparatively, in Credereum, such transparency is avoided. The transaction fired by client A remains unnoticed by another client. Hence, a client doesn't know what transaction modifications are involved in the current block BMT formation. Due to opaqueness in the committed transaction details in Credereum, the clients can't perform the per-block BMT verification. Hence, enabling transparency and holding consensus become necessary at per-transaction and per-block level in Credereum.

4. The decentralization of the blockchain network makes it less prone to failure and shut down. Since multiple servers are connected in a blockchain network, even if a server fails, the rest servers can handle the transactions and keep the blockchain network active. However, Credereum, being centralized, is entirely dependent on one server. If the server fails, no client will be able to process any transaction, and the system will halt.

2.9 Summary

Credereum is a blockchain-based system designed on top of the PostgreSQL database. For every transaction fired by the user, the server builds proof for each modification using Merkle trees. The client verifies the evidence and checks the modified data's correctness to confirm no data tampering was initiated, and finally authenticates the transaction. The digital signature submission signifies the client's agreement to the current database state resulting from his previous actions. Further, at periodic intervals, the block `block-hash` value is computed, which summarizes the entire modification made within the block and gets fed into the trusted storage. Also, the Credereum handles the provenance queries raised by the client and generates the provenance proof. Finally, Credereum having enriching blockchain features has certain drawbacks compared to a general blockchain network.

Chapter 3

Limitations and Proposal

The chapter explains the functionality limitation present within Credereum and also studies its performance compared to PostgreSQL. The functional insufficiency led us to explore the Credereum attack endurance, which further motivated to design SecCred. The performance comparison details the Credereum throughput and provides summarized knowledge of the methodology involved to boost the performance. In particular, ProgCred, PerfCredA, and PerfCredB seeking performance improvements are addressed.

3.1 Introduction

Credereum, possessing enriching features (section 2.2) and blockchain properties (section 2.4), had certain functional limitations (section 2.8) that are being researched and improved. With the hidden full BMT details, the client can't verify the whole BMT formed by the server, which weakens the system's resistance towards malicious activities. Despite showing the old Merkle proof, new Merkle proof, and provenance proof to the client for a queried row R, the system gets prone to fraudulent exercises. Furthermore, SecCred retaining the same architecture, and with additional functionality and variations in the TMT RHV/BMT RHV computation, has been designed to address this issue. Credereum on comparison with PostgreSQL had orders of magnitude dip in the transaction throughput. The reason behind such poor performance has been studied and explained, with a few being TMT/BMT formation, TMT RHV/BMT RHV computation, waiting for lock access, etc. The programming and algorithmic improvements were made to uplift the performance. ProgCred deal with programming modifications, moreover, PerfCredA/PerfCredB deals with the algorithmic changes.

3.2 Functional Limitation

The Credereum, despite owning enriching features, for example, TMT/BMT formation, usage of trusted storage to support immutability, has certain functional limitations that are needed to be addressed to make the system more concrete towards malicious attacks. With the hidden complete BMT details, the server can perform fraudulent actions to destroy the blockchain ledger, and the client remains unaware of it. With the growing timeline and the ledger invalid, the TMT/BMT RHV computation for subsequent blocks gets computed incorrectly, and the fraudulence disseminates ahead. Furthermore, in Credereum, the clients could know the modification history for a row/set-of-rows, using the provenance query functionality. But, certain search space optimization was required to avoid redundant information and its processing in the provenance query. Additionally, the unique indexing constraint in Credereum prevents multiple transactions within the same block from modifying a particular row, which can be further researched to be relaxed. The functionality limitations are addressed in detail in section 3.2.1 to section 3.2.3.

3.2.1 Merkle Tree Display

Credereum, for every transaction fired, generates an old Merkle proof and the new Merkle proof to the clients. The old Merkle proof shows the old row value of the modified row `key-id`'s, and the new Merkle proof displays the respective modified new row value. The client, post verifying the proof's needs to sign the transaction digitally. Credereum, at a periodic interval, forms the block B BMT (say), aggregating all the modifications made by the transactions within the block period. The BMT RHV and `block-hash` are computed and gets stored in the database and trusted storage.

In Credereum, once block B gets built, the clients cannot verify the accounted rows in the block B BMT formation. Using the provenance query, client A of block B can investigate whether or not the respective modified rows have been considered for block B BMT formation. But, since client A does not know the rows modified by the other transactions within block B, client A can easily get tricked for any malicious activity in the BMT creation. In Credereum, the client only has transaction information from the public CREDEREUM_TX_LOG table. The next section, 3.2.1.1, describes a simple scenario in Credereum, where the server performs malicious activity in BMT formation, which remains unacknowledged to the clients. Additionally, with the stated scenario, the reason behind the provenance query displaying each block detail, despite whether or no modifications to the queried row is present, has been addressed.

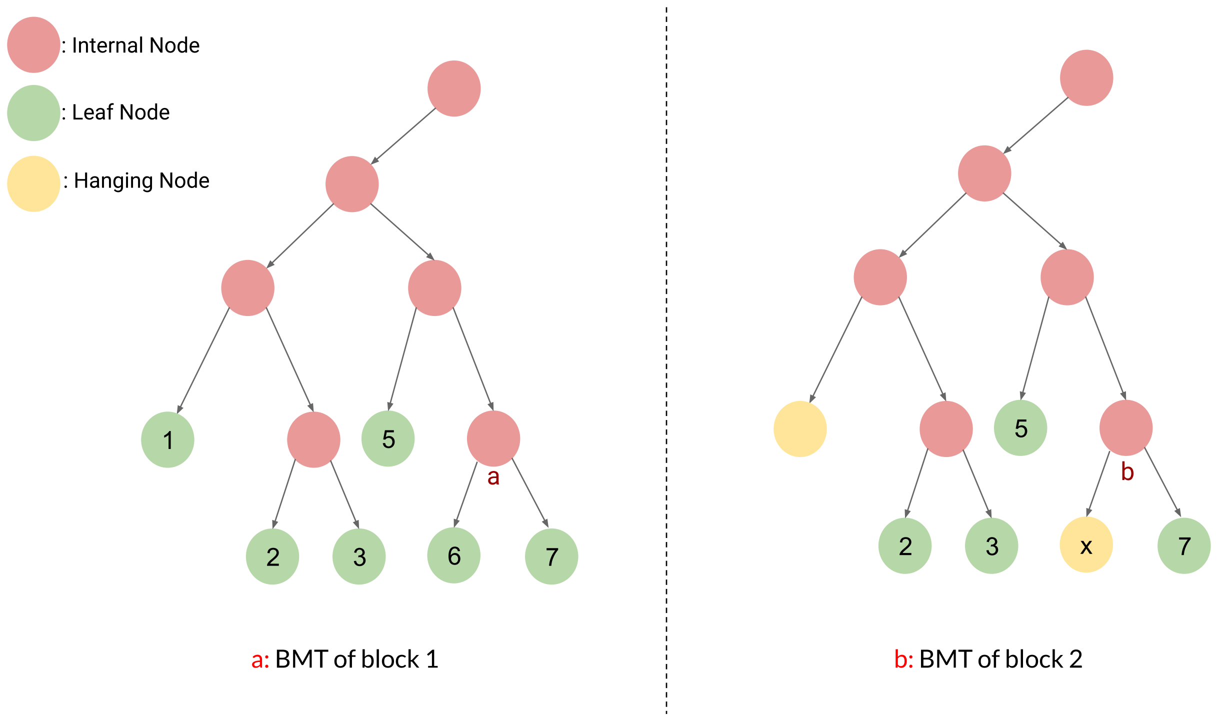


Figure 3.1: Block 1 and Block 2 BMT

3.2.1.1 Malicious Attack

With the hidden complete BMT details, the section details a server performed malevolent action, via which the server gets successful in exercising fraudulence by destroying the ledger. Let, for a BMT shown, we have:

1. The green nodes represent the new modified rows by the transactions within the block.
2. The red nodes represent the BMT internal nodes.
3. The yellow nodes represent the hanging nodes of the BMT.
4. The sky-blue node represents the malicious node of the BMT.

The following explains the malicious attack performed, block-*by*-block:

1. **Block 1 BMT**: In block 1, let's say a set of transactions arrive and modified distinct row `key-ids` [1, 2, 3, 5, 6, 7]. After the block interval, the server forms a block 1 BMT using the six distinct modified row `key-ids`, as shown in the section 'a' of Figure 3.1. The block 1 BMT has no previous BMT to be dependent upon; hence it possesses no

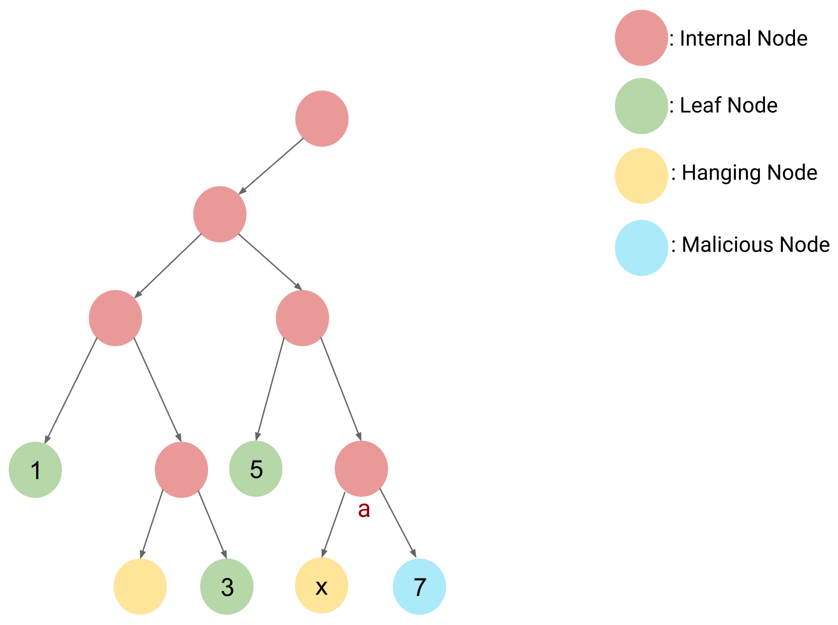


Figure 3.2: Block 3 BMT

hanging nodes. The server computes the BMT RHV and `block-hash` value, and feeds the database and the trusted storage. Considering nodes 6 and 7 of block 1 BMT, the parent node's `node-hash` computed be 'a', as shown. Let the modified row 7, new row value be 1000, and thus, the block 1 BMT node 7 has `node-value` as 1000.

2. **Block 2 BMT**: After block 1 formation, the set of rows modified by transactions gets accounted for in the block 2 formation. Let's say the transactions in block 2 modify rows with `key-ids` [2, 3, 5, 7]. The server, after the block interval, forms block 2 BMT depending upon the previous block 1 BMT structure and using row `key-ids` [2, 3, 5, 7], as shown in Figure 3.1, section 'b'. The server computes the block 2 BMT RHV and `block-hash` value and feeds into the database and the trusted storage. Let the row 7, new row value be 500, and thus, the block 2 BMT node 7 has `node-value` as 500. In block 2, BMT node 7 has been updated, so the `node-hash` value of the leaf node 7 and its parent node gets changed. The parent node of node 7, having `node-hash` as 'a' in BMT of block 1, now gets changed to `node-hash` as 'b' in BMT of block 2, shown in Figure 3.1. The sibling hanging node of node 7 in block 2 BMT, denoted as node x, points to node 6 of BMT block 1, and thus their `node-hash` remains identical.

3. **Block 3 BMT**: The block 3 accounts, the set of rows being modified by the transaction after block 2 gets built. In block 3, say the transactions modified rows [1, 3, 5], and at interval end, the server forms block 3 BMT depending upon block 2 BMT structure and using the row `key-ids` [1, 3, 5], displayed in Figure 3.2. In block 3, the server also maliciously updates the block 3 BMT leaf node 7 with `node-value` as defined in node 7 of block 1 BMT (section 'a' of Figure 3.1), i.e., with `node-value` as 1000. However, the server doesn't modify the `key-id` 7 row value in the database, and it remains 500. The sibling hanging node of node 7, i.e., node x being unmodified, points to node 6 of block 1 BMT. Now, the `node-hash` value of the parent node of node 7 in block 1 and block 3 BMT remains the same, i.e., 'a'. The server computes the block 3 BMT RHV and `block-hash` value and feeds into the database and the trusted storage. Once the block 3 `block-hash` value gets fed into the trusted storage, the clients verify whether the respective modified row `key-ids` were being included in the BMT formation using the provenance query and get satisfied. However, being unknown of the entire rows modified by all the transactions in block 3, the clients can't identify the server's malicious activity in including row `key-id` 7 for the block 3 BMT creation. The server with false BMT has computed false BMT RHV and `block-hash` and has archived it to the immutable trusted storage.

With the growing timeline, each TMT/BMT of future blocks gets formed depending upon the previous block BMT structure, and due to malevolence in block 3, the false TMT RHV and false BMT RHV gets computed for each ahead blocks. The obtained `block-hash` with being incorrect gets stored into the immutable trusted storage. Now, say at block B+n (where n>>>3), if a client queries for row `key-id` 7 value from the database, the server will show the value as 1000, hiding the real database row value of 500. The client further fires a transaction to modify row `key-id` 7 after block 3. The server will generate an old Merkle proof showing block 3 BMT node 7 details (having `node-value` of 1000). Further, the client verifies the old Merkle proof BMT RHV and notices it matching with the database. Seeking evidence valid, the client authenticates by signing with the digital signature, and the server commits the transaction. In Credereum, a client doesn't need to verify the provenance before signing the transaction. The client needs to merely substantiate the old and new Merkle proof, showing old and new data values for the modifications made, thus leading to an attack by the server that went successful.

Further, as stated previously, the entire BMT/TMT details of a block doesn't get displayed in provenance. Now, if the client asks for the provenance of `key-id` 7, and if each block details were not mandatory to be displayed, the server would only show the related TMT and BMT details of block 1, having row `key-id` 7 value as 1000. The server will hide the row `key-id`

7 modification in block 2, and the client remains unacknowledged about it. Hence, each block detail is mandatorily displayed in provenance, proving the modifications made or not to the queried rows.

With provenance, even though the client will get to know the aforementioned malicious activity, the server successfully destroyed the ledger. To re-calculate the `transaction-hash`, each client needs to re-submit the digital signature for all forward block transactions with vexatious verification. With the new TMT/BMT RHV, the trusted storage being immutable, the stored value can't get updated even if re-calculated. Now, likewise, say the server has done malevolent actions to multiple rows, with low hit probability, then the newly computed ledger may still be invalid, and the iteration continues. Without the BMT's complete verification, the client will always be in a dubious stage with the ledger's correctness in a centralized environment.

3.2.1.2 SecCred: ***Exhibit Merkle Data***

In Credereum, a client remains incapable to verify the correctness of the BMT formed by the server due to the absence of per-block transaction modification information. To tackle this situation, SecCred (section 4.2) designed, retains the Credereum architecture but includes changes in the `node-hash` calculation methodology of TMT/BMT nodes and supplements functionality to generate the TMT/BMT details for verification in SHA-256 hash forms. Due hash form, the verifying clients remain unknown about the row `key-id`'s modified and its associated row value. Using the transaction's details output, the client can compute the TMT RHV and `transaction-hash` value and match with the database. The BMT shown can be verified for accounting for each modification made by the transactions within the block and computes BMT RHV and `block-hash` value. The computed `block-hash` value gets equalized with the database and the trusted storage. Hence, with the TMT/BMT display in hash forms, any client can detect malicious activities. SecCred maintains entire blockchain properties held by Credereum and handles provenance.

3.2.2 Provenance Space

The clients in Credereum can query the modification history of a row/set-of-rows, displaying and proving the series of modifications being made by exploring from the *Genesis* block. The provenance query design helps the client verify whether the current state of a row R is the resultant of past modifications and gives add-on support to the old Merkle proof results revealing

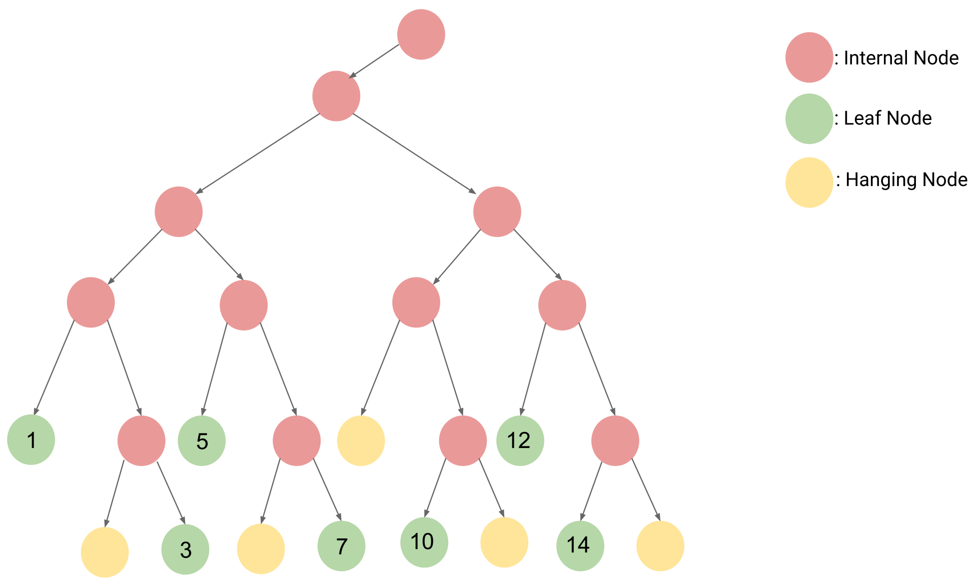


Figure 3.3: Block B BMT

the row R old values. The clients in Credereum can query for any row, and the server generates the provenance proof for the clients to verify. Hence, the displaying of provenance proof signifies beneficial functionality provided by the Credereum. The provenance query in Credereum gets handled by the CREDEREUM_MERKLE_PROOF() function [22], with only input the list of row `key-ids`, for which the provenance has been desired. However, with the growing timeline and the ever-increasing number of block's, the client if had already verified the modification history for a specific row R from the *Genesis* block till block X, then the client might not want to re-verify the modification history for row R, starting from the *Genesis* block. With time, the provenance search space increases, and thus the provenance proof result's generation engrosses time by generating the same results for row R up till block X from the *Genesis* block. The client verification also becomes cumbersome and is undesirable. Therefore, we needed specific optimizations in the search space and result generation for the provenance query. Moreover, the provenance engine designed for Credereum proposed versions addresses and overcomes the above-explained scenario.

3.2.3 Unique Indexing

In Credereum, for every relation A, the instance/row R has a fixed unique `key-id` and is being generated concatenating the relation name A bit-strings with row R identity (primary key) bit-strings (ref. 2). In the TMT/BMT formation, each leaf node represents the modified row `key-id` by-the-transaction/within-the-block. For the set of transactions modifying unique row `key-ids`, say [1, 3, 5, 7, 10, 12, 14] in block B, the BMT shown in Figure 3.3 is created. If multiple same row `key-ids`, say 5, are allowed to modify by-the-transaction/within-the-block, then a TMT/BMT leaf node can't represent multiple modifications to row 5 by-the-transaction/within-the-block. Hence, each node [1, 3, 5, 7, 10, 12, 14] of BMT represents a single modification made to the respective rows [1, 3, 5, 7, 10, 12, 14] in block B. To prevent multiple alterations to the same row R, Credereum adds a constraint that prohibits modifying R more than once by-the-transaction/within-the-block. Due to this constraint, if 'n' clients are firing to alter a particular row R within a block, only one transaction is allowed to commit, and the rest gets aborted. The failed client's competing with the rest needs to re-fire the transaction with a lower likelihood of getting committed.

3.3 Performance Limitation

Credereum on being a blockchain platform, the performance figures play an essential part in its real-world adoption. This section deals with comparing the performance of Credereum with PostgreSQL. With extra work performed in Credereum after Postgres processing, a drop in transaction rate was reasonable, but due to heavy pitfall intensity noticed in throughput, we explored the principal reasons behind such behavior. Further, methodologies focusing on performance boost via programming and algorithmic modifications have additionally been addressed.

3.3.1 Performance Evaluation

Figure 3.4 compares the performance of PostgreSQL with Credereum. The experimentation environment consists of 10 terminals firing transactions to a PG-Tuned PostgreSQL 10.5 database, where each transaction at max modifies 25-rows of the relation with hundred thousand entries. The transaction throughput or performance is measured in *tpe*, which stands for transactions-per-epoch. Here, epoch length is considered 10 minutes. From the figure, when Credereum gets compared with PostgreSQL, a *heavy* – approx *3 orders magnitude dip!* in the performance can be witnessed. Further, the Credereum only with an optimal computational load of the heavy-duty blockchain semantics and no transaction wait-time has been represented as Credereum_UpperBound (Credereum_UB). The figure depicts a *high* – approx *2 orders magnitude dip!* in the performance of Credereum_UB compared to PostgreSQL. With the performance

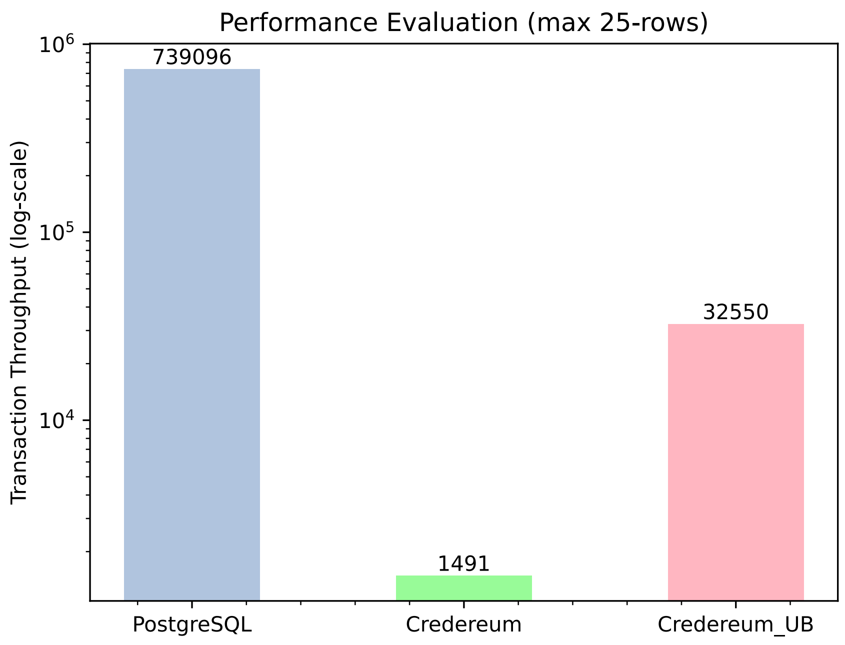


Figure 3.4: PostgreSQL v/s Credereum (TPE)

figures shown for Credereum, its utility in a real-world environment becomes questionable. Hence, significant research has been conducted focusing on boosting the Credereum system's throughput performance.

3.3.2 Credereum Shortcomings

The Credereum retained blockchain properties (section 2.4) including immutability, verification, authentication and non-repudiation. However, the addition of these properties to PostgreSQL via Credereum resulted in heavy orders of the system's performance degradation. With the Credereum engine being examined, the primary reasons for the performance degradation were analyzed.

The following are the set of compelling reasons behind the performance dip in Credereum:

1. In Credereum, for every transaction fired by the client, the transaction process after Postgres processing is handed over to the Credereum. The transaction process in the Credereum first needs to acquire a lock on the CREDEREUM_BLOCK table in ROW-EXCLUSIVE-MODE. Since ROW-EXCLUSIVE-MODE doesn't conflict with self, multiple transactions can

process parallelly after obtaining the lock. However, at the unit intervals, the block process arrives and attempts to procure a lock on the CREDEREUM_BLOCK table in SHARE-ROW-EXCLUSIVE-MODE. Now, SHARE-ROW-EXCLUSIVE-MODE conflicts with the ROW-EXCLUSIVE-MODE and self, signifying the block process works in an isolated manner, with no parallel running transaction or other block processes. Hence, with the block process being active, the transaction process needs to wait for the block process to commit and release the lock. The block process collects and forms a block using the modifications made by the committed-transactions within the block interval, which consumes time. During this time duration, the transaction process waits for the lock access and doesn't get committed or processed. This forms a significant reason, leading to a pitfall in the transaction throughput.

2. In Credereum, after a transaction acquires ROW-EXCLUSIVE-MODE lock on the CREDEREUM_BLOCK table, the server generates `key-id` for the all the rows modified and forms a TMT, depending upon the structure of the previous block BMT. With TMT created, the server finds `node-hash` of each node and computes TMT RHV. The TMT RHV summarizes the modifications made by the transaction. The TMT creation and TMT RHV calculation consume time, and this increases the overall computation cost. With the increase in lock handing time by a transaction process, the waiting transactions in the Read Committed isolation level need to wait longer, resulting in the pitfall of committed-transactions compared to the PostgreSQL.

3. For every transaction fired in Credereum, the server generates a modification proof to the client, displaying the row old values and the corresponding row new values. The row old values get proved using the old Merkle proof, and the row new values are displayed using the new Merkle proof. The old Merkle proof gets built considering the previous block BMT, and for new Merkle proof, the current transaction TMT details are presented. The old Merkle proof and new Merkle proof generation and display at per-transaction level consumes time, leading to an increase in the transaction process's execution time. The increasing computation cost leads to the late lock release and pitfall in the committed-transaction count.

4. For the aforementioned old Merkle proof and new Merkle proof generated by the server, the client needs to verify the evidence. For verification, the client computes the old Merkle proof BMT RHV and the new Merkle proof TMT RHV and matches it with the database. If found equal, the client verifies each supporting node's `node-hash` value of the BMT

with the respective supporting node `node-hash` value from the TMT. The client further checks for no free nodes (orphan nodes) in the BMT and TMT, and if found valid, verify whether the modifications made to the row `key-id`'s from old data value to the new data value is a resultant of the transaction fired. If the client is satisfied with the modifications made, he creates a digital signature and submits it to the server. Thus, the verification workload at the per-transaction level, consumes time and increases the computation cost. With an increase in the transaction execution time, a transaction throughput drop gets witnessed.

5. In Credereum, unique indexing (section 3.2.3) is being created, which avoids two transactions modifying the same row `key-id` within a block. Hence, if transaction A allocated to block B gets committed and has altered row X of table T, any future transactions that modify the same row X of table T in block B interval will eventually get aborted. The constraint also prevents a transaction from changing the same row `key-id` twice within a block. Due to this unique indexing, transactions get aborted, leading to a fall in committed transaction count.

3.3.3 Proposed Modifications

The Credereum, as illustrated, has orders of magnitude dip in the performance compared to PostgreSQL, and a list of potential reasons behind possessing such behavior has been explained. With the focus on uplifting the transaction throughput rate, programming and algorithmic modifications were proposed for Credereum. The changes made need to retain the Credereum supported blockchain properties (section 2.4) and shouldn't violate any blockchain semantics. Section 3.3.3.1 to 3.3.3.3 details a set of modifications being made to raise the performance.

3.3.3.1 ProgCred: *Programming Modification*

In Credereum, the functions written in SQL and C were investigated based on parameters, like holding the maximum number of calls, and having higher execution time (or self-time) per call. These functions were analyzed and further checked for optimization. The CREDEREUM_LONGEST_PREFIX() function [21] initially written in SQL was improved with algorithmic changes and re-programmed in C. The modification made no blockchain semantics violation and boosted the system's performance. Additionally, ProgCred retains the improvement proposed in SecCred (section 3.2.1.2).

3.3.3.2 PerfCredA: ***Algorithmic Modification 1***

PerfCredA deals with the first version, focusing on algorithmic modifications, and gets built considering the ProgCred engine. In ProgCred or earlier versions, a TMT is developed based upon the previous block BMT structure whenever a client fires a transaction. The server generates an old Merkle proof and new Merkle proof, proving the old and new data values. The PerfCredA relaxed this architecture by creating an autonomous TMT and further generating an old Merkle proof and new Merkle proof. The TMT built accounts for each database modification, i.e., UPDATE/INSERT/DELETE, being made by the transaction. The TMT RHV computed is used in the `transaction-hash` value calculation. The relaxed architecture helped raise the transaction throughput without violating any blockchain semantics. The provenance functions were designed and re-written to support the provenance queries in PerfCredA.

3.3.3.3 PerfCredB: ***Algorithmic Modification 2***

PerfCredB deals with an alternative and more radical approach by focusing on algorithmic improvements to the ProgCred engine. In PerfCredB, for each transaction fired, the TMT formation is left aside. Instead, using an iterative methodology, the MODIFICATION-HASH-VALUE (MHV) is computed. The MHV accounts for each modification, i.e., UPDATE/INSERT/DELETE made by the block's transaction. An old Merkle proof and New-Row-Value proof is generated, and further `transaction-hash` value gets computed. The MHV calculated is accounted in the `transaction-hash` value computation. The PerfCredB relaxed architecture uplifted the transaction throughput without violating any blockchain semantics. However, new methodologies and provenance functions were designed to help support provenance proofs. While the MHV computation approach, implemented in PerfCredB, delivers even more significant performance benefits, a down-side is that there is information leakage in provenance proof related to the number of rows updated by a transaction. However, since this leakage deals with quantity and not identity, it is not expected to have profound security implications.

3.4 Pointed Queries

The Credereum, possessing enriching features, has certain shortcomings, with one being the clients facing database visibility issues. With the transaction fired by clients, the server doesn't provide proof for the database state on which the transaction statement executes. The server might use malicious database state D' compared to the original database D and run the clients' fired transactions on it. For the rows modified by the fired transaction in D', the server only shows the old data values proof for the modified `key-ids` while generating the old Merkle proof. The server never proffers evidence for the entire database state on which the transaction gets executed. Hence fraudulent actions can be easily practiced by the server, and the clients get tricked. The Credereum, being centralized, is designed to handle transactions where the client focuses on modifying a particular row of a relation determined using its *primary key* (distinct). With the whole database being highly infeasible for verification, a client in Credereum can't exercise transactions with single/multi-row modification using non-primary key attributes. The client can fire multiple instructions/statements within a transaction in Credereum, where each aims a specific row, determined by its primary key. For m instructions, m distinct row modifications are made.

3.5 Summary

In Credereum, the server keeps hidden the whole BMT information from the clients, due to which the clients can't verify the BMT created. With the help of a provenance query, a client might confirm the inclusion of the respective modified row-set in the BMT formation but, with lacking other transaction's row modification information, the client can't distinguish the malicious activities if exercised. SecCred design helps overcome the scenario and strengthens the systems' resistance towards malevolent actions. The reason behind Credereum's orders of magnitude dip in transaction throughput has been addressed and explained. The programming and algorithmic improvements proposed showed a positive impact in boosting the Credereum systems' performance.

Chapter 4

Functional Improvement

The chapter deals with explaining the functionality advancements made to Credereum. The revealing of the Merkle tree details to the users/clients and reducing the provenance engine's search space have been addressed. Methodology involving the display of Merkle tree details gets introduced in the SecCred. The modifications were made without violating blockchain semantics and retain blockchain properties held initially by the Credereum.

4.1 Introduction

Credereum, being a blockchain platform, has certain functional limitations that need to be addressed to enhance security, system performance, and resistance to malicious activities. The Merkle tree display gets handled to improve system security, and by which the clients can verify the correctness of each block BMT formed. These prevent malicious activities that the server can perform in the BMT formation, and without the Merkle tree display, it would have remained unacknowledged to the clients. The Merkle display technique uses the SHA-256 hash to display the TMT and BMT information, which can be verified and trusted by the clients. The functionality improvement also includes the optimal execution of the provenance query engine and is discussed later in this chapter.

4.2 SecCred: *Exhibit Merkle Data*

In Credereum, whenever a transaction/set-of-transactions gets committed within the current block period, at intervals end, a block process begins. For any active uncommitted transaction X that arrived before the block process, the block process needs to wait for the transaction X to commit/abort. The active uncommitted transactions that appear after the block process will wait for the block process to commit and get accounted for in the next block formation. The block process for the current block creates a BMT depending upon the previous block BMT structure. The BMT formation accounts for each row modified by all the transactions within the block. Once the block's BMT gets developed, the BMT RHV and, subsequently, `block-hash` value is computed.

In Credereum, the above block's BMT formed does not get displayed to the user and is kept hidden. Hence, the server can practice certain malicious activities in the BMT creation without getting noticed, as explained in section 3.2.1.1. So, the user inherently trusts the server with per block BMT formation.

A feature addition of displaying a block's BMT and all the allocated transaction's TMT in the Credereum engine is performed by preserving privacy and using the SHA-256 hash. The new Credereum design, with TMT and BMT display, is named as SecCred. In Credereum, some unfreed pointers P, resulting in memory leaks, were also witnessed. In SecCred, those pointers P were also freed and deallocated.

4.2.1 Merkle Tree: Hash Computation

In Credereum, a TMT and BMT are formed for a transaction process and a block process, respectively, depending upon the previous block BMT structure. In SecCred, the TMT and BMT structure formation remains the same as of Credereum. However, on Merkle tree formation, the Merkle tree RHV calculation methodology in SecCred differs.

In Credereum for a given Merkle tree, either TMT or BMT, as shown in Figure 4.1, the `node-hash` of each node (if not known), is computed as:

- hash(leaf node) = SHA256 (row `key-id` + ':' + `node-value`)

- hash(internal node) = SHA256 (children-1 `node-key` + ':' + `node-hash` of children-1 + ',' + children-2 `node-key` + ':' + `node-hash` of children-2)

In SecCred, for a given Merkle tree, either TMT or BMT, as shown in Figure 4.1, the `node-hash` of each node (if not known), is computed as:

- hash(leaf node) = SHA256 (row `key-id` + ':' + `node-value`)

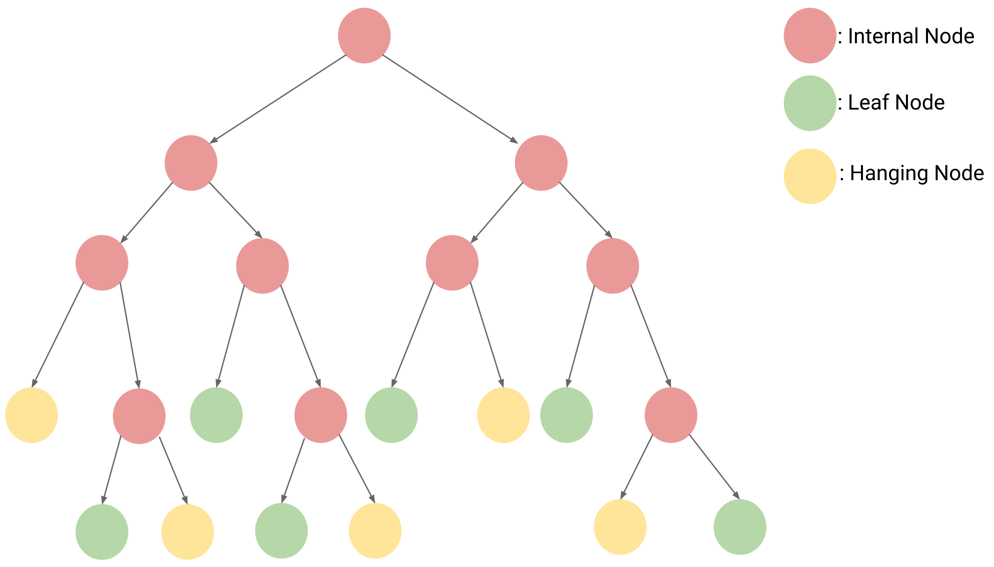


Figure 4.1: Transaction/Block Merkle tree

- hash(internal node) = SHA256 (SHA256 (children-1 `node-key`) + ‘:’ + `node-hash` of children-1 + ‘,’ + SHA256 (children-2 `node-key`) + ‘:’ + `node-hash` of children-2)

In the above formula, the `key-id` bit-strings represents the row R modified by the transaction. Further, the leaf nodes’ `node-value` retains the row R new data. Also, as stated earlier, for a leaf node, the `node-key` is the row `key-id`. Now, as shown in Figure 4.1, the Merkle tree might possess hanging nodes (yellow nodes), which links the current Merkle tree to the previous block’s BMT. The `node-hash` value for such hanging nodes is already known and gets used in the `node-hash` calculation of its associated parent node in the Merkle tree. In SecCred, each internal node, as shown, considers the hash of `node-key` of each child, rather than just the `node-key`, and the rest remains the same.

4.2.2 TRANSACTION_AND_BLOCK_DATA relation

The feature addition of displaying a block’s TMT and BMT in Credereum with preserving privacy lead SecCred to display the details of the attributes in the TRANSACTION_AND_BLOCK_DATA relation, shown in Table 4.1. The server displays all the associated TMT’s and BMT details for the queried block via TRANSACTION_AND_BLOCK_DATA table entries. The block_num attribute denotes the block to which the TMT/BMT belongs, and the leaf attribute denotes whether a node is a leaf node or not. Also, for a BMT, the transaction_id is NULL. Now, the remaining blue-highlighted attributes from Table 4.1, i.e., key_hash, nhash, left_child, and right_child are represented in their SHA-256 hash forms. The key_hash attribute denotes the hash of `node-key` (i.e., `node-key` hash), and nhash attribute indicates a nodes' `node-hash` value. Attributes left_child and right_child denotes both the children of a specific node, and retains the hash of respective children's `node-key`.

Relation	Attributes
TRANSACTION_AND_BLOCK_DATA	block_num, transaction_id, key_hash, nhash, left_child, right_child, leaf

Table 4.1: TRANSACTION_AND_BLOCK_DATA schema

With the given details from the TRANSACTION_AND_BLOCK_DATA relation, any client can compute the TMT and BMT structure in its hash-forms. Due to hash, the verifying clients can't infer from TMT/BMT, the rows modified by another client (via leaf node's `node-key`) and their associated `node-value`; hence it remains hidden.

4.2.3 Execution and BMT Proof

In SecCred, similar to Credereum, whenever a client fires a transaction, a TMT gets built with the previous block BMT dependency. The TMT RHV is computed as discussed in section 4.2.1. An old Merkle proof and new Merkle proof gets generated detailing the modified rows old and new values, respectively. After verifying the proof, the client digitally signs the transaction and sends the digital signature to the server. The server, after verifying the signature, computes the `transaction-hash` value as:

- `transaction-hash` = SHA256 (previous block BMT RHV + current transaction's TMT RHV + public key + digital signature)

The `transaction-hash` value is updated in the CREDEREUM_TX_LOG table, and the server commits the transaction. Similarly, say a set of transactions fired by multiple client's commits. At unit intervals, the block process arrives and computes a BMT depending upon the previous block BMT structure. The BMT RHV is computed, as stated in section 4.2.1, and the `block-hash` value is further calculated as:

- `block-hash` = SHA256 (previous block `block-hash` + all transaction's `transaction-hash` + BMT RHV)

The computed BMT RHV and `block-hash` value is further stored in the CREDEREUM_BLOCK table and the block process commits. The `block-hash` value gets finally archived to the trusted storage.

Unlike Credereum, the SecCred on request displays the above formed TMT's and BMT information, with `node-key`'s in hash formats, and is verifiable by all the clients, making the system more secure and concrete. For proof, the TRANSACTION_AND_BLOCK_DATA relation (Table 4.1) details (say, set S), introduced in section 4.2.2, gets displayed. Given the details, any client can construct the TMT and BMT structure and know the TMT and BMT leaf nodes. Section 4.2.4 states the reason why displaying the TMT is crucial in proving the block's BMT.

For a particular TMT, the client can compute the TMT RHV from the details output and match it with S and digital signature (obtained from public CREDEREUM_TX_LOG relation). For the BMT displayed, the client can likewise compute the BMT RHV and check with S. While verification, the TMT/BMT internal nodes' `node-hash` (nhash, Table 4.1) can be calculated and verified using:

- hash(internal node) = SHA256 (left_child + ':' + nhash of Left-Child + ',' + right_child + ':' + nhash of Right-Child)

From the above formula, the blue-highlighted texts represent the attributes of TRANSACTION_AND_BLOCK_DATA relation (Table 4.1), where left_child, right_child and nhash are represented in SHA-256 hash forms.

Now, while computing TMT RHV/BMT RHV, each node's `node-hash` is matched with the details from set S. The clients with the TMT details from S can verify no multiple modifications to same rows made intra-transaction and inter-transaction, by seeking TMT's leaf node whether or not holding distinct key_hash values. Further, each TMT leaf node details are checked whether or not matched with the current block BMT and vice-versa. A check for no orphan nodes is additionally made in the TMT and BMT displayed.

Now, very-importantly, for the hanging nodes of TMT and BMT, the server additionally shows through TRANSACTION_AND_BLOCK_DATA relation (Table 4.1), the path nodes and supporting nodes of the previous block BMT. The path nodes contain nodes from the to-be-verified hanging nodes to the previous block BMT root node. And, the supporting nodes with the sibling's aide help calculate the `node-hash` of the parent node. The clients can verify the correctness of the `node-hash` of TMT and BMT hanging nodes by computing the previous block BMT RHV and further `block-hash` value. The `block-hash` value computed is matched with the database and the immutable trusted storage.

Further, now with hanging nodes verified, the clients can compute all the transaction's `transaction-hash` and block's `block-hash` value and confirm with the one stored in the CREDEREUM_TX_LOG table and CREDEREUM_BLOCK table, respectively. The computed `block-hash` is mandatorily matched with the trusted storage, and if valid, this signifies S to be authentic. With even a single bit-modification, a change in the `block-hash` can be easily witnessed. Hence, with given TMT and BMT details, any client can verify the BMT and check its validity.

4.2.4 TMT Display Necessity

For proving the block B BMT created to be authentic, the transaction's TMT associated with block B needs to be displayed. With an option being to show only the modifications made by a particular transaction (leaf nodes of TMT) in hash forms and make it public, the server might display a client, the different set of alterations related to a transaction, and the client remains victimized. By just revealing the modifications made, the client can't compute the TMT RHV to verify whether the server's details with respect to a transaction are valid or not. To overcome this, the per transaction TMT structure needs to be displayed. With the TMT structure revealed, the client can verify TMT RHV and `transaction-hash` and check hanging nodes authenticity. The orphan nodes' presence in TMT can be additionally checked. Further, the client can compute the `block-hash` value and compare it with the trusted storage.

From section 3.2.1.1, a case arises where the server maliciously updates a leaf node (Figure 3.2) of BMT and holds it hidden to practice malicious actions. Without TMT display, even if the server has shown the BMT structure, the server might show client A that the malicious node has been the result of the modifications made by client B and vice-versa. Now without TMT display, client A/client B needs to trust the server for its authenticity and fall into the trap. If client A/client B gets shown the TMT structure related to client B/client A, then because of the client's verification, the server can't cheat and the fraudulent actions get prevented. Hence, to verify BMT's authenticity, the associated TMT must be displayed to make proof complete.

4.2.5 BMT Sample Results

Say, there are four transactions T1, T2, T3, and T4, which users were firing, one after the other, and on commit, were used in the block formation of block 1, block 2, block 2, and block 3, respectively. The query-set related to each transaction is:

T1:

```
INSERT INTO WAREHOUSE (id, w_tax) VALUES (1, 24.50);
INSERT INTO WAREHOUSE (id, w_tax) VALUES (5, 22.95);
INSERT INTO WAREHOUSE (id, w_tax) VALUES (3, 27.85);
INSERT INTO WAREHOUSE (id, w_tax) VALUES (6, 25.65);
INSERT INTO WAREHOUSE (id, w_tax) VALUES (12, 28.90);
```

T2:

```
UPDATE WAREHOUSE set w_tax=w_tax-0.007 where id=6;
UPDATE WAREHOUSE set w_tax=w_tax-0.007 where id=12;
INSERT INTO WAREHOUSE (id, w_tax) VALUES (8, 29.50);
INSERT INTO WAREHOUSE (id, w_tax) VALUES (4, 28.25);
```

T3:

```
INSERT INTO CUSTOMER (id, c_discount) VALUES (3, 24.50);
INSERT INTO CUSTOMER (id, c_discount) VALUES (4, 21.20);
INSERT INTO CUSTOMER (id, c_discount) VALUES (5, 26.10);
INSERT INTO CUSTOMER (id, c_discount) VALUES (6, 23.55);
INSERT INTO CUSTOMER (id, c_discount) VALUES (7, 29.55);
```

T4:

```
UPDATE WAREHOUSE set w_tax=w_tax+0.001 where id=3;
UPDATE WAREHOUSE set w_tax=w_tax+0.001 where id=6;
UPDATE WAREHOUSE set w_tax=w_tax+0.001 where id=12;
INSERT INTO WAREHOUSE (id, w_tax) VALUES (7, 18.75);
INSERT INTO WAREHOUSE (id, w_tax) VALUES (2, 15.95);
UPDATE CUSTOMER set c_discount=c_discount+0.05 where id=3;
UPDATE CUSTOMER set c_discount=c_discount+0.05 where id=7;
```

The above transaction's either INSERT/UPDATE the WAREHOUSE table, or the CUSTOMER table, or both. For each UPDATE/INSERT/DELETE, a node in the TMT and BMT is created, with the new value. Using the above example, after the transaction's commit and post block formation, any user can verify the BMT correctness and the block's `block-hash` value computation made by the server.

For a block B BMT, we have the following:

1. Red nodes represent the BMT internal nodes or path nodes (nodes that lead to the root node from leaf nodes).
2. Green nodes represent the new modified rows by the transactions within block B.
3. Yellow nodes represent the hanging nodes in BMT of block B, which references the previous block B-1 BMT node.
4. Orange node represents the referenced node in block B-1 BMT, corresponding to the hanging node of BMT of block B.
5. Blue nodes represent the supporting nodes in block B-1 BMT, provided for calculating the parent node's `node-hash`.

Using the above information, whenever a client queries for block B BMT proof, all transaction's TMT and BMT of block B are output, as explained in section 4.2.3. Now, the following details the formation and client-end verification for each block's BMT. For simplicity purpose, let's say for each block B BMT-Query (discussed below), with the evidence presented, the client has computed all the respective transaction's TMT RHV and `transaction-hash` value and verified with the CREDEREUM_TX_LOG table. Further, the client has also checked the authenticity/correctness of the hanging nodes of each TMT by verifying it with the previous block BMT details.

1. **Block 1 BMT-Query**: When the fired transaction T1 gets committed, the block process forms a BMT depending upon the previous block BMT structure. Block 1, being the first data block of the ledger, has no previous BMT to be dependent upon. Hence, the block process, after autonomously forming block 1 BMT, computes the BMT RHV and the `block-hash` value and feeds into the database. The block process eventually commits

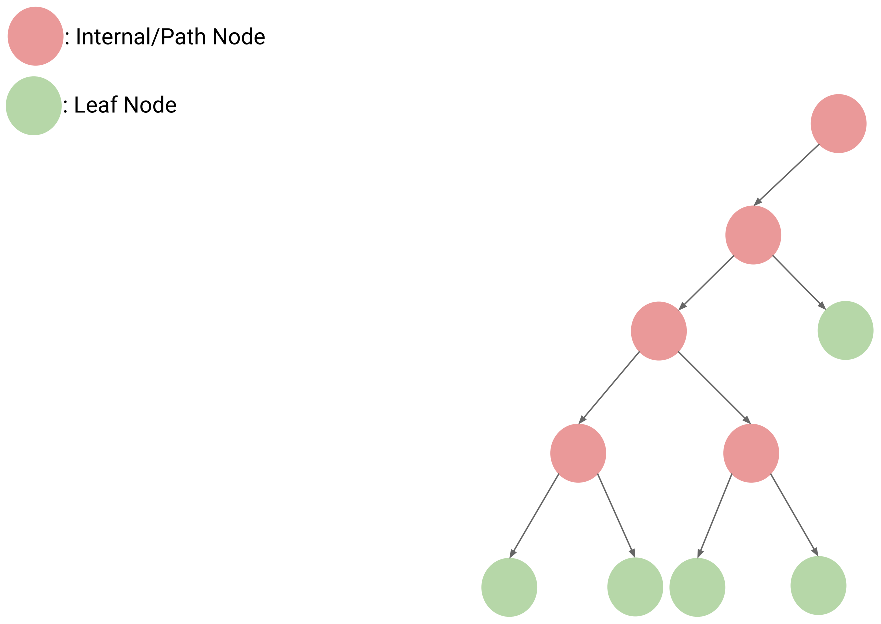


Figure 4.2: Block 1 BMT-Query Output - BMT of block 1

and the computed block 1 `block-hash` value gets archived to the trusted storage.

Now, any user can query the server to display the block 1 BMT proof. Apart from T1 TMT details, the server displays the previous block 0 entry and current block 'block 1' BMT, shown in Figure 4.2. For each node shown in Figure 4.2, as stated, from Table 4.1, the server displays the associated block number, the hash of `node-key`, the `node-hash` value, the hash of `node-key` of children's node, and indicates whether the node is a leaf or not.

The green nodes of the BMT of block 1 denote the five different rows modified by transaction T1. Block 1, being the first data block of the ledger and having no previous BMT, doesn't possess any hanging nodes. Now, once all the leaf nodes of block 1 BMT is cross-verified with TMT of T1, the user can compute the block 1 BMT RHV from the details given and match it with the value stored in the database/TRANSACTION_AND_BLOCK_DATA table. If valid, the user further computes the block 1 `block-hash` value and compares it with the database and the trusted storage. If match occurs, this signifies the BMT formation of block 1 made by the server is valid.

2. **Block 2 BMT-Query**: When the fired transactions T2 and T3 gets committed, the

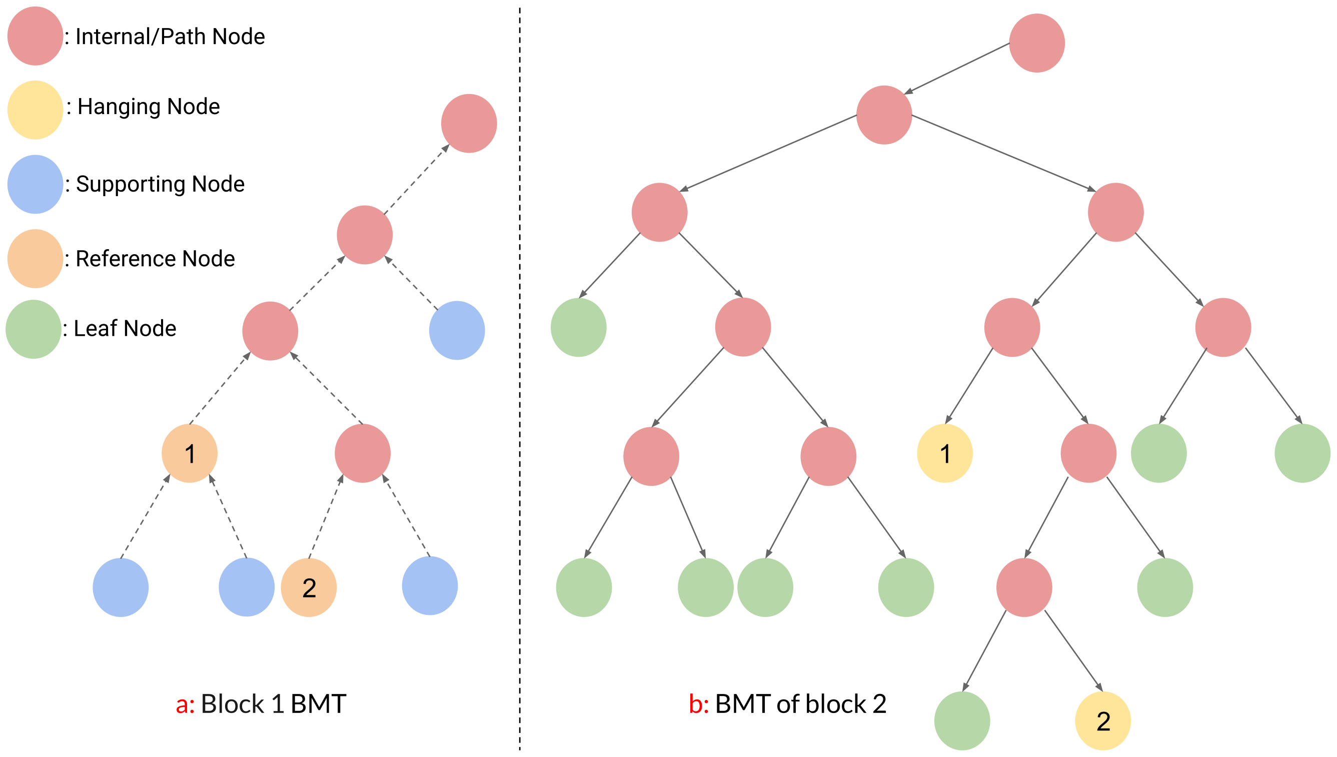


Figure 4.3: Block 2 BMT-Query Output

block process forms a BMT for block 2 depending upon the previous block 'block 1' BMT structure. The block process obtains the BMT RHV, computes the block 2 `block-hash`, and stores it in the database. The block process commits and adds the block 2 `block-hash` to the trusted storage.

After block formation of block 2, any user can query the server to display block 2 BMT proof. The server, apart from T2 and T3 TMT structure, as shown in section 'b' of Figure 4.3, displays the current block 'block 2' BMT structure. Since the BMT formation of block 2 depends upon the BMT structure of block 1, block 2 BMT retains hanging nodes, illustrated in section 'b' of Figure 4.3. For each node in Figure 4.3, the server displays the associated block number, the hash of `node-key`, the `node-hash` value, the hash of `node-key` of the children's nodes, and indicates whether the node is a leaf or not, as stated in Table 4.1.

For the green nodes of section 'b' of Figure 4.3, any user can cross-verify the `node-key` hash and the `node-hash` from the respective details of T2 and T3 TMT leaf nodes. The BMT green nodes of block 2 denote the nine distinct rows, modified by transaction T2 and T3 combined. Additionally, block 2 BMT holds hanging nodes (marked 1 and 2), shown in section 'b' of Figure 4.3. The hanging nodes' existence in the previous block BMT needs to be proven to establish the hanging nodes' validity. For this purpose, the previous block 'block 1' BMT details containing the referenced node (marked 1 and 2) corresponding to the hanging node, and supporting nodes and path nodes are displayed. If the reference node is an internal node of the BMT, then the details comprising `node-key` hash and `node-hash` of both the children's are output, as shown for reference node (marked 1) in the section 'a' of Figure 4.3. Else, if the reference node is a leaf node of the BMT, then the `node-key` hash and `node-hash` details of the reference node are output, as shown for reference node (marked 2) in the section 'a' of Figure 4.3. Using the given details, the user can compute the BMT RHV and `block-hash` value of block 1 and match it with the database and the trusted storage. If the match occurs, this signifies the hanging nodes exist and are valid. The user can now verify the leaf nodes of block 2 BMT, compute the block 2 BMT RHV, and match it with the database/TRANSACTION_AND_BLOCK_DATA table. If found matching, the user can further calculate the block 2 `block-hash` value and confirms it with the database and the trusted storage. Once verified, the client can be assured of the BMT formation of block 2 to be valid.

3. **Block 3 BMT-Query**: When the fired transaction T4 gets committed, the block process forms a BMT for block 3 depending upon the previous block 'block 2' BMT structure. The block process finds the BMT RHV and computes the block 3 `block-hash`, which gets stored in the database. The block process commits and adds the block 3 `block-hash` to the trusted storage.

   With the block creation of block 3, any user can query for block 3 BMT proof. Apart from T4 TMT structure, the server displays the current block 'block 3' BMT, as shown in Figure 4.4. Since the BMT formation of block 3 depends upon the BMT structure of block 2, block 3 BMT retains hanging nodes. For each node in Figure 4.4 and Figure 4.5, the server displays the block number, the `node-key` hash, the `node-hash`, the `node-key` hash of the children's nodes, and indicates whether the node is a leaf/not from Table 4.1.

   The green nodes of the BMT of block 3 (Figure 4.4) denote the seven distinct rows, being modified by transaction T4, and any client can cross-verify the BMT leaf node's `node-key` hash and the `node-hash` value from the respective T4 TMT leaf nodes. Block 3 BMT formation depends upon the previous block 'block 2' BMT structure, hence possesses hanging nodes (marked 1, 2, 3, 4, and 5), shown in Figure 4.4. To prove the validity of the hanging nodes, the previous block 'block 2' BMT details containing the referenced node (marked 1, 2, 3, 4, and 5) corresponding to the hanging node, and supporting nodes

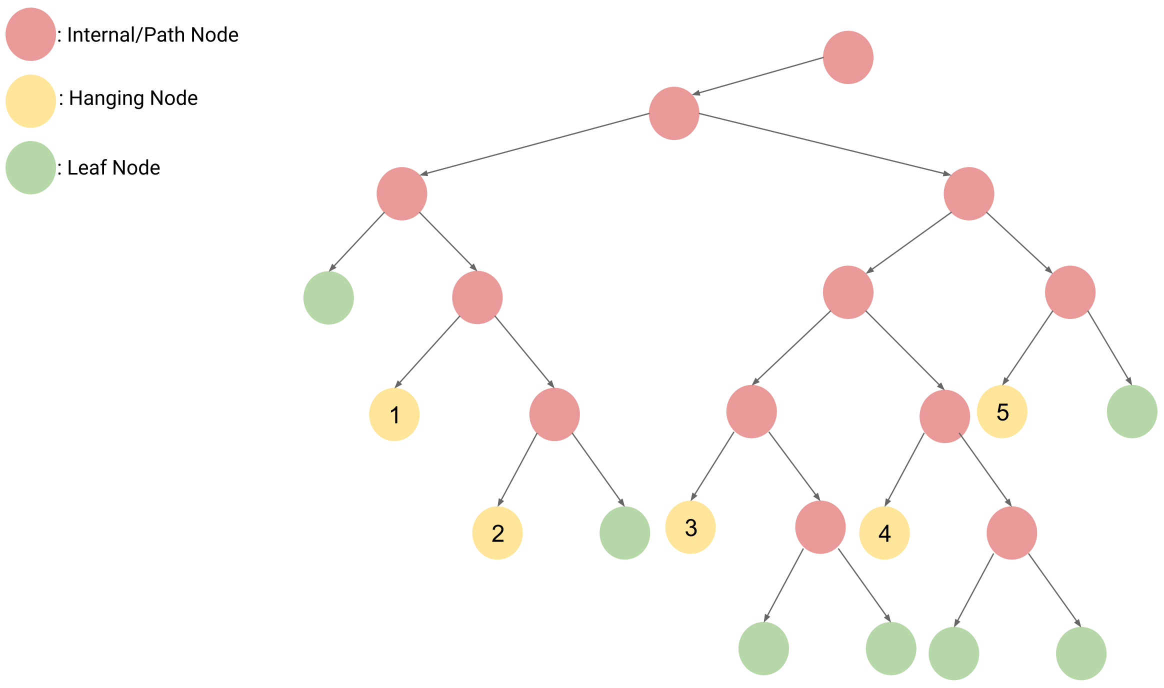


Figure 4.4: Block 3 BMT-Query Output (Part-1) - BMT of block 3

and path nodes are displayed, as shown in Figure 4.5. Using the given details, the user can compute the BMT RHV and `block-hash` value of block 2 and match it with the database and the trusted storage. If the match occurs, this signifies the hanging nodes are authentic. However, it can be observed that, while comparing the block 2 BMT shown in section 'b' of Figure 4.3 and Figure 4.5 remains the same, except for an extra addition of reference node (marked 3) and it's sibling in Figure 4.5. This happens because the hanging node (marked 1) in section 'b' of Figure 4.3 refers to the block 1 BMT referenced node (marked 1) in the section 'a'. Hence, the reference node (marked 3) and its sibling in Figure 4.5 corresponds to both the children of the referenced node (marked 1) in the section 'a' of Figure 4.3. The hanging nodes of a BMT, thus, link one block BMT to the previous block BMT.

The user can now verify the leaf nodes of block 3 BMT, compute the block 3 BMT RHV, and match it with the database/TRANSACTION_AND_BLOCK_DATA table. If the match occurs, the user can further calculate the block 3 `block-hash` value and compare it with the database and the trusted storage. If found valid, this signifies the block 3 BMT creation by the server is correct.

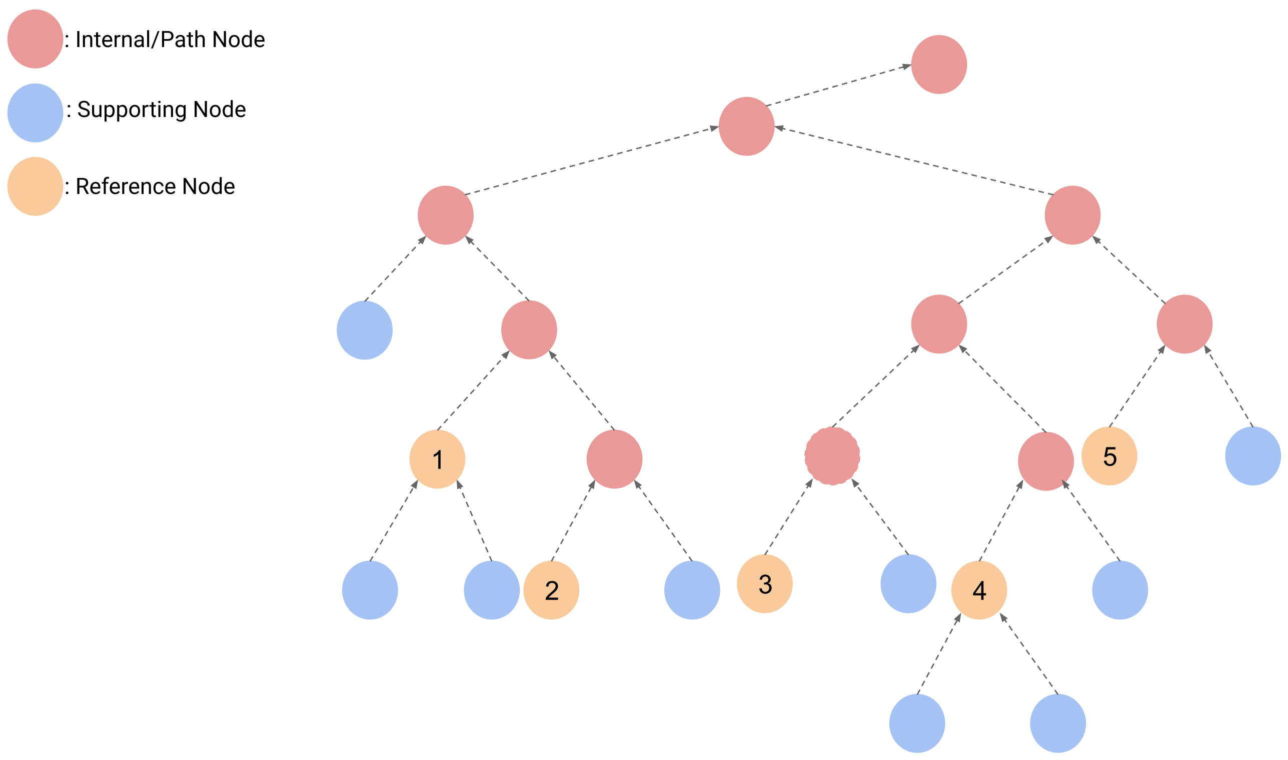


Figure 4.5: Block 3 BMT-Query Output (Part-2) - Block 2 BMT

4.2.6 BMT Validity

The section details a simple scenario of a malicious activity being performed by the server within a block and addresses how a client can trace and acknowledge such an act.

Say, at unit intervals, a block process arrives and forms a block B considering all the modifications made by the committed transactions within the block period. Figure 4.6 displays the block B BMT created by the server. From Figure 4.6, all the green nodes are valid modifications, being made by the transactions allocated to block B. However, say the server has exercised malicious activity in block B BMT formation by updating the `node_value` of a leaf node corresponding to the row `key-id` R, which has not been modified by any transaction in block B. Further, say node A represents the row `key-id` R in block B BMT formation, shown in Figure 4.6. The server computes the BMT RHV and `block-hash` value for block B and feeds it into the database. The block process commits, and the `block-hash` value gets archived into the trusted storage.

Now, whenever the client queries for the block B BMT proof, the server apart from TMT details shows the BMT data as internal nodes (red nodes), leaf nodes (green nodes), and the

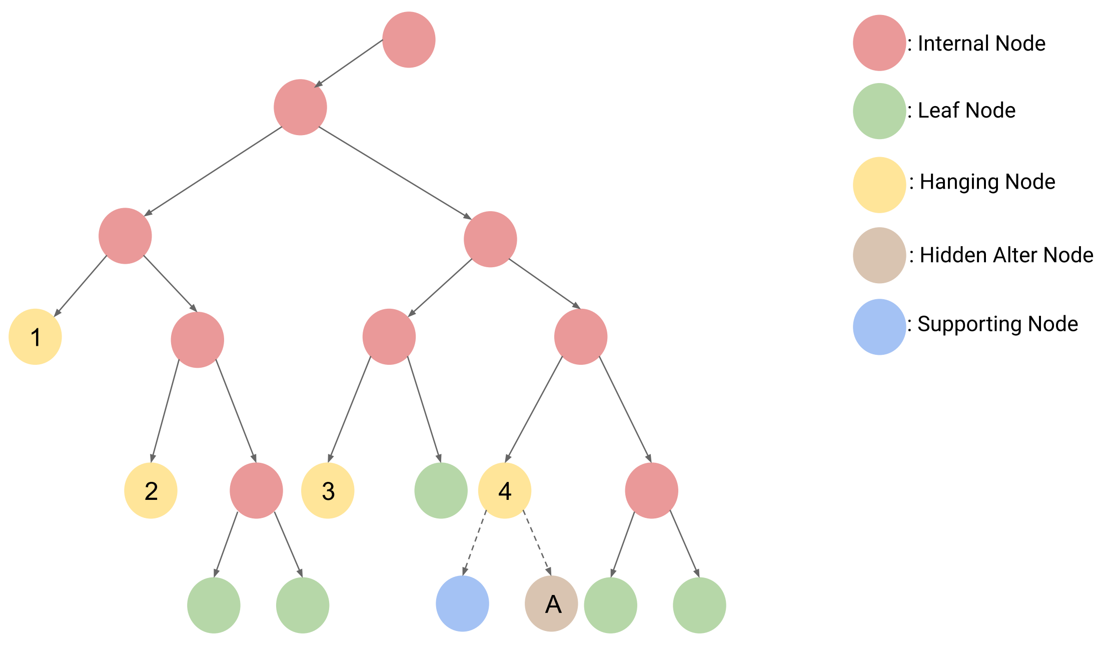


Figure 4.6: Block B BMT

hanging nodes (yellow nodes) shown in Figure 4.6. The server doesn't display node A and its sibling and shows only the parent node 4. The client verifies block B transaction's TMT RHV, and `transaction-hash` and check the authenticity of its each hanging node. Now, using the shown details, the client will first check whether each BMT green node and the `node-hash` details associated are matched with the TMT leaf node, and vice-versa. After verification, with the details displayed, the client further computes the block B BMT RHV and `block-hash` value and matches with the database and trusted storage. The client will discover the details matching, and lastly, check for the hanging nodes' authenticity before declaring the BMT to be valid.

Now, for each hanging node of BMT of block B, its reference node in the previous block B-1 BMT has to be shown by the server. Hence, for each hanging node of BMT of block B, the output shown in Figure 4.7 are displayed. Figure 4.7 shows the BMT of block B-1 with reference nodes (orange nodes), internal nodes (red nodes), and supporting nodes (blue nodes). The client using block B-1 BMT details obtains the BMT RHV and `block-hash` value of block B-1 and matches with the database and trusted storage. The `node-hash` value for each hanging node [1, 2, 3, 4] of BMT of block B, shown in Figure 4.6, is matched with the `node-hash` of respective reference node [1, 2, 3, 4] of BMT of block B-1 in Figure 4.7. If the `node-hash`

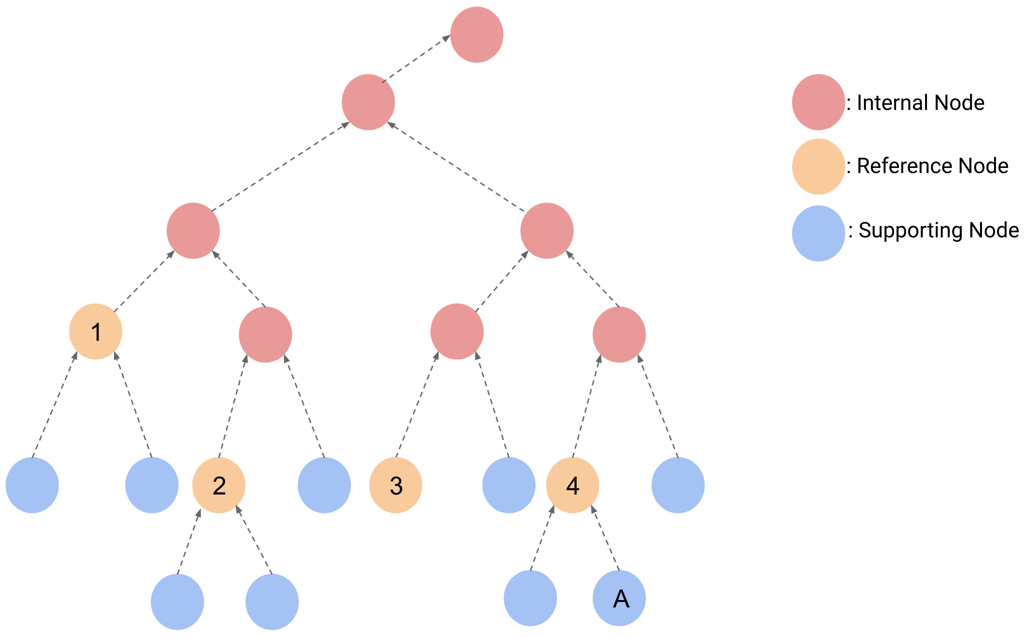


Figure 4.7: Block B-1 BMT - Hanging node verification

of each hanging node matches, this signifies the hanging node `node-hash` used in the BMT formation of block B is authentic. However, in the current case, only the `node-hash` of hanging node [1, 2, 3] of block B BMT will match with the respective reference node of BMT of block B-1. Because of the modification made in the right child of node 4 of block B BMT (Figure 4.6), the `node-hash` of node 4 gets changed. Thus, the `node-hash` of hanging node 4 of block B BMT will differ from the reference node 4 of block B-1 BMT (Figure 4.7). Hence, the client will witness a change in the `node-hash` value of node 4 in block B BMT compared to block B-1 BMT and acknowledge malicious activity practiced.

Now, if the `node-hash` for node 4 of the block B-1 BMT (Figure 4.7) and node 4 of block B BMT (Figure 4.6) is shown equal by the server, then the calculated BMT RHV and `block-hash` value of block B-1 BMT will differ from the database and the trusted storage, and the client gets alarmed. The client can also query for the provenance to judge for any malicious activity performed by the server. Hence, if the server performs malicious activity by not acknowledging any row being modified by the committed transaction within the block in BMT formation or performing malicious activities in BMT creation, then the client can easily witness such exercises with the BMT information and proof displayed.

4.3 Provenance Engine

In Credereum, whenever a client fires provenance queries for any particular row `key-id`, the provenance engine always searches the records from the *Genesis* block. However, as explained, if a client has already verified the row `key-id` R provenance results up to block B-K ($1 \leq K \leq B-1$), the client might not want/require to re-verify the provenance results for the row `key-id` R from *Genesis* block to block B-K. Instead, the client might want to verify the provenance results for row `key-id` R only from block B-K+1 to the latest formed block B. Since the provenance result generation and verification consume time, optimization in the search was required. The optimization made help the provenance engine to search for records in a particular block bandwidth with a lower bound and upper bound as block number being input. The modifications made to the Credereum, i.e., SecCred (section 4.2), ProgCred (section 5.2), PerfCredA (section 5.3), and PerfCredB (section 5.4), all have the provenance engine functionality of receiving block number from the clients, specifying the block start and end, for which the provenance search is desired. If no block number is specified, the provenance engine by default searches from the *Genesis* block to the current block B. The discussed functionality added led to the modifications in the provenance engine code.

4.4 Summary

The Credereum on the block's BMT formation doesn't display the whole BMT and is hidden, leading to the server's malicious activities, which remain unacknowledged to the clients. The SecCred (section 4.2) shows the BMT in hash forms on the client's request, which the client can verify, and any malicious activity can get easily detected. The BMT is displayed, keeping the other client's privacy protected and with the application of SHA-256 hash. The SecCred can be stated as Credereum with additional functionality to display the BMT details. Further, SecCred led changes in TMT/BMT RHV calculation methodology and displays the TRANSACTION_AND_BLOCK_DATA relation. The provenance engine (section 4.3) search space was reduced to boost provenance query response time and lower the client's verification time.

Chapter 5

Performance Improvement

This chapter explains the performance improvements made to Credereum, focusing on programming optimization and algorithmic modifications. The proposed versions, namely, ProgCred, PerfCredA, and PerfCredB, are explained. ProgCred introduces programming improvements, while PerfCredA and PerfCredB explain algorithmic changes. The programming/algorithmic modification was made without violating blockchain semantics and retains the blockchain properties held by the Credereum. The changes proposed made a positive impact in boosting the committed-transaction throughput.

5.1 Introduction

In a blockchain system, the speed-efficiency/transaction-rate is one of the key parameters through which the system utility gets examined. The Credereum, as stated, has orders of magnitude degradation in transaction throughput compared to the native PostgreSQL. Research was performed to increase the throughput performance in Credereum, thus leading to programming and algorithmic modifications. The functions were analyzed, and optimization was proposed to uplift the transaction count. The TMT/BMT formation at per transaction-level/block-level was researched, and modifications were aimed without violating blockchain semantics. With TMT RHV used natively to summarize the transaction's alterations, new methodologies were designed to hold the same accompanying a rise in performance. The algorithmic changes have led to alterations in the handling of the provenance query raised by the client. With all the changes proposed, a boost in transaction throughput was witnessed, along with all the blockchain properties held.

5.2 ProgCred: *Programming Modification*

This section deals with the first phase of changes made with a focus on performance enhancement. ProgCred retains the modifications made in SecCred and deals with the programming changes to the Credereum functions/procedures. The Credereum functions were written in SQL and C language. Based on the total execution-time per call (or total self-time per call) and the total number of calls made (obtained from PostgreSQL *Statistics Collector*), the functions were analyzed, seeking optimization to enhance the performance. The function with the maximum call counts was considered with the highest priority for optimization analysis. The modifications made in ProgCred showed a positive impact without violating blockchain semantics and retains the blockchain properties sustained initially by the Credereum. In ProgCred, similar to the Credereum (section 2.5), the transactions fired by multiple clients to PostgreSQL processes parallelly under Read Committed (RC) isolation level.

Function	**Calls**	**Total_Time_Per_Call** (micro-secs)	**Self_Time_Per_Call** (micro-secs)
credereum_longest_prefix	9684313	140	140
credereum_sha256	527334	1	1
credereum_merklix_get_hash	310924	54	52
credereum_merklix_insert	40768	6469	653
credereum_acc_trigger	20595	19	19
credereum_get_relation_id	20544	3	3
...	...	...	...
credereum_sign_transaction	1025	545	542
credereum_apply_transaction	1025	4677239	4486824
credereum_pack_block	105	5684609	4887006

Table 5.1: Credereum - Functions list

In Credereum, as stated, the functions were written in SQL and C language. These functions were analyzed and ordered based on the total number of calls made to them for a given time frame. Table 5.1 details the list of some Credereum functions (SecCred incorporated), being arranged in decreasing order of Calls. The Calls indicate the number of times the function X being called, the Self_Time_Per_Call denotes the function X average self-execution time excluding the execution-time of any further called sub-function by X, and the Total_Time_Per_Call signifies the average execution time of a function X summing the self-execution time and execution-time of any further called sub-function by X. The Total_Time_Per_Call and the Self_Time_Per_Call shown in Table 5.1 are measured in micro-seconds.

Table 5.1 shows that the CREDEREUM_LONGEST_PREFIX() function got the maximum number of calls and was initially written in SQL. The CREDEREUM_LONGEST_PREFIX() function computes the longest common prefix between the two given bit-strings. In ProgCred, the CREDEREUM_LONGEST_PREFIX() with algorithmic changes and inclusion of bitwise-level operations was re-written in C language. These changes made a positive impact and improved throughput. The CREDEREUM_SHA256() function (Table 5.1) computes the hash of any data using SHA-256 and has a self-execution time of 1 microsecond. The CREDEREUM_MERKLE_INSERT() function (Table 5.1), with the help of the CREDEREUM_LONGEST_PREFIX() function, forms TMT/BMT structure by depending upon the previous block BMT structure. The CREDEREUM_MERKLE_GET_HASH() function (Table 5.1) helps compute the TMT RHV and BMT RHV after the TMT and BMT formation, respectively. The CREDEREUM_SIGN_TRANSACTION() function (Table 5.1) checks the digital signature submitted by the client to the server. After verification, the function computes the `transaction-hash`, stores it in the database, and commits the transaction.

The Self_Time_Per_Call can be found at high peaks for CREDEREUM_APPLY_TRANSACTION() function and CREDEREUM_PACK_BLOCK() function, from Table 5.1. The major reason for such high self-execution time is the lock's access by these functions. Each transaction process before TMT formation acquires a ROW-EXCLUSIVE lock on the CREDEREUM_TX_LOG table in CREDEREUM_APPLY_TRANSACTION() function. Each block process before BMT formation, acquires a SHARE-ROW-EXCLUSIVE lock on the CREDEREUM_TX_LOG table and CREDEREUM_BLOCK table in CREDEREUM_PACK_BLOCK() function. Because of the ROW-EXCLUSIVE lock, multiple transactions can process parallelly. However, if a block process executes and acquires SHARE-ROW-EXCLUSIVE lock, no other transaction/block process (if exist) can execute simultaneously/parallelly.

If a transaction process acquires the ROW-EXCLUSIVE lock, the block process waits for the lock release. Similarly, if a block process acquires the SHARE-ROW-EXCLUSIVE lock, the transaction process waits for the lock release. Because of the waiting time, the Self_Time_Per_Call for CREDEREUM_APPLY_TRANSACTION(), and CREDEREUM_PACK_BLOCK(), is massive compared to other functions.

5.2.1 Function Optimization

The ProgCred deals with programming changes to the CREDEREUM_LONGEST_PREFIX() function (from Table 5.1) and was initially written in SQL. In particular, the SQL version of the

CREDEREUM_LONGEST_PREFIX() function [21] takes two `key` bit-strings as input and output the longest common prefix between the `keys`. The SQL variant checks each bit of the two `key` bit-strings until the shortest length between the two `key`'s. If the bit matches, it's appended to the resultant bit-string, and on un-matched bits/end-of-loop, the SQL returns the resultant bit-string. Hence, in the initial SQL version of the CREDEREUM_LONGEST_PREFIX() function comparison between two `keys` is made at the bit-by-bit level to generate the longest common prefix. The provided code for the CREDEREUM_LONGEST_PREFIX() function SQL version [21] is shown below:

```
CREATE OR REPLACE FUNCTION credereum_longest_prefix(k1 varbit, k2 varbit)
    RETURNS varbit AS $$
DECLARE
  len     int;
  i     int;
  prefix    varbit;
BEGIN
  len := least(length(k1), length(k2));
  i := 1;
  prefix := ''::varbit;
  WHILE substring(k1, i, 1) = substring(k2, i, 1) AND i <= len LOOP
    prefix := prefix || substring(k1, i, 1);
    i := i + 1;
  END LOOP;
  RETURN prefix;
END;
$$ LANGUAGE plpgsql STRICT;
```

The above CREDEREUM_LONGEST_PREFIX() function, SQL version, was converted to C function, with programming changes and bitwise-level operations. The C variant, similar to SQL, inputs two `key` bit-strings and computes the `keys`' longest common prefix. With a well-known approach, the CREDEREUM_LONGEST_PREFIX() C version compares the two `keys` in a byte-by-byte fashion, with upper bounds computed considering the shortest length between the `keys`. Commencing from MSB (Most Significant Byte), the two `keys`' corresponding bytes get compared using the bitwise Exclusive-OR (XOR) operation. If the compared byte of the two `keys` matches, the prefix length is incremented by 8. If the byte does not match, then the bitwise-level operations are performed to obtain the mismatch bit location for the given byte and the number of bits before the mismatched bit gets added to the prefix. The C version CRED-

EREUM_LONGEST_PREFIX() function, at the mismatch byte occurrence/upper-bound, outputs the bits up till the prefix counts.

With long length bit-strings, on average 125 bits, used for evaluation, the conversion of the CREDEREUM_LONGEST_PREFIX() function from SQL to C made a positive impact and improved the performance. The ProgCred retains all the blockchain properties held by Credereum and is described in section 6.3. Section 7.2 explains the provenance query handling by ProgCred.

5.3 PerfCredA: *Algorithmic Modification 1*

This section deals with the second phase of changes made, focusing on performance enhancement. In ProgCred (and, preceding), when a client fires a transaction to the Postgres, a TMT gets developed with the previous block BMT structure dependency. As proof of database modification, an old Merkle proof and new Merkle proof get generated to the client. The new Merkle proof constitutes the current transaction TMT details. The PerfCredA deals with algorithmic improvements to the ProgCred system. In PerfCredA, the aforementioned architectural complexity is eased by eliminating the previous block BMT dependency for per-transaction TMT creation. However, the proposed methodology has led to changes in the provenance output, and further, verification at the client-end. By relaxing the architecture, the blockchain semantics initially possessed by Credereum gets retained, and the modification made a positive impact in boosting the transaction throughput. In PerfCredA, similar to ProgCred/Credereum (section 2.5), the transactions fired by multiple clients to PostgreSQL processes parallelly under Read Committed (RC) isolation level.

5.3.1 Architecture Design

In PerfCredA, whenever a client fires a transaction X, the transaction first processes in the Postgres engine with the Read Committed isolation level, and the control is handed over to the PerfCredA engine shown in Figure 5.1. In the PerfCredA engine, the transaction X process first builds its TMT independently, i.e., without depending upon the previous block BMT structure. The independent building of TMT per transaction level leads to modifications in the CREDEREUM_MERKLE_INSERT() function (Table 5.1), responsible for building the Merkle tree. With TMT independent, the TMT only accounts for the client's alterations, with new row value entries. Upon completing the TMT structure, the TMT RHV gets computed in

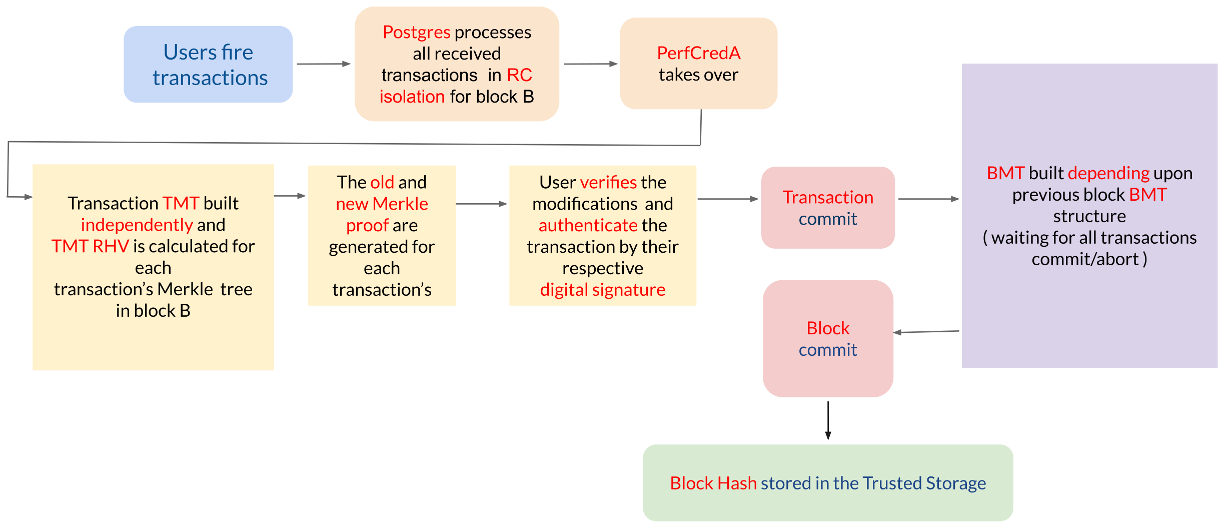

Users fire transactions
Postgres processes all received transactions in RC isolation for block B
PerfCredA takes over
Transaction TMT built independently and TMT RHV is calculated for each transaction's Merkle tree in block B
The old and new Merkle proof are generated for each transaction's
User verifies the modifications and authenticate the transaction by their respective digital signature
Transaction commit
BMT built depending upon previous block BMT structure (waiting for all transactions commit/abort)
Block commit
Block Hash stored in the Trusted Storage

Figure 5.1: PerfCredA workflow

the naive recursive methodology. For computing the TMT each node's `node-hash` value, the methodology remains the same as discussed in SecCred (section 4.2.1).

For each UPDATE/INSERT/DELETE made by transaction X, an old Merkle proof and new Merkle proof get generated to the client. The old Merkle proof presents the data values of a row before modification with evidence and is verifiable. The old Merkle proof gets constructed using the previous block BMT, similar to Credereum (Figure 2.7). The new Merkle proof shows the new row values after modifications made by transaction X. The new Merkle proof displays the current transaction X TMT details. Figure 5.2 shows the old Merkle proof (left tree) and new Merkle proof (right tree), say generated by the server for transaction X. The left tree contains modified leaf nodes (grey nodes), path nodes from leaf to root nodes (purple nodes), and supporting nodes (blue nodes). Similarly, the right tree contains new altered leaf nodes (green nodes) and internal nodes (red nodes). Except for modified leaf nodes (grey nodes) and new altered leaf nodes (green nodes), for each node, only the `node-key`, and `node-hash` values are shown, which helps compute the Merkle tree RHV. While, for modified leaf nodes (grey nodes) and new altered leaf nodes (green nodes), all the details constituting `key-id` (i.e., `node-key`) and associated `node-value` are shown.

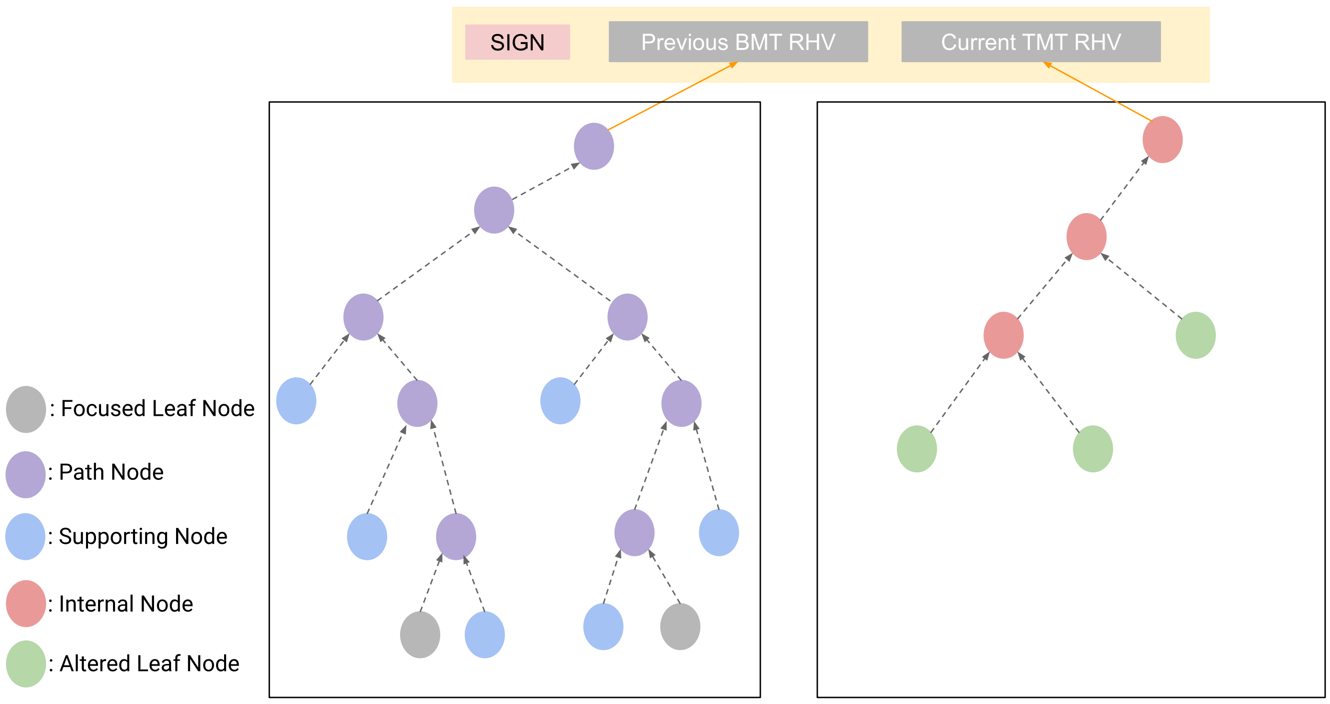


Figure 5.2: PerfCredA Merkle proof generation

Now, for each modified leaf node (grey node), showing the old data values, the client first checks its validity by computing the previous block BMT RHV and matches it with the database. If the match occurs, the client calculates and verifies the transaction X TMT RHV shown. The client examines for orphan nodes' existence in the BMT and TMT of old Merkle proof and new Merkle proof, respectively. For 'n' modified leaf nodes (grey node), there will be respective 'n' altered leaf nodes (green node), signifying each UPDATE/DELETE performed by transaction X. If there are any extra altered leaf nodes (green node) present in transaction X TMT, then those nodes signify the INSERT performed by the transaction X. For such an INSERT operation, the old Merkle proof gives proof of its non-existence in the database.

The user post verifying whether the transaction fired by him resulted in changes from the modified leaf node to the new altered leaf node authenticates the transaction by signing with his digital signature. The digital signature gets generated by concatenating the old Merkle proof (left tree) BMT RHV and new Merkle proof (right tree) TMT RHV and is signed using its private key. The server verifies the signature, and if found valid, computes the `transaction-hash` similar to ProgCred/SecCred (section 4.2.3) and commits the transaction.

From Figure 5.3, we can see that the `transaction-hash` gets computed concatenating the

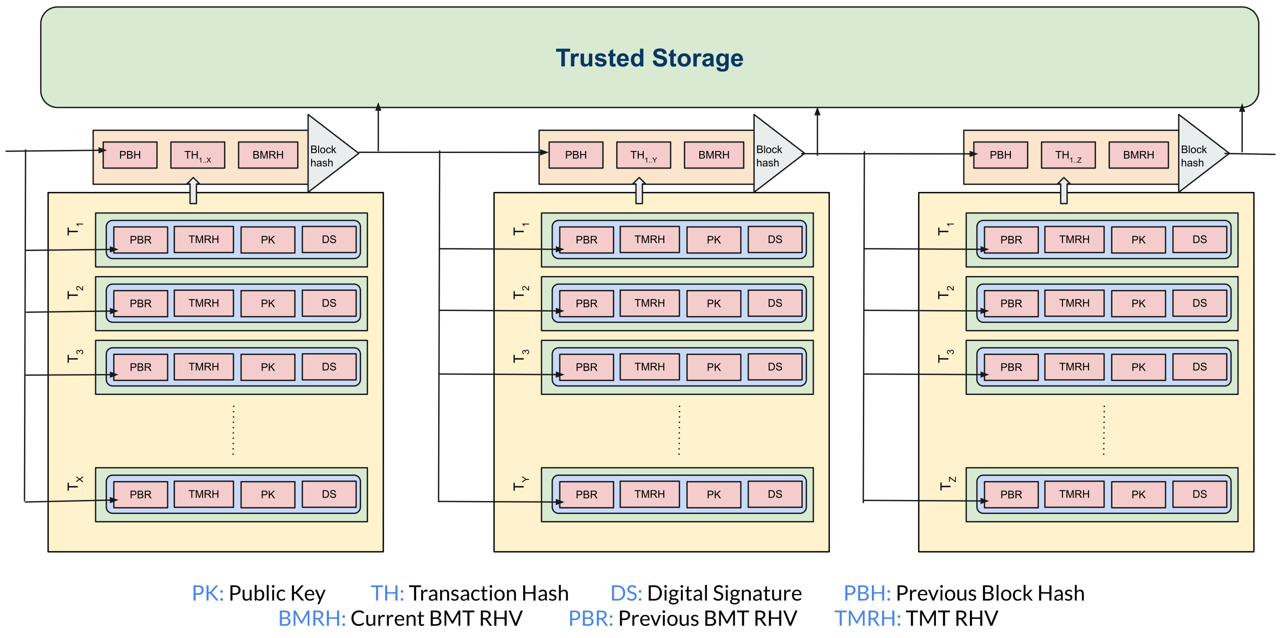


Figure 5.3: PerfCredA block creation

previous block BMT RHV value, current transaction TMT RHV, public key, and the digital signature. At unit intervals, the block formation process waits for all the transactions within the block period to commit. The block process then begins collecting all the rows altered by all the transactions within the block period. For constructing the BMT, the block process calls the CREDEREUM_BLOCK_MERKLE_INSERT() function, which forms the current block's BMT by depending upon the previous block's BMT structure. Hence, the block's BMT formation methodology remains the same for the Credereum, SecCred, ProgCred, and PerfCredA. On completion of the current block BMT structure, the BMT RHV is computed recursively. For calculating the BMT each node's `node-hash` value, the approach remains the same as discussed in SecCred (section 4.2.1). The computed BMT RHV is accounted in the current block `block-hash` value, along with each allocated transaction's `transaction-hash` value and previous block `block-hash` value. The previous block `block-hash` value accounting in the current block `block-hash` value helps establish the ledger chain. The current block `block-hash` value calculated is stored in the database and the trusted storage. The client can further query the block's BMT formed, similar to the SecCred (section 4.2.3), and verify the results.

PerfCredA retains all the blockchain properties held by Credereum and has been addressed in section 6.4. The algorithmic alteration has led to modifications in handling the PerfCredA provenance queries and is explained in section 7.3.

5.3.2 Architecture Validity

Let's say the transaction T fired by the client A, UPDATE/DELETE 'n' rows, and INSERT 'm' rows, where each row is distinct and on commit will get accounted in the block formation of block B.

In Credereum, SecCred, and ProgCred, for each UPDATE/INSERT/DELETE made by transaction T, an old Merkle proof and new Merkle proof is displayed to client A. The old Merkle proof gets developed using the previous block B-1 BMT and displays the old data values of distinct rows being UPDATE/DELETE by transaction T. And, for INSERT, a proof of non-existence gets shown. For 'n' unique rows that client A wants to UPDATE/DELETE, the old Merkle proof should show the respective 'n' old values. Client A verifies the 'n' targeted row values shown in old Merkle proof and confirms it by obtaining the block B-1 BMT RHV. The new Merkle proof shows the transaction T TMT details. For each UPDATE/INSERT/DELETE performed by transaction T, a respective leaf node in TMT gets created. The TMT being dependent also had hanging nodes, which links to the previous block B-1 BMT node. The previous block B-1 BMT dependency, in transaction T TMT formation, symbolizes n+m rows being modified by transaction T, and the rest of the database rows remain unmodified. Hence, the main idea of the previous block BMT dependency in TMT formation was to show that the non-targeted database rows remain unchanged/unaltered. The user verifies the old Merkle proof and the new Merkle proof. If found valid, then client A digitally signs the transaction.

In PerfCredA architectural design, the per transaction TMT created was made independent and had no structural dependency on the previous block B-1 BMT. The ideology is the client A is only accountable/responsible for the modifications being made by him. In PerfCredA, for each UPDATE/INSERT/DELETE made by transaction T, an old Merkle proof and new Merkle proof is displayed to client A. The old Merkle proof generation remains the same as discussed above for Credereum, SecCred, and ProgCred. However, in new Merkle proof, the transaction T TMT is developed independently using n+m modified rows. Hence, client A is accountable to only n+m distinct rows targeted by him. If the server makes any additional row modification, then that row has to be added in the transaction T TMT. Client A can easily catch any extra row modification with changes in the TMT, and with variations than the intended TMT, it will not sign the transaction and the transaction gets aborted. Hence, if the server modifies fewer/more rows than the targeted, such malicious activity will be reflected in the TMT formed, and the server will get caught. If the TMT formed by the server is valid and has leaf nodes only corresponding to targeted n+m modified distinct rows, then client A will verify the modifications and create a digital signature using his private key. The digital signature is sent to the server and gets accounted for in the transaction T `transaction-hash` computation. The `transaction-hash` computed is used in block B `block-hash` value computation. Once block B gets formed, client A can verify the transaction T TMT and its inclusion in block B BMT formation through provenance. If any malicious activity gets found, client A can raise the alarm.

Even if the server forms a false TMT and computes it RHV, and keeps it hidden from client A, the server can't generate the digital signature for the modifications made because it doesn't have the private key of client A. Hence, without a digital signature, the transaction T `transaction-hash` can't be computed and made public for other clients to verify. Therefore, the server gets bound from performing malicious activities in PerfCredA.

5.4 PerfCredB: *Algorithmic Modification 2*

The section deals with an alternative improvement to ProgCred, compared to PerfCredA, focusing on boosting committed-transaction throughput. In ProgCred, when a client fires a transaction, a TMT is developed depending upon the previous block BMT structure. As proof of database modification, an old Merkle proof and new Merkle proof get generated to the client for the transaction fired. The PerfCredB deals with an alternative algorithmic modification to the ProgCred. In PerfCredB, the above architectural complexity gets relaxed by removing the per-transaction TMT formation, and alternatively, using an iterative methodology for summarizing the transaction modifications. Further, the changes made in PerfCredB led to modification in handling and providing proof for the provenance queries raised by the client. By relaxing the architecture, the blockchain semantics initially possessed by Credereum is retained, and modification made a positive impact in uplifting the transaction throughput. In PerfCredB, similar to PerfCredA/ProgCred/Credereum (section 2.5), the transactions fired by multiple clients to PostgreSQL, processes parallelly under Read Committed (RC) isolation level.

5.4.1 Architecture Design

Whenever a client fires a transaction T in PerfCredB, the transaction first processes in the Postgres engine with the Read Committed isolation level. The control is later handed over

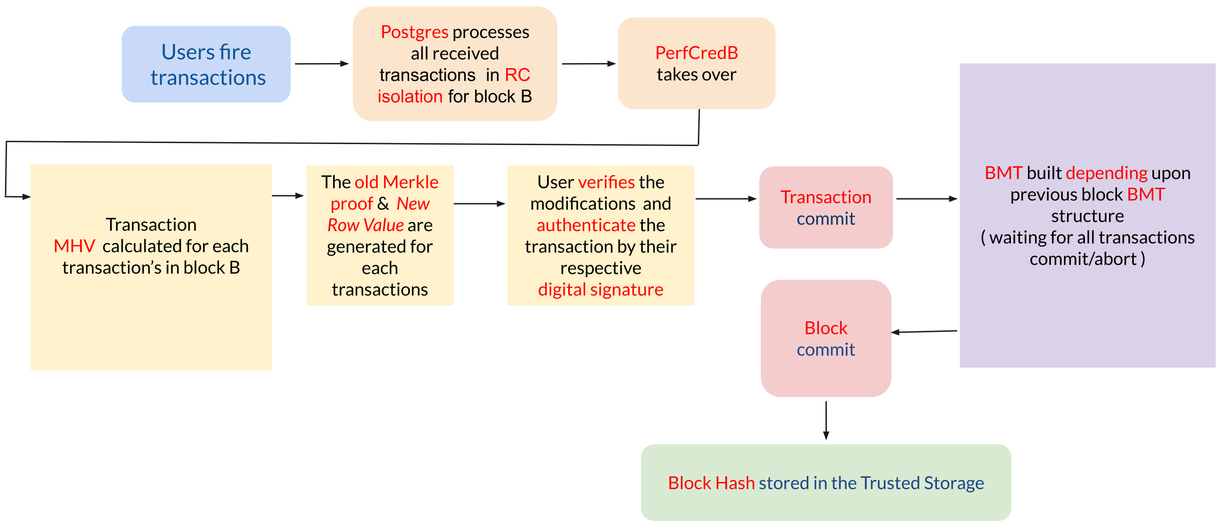

Users fire transactions
Postgres processes all received transactions in RC isolation for block B
PerfCredB takes over
Transaction MHV calculated for each transaction's in block B
The old Merkle proof & New Row Value are generated for each transactions
User verifies the modifications and authenticate the transaction by their respective digital signature
Transaction commit
BMT built depending upon previous block BMT structure (waiting for all transactions commit/abort)
Block commit
Block Hash stored in the Trusted Storage

Figure 5.4: PerfCredB workflow

to the PerfCredB engine, as shown in Figure 5.4. In the PerfCredB engine, the transaction T processes by iterative methodology and calculate the `sub-hash` and `hash` value of each modified row `key-id` and computes the MODIFICATION-HASH-VALUE (explained section 5.4.2). The MODIFICATION-HASH-VALUE (MHV) accounts for the `key-id` and data value pairs of each modified row and summarizes the transaction T's modifications.

For modifications made by transaction T, an old Merkle proof and New-Row-Value proof get generated to the client. Figure 5.5 shows the old Merkle proof (left section) and New-Row-Value proof (right section), generated by the server for transaction T. The old Merkle proof presents the data values of a row before modification with evidence and is verifiable. The New-Row-Value proof shows the new data values after changes made by transaction T. The left section contains a Merkle tree with modified leaf nodes (grey nodes), path nodes from leaf to root nodes (purple nodes), and supporting nodes (blue nodes). Similarly, the right section contains new altered row value details (green circles), and the MHV (orange circles) details of transaction T. For the old Merkle proof, except for the modified leaf nodes (grey nodes) of the Merkle tree, for each node, only the `node-key`, and `node-hash` are shown, which helps compute the Merkle tree RHV. While for modified leaf nodes (grey nodes) of the Merkle tree

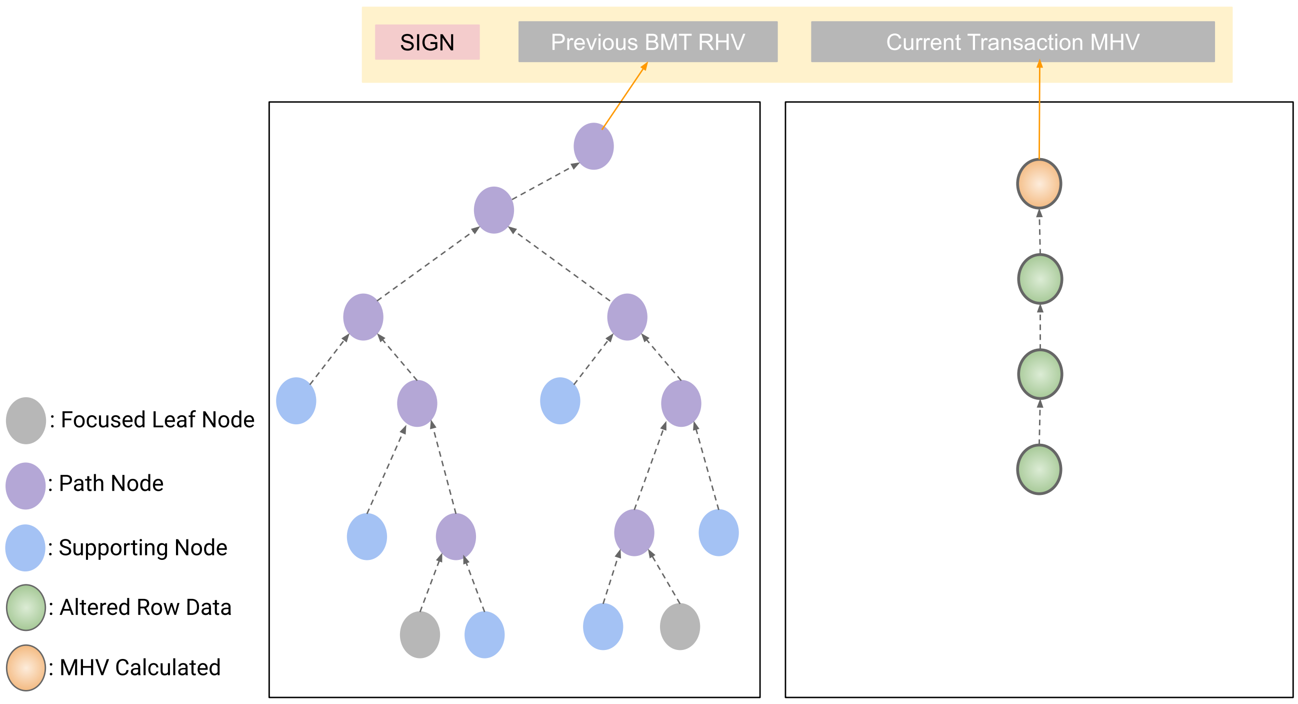


Figure 5.5: PerfCredB proof generation

and new altered row data (green circles), all details, including the `key-ids` and associated value, get shown.

With the given old-values of rows modified, the client first validates the old Merkle proof shown by computing and verifying each node's `node-hash` value. The computed previous block BMT RHV is matched with the database. If the match is valid, the client using the New-Row-Value proof details, calculate the MHV and compares it with the given (orange circle, Figure 5.5). If the match occurs, for each UPDATE/DELETE made by transaction T, the client further compares the leaf nodes of old Merkle proof with the respective row data displayed in the New-Row-Value proof. For each INSERT, the new values are checked and verified for inclusion in the MHV calculation. If the modifications made by the server for the transaction T are found valid, the client authenticates the transaction by signing with his digital signature. The digital signature is generated by concatenating the old Merkle proof (left section) RHV and New-Row-Value proof (right section) MHV and signing it using a private key. The server verifies the signature, and if valid, computes the `transaction-hash` value varying PerfCredA/ProgCred/SecCred.

From Figure 5.6, we can perceive that the `transaction-hash` is computed by concate-

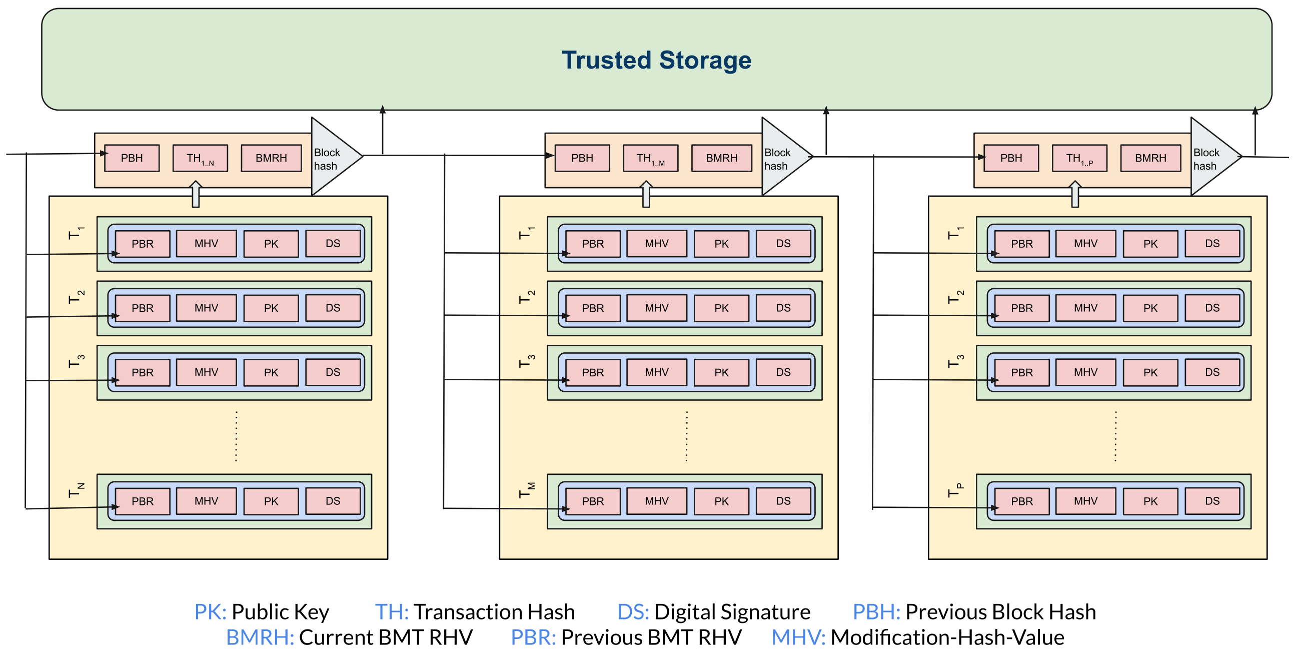


Figure 5.6: PerfCredB block creation

nating the previous block BMT RHV, current transaction MHV, public key, and the digital signature. The `transaction-hash` value calculated is fed into the database, and the server commits the transaction. At periodic intervals, the block process begins and waits for all the transactions within the block period to commit. The block process then begins collecting all the rows modified by all the transactions within the block. The block BMT formation takes place by depending upon the previous block BMT structure, and further, the current block BMT RHV is calculated similarly to SecCred (section 4.2.1). It can be observed that, for each, Credereum, SecCred, ProgCred, PerfCredA, and PerfCredB, the BMT structure formation of the current block depends upon the previous block BMT structure, and hence remains identical. The `block-hash` value calculation of the current block in PerfCredB remains the same for PerfCredA, ProgCred, SecCred (section 4.2.3) and is shown in Figure 5.6. The accounting of the previous block `block-hash` value helps establish the ledger chain. The current block `block-hash` value calculated is stored in the database and the trusted storage. The client can further query the block's BMT formed and verify the results similar to SecCred (section 4.2.3).

The PerfCredB preserves all Credereum possessed blockchain properties and has been described in section 6.5. The architectural redesign in PerfCredB resulted in adjustments to the handling and providing proof for the client's provenance queries and is explained in section 7.4.

5.4.2 MHV Algorithm

In PerfCredB, let the transaction T fired by a client, modifies the rows row 1, row 2, row 3, ..., row N of a table in the Postgres engine. After Postgres processing, the control gets handed over to the PerfCredB engine. In PerfCredB, an MHV is being calculated iteratively for transaction T, as shown in Algorithm 1. The `hash`, `sub-hash`, and `previous` variable are initialized, and a loop is executed for each modified row. Within each loop, after computing the hash of row `key-id` (i.e., `key-hash`), a `sub-hash` value is calculated (by computing hash of concatenated row `key-id` and associated value). Finally, the `hash` variable is computed considering the previous computed `hash` value, `key-hash`, and `sub-hash` calculated for the modified row `key-id`. The `hash` value obtained with other associated details is fed into the CREDEREUM_MERKLIX table (with key_hash, sub_hash, and previous attributes added), and post updating `previous`, the loop resumes from the start. Now, once the loop gets executed for each modified row `key-id`, an MHV is computed. The MHV calculation accounts for the current transaction transaction-id and the calculated `hash`. The MHV computed with other associated details is fed into the CREDEREUM_MERKLIX table.

Algorithm 1 : MODIFICATION-HASH-VALUE Calculation

```
$hash = ‘’$
$sub\text{-}hash = ‘’$
$previous = None$
while $each\ row\ in\ \ row\ 1, row\ 2, row\ 3, ..., row\ N$ : do
    $key\text{-}hash = SHA256\ (row\ key\text{-}id)$
    $sub\text{-}hash = SHA256\ (row\ key\text{-}id + ‘:’ + row\ value)$
    $hash = SHA256\ (hash + key\text{-}hash + sub\text{-}hash)$
    $store\ (key\text{-}id,\ key\text{-}hash,\ value,\ sub\text{-}hash,\ hash,\ previous)$
    $previous = row\ key\text{-}id$
end while
$MHV = SHA256\ (hash + transaction{-}id)$
$store\ (MHV,\ previous)$
```

Unlike ProgCred, which uses the transaction's TMT RHV in `transaction-hash` value calculation, the PerfCredB uses MHV in the `transaction-hash` value calculation. The MHV calculation complexity is simpler compared to the TMT formation and TMT RHV calculation.

5.5 Performance Evaluation

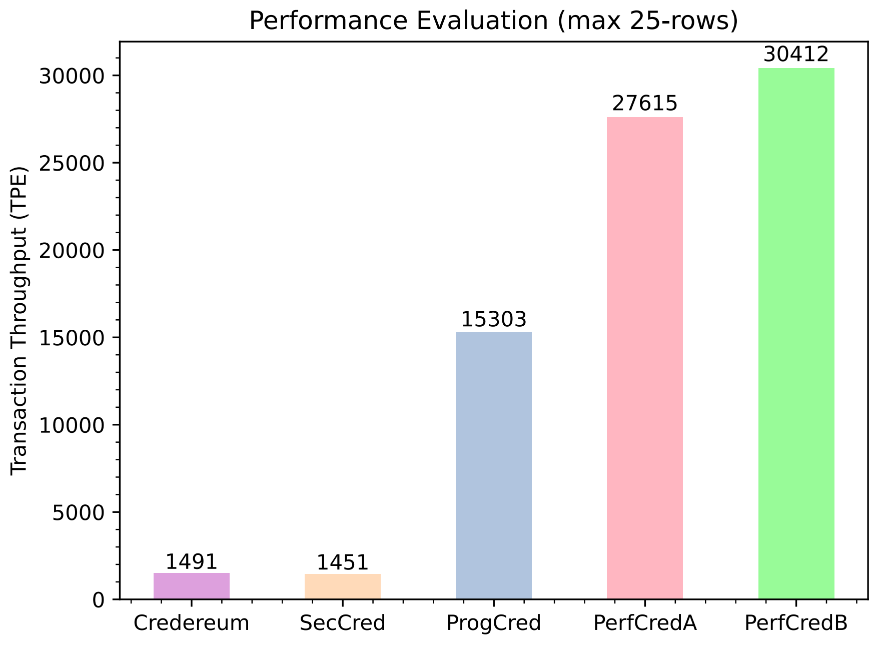


Figure 5.7: Comparative TPE Analysis

The Credereum, SecCred, ProgCred, PerfCredA, and PerfCredB performances were evaluated with 10 terminals firing to a database with hundred thousand entries and are shown in the Figure 5.7. Each transaction fired by terminals to the database modifies at max 25-rows. Each writes get accompanied by a read operation to perform verification of the modification proof generated. From the figure, we can perceive that the performance of SecCred lies approximately equal to the Credereum performance. The primary reason is the low-lying processing time to generate the SHA-256 hash, and due to which no heavy dip gets witnessed in throughput rate. The ProgCred with programming modifications achieves a significant boost to the system performance. The PerfCredA with independent TMT lightens the transaction process work in TMT creation, TMT RHV computation, modification proof generation, and verification, leading to a positive impact on performance improvements. A further performance rise can be witnessed with MHV computation in PerfCredB. It has been observed that with varying settings, PerfCredB beats PerfCredA in achieving better system performance.

5.6 Summary

The Credereum has shown massive magnitude degradation in committed-transaction throughput when compared to PostgreSQL. Methodologies were developed focusing on programming and algorithmic modifications to enhance performance. ProgCred focuses on programming optimization to the maximum called CREDEREUM_LONGEST_PREFIX() function. PerfCredA, and PerfCredB, deals with relaxing and improving the architecture without violating blockchain semantics. PerfCredA explains removing the previous block BMT dependency in the per TMT formation, while PerfCredB simplifies further by using MHV calculation in generating the `transaction-hash`. The MHV calculated summarizes the modification made by the transaction in PerfCredB. The performance results of each ProgCred, PerfCredA, and PerfCredB have been additionally discussed.

Chapter 6

Blockchain Semantics Retention

The chapter explains the blockchain properties retention by the Credereum proposed versions, responsible for functionality addition and performance improvements. SecCred, ProgCred, PerfCredA, and PerfCredB are explained to maintain the Credereum held blockchain properties, namely immutability, verification, authentication, and non-repudiation. The retention and correctness of the blockchain properties remain crucial for the proposed versions to be termed valid.

6.1 Introduction

Credereum, possessing enriching features (section 2.2), needed improvements seeking security advancement and performance hike. The functionality enhancement made by displaying the TMT and BMT in hash forms provides resistance to the malicious activities and is dealt with in SecCred (section 4.2). The changes in Merkle tree `node-hash` computation in SecCred is explained to retain the Credereum held blockchain properties. Further, ProgCred (section 5.2), dealing with programming alterations to the CREDEREUM_LONGEST_PREFIX() function has been addressed. PerfCredA (section 5.3) with dependency removal on previous block BMT per TMT formation, and PerfCredB (section 5.4) instead of TMT formation using MHV calculation (section 5.4.2), are explained for non-violating any Credereum held blockchain properties.

6.2 SecCred: Blockchain Properties

The following details the blockchain properties held by the SecCred, which deals with the functionality improvement:

1. **Immutability**: The SecCred, designed with functionality advancement to the Credereum, proposes changes in the `node-hash` computation of TMT/BMT nodes (section 4.2.1). Each TMT/BMT formed in SecCred depends upon the previous block BMT structure. Unlike Credereum, the internal nodes' `node-hash` computation of TMT/BMT considers the children `node-key` in SHA-256 hash form. For each UPDATE/INSERT/DELETE made by transaction T, a leaf node in TMT exists, and TMT RHV gets computed. The TMT RHV is used in the `transaction-hash` computation. Hence, whenever a fraudulent person modifies any TMT leaf nodes, then the TMT RHV and transaction T `transaction-hash` changes. With alteration in `transaction-hash` value, the `block-hash` value of block B and all forward blocks gets changed. Even if the fraudulent person gets successful in re-computing the `block-hash` value of block B and all forward blocks, the computed `block-hash` remains different from the stored original `block-hash` digest at the immutable trusted storage. Hence, any retroactive modification made to the database contents by a fraudulent person can be easily detected. Thus, SecCred with functionality improvements ensures the immutability property.

2. **Verification**: In SecCred, for every transaction fired by the client, a modification proof, namely old Merkle proof and new Merkle proof, is generated by the server to the client. The old Merkle proof shows the old data values of rows, UPDATED/DELETED by the client, using the previous block B-1 BMT. For each INSERT, the old Merkle proof also proves the non-existence of such rows in the database. The new Merkle proof shows the transaction T TMT, displaying each UPDATE/INSERT/DELETE performed by transaction T. The client first verifies the old Merkle proof BMT by computing each nodes' `node-hash` value and confirms with the given. With the nodes' `node-hash` computation logic changes in SecCred, the client needs to calculate the nodes' `node-hash` with the proposed technique in SecCred; else, it will remain unmatched. The client verifies the new Merkle proof TMT by computing and matching the nodes' `node-hash` with the provided. For computing the TMT nodes' `node-hash`, the client needs to use the proposed technique in SecCred. Further, the orphan nodes check is made in BMT and TMT displayed. The client verifies the supporting nodes of old Merkle proof BMT with the respective supporting nodes of new Merkle proof TMT and vice-versa. If the match occurs, the client confirms the modifications from old Merkle proof BMT leaf nodes to the respective new Merkle proof TMT leaf nodes. If found valid, the client validates the transaction by signing with his digital signature. In SecCred, the client has also been given the provision of provenance queries with added search space optimization (section

4.3, section 7.2). SecCred, thus retains the verification property.

3. **Authentication**: In SecCred, for every transaction fired, the client needs to submit the digital signature to the server to authenticate the modifications. The digital signature is generated by concatenating the old Merkle proof BMT RHV and new Merkle proof TMT RHV and signing using the client's private key. The server verifies the digital signature, and if found valid, commits the transaction; else, it aborts the transaction. Hence, SecCred preserves the authentication property.

4. **Non-repudiation**: In SecCred, the client post verifying the old Merkle proof, and the new Merkle proof needs to authenticate the transaction by signing with his digital signature. The digital signature uses the client's private key. Hence, a situation can never occur where a client can deny the authorship or the transaction's validity being fired by him. Thus, SecCred holds the non-repudiation property.

   Hence, with the above analysis, SecCred retains all the blockchain semantics held initially by the Credereum.

6.3 ProgCred: Blockchain Properties

The following details the blockchain properties held by the ProgCred, which deals with the programming modifications:

1. **Immutability**: In ProgCred, whenever a transaction gets fired, a TMT is developed depending upon the previous block BMT structure. The TMT formation requires determining the longest common prefix between the two given `key` bit-strings. The CREDEREUM_LONGEST_PREFIX() function (Table 5.1) initially in SQL was re-written in C to output the longest common prefix using byte-level comparison and bit-wise operations for the unmatched byte. The output of the CREDEREUM_LONGEST_PREFIX() C variant remains the same as the initial but holds quick computation. Hence, similar to Credereum (explained section 2.4), with changes in transaction T details of block B by a fraudulent person, the TMT RHV and `transaction-hash` value of transaction T alters. In ProgCred, the `transaction-hash` value is accounted for in the block B `block-hash` computation. Hence, with changes in the `transaction-hash` of transaction T, the `block-hash` of block B and all forward blocks gets changed. However, the newly

calculated `block-hash` digest of block B and all the forward blocks will differ from the stored original `block-hash` digest at the immutable trusted storage. Hence, any retroactive modification of the database contents by a fraudulent person can be easily detected. Thus, ProgCred ensures and retains the immutability property.

2. **Verification**: In ProgCred, with programming modifications proposed to the CREDEREUM_LONGEST_PREFIX() function, the architecture remains the same as of Credereum. Hence, in ProgCred, whenever a client fires a transaction to the Postgres engine, an old Merkle proof and the new Merkle proof is output. The client verifies the old Merkle proof shown by computing each node's `node-hash` value and BMT RHV and matches the evidence. The client further verifies the new Merkle proof TMT RHV, orphan nodes, the supporting nodes and checks the modifications. If the modification made is correct, the client validates the transaction by signing with his digital signature. In ProgCred, the client has also been given the provision of provenance queries with added search space optimization (section 4.3, section 7.2). Thus, ProgCred retains the verification property.

3. **Authentication**: In ProgCred, the modification made by programming changes to the CREDEREUM_LONGEST_PREFIX() function has no impact on the authenticity feature possessed by the Credereum. Hence, in ProgCred, for every transaction being fired, after verification of proof's generated by the server, the client needs to authenticate by signing with his digital signature, using a private key. The server verifies the digital signature, and if found valid, commits the transaction else, aborts. Hence, ProgCred holds the authentication property.

4. **Non-repudiation**: In ProgCred, with modifications to CREDEREUM_LONGEST_PREFIX() function, the non-repudiation feature remains unchanged. In ProgCred, for each transaction fired, the client needs to submit the digital signature, reflecting the client's approval in the modifications made by the server to be valid, and signs using its private key. Hence, a situation can never occur where a client can deny the authorship or the validity of the transaction fired by him. Therefore, ProgCred retains the non-repudiation property.

Thus, with the above explanation, ProgCred retains all the blockchain semantics held initially by the Credereum.

6.4 PerfCredA: Blockchain Properties

The following details the blockchain properties possessed by PerfCredA, which deals with the algorithmic modifications:

1. **Immutability**: In PerfCredA, for every transaction T being fired by the client, a TMT is developed without depending upon the previous block BMT. The TMT leaf nodes account for each of the UPDATE/INSERT/DELETE performed by the transaction T. The TMT RHV is computed and considered in the transaction T `transaction-hash` value calculation. Hence, whenever a fraudulent person modifies any TMT leaf nodes representing UPDATE/INSERT/DELETE, then the TMT RHV and transaction T `transaction-hash` changes. Moreover, the `transaction-hash` value of transaction T is accounted for block B `block-hash` calculation. Hence, with changes in the transaction T `transaction-hash`, block B and all forward block's `block-hash` gets changed. Even if the fraudulent person succeeds in validating the chain by re-calculating the `block-hash` digest of all the subsequent blocks, the newly calculated `block-hash` digest will differ from the stored original `block-hash` digest at the immutable trusted storage. Hence, any retroactive modification of the database contents by a fraudulent person can be easily detected. Thus, PerfCredA with algorithmic improvements ensures and retains the immutability property.

2. **Verification**: In PerfCredA, with independent TMT formation for each transaction fired, the verification logic for the old Merkle proof and new Merkle proof changes. Now, for the old Merkle proof generated, the client first obtains each node's `node-hash` and BMT RHV. The `node-hash` computed is matched with the evidence displayed. If valid, the client likewise obtains and verifies the new Merkle proof TMT RHV. Orphan node checks are made in the BMT and TMT shown. In PerfCredA, since the TMT doesn't depend upon the previous block BMT structure, the TMT doesn't retain any hanging nodes. Hence, no comparison of supporting nodes is made, with old Merkle proof BMT, differing Credereum, SecCred, and ProgCred. Finally, the client for each UPDATE/DELETE checks the validity of the modifications from the leaf nodes of the old Merkle proof BMT to the respective leaf nodes of the new Merkle proof TMT. The client further checks for each INSERT the value in new Merkle proof TMT being made by T. If found valid, the client validates the transaction by signing with his digital signature. In PerfCredA, the client has also been given the provision of provenance queries with added search space optimization (section 4.3, section 7.3). Thus, PerfCredA holds the verification property.

3. **Authentication**: The client in PerfCredA, similar to Credereum, after verification of the old Merkle proof and new Merkle proof, needs to authenticate the transaction with his digital signature using a private key. The digital signature gets generated, concatenating the old Merkle proof BMT RHV and new Merkle proof TMT RHV. The TMT RHV summarizes the modifications made by transaction T and has been validated by the client. The server verifies the digital signature, and if found valid, commits the transaction; else, it aborts the transaction. Hence, PerfCredA retains the authentication property.

4. **Non-repudiation**: In PerfCredA, with algorithmic modifications proposing the independent TMT formation per transaction, it doesn't led changes in the client's digital signature submission to the server to commit the transaction. The digital signature generated reflects the client's approval in the server's modifications to be termed valid and it uses the client's private key. Hence, a situation can never occur where a client can deny the authorship or the validity of the transaction being fired by him. Thus, PerfCredA preserves the non-repudiation property.

   Hence, with the above analysis, PerfCredA retains all the blockchain semantics retained initially by the Credereum.

6.5 PerfCredB: Blockchain Properties

The following details the blockchain properties being retained by PerfCredB, which deals with algorithmic modifications:

1. **Immutability**: For every transaction being fired in PerfCredB, unlike the Credereum/SecCred/ProgCred/PerfCredA, no TMTs are formed. Instead, an MHV (section 5.4.2) gets calculated, which accounts for each UPDATE/INSERT/DELETE performed by the transaction T. The MHV gets calculated using the SHA-256 hash and summarizes the modifications made. Whenever a fraudulent person modifies any of the UPDATE/INSERT/DELETE performed by T, the MHV calculation gets changed. The MHV gets accounted for in the `transaction-hash` computation; hence any change in MHV leads the transaction T `transaction-hash` value to get changed. The `transaction-hash` of transaction T is further accounted for block B `block-hash` calculation. With changes in the `transaction-hash`, the `block-hash` of block B and all the forward blocks get changed. Even though, if the fraudulent person gets successful in re-computing the

`block-hash` of block B and all forward blocks, and validating the chain, the newly calculated `block-hash` digest of block B and all the forward block's will differ from the stored original `block-hash` digest at the immutable trusted storage. Hence, any retroactive modification of the database contents by a fraudulent person can be easily detected. Thus, PerfCredB ensures and retains the immutability property.

2. **Verification**: In PerfCredB, the server generates an old Merkle proof and the New-Row-Value proof for every transaction fired. The old Merkle proof uses the previous block BMT to display and prove the row's old values, which were being UPDATED/DELETED by the client. For each INSERT, a non-existence proof gets shown. The New-Row-Value proof shows the new values for each UPDATE/INSERT/DELETE, being performed by transaction T, and summarizes the modification by displaying the MHV. The client first verifies the leaf nodes of old Merkle proof BMT by calculating each node's `node-hash` and BMT RHV. The `node-hash` gets matched with the evidence shown. If valid, the client further checks the value of new modified row from New-Row-Value proof and computes and compares the MHV with the displayed. The client further, for each UPDATE/DELETE, checks the modifications from the leaf nodes of old Merkle proof BMT to the respective entry in the New-Row-Value proof and vice-versa. The client for each INSERT checks the respective entry in the New-Row-Value proof. If found valid, the client validates the transaction by signing with his digital signature. In PerfCredB, the client has also been given the provision of provenance queries with quick computation and added search space optimization (section 4.3, section 7.4). Thus, PerfCredB retains the verification property.

3. **Authentication**: The client in PerfCredB, after the server's modifications are verified, the client needs to authenticate the transaction by signing with his digital signature. The digital signature is computed by concatenating the old Merkle proof BMT RHV and the New-Row-Value MHV and signing with the client's private key. The server verifies the digital signature, and if found valid, commits the transaction; else, it aborts the transaction. Hence, PerfCredB preserves the authentication property.

4. **Non-repudiation**: The replacement of TMT formation per transaction with MHV computation per transaction in PerfCredB, does not change the digital signature submission requirement to commit the transaction. The digital signature generation requires the client's private key. Hence, a situation can never occur where a client can deny the authorship or the validity of the transaction being fired by him. Thus, PerfCredB holds the non-repudiation property.

Therefore, with the above analysis, PerfCredB retains all the blockchain semantics held initially by the Credereum.

6.6 Summary

The SecCred dealing with functionality enhancement, and ProgCred, PerfCredA, PerfCredB dealing with performance improvements, has been shown to retain the immutability, verification, authentication, and non-repudiation property, held initially by the Credereum. The immutability property assures any malicious activity performed at the per transaction/block level can be easily detected. The verification property explain the displaying of the modification proofs and methodology indulge in verifying and validating the generated proof. The authentication property involves submitting a digital signature using the client's private key, which helps support the non-repudiation property.

Chapter 7

Provenance Handling

With provenance, the queried data authenticity and source can be proved using the auditable records of the modifications made from the *Genesis* block. The chapter deals with explaining the methodology involved in handling the provenance query raised by the client. The Credereum, SecCred dealing functionality additions, and ProgCred holding the programming improvements, deals provenance query likewise. Further, the provenance handling by PerfCredA and PerfCredB retaining algorithmic modifications has been addressed. The algorithmic improvements lead to alterations in the handling and output generation of the provenance query. Each section details sample results of the provenance query output in the respective system and has been analyzed and compared.

7.1 Introduction

Provenance query handling plays an essential role in the blockchain system design. With this, a particular row's existence gets proved by displaying and providing evidence for the historical set of modifications made. Credereum, being a blockchain system, provides this functionality through which the client can query and verify the history of alterations performed on a particular row/set-of-rows. The trusted storage, being immutable, plays an essential role in proving the modifications. The different phases of changes made to Credereum all provide the provenance query handling functionality. SecCred and ProgCred, retaining the same architecture as Credereum, establishes the same methodology but with stated provenance search optimizations. The PerfCredA and PerfCredB provide provenance query functionality, but due to variation in architectures, their provenance output differs. The autonomous TMT creation in PerfCredA and MHV (section 5.4.2) computation in PerfCredB led to changes in the algorithm to handle and verify provenance proof.

7.2 Credereum/SecCred/ProgCred

The client in Credereum, SecCred, and ProgCred has been facilitated to query the modification history of a row/set of rows with evidence through provenance query. The methodology involved in displaying the provenance query results for Credereum/SecCred/ProgCred remains the same, but with added functionality in SecCred and ProgCred, as explained in section 4.3. SecCred and ProgCred with functionality advancement and programming changes, respectively, have no variation in their architecture; hence their provenance engine remains same.

7.2.1 Provenance Explanation

In Credereum, SecCred, and ProgCred, the CREDEREUM_MERKLE_PROOF() function handles the client's provenance query. For each row `key-id` R being queried, the mentioned CREDEREUM_MERKLE_PROOF() function searches and outputs all the [key, transaction_id, block_num, children, leaf, hash, value] attribute entries by probing the CREDEREUM_MERKLIX relation (Table 1.2), where the key is R and transaction_id is NOT NULL. The CREDEREUM_MERKLE_PROOF() function for each entry above, with transaction_id as T, block_num as B, and key as K:

1. outputs [key, block_num, transaction_id, children, leaf, hash] details by searching the CREDEREUM_MERKLIX table for transaction_id equal to T, block_num equal to B, and key not equal to K. The details fetched belongs to the path nodes (nodes from leaf nodes to the root node) and supporting/hanging nodes of transaction T TMT.

2. outputs [key, block_num, transaction_id, children, leaf, hash, value] details by searching the CREDEREUM_MERKLIX table for the transaction_id as NULL, block_num equal to B, and key equal to K. The details fetched belongs to the block B BMT leaf nodes. Thus for a hit block, it shows the value (i.e., `node-value`) of modified K present in the BMT.

3. outputs [key, block_num, transaction_id, children, leaf, hash] details by searching the CREDEREUM_MERKLIX table for the transaction_id as NULL, block_num equal to B, and key not equal to K. The details fetched belongs to the path nodes (nodes from leaf nodes to the root node), and the supporting nodes of BMT for the given block number.

For each hit block B, the server also generates TMT of the rest transactions (where transaction_id $\neq$ T). For block M, where a queried row `key-id` K has not been modified, the server additionally generates proof by displaying the block M BMT and all the related TMT details.

Thus for a queried row R, the server for each block generates the BMT and each TMT to prove whether R has been modified/not. While displaying the proofs, the minimal disclosure retention property (section 2.6) is maintained.

The client queries and stores the public CREDEREUM_BLOCK table and CREDEREUM_TX_LOG table entries. For each transaction-id, the client verifies the previous block BMT RHV details. With the CREDEREUM_TX_LOG results, the client computes and verifies each transaction-id X `transaction-hash` value, as:

- `transaction-hash` = SHA256 (previous block BMT RHV + transaction X TMT RHV + public key + digital signature)

Once the `transaction-hash` of each transaction-id matches, the client for each block Y verifies whether the previous block Y-1 `block-hash` accounted is valid/not. If valid, the client computes the `block-hash` of Y and matches it with the details fetched from the CREDEREUM_BLOCK table and trusted storage. The client can compute and verify the `block-hash`, as:

- `block-hash` = SHA256 (previous block Y-1 `block-hash` + all constituting transaction's `transaction-hash` + block Y BMT RHV)

With provenance proof displaying the row `key-id` (key) r_1 was modified by the transaction T_t of block number B_k, the client using the provenance results can verify its authenticity. To validate that the modification was made by transaction T_t of block number B_k, the client can compute the transaction T_t TMT each node's `node-hash` value (i.e., hash) and verify RHV with the proof provided. If found valid, the computed RHV is matched with the verified details from the CREDEREUM_TX_LOG table. Similarly, for the rest transaction-id of B_k, the client verifies and expects the *same* old value of r_1.

From the proof provided, the client needs to further verify the BMT of block B_k, to see the accounting of row r_1 in the block B_k formation and `block-hash` calculation. The client can compute each BMT node's `node-hash` value (i.e., hash) and match it with the provenance results. The computed BMT RHV is used to calculate the block B_k `block-hash` value and compare with the verified CREDEREUM_BLOCK table entries. If the value matches, this signifies that row r_1 was being modified and used in the block formation of block B_k. Likewise, for block M, which doesn't have any modification to row r_1, the client can verify the block M BMT and each associated transaction-id TMT details, expecting the *same* old value of r_1.

With the given output from the CREDEREUM_MERKLE_PROOF() function, a client has verified that the transaction T_t of block number B_k has modified row r_1 and has been used in the BMT formation and `block-hash` calculation for block B_k. Similarly, the client can verify the rest queried rows result from the provenance output.

7.2.2 Sample Results

In Credereum/SecCred/ProgCred, let's consider there are four transactions T1, T2, T3, and T4, which were being fired by the clients, one after the other, and on commit, were used in the block formation of block 1, block 2, block 2, and block 3, respectively. Each transaction, T1, T2, T3, and T4 remains the same, as defined in section 4.2.5. The transactions either INSERT/UPDATE the WAREHOUSE table, or the CUSTOMER table, or both. For each UPDATE/INSERT/DELETE, a node in the TMT and BMT is created, with the new `node-value` entries. For DELETE, the `node-value` is NULL, for INSERT, the `node-value` is the new fed data in the relation, and for an UPDATE, the `node-value` is the new modified data.

For a TMT, we have the following:

1. The green nodes display provenance queried rows (leaf nodes) within a transaction TMT.
2. The red nodes exhibit the TMT internal nodes.
3. The blue and yellow nodes represent the supporting and hanging nodes, respectively, which gets used to compute and verify the parent node's `node-hash`.
4. The dashed edges mean the child details can be fetched from the previously shown TMT/BMT details from the provenance results. Hence duplicate entries are avoided.

For a BMT, we have the following:

1. The green nodes illustrate provenance queried rows (leaf nodes) within a block BMT.
2. The red nodes display the BMT internal nodes.
3. The blue nodes represent the supporting nodes used for calculating and verifying the `node-hash` of the parent node.
4. The dashed edges mean the child details can be fetched from the previously shown BMT details from the provenance results. Hence duplicate entries are avoided.

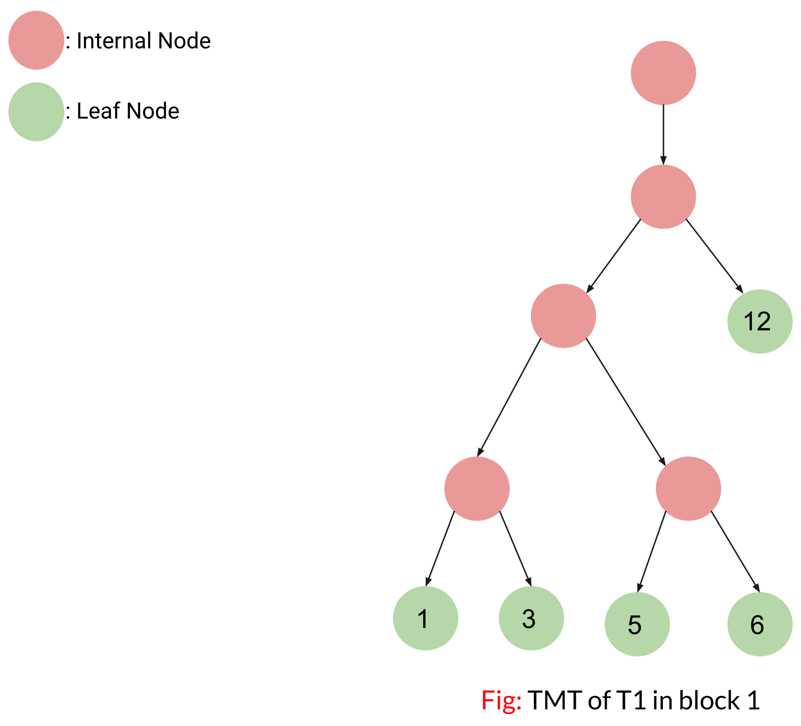


Figure 7.1: Provenance outcome for transaction T1 of block 1

With the above, say the client for each transaction T1, T2, T3, and T4 before commit, queries the provenance for respective each modified (inserted/updated/deleted) row `key-ids`:

1. **Block 1, Transaction T1**: For the transaction T1 fired by the client, a TMT gets formed, and an old Merkle proof and new Merkle proof are generated. For the provenance proof of the modified row `key-ids`, a block 0 entry and the outcome shown in Figure 7.1 are displayed. Now, with block 1 being the first block, there are no previous modifications to the rows of the WAREHOUSE table with id equal to 1, 5, 3, 6, and 12. Further, the provenance output displays the current transaction T1 TMT to output the new data values after modification and is shown in Figure 7.1. The TMT's leaf node show the WAREHOUSE rows, with id equal to 1, 5, 3, 6, and 12 modified. The client can verify the modifications, and if found correct, digitally signs the transaction.

2. **Block 2, Transaction T2**: For the transaction T2 fired by the client, a TMT gets formed, and an old Merkle proof and new Merkle proof are generated. The provenance outcome for the modified row `key-ids` queried by the client is shown in Figure 7.2. Additionally, an initial block 0 entry also gets output. Now, the transaction T2 modifies

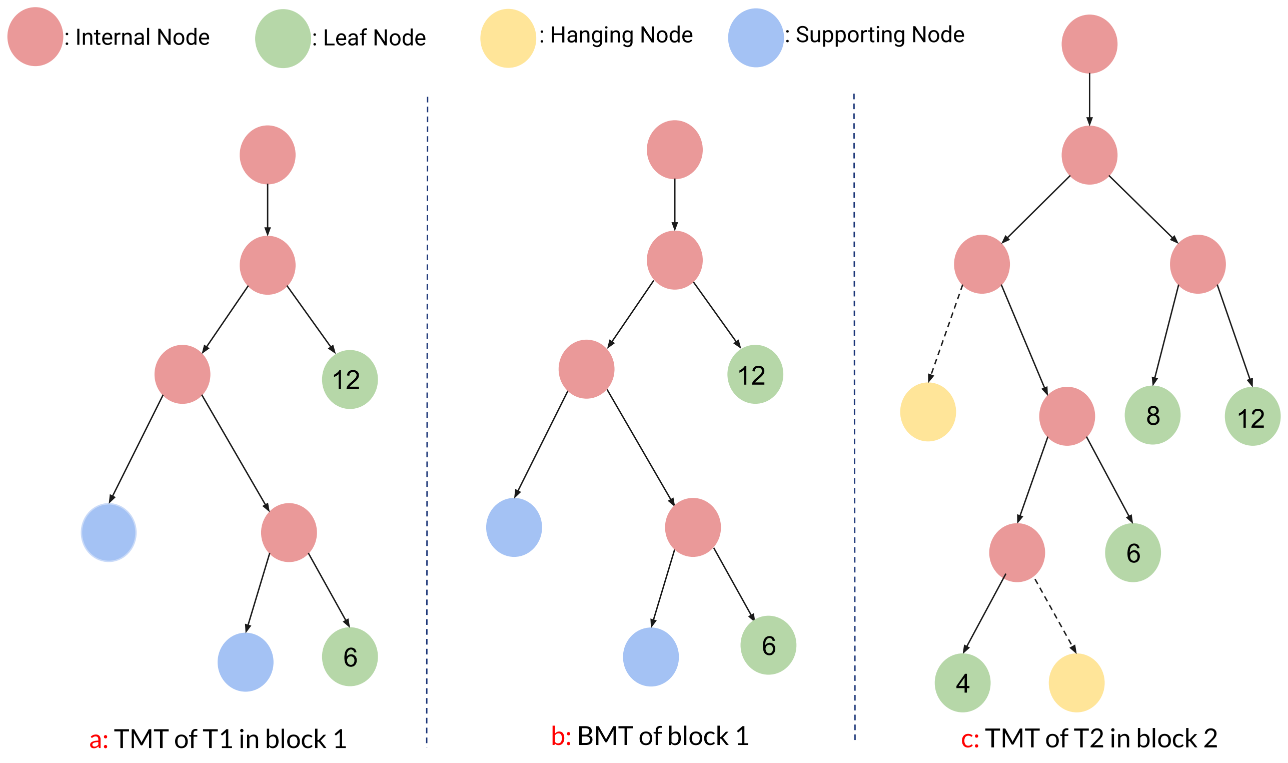


Figure 7.2: Provenance outcome for transaction T2 of block 2

WAREHOUSE rows with id values 6 and 12, which was earlier modified by the transaction T1 at block 1. So the provenance query displays the commonly modified rows, i.e., WAREHOUSE rows with id values 6 and 12, of the TMT of T1 and the BMT of block 1, as shown in section 'a' and 'b' of Figure 7.2, respectively. The BMT of block 1 displays the inclusion of WAREHOUSE rows with id values 6 and 12 in the block formation. For the TMT shown, the client can compute the TMT RHV and the `transaction-hash` value. Given a BMT, the client can compute the BMT RHV and `block-hash` value and match it with the database and the trusted storage. As shown in the section 'c' of Figure 7.2, the provenance proof also outputs the current transaction T2 TMT and displays the new modified data values. The leaf nodes of TMT of T2 shows the WAREHOUSE rows with id equal to 6, 12, 8, and 4 modified. The TMT of T2 depends upon the previous block, i.e., block 1 BMT structure, and hence holds hanging nodes (yellow nodes) attached to it. The client, after verifying proofs and changes, sends a digital signature to the server.

3. **Block 2, Transaction T3**: For the transaction T3 fired by the client, a TMT gets formed and, an old Merkle proof and new Merkle proof are generated. The provenance outcome for the past data values of the modified row `key-ids`, being queried by the client, includes

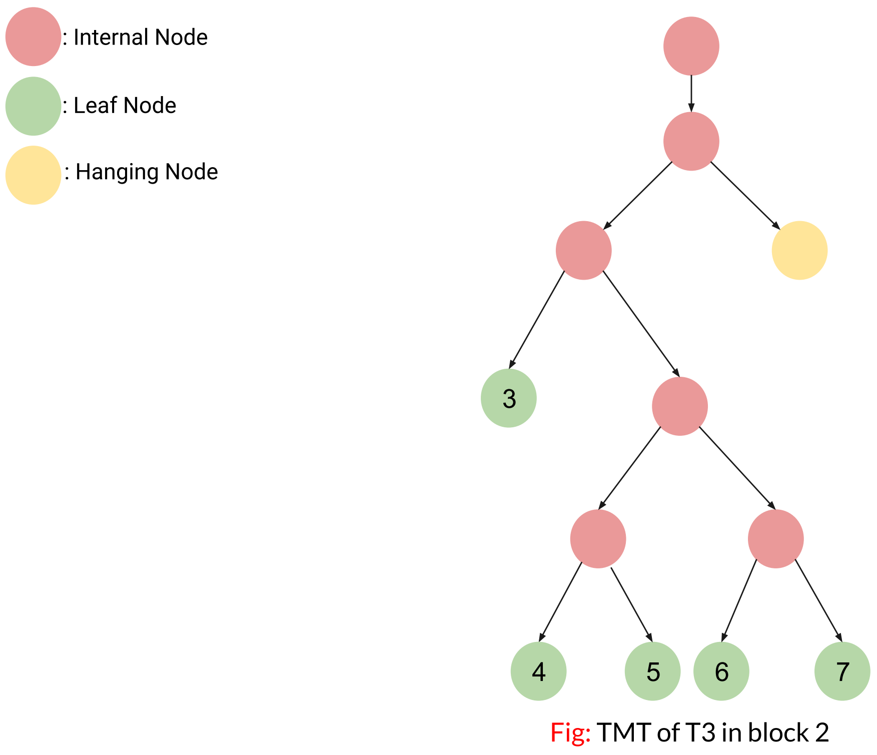


Figure 7.3: Provenance outcome for transaction T3 of block 2

block 0 entry, Merkle tree details of block 1, and the output shown in Figure 7.3. Now, the transaction T3 modifies the CUSTOMER table rows, which were not altered by any transaction earlier. Considering block 1, because of no common prefix with queried `key-ids`, only the root node and the child attached to the root node of TMT of T1 and BMT of block 1 are displayed. With the details output, the client can respectively compute T1 TMT RHV and block 1 BMT RHV. Further, using the computed T1 `transaction-hash` value and block 1 BMT RHV, the `block-hash` of block 1 can be calculated and matched with the database and the trusted storage. Further, the server also outputs the current transaction T3 TMT as shown in Figure 7.3. The leaf node entries of TMT show the CUSTOMER rows with id equal to 3, 4, 5, 6, and 7 modified. The TMT of transaction T3 depends upon the previous block, i.e., block 1 BMT structure, and hence has a hanging node (yellow node) attached to it. The client verifies the proofs and modifications, and if found valid, submits the digital signature to the server.

4. **Block 3, Transaction T4**: For the transaction T4 fired by the client, a TMT is formed and, an old Merkle proof and new Merkle proof gets generated. The provenance results for the preceding data values of the modified row `key-ids`, queried by the client, are

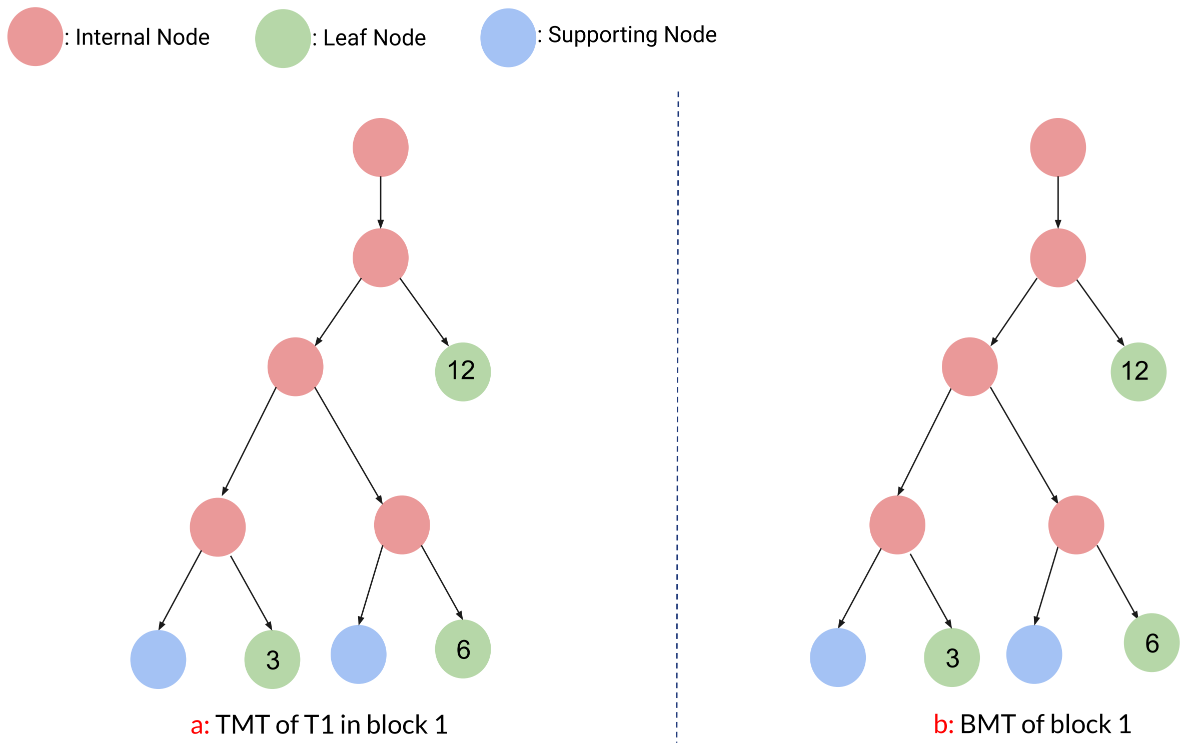


Figure 7.4: Provenance outcome for transaction T4 of block 3 (Part-1)

displayed in Figures 7.4 to 7.7. Additionally, an initial block 0 entry also gets output by the server. Now, transaction T4 alters WAREHOUSE rows with id equal to 3, 6, 12, 7, and 2, and the CUSTOMER rows with id equal to 3 and 7. Commencing the search from initial block, transaction T1 in block 1 has modified WAREHOUSE rows with id equal to 3, 6, and 12. As shown in section 'a' and 'b' of Figure 7.4, the TMT of T1 and BMT of block 1 are displayed, showing the value details of WAREHOUSE id entries equal to 3, 6, and 12.

In block 2, transaction T2 has modified WAREHOUSE rows with id equal to 6 and 12. Further, transaction T3 of block 2 has changed CUSTOMER rows with id equal to 3 and 7. From section 'a' and 'b' of Figure 7.5, the TMT of T2 and T3 is shown, respectively, with commonly modified rows with T4. The BMT of block 2 gets further displayed in Figure 7.6 to show the inclusion of commonly modified rows of T2 and T3 with T4 in the block 2 formation. The leaf nodes of BMT have entries written in red (CUSTOMER table modifications made by T3) and black (WAREHOUSE table modifications made by T2).

For each transaction's TMT shown, the client can compute the TMT RHV and the `transaction-hash` value. For each block's BMT shown, the client can compute the BMT RHV and `block-hash` value and match it with the database and the trusted

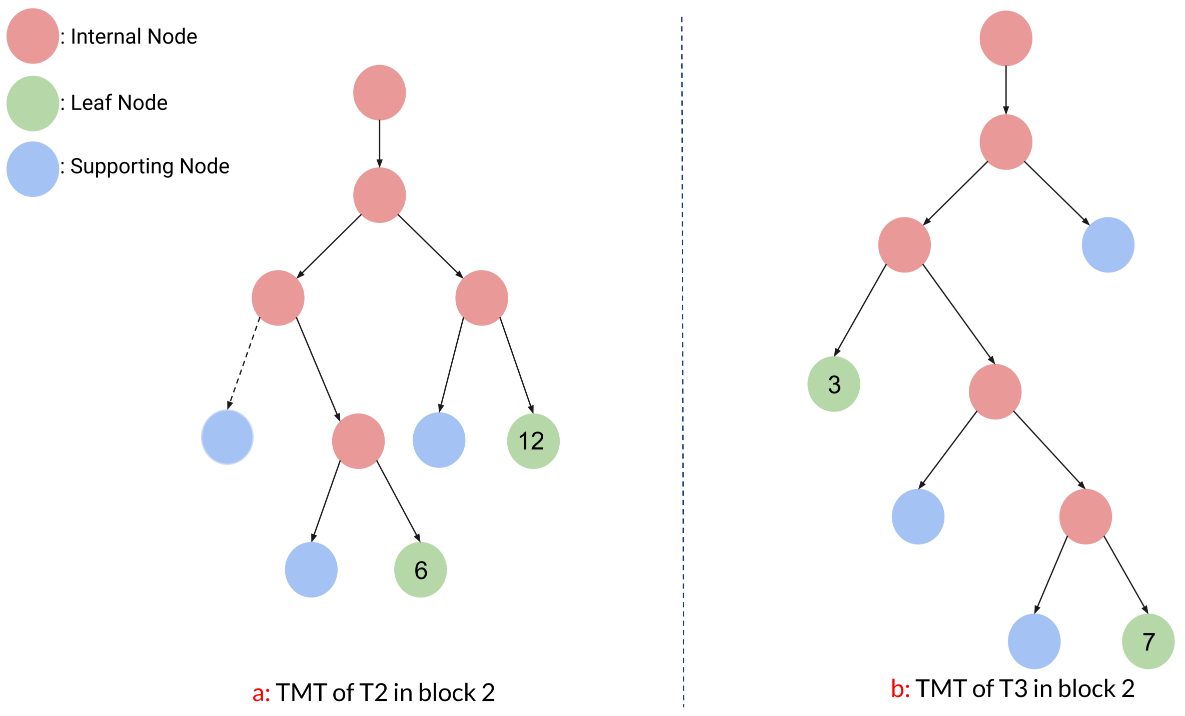


Figure 7.5: Provenance outcome for transaction T4 of block 3 (Part-2)

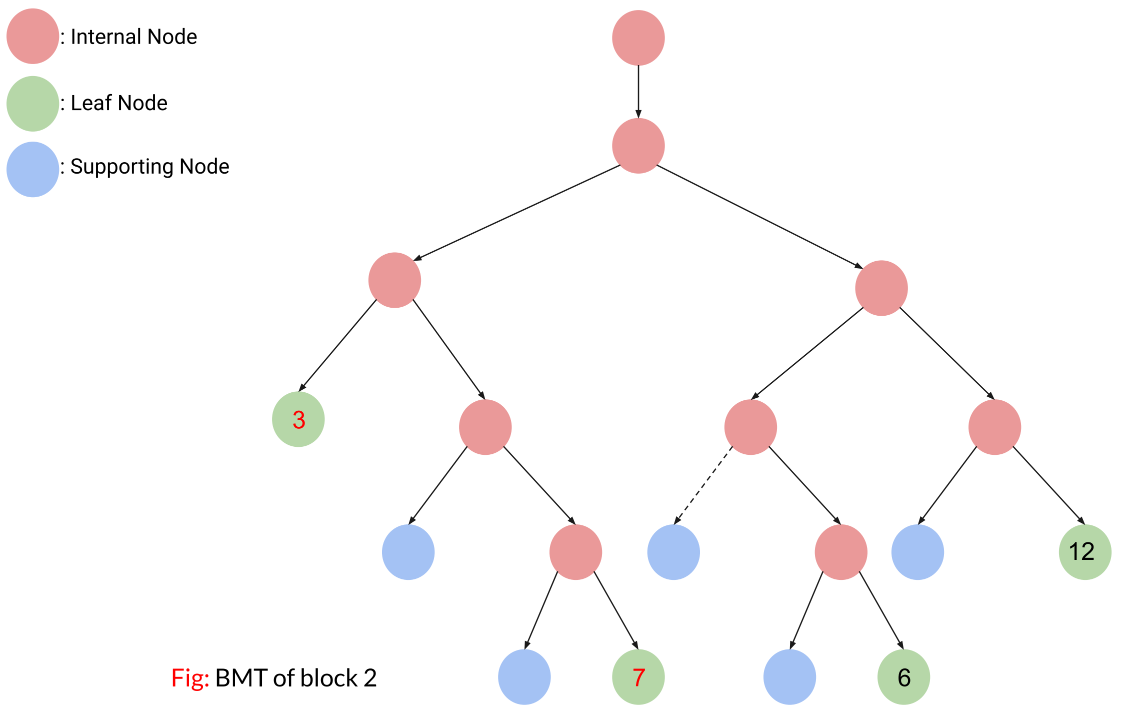


Figure 7.6: Provenance outcome for transaction T4 of block 3 (Part-3)

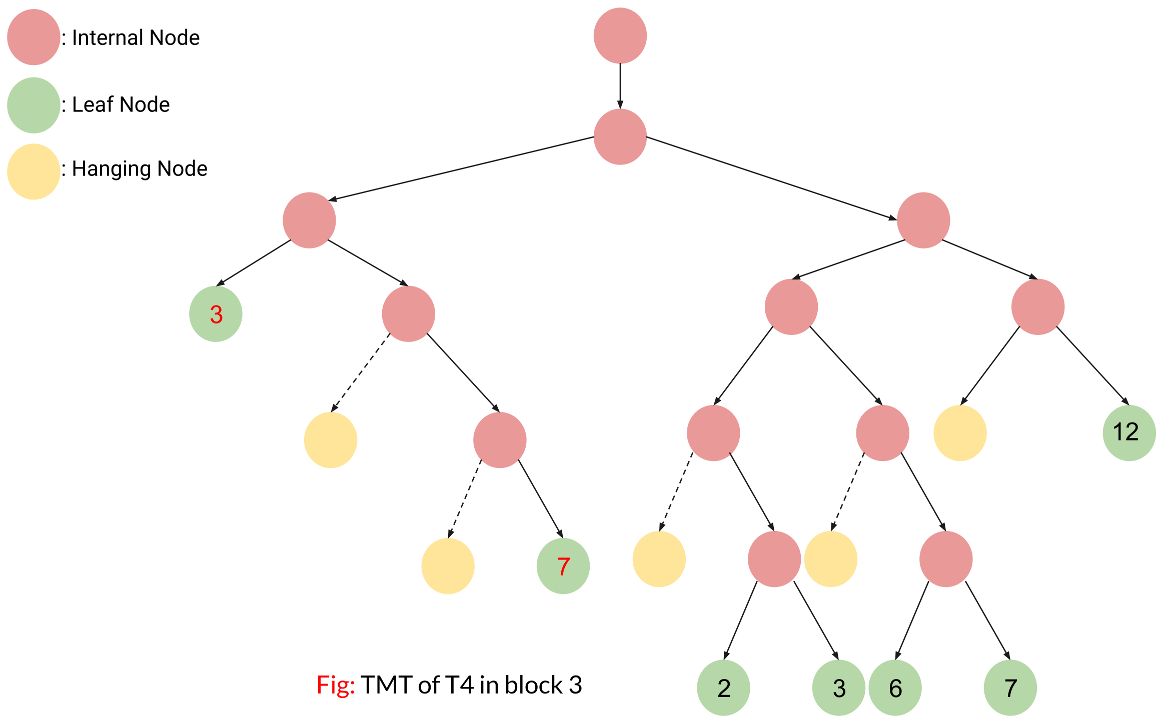


Figure 7.7: Provenance outcome for transaction T4 of block 3 (Part-4)

storage. The provenance proof results also display the current transaction T4 TMT as shown in Figure 7.7. The leaf node entries of TMT of transaction T4 show the CUSTOMER rows with id values 3 and 7 (red entries) and WAREHOUSE rows with id values 3, 6, 12, 7, and 2 (black entries) modified by T4. The TMT of T4 depends upon the previous block, i.e., block 2 BMT structure, and hence has hanging nodes (yellow nodes) attached to it. The client, after verifying the proofs and modifications, digitally signs the transaction.

7.3 PerfCredA

The PerfCredA, dealing with algorithmic changes, provides the clients with evidence for the queried rows' past modifications using provenance. The search space optimization, as is explained in section 4.3, is facilitated in PerfCredA. Any client can raise the provenance query by providing the set of row `key-ids` for which the proof is desired.

7.3.1 Provenance Explanation

The PerfCredA design has no previous block BMT dependency at the per transaction TMT formation. In PerfCredA, the provenance query is handled by the PROVENANCE_PROOF_PCA() function, which intakes the set of row `key-ids` for which the provenance is desired and the range block number's (section 4.3) with default as *Genesis* block and the current block. The PROVENANCE_PROOF_PCA() function output remains similar to section 7.2.1 but with variations in the TMT display. With the TMT structure modifications in PerfCredA, the TMT holds no hanging nodes. With its absence, the exploration of the TMT hanging nodes in the verification logic gets avoided. While probing provenance results in PerfCredA, if within a block, the queried row R has not been modified, then the block BMT is explored expecting the *same* old value of R, and the TMT's in the block gets surveyed expecting *no-entry* for row R. Further, in PerfCredA, the `transaction-hash`, and `block-hash` calculation formula remains the same. The client can likewise (section 7.2.1) verify the details from the public CREDEREUM_BLOCK and CREDEREUM_TX_LOG relations.

7.3.2 Sample Results

The transactions being fired by the clients sequentially to PerfCredA, be T1, T2, T3, and T4, which on commit will be accounted in block 1, block 2, block 2, and block 3 formations, respectively. The transaction details of T1, T2, T3, and T4 remain the same as defined in section 4.2.5. In PerfCredA, for each UPDATE/INSERT/DELETE, a node in the TMT and BMT is created with the new `node-value` entries. For DELETE, the `node-value` is NULL, for INSERT, the `node-value` is the new fed data in the table, and for an UPDATE, the `node-value` is the new modified data. In Credereum/SecCred/ProgCred, the per transaction TMT and per block BMT depend upon the previous block BMT structure. But, in PerfCredA, as we will see, a transaction's TMT will be constructed independently, i.e., without being dependent on the previous block BMT. Hence, there won't be any hanging nodes in any of the transaction's TMT. However, each block in the PerfCredA develops its structure depending upon the previous block BMT structure and will retain hanging nodes.

For a TMT, we have the following:

1. The green nodes exhibit provenance queried rows (leaf nodes) within transaction's TMT.
2. The red nodes display the TMT internal nodes.

3. The blue nodes represent the supporting nodes used to calculate and verify the `node-hash` of the parent node.

For a BMT, we have the following:

1. The green nodes exhibit provenance queried rows (leaf nodes) within block's BMT.
2. The red nodes represent the BMT internal nodes.
3. The blue nodes display the supporting nodes used to calculate and verify the `node-hash` of the parent node.
4. The dashed edges mean the child details can be fetched from the previously shown BMT details from provenance proof. Hence duplicate entries are avoided.

With above details, say the client for each transaction T1, T2, T3, and T4 before commit, fires the provenance query for each respective modified (inserted/updated/deleted) row `key-ids`:

1. **Block 1, Transaction T1**: For the transaction T1, fired by the client, a TMT gets formed, and an old Merkle proof and new Merkle proof are generated. The client, when queries the provenance for the row `key-id`'s modified, the block 0 entry and outcome shown in Figure 7.8 gets displayed. Now, since block 1 is the first data block, there are no previous modifications to the rows of the WAREHOUSE table with id equal to 1, 5, 3, 6, and 12. Further, the provenance displays the current transaction T1 TMT to exhibit the new data values after modification, shown in Figure 7.8. The TMT's leaf node entries show the WAREHOUSE rows with id equal to 1, 5, 3, 6, and 12 modified. We can observe that the transaction T1 TMT of PerfCredA shown in Figure 7.8 remains the same as of transaction T1 TMT in Credereum/SecCred/ProgCred shown in Figure 7.1. The reason is, in Credereum/SecCred/ProgCred, T1 being the transaction of block 1, it has no previous block's BMT to be dependent upon, hence designs autonomously. Therefore, in general, all the transactions of block 1 in Credereum/SecCred/ProgCred and PerfCredA will develop its TMT independently and remain the same. The client, after verifying the proofs, and modifications can digitally sign the transaction.

2. **Block 2, Transaction T2**: For the transaction T2 fired by the client, a TMT gets formed, and an old Merkle proof and new Merkle proof are generated. Now, for the provenance query fired by the client concerning the modified row `key-id`'s, the output

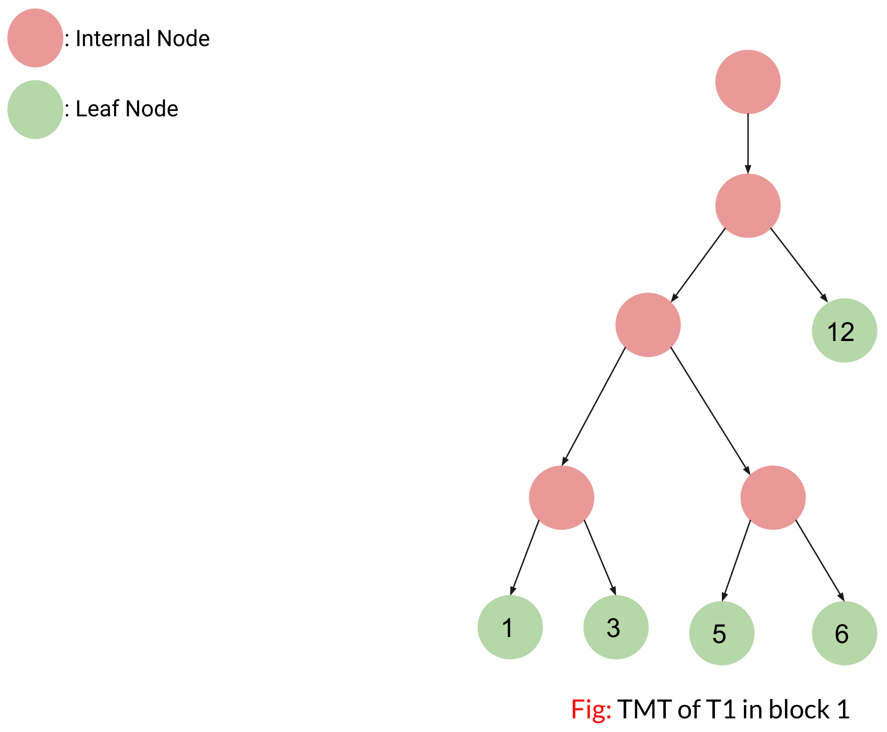


Figure 7.8: PerfCredA provenance outcome for transaction T1 of block 1

shown in Figure 7.9 is displayed. Additionally, an initial block 0 entry also gets shown. Now, the WAREHOUSE rows with id values 6 and 12 were earlier modified in block 1 by T1. Thus, the provenance query displays the value of the commonly modified rows, i.e., WAREHOUSE id equal to 6 and 12, of the TMT of T1 and the BMT of block 1, as shown in section 'a' and 'b' of Figure 7.9, respectively.

The BMT of block 1 shows the inclusion of WAREHOUSE rows with id values 6 and 12 in the block formation. From the TMT, the client can compute TMT RHV and `transaction-hash` value. Given a BMT, the client can compute BMT RHV and `block-hash` value and match it with the database and the trusted storage. The server for provenance proof, as shown in section 'c' of Figure 7.9, also outputs the current transaction T2 TMT to display the new modified data values. The leaf nodes of TMT of transaction T2 show the WAREHOUSE rows with id equal to 6, 12, 8, and 4 modified. The TMT of transaction T2 doesn't depend upon the previous block, i.e., block 1 BMT structure, and hence holds no hanging nodes. The TMT of transaction T2 of PerfCredA can be observed differently from the TMT of transaction T2 of Credereum/SecCred/ProgCred shown in Figure 7.2, which depends on the previous block BMT structure and contains the hanging

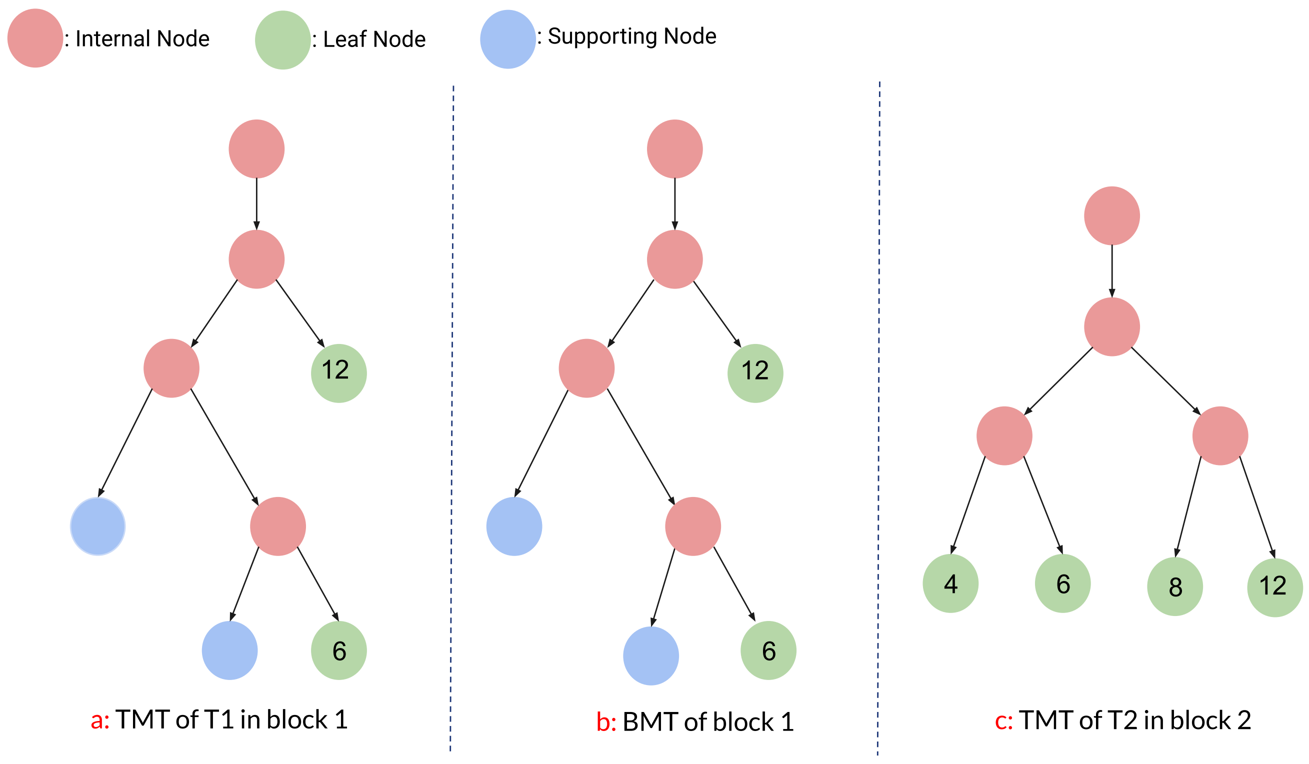


Figure 7.9: PerfCredA provenance outcome for transaction T2 of block 2

nodes (yellow nodes). The client, after verifying the transaction's modification and proof displayed, can authenticate the transaction.

3. **Block 2, Transaction T3**: For the transaction T3 fired by the client, a TMT gets formed, and an old Merkle proof and new Merkle proof are generated. Now, with regard to the provenance queried for the modified row `key-ids`, the generated result includes block 0 entry, block 1 Merkle details, and the output shown in Figure 7.10. The relation modified by transaction T3 was not altered by any transaction since the *Genesis* block. Now, concerning block 1, with no common prefix matching with the queried `key-ids`, the provenance outputs the root node and the child attached, for TMT of T1 and BMT of block 1. The client using details can compute TMT RHV and `transaction-hash` of T1, and further BMT RHV and `block-hash` value of block 1. The computed `block-hash` can be matched with the database and trusted storage. The server for provenance also displays the current transaction T3 TMT, shown in Figure 7.10. The leaf nodes of TMT of T3 show the CUSTOMER rows with id equal to 3, 4, 5, 6, and 7 modified. The TMT of transaction T3 doesn't depend upon the previous block, i.e., block 1 BMT structure, and hence has no hanging node attached to it. The TMT of transaction T3 of PerfCredA

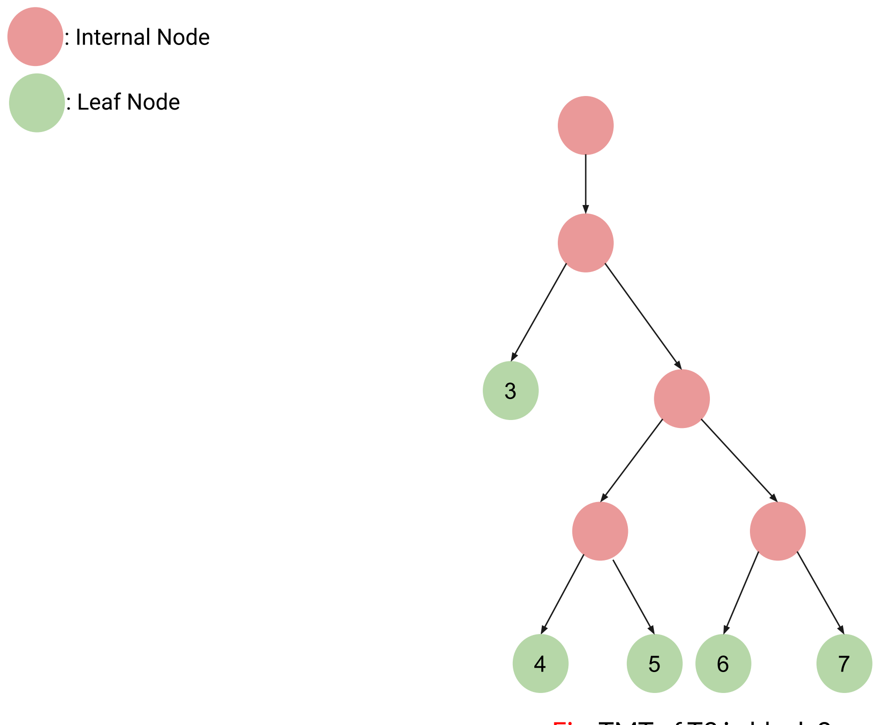


Fig: TMT of T3 in block 2

Figure 7.10: PerfCredA provenance outcome for transaction T3 of block 2

can be seen differently from the TMT of transaction T3 of Credereum/SecCred/ProgCred shown in Figure 7.3, having hanging node (yellow node) attached to it. The client, after verifying the proofs and modifications made, can digitally sign the transaction.

4. **Block 3, Transaction T4**: For the transaction T4 fired by the client, a TMT gets formed, and an old Merkle proof and new Merkle proof are generated. For the client's provenance query, to determine the history of modified row `key-ids`, the details shown in Figures 7.11 to 7.14 are displayed. Additionaly, an initial block 0 entry also gets shown. Now, transaction T4 modifies WAREHOUSE rows with id equal to 3, 6, 12, 7, and 2, and the CUSTOMER rows with id equal to 3 and 7. Seeking from *Genesis* block, the transaction T1 in block 1 has modified WAREHOUSE rows with id equal to 3, 6, and 12. The provenance query outputs T1 TMT and block 1 BMT, with the value details of WAREHOUSE id equal to 3, 6, and 12, shown in section 'a' and 'b' of Figure 7.11, respectively. It can be observed that the provenance proof results containing block 1 information for transaction T4 in PerfCredA remain the same as of transaction T4 in Credereum/SecCred/ProgCred, shown in Figure 7.4 section 'a' and 'b'. The reason behind such behavior, as explained, is the block 1 each transaction's TMT and the block's BMT in Credereum/SecCred/Prog-

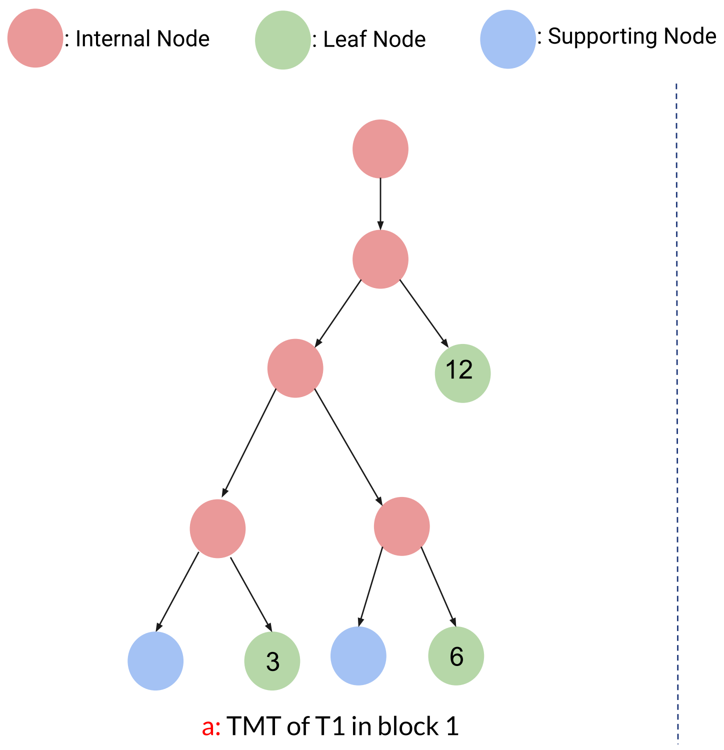


Figure 7.11: PerfCredA provenance outcome for transaction T4 of block 3 (Part-1)

Cred getting developed in the absence of a previous block BMT structure, and thus, builds independently. Hence, in Credereum/SecCred/ProgCred, the Merkle tree structure for each TMT and BMT for block 1 remains the same as for PerfCredA.

In block 2, transaction T2 has modified WAREHOUSE rows with id equal to 6 and 12. Also, transaction T3 of block 2 has changed CUSTOMER rows with id equal to 3 and 7. As shown in section 'a' and 'b' of Figure 7.12, the TMT of T2 and T3 is shown, respectively, with commonly modified rows with T4. The BMT of block 2 is further displayed in Figure 7.13 to demonstrate the inclusion of commonly modified rows of T2 and T3 with T4 in the block formation. The leaf node of BMT has entries written in red (CUSTOMER table modifications made by T3) and black (WAREHOUSE table modifications made by T2). For the TMT of T1, T2, and T3, the client can calculate the TMT RHV and `transaction-hash` value. With given BMT of block 1, and block 2, the client can compute the BMT RHV and `block-hash` and match it with the database and trusted storage.

The server for provenance also outputs the current transaction T4 TMT details displayed in Figure 7.14. The leaf node entries of T4 TMT shows the CUSTOMER rows with id value 3 and 7 (red entries) and WAREHOUSE rows with id value 3, 6, 12, 7, and 2

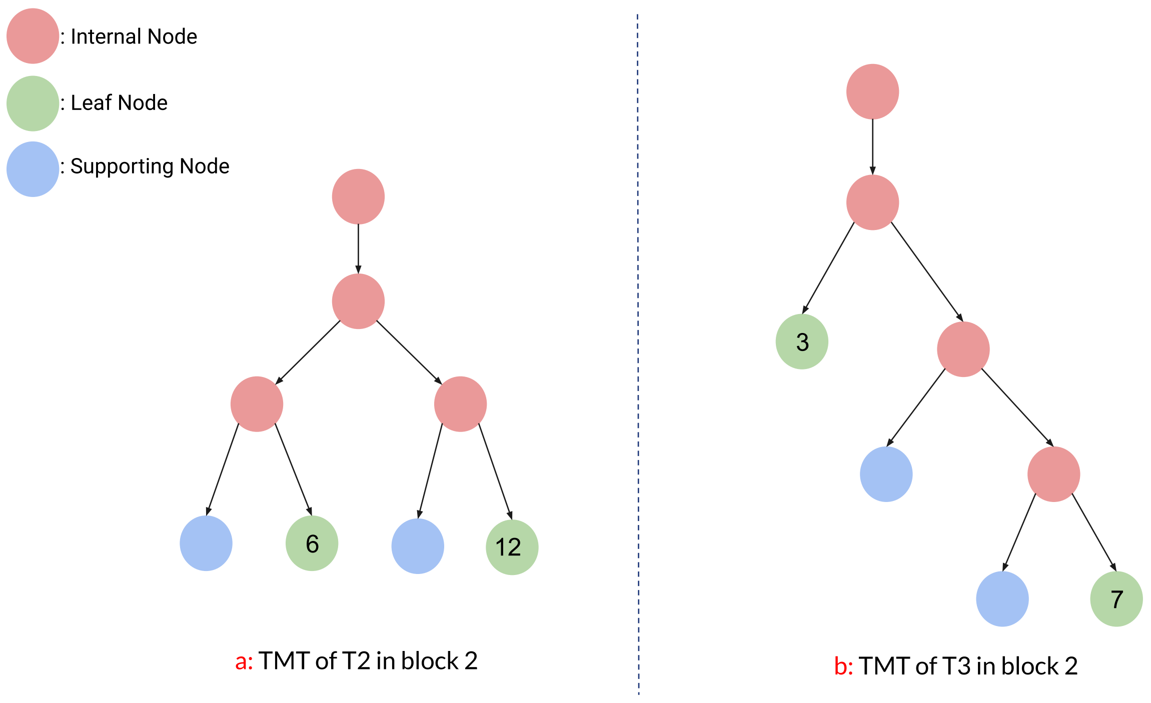


Figure 7.12: PerfCredA provenance outcome for transaction T4 of block 3 (Part-2)

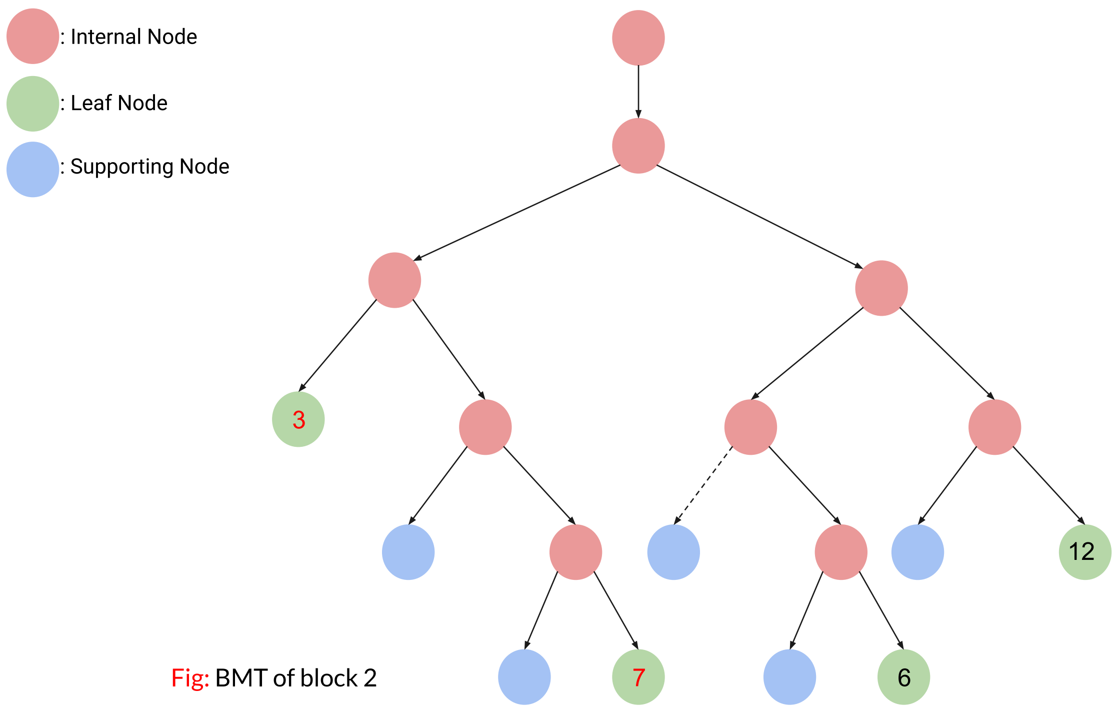


Figure 7.13: PerfCredA provenance outcome for transaction T4 of block 3 (Part-3)

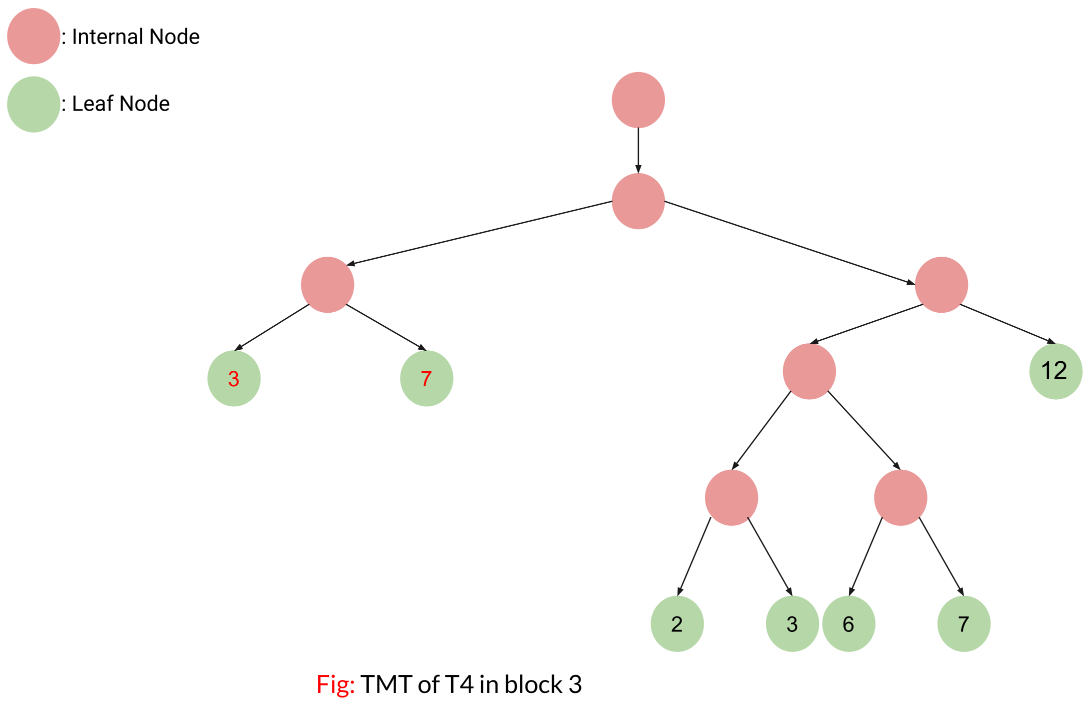


Figure 7.14: PerfCredA provenance outcome for transaction T4 of block 3 (Part-4)

(black entries) modified. The T4 TMT does not depend upon the previous block, i.e., block 2 BMT structure, and hence has no hanging nodes attached to it. The TMT of transaction T4 of PerfCredA can be seen differently from the TMT of transaction T4 of Credereum/SecCred/ProgCred shown in Figure 7.7, which has a dependency on the previous block BMT structure and holds hanging nodes (yellow nodes). It can also be observed, in PerfCredA with a growing timeline, the transaction's TMT structure is simpler compared to a similar transaction's TMT structure in Credereum/SecCred/ProgCred. Hence, per transaction computation cost in TMT formation and TMT RHV calculation decrements. The client in PerfCredA, after verifying the proofs and modifications from Figures 7.11 to 7.14, can digitally sign the transaction.

7.4 PerfCredB

The algorithmic modifications in PerfCredB have led to changes in the displaying and proving of the clients' provenance query results. As mentioned, with no TMT formation, the PerfCredB with MHV calculation (section 5.4.2) summarizes the transaction's alterations. The MHV computation has simplified the provenance result generation, and additionally, has reduced the provenance response time (section 8.2.3). Any client can raise the provenance queries, and the server gives proof for the modification history of queried row `key-ids`. PerfCredB retains the search space optimization (section 4.3) proposed for the provenance query.

7.4.1 Provenance Explanation

The PROVENANCE_PROOF_PCB() function in PerfCredB handles the provenance query raised by the client. The PROVENANCE_PROOF_PCB() function inputs row `key-ids` for which the provenance is desired, and the range block numbers (section 4.3) with default as *Genesis* block and the current block. For each row `key-id` X being queried, the PROVENANCE_PROOF_PCB() function generates and displays the [key, transaction_id, block_num, sub_hash, previous, hash, value] attribute entries by probing the CREDEREUM_MERKLIX table, where key is X and transaction_id is NOT NULL. The PROVENANCE_PROOF_PCB() function for each entry above, with transaction_id as T, block_num as B, and key as K:

1. outputs [key_hash, block_num, transaction_id, sub_hash, previous, hash] entries by probing the CREDEREUM_MERKLIX table for the transaction_id as T, block_num equal to B, and key not equal to K. The details fetched belongs to the other row `key-id`'s modified by T at block B.

2. outputs [key, block_num, transaction_id, children, leaf, hash, value] entries by probing the CREDEREUM_MERKLIX table for the transaction_id as NULL, block_num equal to B, and key equal to K. The details fetched belongs to the BMT leaf nodes of block B.

3. outputs [key, block_num, transaction_id, children, leaf, hash] entries by probing the CREDEREUM_MERKLIX table for the transaction_id as NULL, block_num equal to B, and key not equal to K. The details fetched belongs to the path nodes and supporting nodes of BMT of block B.

The PerfCredB, for each transaction_id ($\neq$ T) of block B, the server also displays the row `key-id` details in their hash formats, i.e., [key_hash, block_num, transaction_id, sub_hash, previous, hash]. For block M, where the queried key-ID has not been modified, the server displays the BMT details with minimal disclosure retention property (section 2.6). Each transaction details of block M is also additionally shown in their hash formats. Hence, for the provenance proof generation, each block and its respective transaction details are utilized.

Similar to Credereum (section 7.2.1), the client queries and verifies the CREDEREUM_BLOCK and CREDEREUM_TX_LOG table details. The details verified and stored helps evaluate the correctness of the provenance query results.

The client with the provenance results, say it verifies the queried row `key-id` (key) R, with details showing the modification being made by the transaction T_t of block number B_k. From the row-set S modified by T_t of B_k, only row R key and value are shown, and for the rest of rows (S-R), the key_hash, sub_hash and hash values are displayed by the PROVENANCE_PROOF_PCB() function. The client from the T_t details verifies no-duplicate/multi-modification by equalizing the shown key_hash's with those of R. If found valid, the client can compute the MHV and `transaction-hash` value of transaction T_t in B_k, using Algorithm 2.

Algorithm 2 : *MHV and transaction_hash verification*

```
$hash = `'$
$sub\_hash = `'$
while each row being modified by $T_t$ i.e. $r'_1, r'_2, r'_3, ..., r'_m$ in order do
    if sub_hash is known then
        $hash = SHA256\ (hash + key\_hash + sub\_hash)$
    else
        $key\_hash = SHA256\ (row\ key\text{-}id)$
        $sub\_hash = SHA256\ (row\ key\text{-}id + `{:}' + row\ value)$
        $hash = SHA256\ (hash + key\_hash + sub\_hash)$
    end if
end while
$MHV = SHA256\ (hash + transaction\_id)$
$transaction\_hash = SHA256\ (B_k\text{-}1\ BMT\ RHV + MHV + PK + DS)$
```

Algorithm 2 uses an iterative methodology to compute the hash of each row in S being modified by T_t in an order obtained from the previous attribute shown. For row R, the client verifies the `key-id` (key) bit-string and the value associated and computes key_hash, sub_hash and hash. For row-set S-R, the client, with the known key_hash and sub_hash value, compute the hash. The MHV is being calculated after the iterative loop ends and further gets used to calculate the T_t `transaction-hash` value. The block B_k-1 BMT RHV, the public key (PK), and the digital signature (DS) are also included in the `transaction-hash` value computation. If the MHV and `transaction-hash` for T_t matches the verified CREDEREUM_TX_LOG table entries, then the client further verifies the rest of B_k transactions, shown in provenance results for *non-inclusion/non-modification* of R.

In PerfCredB likewise, unique indexing is created with the constraint that a row can be modified at most once within a block. The client verifies the BMT of block B_k being output by PROVENANCE_PROOF_PCB() function to check the involvement of row R in the B_k formation. With the details output, the client verifies the R `key-id` (key) bit-string and the value associated with the BMT leaf node and computes the B_k BMT RHV. The BMT RHV gets matched with the verified CREDEREUM_BLOCK entries. With BMT RHV verified, the client can further calculate and verify the `block-hash` value of B_k as:

- `block-hash` = SHA256 (previous block B_k-1 `block-hash` + all constituting transaction's `transaction-hash` + BMT RHV)

If the BMT RHV and `block-hash` match with the verified base relations, then this signifies that row R has been modified and is accounted for in the block formation of block B_k.

Similarly, the client for non-hit block's M verifies the BMT and each related transaction details for the *non-modification* of row R. For block M, the BMT gets explored expecting the *same* old value of R, and each transaction of M are examined expecting *no-entry* for row R. The client using the same above technique can confirm the provenance proof results of each queried row `key-id`.

PerfCredB, with the removal of TMT, and replacement with MHV, has led to a rise in performance. However, while displaying the provenance, the non-queried row counts modified within a transaction are visible. With only the counts visible and not the identity of the non-queried `key-ids` of a transaction, no serious security breaches are expected.

7.4.2 Sample Results

In PerfCredB, say transaction T1, T2, T3, and T4, is being sequentially fired by the clients, and on commit will be accounted in block 1, block 2, block 2, and block 3, respectively. Each transaction is detailed in section 4.2.5. For each UPDATE/INSERT/DELETE, an entry with the new value is made in CREDEREUM_MERKLIX table, and further considered in the transaction's MHV calculation. For DELETE, the value is NULL, for INSERT, the value is the new fed data in relation, and for an UPDATE, the value is the new modified data. In Credereum/SecCred/ProgCred/PerfCredA, for each transaction, a TMT was developed and TMT RHV was computed. However, as will see, in PerfCredB, no such TMT gets formed, and instead, the modification rows are accounted for in the MHV calculation at the per-transaction level. The per-block BMT formation remains the same for Credereum, SecCred, ProgCred, PerfCredA, and PerfCredB, with being dependent upon the previous block BMT structure, and thus retains hanging nodes.

For a transaction's MHV verification, we have the following:

1. The green circles demonstrate the provenance queried rows (altered rows) and displays all the details, i.e., block_num, transaction_id, row key, value, previous, sub_hash, and hash value from the CREDEREUM_MERKLIX relation.

2. The orange circle represents the transaction's MHV.

3. The light-blue circles represent the supporting key_hashs (hidden rows), used in proofs for helping to compute and verify the transaction MHV.

4. The dark-blue inverted triangle indicate the hash calculation stage of each row.

For a BMT, we have the following:

1. The green nodes display the provenance queried rows (leaf nodes) within block's BMT.

2. The red nodes exhibit the BMT internal nodes.

3. The blue nodes represent the supporting nodes used to calculate and verify the parent node `node_hash`.

4. The dashed edges mean the child details can be fetched from the previously shown BMT details from the provenance results. Hence duplicate entries are avoided.

For the simplicity of explanation, corresponding to a transaction T, we define MHV-set as the output generated by the server, using transaction T details (from CREDEREUM_MERKLIX relation), to provide proofs to the clients for provenance. Now, with all the above information mentioned, say the client for each transaction T1, T2, T3, and T4 before commit, fires the provenance query for respective each modified (inserted/updated/deleted) row `key-ids`:

1. **Block 1, Transaction T1**: For the transaction T1, fired by the client, after making the database's modifications, an MHV gets calculated, and an old Merkle proof and New-Row-Value proof are generated. Now, considering the provenance raised by the client for the modified row `key-id`'s, the block 0 entry and the details shown in Figure 7.15 are displayed. With block 1 being the first data block, no previous modifications to the rows of the WAREHOUSE table with id equal to 1, 5, 3, 6, and 12 was made. Further, for

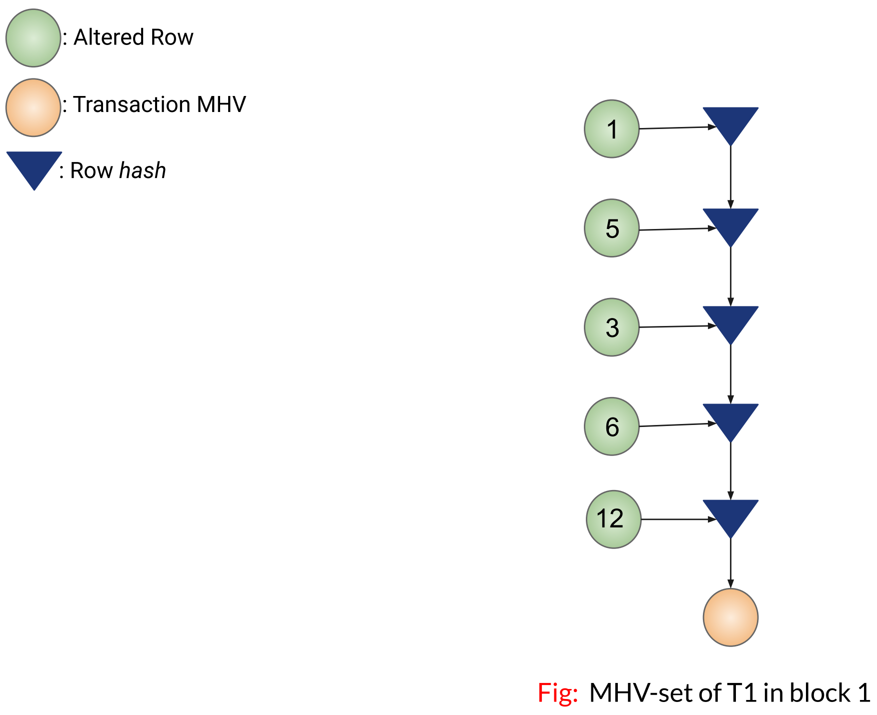


Figure 7.15: PerfCredB provenance outcome for transaction T1 of block 1

provenance query to display the new data values from changes made, shows the current transaction T1 MHV-set, which includes all modified row entries (green circles) along-with the MHV entry (orange circle), shown in Figure 7.15. The details displayed for each modified row entry include block_num, transaction_id, key (`key-id`), value, previous, sub_hash, and hash.

From Figure 7.15, we can see that the green circles represent each modified WAREHOUSE row with id equal to 1, 5, 3, 6, and 12. Considering the WAREHOUSE id value 1, its hash value is calculated in the dark-blue inverted triangle and checked with the provenance results. The verified hash is further forwarded and included in the hash calculation of the next modified row, i.e., WAREHOUSE row with id as 5. Hence, in this manner, the hash value of each modified row is computed and verified. The final obtained hash of WAREHOUSE id value 12 is passed for the MHV calculation. The calculated MHV summarizes all the modifications made by the transaction T1. The client, after verifying the changes, digitally signs the transaction.

2. **Block 2, Transaction T2**: For the transaction T2 fired, after making modifications in the database and with MHV calculated, an old Merkle proof and New-Row-Value proof

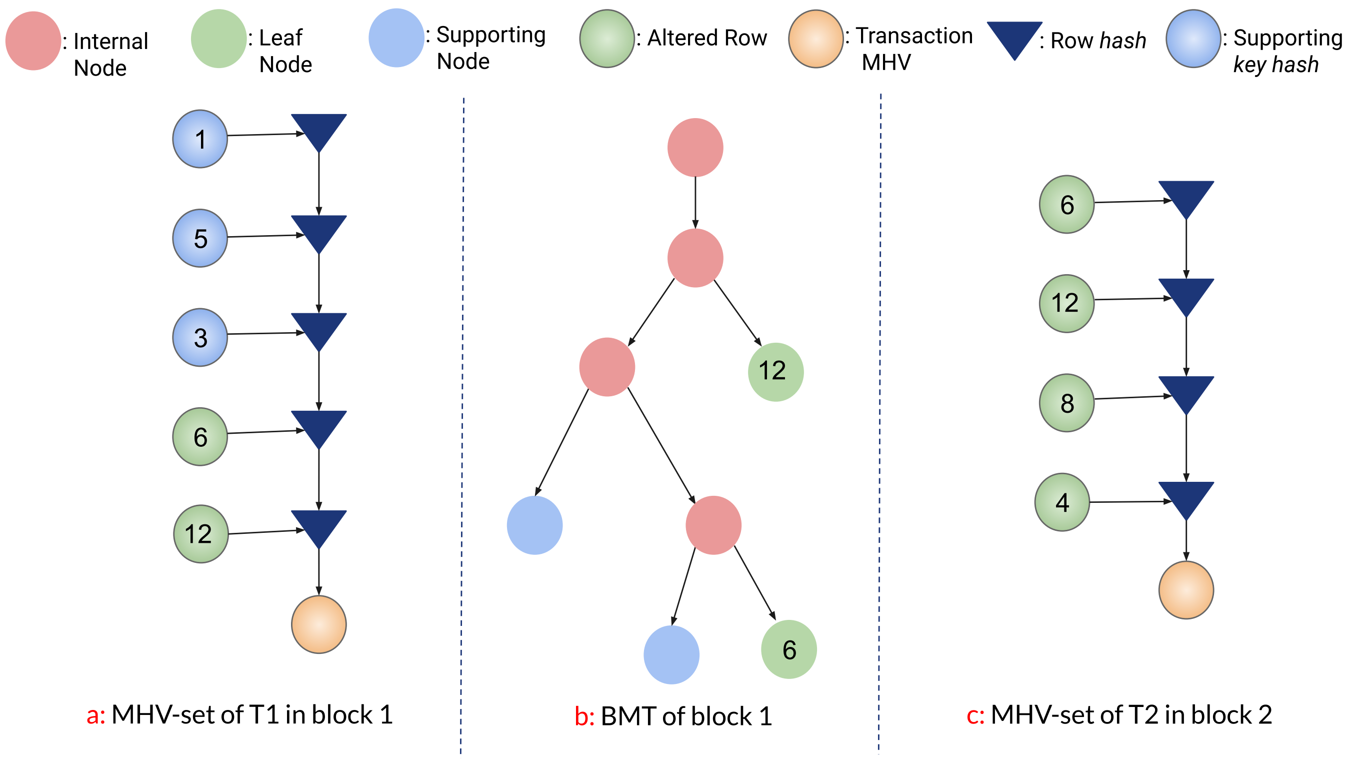


Figure 7.16: PerfCredB provenance outcome for transaction T2 of block 2

gets generated. The provenance outcome for the modified row `key-ids`, queried by the client, is displayed in Figure 7.16. Additionally, an initial block 0 entry also gets shown. Now, the transaction T2 modifies WAREHOUSE rows with id values 6 and 12, which was earlier modified by transaction T1 at block 1. So the provenance query displays all details of the commonly modified rows, i.e., WAREHOUSE rows with id values 6 and 12, in the MHV-set of T1 (green circles) and the BMT of block 1, as shown in section 'a' and 'b' of Figure 7.16, respectively. We can also observe from the section 'a' of Figure 7.16 that while proving the modification of WAREHOUSE rows with id values 6 and 12 by T1, the rest row's (light-blue circles) modified by T1 is kept hidden, and only their key_hash and sub_hash/hash are shown.

The BMT of block 1 shows the inclusion of WAREHOUSE rows with id values 6 and 12 in the block formation. Given a BMT, the client can compute `block-hash` value and match it with the one stored in the database and the trusted storage. As stated earlier, for each Credereum, SecCred, ProgCred, PerfCredA, and PerfCredB, a block's BMT depends on the previous block BMT structure. Hence, the section 'b' of Figure 7.16 displaying the inclusion of WAREHOUSE id values 6 and 12 in BMT formation of block 1 remains the

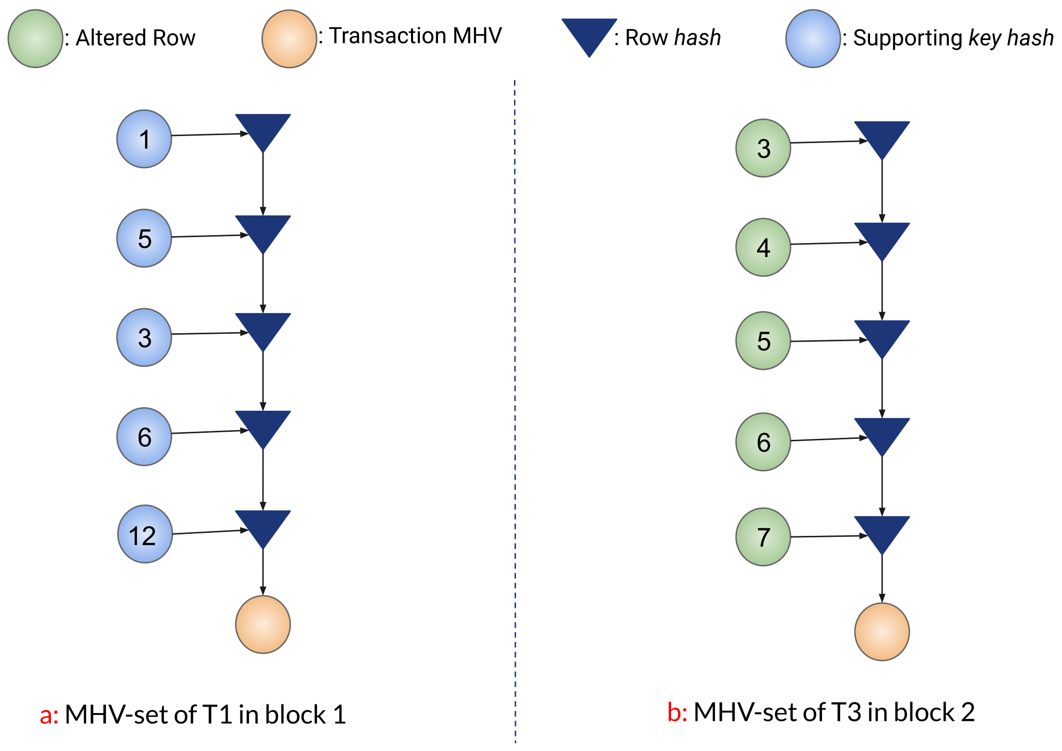


Figure 7.17: PerfCredB provenance outcome for transaction T3 of block 2

same as of section 'b' of Figures 7.2 and 7.9. The provenance proof to display the new modified data values also outputs the current transaction T2 all modified rows entries (green circles) along-with the MHV entry (orange circle), as shown in the section 'c' of Figure 7.16. The client, after verifying the proofs, and modification can digitally sign the transaction.

3. **Block 2, Transaction T3**: For T3, with modifications made and MHV calculated, an old Merkle proof and New-Row-Value proof gets generated. For the provenance proof of the modified row `key-ids`, block 0 entry, BMT of block 1, and the details shown in Figure 7.17 are displayed to the client. Now, the transaction T3 modifies the CUSTOMER table rows, which were not altered by any transaction earlier. Searching from the initial block, at block 1, with T1 having no common row modification with T3, all the rows modified by T1 are displayed by provenance in their hash form, shown in section 'a' of Figure 7.17. Now, considering BMT of block 1, with no common prefix with queried `key-ids`, only the root node and the child attached to the root node gets displayed.

   Using block 1 BMT RHV and MHV details from section 'a' of Figure 7.17, the client can compute the `block-hash` value of block 1 and match it with the database and the

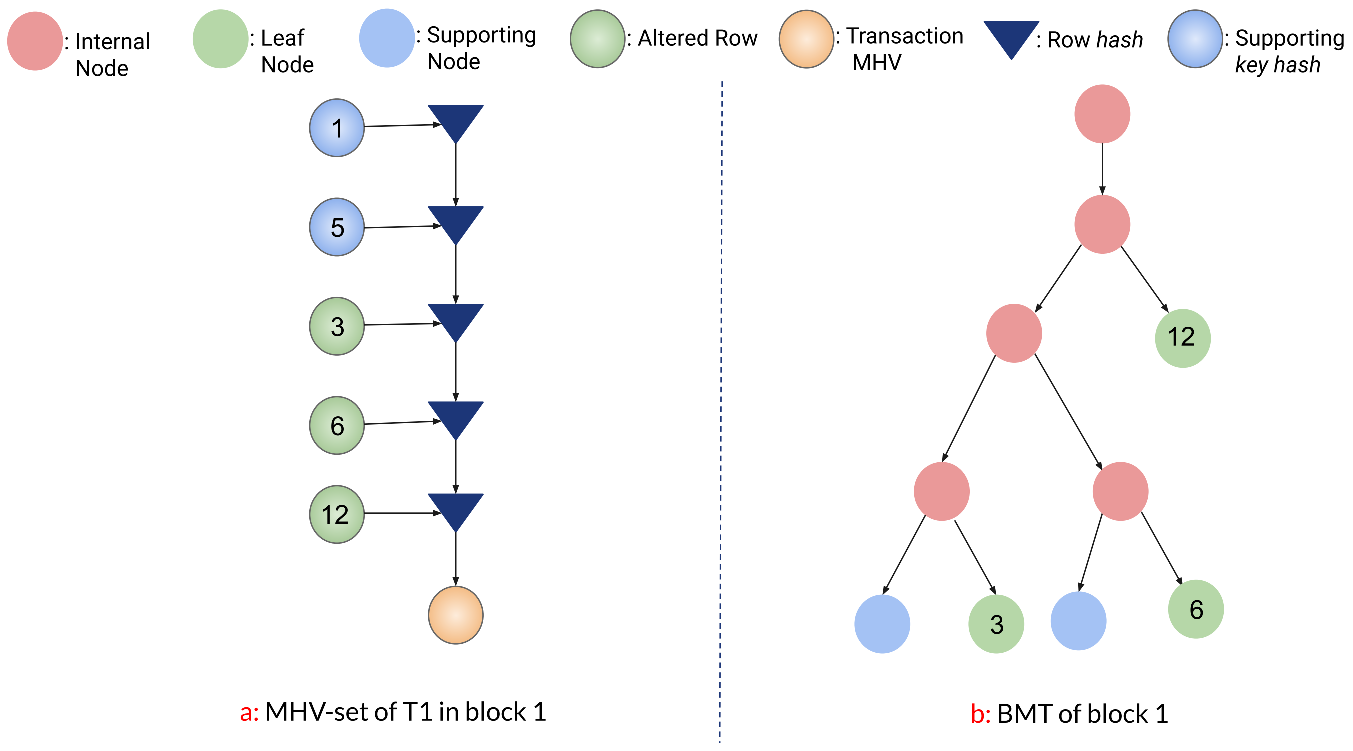


Figure 7.18: PerfCredB provenance outcome for transaction T4 of block 3 (Part-1)

immutable trusted storage. Further, the provenance also displays the new modified data values, which includes current transaction T3 all modified row entries (green circles) along with the MHV entry (orange circle), as shown in the section 'b' of Figure 7.17. The client, with proofs and modifications verified, digitally signs the transaction.

4. **Block 3, Transaction T4**: For the transaction T4, after database modification and MHV computation, an old Merkle proof and New-Row-Value proof gets generated. For the provenance proof of the modified row `key-ids`, the details shown in Figures 7.18 to 7.20 are displayed to the client. Additionally, an initial block 0 entry also gets output. Now, transaction T4 modifies WAREHOUSE rows with id equal to 3, 6, 12, 7, and 2, and the CUSTOMER rows with id equal to 3 and 7. Searching from the *Genesis* block, the transaction T1 in block 1 has modified WAREHOUSE rows with id equal to 3, 6, and 12. So the provenance query displays all details of the commonly modified rows, i.e., WAREHOUSE rows with id values 3, 6, and 12, in the MHV-set of T1 (green circles) and the BMT of block 1, as shown in section 'a' and 'b' of Figure 7.18, respectively. It can also be observed, from the section 'a' of Figure 7.18, that while proving the modification of WAREHOUSE rows with id values 3, 6, and 12 by T1, the rest row's (light-blue circles)

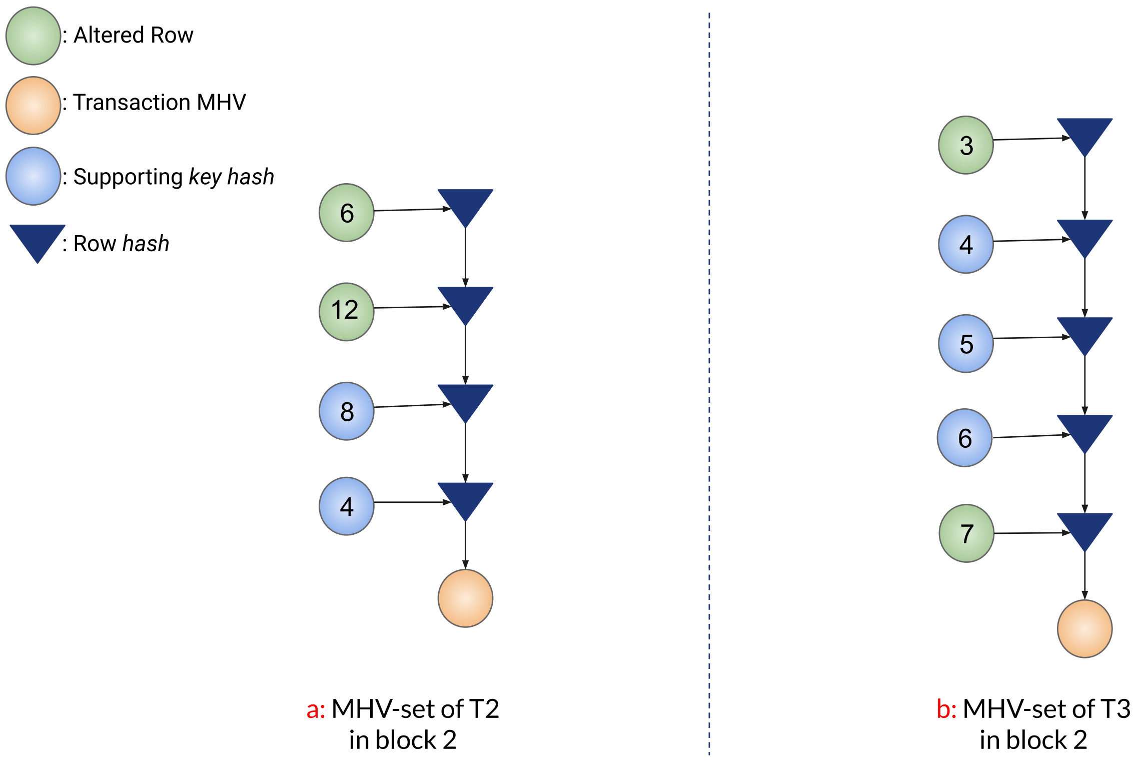


Figure 7.19: PerfCredB provenance outcome for transaction T4 of block 3 (Part-2)

modified by T1 is kept hidden, and only their key_hash and sub_hash/hash gets shown.

Now in block 2, transaction T2 has modified WAREHOUSE rows with id equal to 6 and 12. Also, transaction T3 of block 2 has changed CUSTOMER rows with id equal to 3 and 7. So, as shown in section 'a' and 'b' of Figure 7.19, all details of the commonly modified rows, i.e., WAREHOUSE rows with id values 6 and 12 in the MHV-set of T2 (green circles), and CUSTOMER rows with id values 3 and 7 in the MHV-set of T3 (green circles), are displayed respectively. From section 'a' of Figure 7.19, we can notice that while proving the modification of WAREHOUSE rows with id values 6 and 12 by T2, the rest row's (light-blue circles) modified by T2 is kept hidden. Similarly, from section 'b' of Figure 7.19, it can be observed that, while proving modification of CUSTOMER rows with id values 3 and 7 by T3, the rest row's (light-blue circles) modified by T3 is kept hidden.

The BMT of block 2 is further displayed in section 'a' of Figure 7.20 to show the inclusion of commonly modified rows of T2 and T3 with T4 in the block formation. The leaf node of BMT has entries written in red (CUSTOMER table modifications made by T3) and black (WAREHOUSE table modifications made by T2). We can also observe that the BMT of block 2 shown in section 'a' of Figure 7.20 remains the same as BMT of

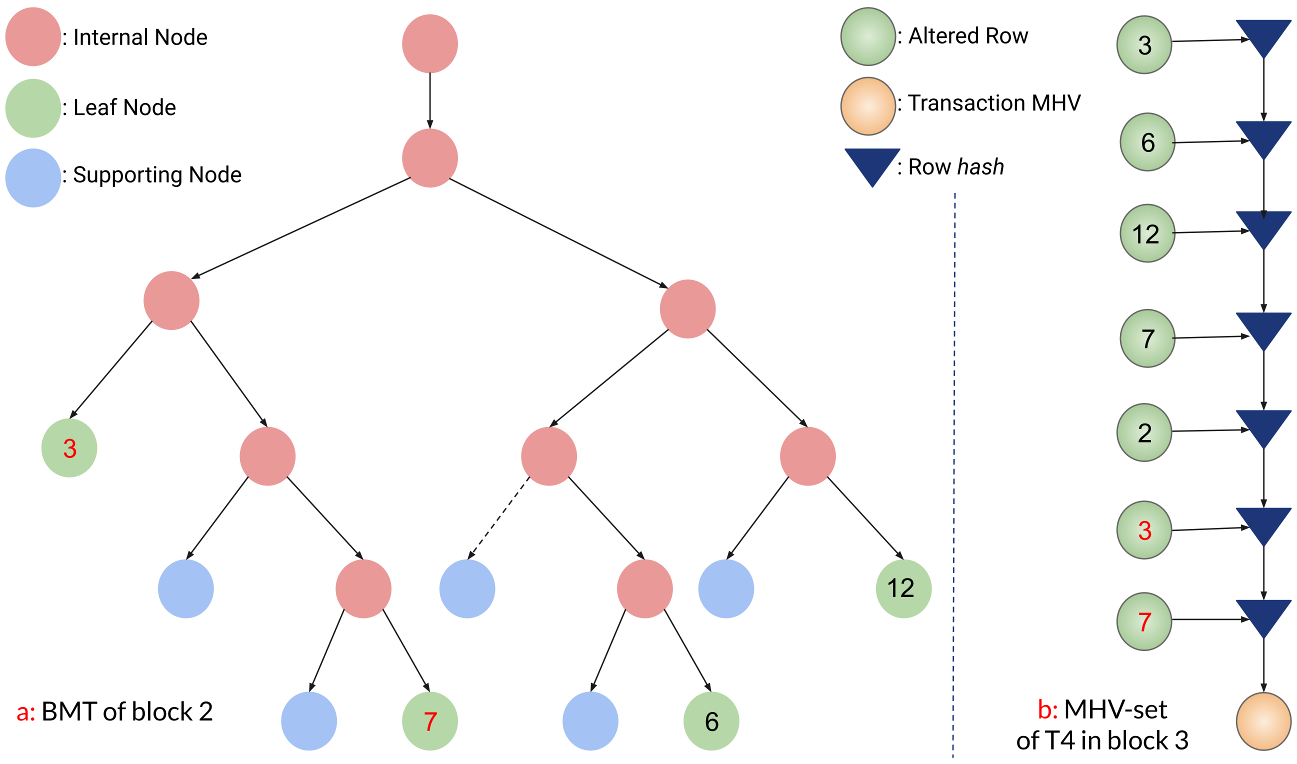


Figure 7.20: PerfCredB provenance outcome for transaction T4 of block 3 (Part-3)

block 2 shown in Figures 7.6 and 7.13, because in each case the BMT of the current block (block 2) depends upon the previous block (block 1) BMT structure. Hence, for each, Credereum, SecCred, ProgCred, PerfCredA, and PerfCredB, the BMT structure for each block will remain same.

For each T1, T2, and T3, the client can compute the respective MHV and the `transaction-hash` value using the details provided. For each block, block 1 and block 2 BMT, the client can compute BMT RHV, and further calculate the `block-hash` value and match it with the database and the immutable trusted storage. Further, the provenance query to display the new modified data values also outputs the current transaction T4 all modified row entries (green circles) along-with the MHV entry (orange circle), as shown in section 'b' of Figure 7.20. The entries written in red represents CUSTOMER table modifications, while the black entries represent WAREHOUSE table modifications made by T4. The client after verifying the proofs and modification from Figures 7.18 to 7.20 can digitally sign the transaction.

7.5 Summary

The Credereum provides provenance query functionality, by which the clients can query the modification history for a row/set-of-rows. With TMTs and BMT help, the client computes the block digest and matches the database and the trusted storage. SecCred and ProgCred show similar provenance output as Credereum because of the same underlying architecture. Further, considering provenance proof generated, PerfCredA, with algorithmic modifications, has variations in the display of TMT when compared with the native Credereum. Additionally, PerfCredB uses MHV to summarize the transaction alterations, which led us modify existing algorithms to check the authenticity of the generated provenance results.

Chapter 8

Experiment and Analysis

This chapter details the experiments conducted to evaluate the performance of the Credereum, and the different modification versions, i.e., SecCred, ProgCred, PerfCredA, and PerfCredB. Varying row modification counts made by transactions are used to create different experimental settings to evaluate the performances. Furthermore, the provenance response time for different systems has been analyzed and discussed.

8.1 Experimentation Setup and Details

The experiments performed in HP Z440 Workstation have Intel Xeon CPU E5-1660 v4 @ 3.20GHz, 32 GB memory, and Ubuntu 16.04 LTS. With transaction firing from terminals to PG-Tuned PostgreSQL 10.5, the average committed-transaction throughput is examined, with a time interval of 10 minutes. Moreover, a synthetic transaction generator was developed for targeted row alterations. Row numbers are chosen uniformly and randomly. If the row exists in the relation, it gets deleted 25% of the time and updated 75% of the time. If not, the row gets inserted. Each transaction fired modifies a maximum of n number of rows. For each of the m ($m \leq n$) write operations conducted, respective m reads were made for verification.

8.2 Performance Analysis

The section shows and compares the performance of Credereum, SecCred, ProgCred, PerfCredA, and PerfCredB using different experiment settings in a closed system environment. A varying number of row modifications made by a transaction are used to study the proposed systems' performance. Additionally, the performances of the Credereum and its proposed versions are evaluated in an open system setting. Further, the response time of provenance queries for each design has been analyzed and discussed.

8.2.1 Varying Row Alteration (Closed System)

This section evaluates and compares the Credereum and proposed system's performance in a closed setting. Previously, it was shown in Figure 3.4 that the Credereum blockchain semantics are obtained at an enormous cost in performance – an orders-of-magnitude degradation in transaction throughput as compared to the native PostgreSQL. With the performance of Credereum considered a base, the modification made helped achieve a high transaction throughput by retaining all the essential functionalities of providing modification and provenance proofs. With varying systems and varying max-modified rows by a transaction, the performances evaluated are illustrated in Table 8.1.

Row Modified →	**1 row**	**max 5-rows**	**max 10-rows**	**max 15-rows**	**max 25-rows**
PostgreSQL TPE →	2420406	1682993	1391740	1092442	739096
Credereum TPE →	55398	14642	5758	3258	1491
SecCred TPE →	55252	14592	5613	3236	1451
ProgCred TPE →	104483	50949	32792	23670	15303
PerfCredA TPE →	241754	106690	63438	45437	27615
PerfCredB TPE →	245389	111546	67900	48102	30412

Table 8.1: TPE Analysis (Closed System)

Table 8.1 shows the performance of Credereum, SecCred, ProgCred, PerfCredA, and PerfCredB for 1-row and a maximum of 5-rows, 10-rows, 15-rows, and 25-rows alterations made by a transaction. For an addressed system, the exponential distribution gets observed. Now, each case's transactions were being fired from 10 terminals to a database with hundred thousand entries. From the table, we can observe a performance boost compared to Credereum. With cheap SHA-256 cryptographic hash calculation, SecCred can be observed to attain a similar performance number compared to Credereum. Hence, the functionality addition hasn't led to a decrement in the transaction throughput with huge counts. The ProgCred holding SQL to C translation of CREDEREUM_LONGEST_PREFIX() function with programming changes can be observed to have positively boosted the performance numbers. With CREDEREUM_LONGEST_PREFIX() being the maximum called function, any optimization rectifications were very likely to have drastic performance improvements. The PerfCredA with independent TMT has lightened the transaction's exercise to sustain blockchain semantics, leading to a massive rise in performance numbers. With MHV proposed, the PerfCredB can be perceived to have better performance compared to PerfCredA. For varying settings, PerfCredB has been frequently observed to have better performance compared to PerfCredA.

8.2.2 Varying Row Alteration (Open System)

The Credereum and its proposed versions were evaluated in an open system setting to examine their throughput rate. With transactions firing at a rate of 100 tps to a database with hundred thousand entries and with *max_connection* parameter set to 2500, the performances of Credereum, SecCred, ProgCred, PerfCredA, and PerfCredB has been evaluated. For Credereum and SecCred, due to the heavy time-consuming block formation process, the new transactions are made to wait, and subsequently, thrashing gets observed. The rest proposed version system's performance has been demonstrated in Figure 8.1. Each transaction fired modifies a single row of the relation. From the figure, we can observe a performance boost with the proposed versions. PerfCredB delivers the highest performance number compared to all the rest available peer systems.

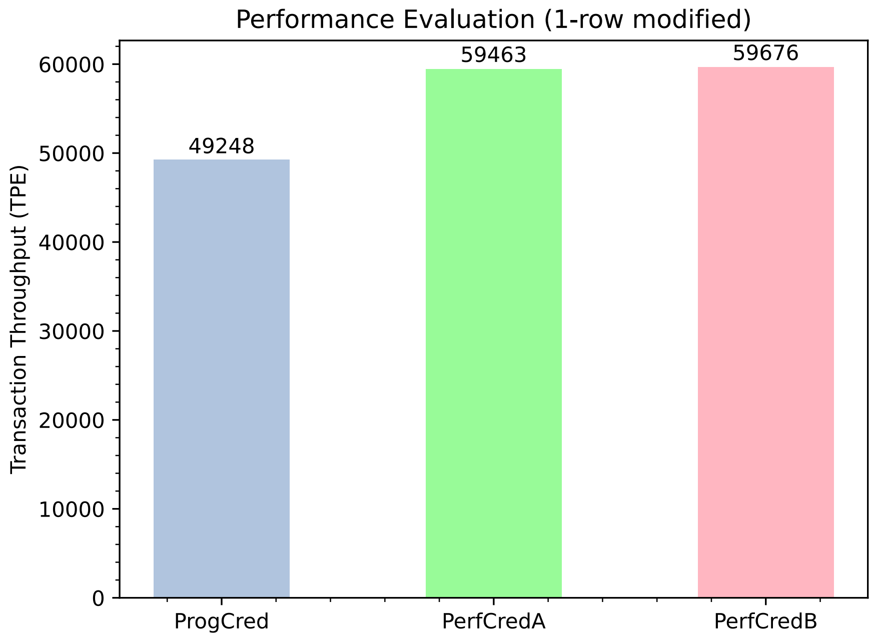


Figure 8.1: TPE Analysis (Open System)

8.2.3 Provenance Analysis

With equal counts of thousands of transactions committed, the response time of a provenance query fired by the client has been evaluated. The experiment setting includes transactions firing from 10 terminals to a database with hundred thousand entries, where each transaction at max modifies distinct 25-rows. Table 8.2 illustrates the response time of a query asking provenance for distinct 1-row, 10-rows and 25-rows in the Credereum, ProgCred, PerfCredA, and PerfCredB system. On experiments, SecCred is observed to have a similar provenance response time compared to Credereum. With PerfCredA holding small autonomous TMT and with iterative methodology in PerfCredB, we can witness a notable difference in provenance execution. Further, PerfCredB, due to avoiding TMTs, beats Credereum by huge margins in provenance response time.

Row Queried →	**1 row**	**10-rows**	**25-rows**
Credereum →	823	3035	7516
ProgCred →	1514	2496	3297
PerfCredA →	781	1709	2228
PerfCredB →	111	273	426

Table 8.2: Average Provenance Response Time (secs)

8.3 Summary

The Credereum, to sustain the blockchain semantics, had led to a massive transaction throughput degradation compared to the naive PostgreSQL. With the performance being a vital parameter for acceptance and deployment of the blockchain model in a real-world environment, the Credereum with poor performance was highly inadequate. With the research focused on functionality and performance improvements, SecCred, ProgCred, PerfCredA, and PerfCredB proposed were evaluated in terms of committed transactions in a given time interval. The chapter depicts the performance of each varying modification model with experimental parameter variations. The PerfCredA and PerfCredB are seen to have a comparatively very high performance compared to Credereum. Further, the PerfCredB can be observed to outperform PerfCredA in all the experiments evaluated. Additionally, PerfCredB has very low provenance response time compared to the rest addressed peers.

Chapter 9

Conclusions and Future Work

The chapter presents an abstracted glimpse of the research work and lists potential future works. Additionally, the Credereum positives and limitations have been addressed. Further, the exercises to uplift the barriers are summarized, detailing security enhancement methodologies and performance up-gradation techniques. Concerning future works, the chapter proposes making the system distributed and decentralized, holding no transaction-type constraints, unique indexing relaxation, and so forth.

Credereum, being a private blockchain-enabled platform, is designed to be implemented on top of native PostgreSQL. With notable features like forming Merkle tree at the per transaction and block level, generating old Merkle proof and new Merkle proof (to prove the old data values and the respective new data values), usage of trusted storage, provenance query handling, etc., the Credereum attains significant blockchain properties, specifically, immutability, verification, authentication, and non-repudiation. Now, with a single bit malicious modification made within block B at TMT/BMT level, the `block-hash` of block B and all subsequent forward blocks need to be recalculated to validate the ledger. However, the newly computed `block-hash` will differ from the stored digest at the immutable trusted storage, and thus immutability gets guaranteed in Credereum. Also, for every row being altered by the transaction, the server-generated old Merkle proof and new Merkle proof needs to be verified by the client. The client can also query for the modification history of a row/set-of-rows using provenance query. Additionally, for the transaction to commit, the client needs to submit a digital signature using its private key to authenticate the transaction, which further helps attain non-repudiation property. With most well-known blockchain platforms being decentralized and distributed, Credereum is currently centralized. Due to this, Credereum, suffering from poor transaction throughput performance, encounters some traditional and blockchain-based consequences, like single server dependency, no resilience to data loss, no consensus, and so forth.

9.1 Conclusions

Credereum, despite holding valuable features has certain shortcomings, covering resistance to malicious actions, system optimization and performance numbers. In Credereum, for each BMT built, the server keeps the entire BMT details hidden. Although, with the help of a provenance query, the client can verify whether or not the respective modifications made have been accounted in the BMT creation. However, this functionality doesn't provides prevention from all the malicious exercises practiced. Examining the client is unaware of the modifications being performed by the rest clients within a block, to provide prevention from fraudulent activities, the server needs to public the TMT and BMT details, which outlines the core trait of SecCred. The SecCred generates the per block TMT and BMT structure in the SHA-256 hash form, which is easily verifiable. Further, the SecCred has led to changes in the Merkle tree node's `node-hash` computation methodology. Moreover, with the stated changes in SecCred, the SecCred retains all the blockchain properties held by the Credereum.

Now, when the performance of Credereum was evaluated and compared with PostgreSQL, a substantial orders of magnitude degradation in transaction throughput was witnessed. The research was being conducted focusing on performance improvement, leading to specific programming optimization and algorithmic alterations. Considering programming optimization, the Credereum functions were analyzed based on maximum call counts, execution-time/call, and self-time/call metrics. The CREDEREUM_LONGEST_PREFIX() function with the algorithm to determine the longest common prefix among the given two bit-strings was observed to achieve the maximum calls. ProgCred dealing with programming optimization has led to modifications in the CREDEREUM_LONGEST_PREFIX() function, with byte-level comparison and engagement of bitwise operations. The stated improvements had made a positive impact on performance up-gradation. Further, ProgCred retains the alterations made in SecCred and maintains all the Credereum held blockchain properties.

Modifications dealing with algorithmic changes focus on optimizing transaction handling in the Credereum architecture. In Credereum, for every transaction being fired, a TMT is being developed by depending upon the previous block BMT structure. Exploring a redundancy with the previous BMT dependency, PerfCredA, with inheriting alterations made in ProgCred, have made TMT develop autonomously. PerfCredA retains all the blockchain properties and semantics held by the Credereum. Further, PerfCredB deals with an alternative algorithmic modification to ProgCred, where instead of TMT formation per transaction, an MHV gets calculated. The MHV summarizes all the modifications being made by the transaction, and also additionally eases the handling of provenance queries. The MHV computed is accounted for in the transaction `transaction-hash` value computation. For every client's modification, an old Merkle Proof and New-Row-Value proof are shown to the client. Further, the PerfCredB retains all the blockchain properties being possessed by Credereum. The algorithmic modifications in PerfCredA and PerfCredB, positively led to an increase in the evaluated performance. Also, the algorithmic alterations proposed led to changes in the provenance output for PerfCredA and PerfCredB. A search space optimization has been included in the provenance query engine for all the proposed Credereum versions.

9.2 Future Work

The following lists the set of future works which could be further researched to enhance the system performance, utility and security:

1. **Decentralized and Distributed Environment**: The Credereum and the proposed versions deal with the centralized server, leading to certain limitations, like single server dependency, scalability issues, transaction-type constraint, no consensus, no resilience to data loss, etc. Whereas, in a decentralized environment, even if a server fails, the rest active servers can handle the live transactions, and the data loss can get retrieved from the rest. Further, the decentralized environment uses consensus to judge the transaction's modifications, which leads to relaxing transaction-type constraints and also enhances the client's trust. The distributed environment can help scale the system's performance and lower response time.

   Credereum is a permissioned centralized blockchain implementation that uses immutable trusted storage to store the per block cryptographic digest. Being a centralized system, the Credereum currently operates with a single server and multiple clients. However, considering a distributed database environment with multiple servers working in Credereum, we, apart from client and server, propose an additional data-server entity. The data-server is responsible for handling replicated distributed databases and achieving consensus. In a permissioned setting, whenever a client wants to fire a transaction, it broadcasts its transaction along with the digital signature to all the available servers. Each server verifies the transactions, orders the transaction, and achieves a consensus on the transaction order within the block. Further, for each transaction in the block, a server broadcasts the transaction to each associated data-server entity and awaits its response. The data-server entity for its generated result obtains a consensus with the rest of the data-server entities possessing the same database. Further, the consensus achieved result is returned by the data-server to the server. On the complete execution of each transaction within a block, the server computes the block digest and achieves a consensus with the rest servers. In this manner, distributed databases with the decentralized setting can be applied for Credereum. However, efficient methods of handling the distributed database environment for Credereum are kept for study in future research work.

Considering a decentralized environment in Credereum with multiple servers connected, we propose holding an additional ordering-servers set (similar to the *order-then-execute* approach of *Blockchain Meets Database* paper [28] by Senthil Nathan et.al.). The ordering-server will be responsible for collecting the transactions, checking their authenticity, ordering them for a block, and achieving a consensus. So, in a decentralized setting, Credereum has the client, server, and ordering-server entity. Now, whenever a client fires a transaction to a server, it also submits its digital signature to the server. On successful verification, the server broadcasts the client-fired transaction to the ordering-server entities. Further, the ordering-server forms a block B from the transactions received and holds a consensus on the ordering of the transactions within the block. Then, the ordering-server entities broadcast this block B to each server, where every server executes and commits in order the transactions listed within block B. Further, the server computes the block digest of block B and broadcasts it to the ordering-server. The ordering-server performs a consensus and adds the valid block digest to the next block B+1. Each server can verify the consensus-held block digest of block B after receiving block B+1 from the ordering-server. In this manner, all the servers can retain the same ledger with the help of consensus. However, the efficient design of the decentralized environment for Credereum will be of high focus for future research work.

2. **Locking Mechanism**: The transaction fired in Credereum and proposed versions needs to acquire a lock, with conflicting behavior, from the locks possessed by the block process. Thus, the transaction process needs to wait for the block process completion to acquire the lock and process. The waiting time of a transaction process to acquire a lock from the block process consumes a significant portion of transaction execution time. This environment can be researched forward, where block and transaction processes can execute in parallel. With a parallel environment, there will be no longer a wait for accessing the locks, thus leading to a hike in performance numbers.

3. **Merkle Proof Generation**: A proof for the database modification is generated for every fired transaction, providing evidence for the row's old data values and new data values. The old Merkle proof displaying the old data values uses the previous block BMT

structure and is expensive. With better techniques for the old data proof generation, a reduction in transaction computation cost can be achieved.

4. **Block Creation**: For every block created, a BMT is constructed depending upon the previous block BMT structure, which is expensive. Research can be done forward in designing new efficient methodologies for displaying and proving the modifications made within a block. With this cause, the per-block formation will be quick, and due to the isolated running environment required by the block process, it will lead to an early release of the held locks, and thus more transactions can get efficiently processed.

5. **Unique Indexing Relaxation**: The Credereum and proposed versions each possess a unique indexing constraint, explained in section 3.2.3. Due to this, multiple transactions cannot modify the same row within a block. Research can be done further where new BMT architecture is designed, allowing multiple modifications to a row within a block. With this, numerous transactions can modify the same row, and with no aborts due to constraint, the performance will hike.

6. **SSI Complaint Architecture**: Considering Credereum and proposed versions, each underlying architecture of the system is designed for processing transactions in a Read Committed (RC) isolation level. New architectures can be researched to handle more secure SERIALIZABLE SNAPSHOT ISOLATION (SSI) locks. With SSI, all the concerns held by RC isolation can be avoided.

Apart from the aforementioned possibilities, research directed towards modifying underlying architecture should be motivated to upgrade system performance and security.

Bibliography

[1] Double-spending. URL https://en.wikipedia.org/wiki/Double-spending. 2

[2] Ethereum. URL https://ethereum.org/en/. ii, 2

[3] What is a Smart Contract? https://github.com/ethereumbook/ethereumbook/blob/develop/07smart-contracts-solidity.asciidoc#what-is-a-smart-contract. 2

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190538. URL https://doi.org/10.1145/3190508.3190538. 2

[5] Dave Bayer, Stuart Haber, and W. Scott Stornetta. Improving the Efficiency and Reliability of Digital Time-Stamping. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II*, pages 329–334, New York, NY, 1993. Springer New York. ISBN 978-1-4613-9323-8. URL https://doi.org/10.1007/978-1-4613-9323-8_24. 2

[6] BigchainDB. URL https://www.bigchaindb.com/. 7

[7] Apache Cassandra. URL https://cassandra.apache.org. 7

[8] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association. ISBN 1880446391. URL https://dl.acm.org/doi/10.5555/296806.296824. 4

[9] chainifyDB. URL https://www.chainifydb.com/. 8

[10] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A Shared Database on Blockchains. *Proc. VLDB Endow.*, 12(11):1597–1609, July 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342636. URL https://doi.org/10.14778/3342263.3342636. ix, 6, 7

[11] Elasticsearch. URL https://www.elastic.co/products/elasticsearch. 7

[12] Hal Finney. Reusable Proofs of Work (RPOW). URL https://nakamotoinstitute.org/finney/rpow/index.html. 2

[13] Consistency guarantees. URL https://rethinkdb.com/docs/consistency/. 7

[14] Community guides and resources. URL https://ethereum.org/en/learn. ii

[15] Stuart Haber and W. Scott Stornetta. How to Time-Stamp a Digital Document. *J. Cryptol.*, 3(2):99–111, January 1991. ISSN 0933-2790. doi: 10.1007/BF00196791. URL https://doi.org/10.1007/BF00196791. 2

[16] Cryptographic hash function (CHF). URL https://en.wikipedia.org/wiki/Cryptographic_hash_function. 4, 12

[17] Apache HBase. URL https://hbase.apache.org. 7

[18] Changefeeds in RethinkDB. URL https://rethinkdb.com/docs/changefeeds. 7

[19] Credereum - blockchain-enabled Postgres. URL https://pgconf.ru/en/2018/110824. 10, 17

[20] What is Ethereum? URL https://docs.ethhub.io/ethereum-basics/what-is-ethereum/. 2

[21] Alexander Korotkov. *credereum_longest_prefix()* function, 2018. https://github.com/postgrespro/pg_credereum/blob/3a227a6eac4049378203d53e9b86d6aa6ba065b6/pg_credereum--0.1.sql#L108. 45, 67

[22] Alexander Korotkov. *credereum_merkle_proof()* function, 2018. https://github.com/postgrespro/pg_credereum/blob/3a227a6eac4049378203d53e9b86d6aa6ba065b6/pg_credereum--0.1.sql#L459. 41

[23] Alexander Korotkov. Credereum - github repository, May 2018. https://github.com/postgrespro/pg_credereum. ii, 17

[24] Riak KV. URL https://docs.basho.com/riak. 7

[25] Trent McConaghy, Rodolphe Marques, Andreas Muller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. BigchainDB: A Scalable Blockchain Database. URL https://mycourses.aalto.fi/pluginfile.php/378362/mod_resource/content/1/bigchaindb-whitepaper.pdf. ix, 7, 8

[26] MongoDB. URL https://www.mongodb.org. 7

[27] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. URL https://bitcoin.org/bitcoin.pdf. 2

[28] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proc. VLDB Endow.*, 12(11):1539–1552, July 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342632. URL https://doi.org/10.14778/3342263.3342632. ix, 5, 6, 124

[29] Dan R. K. Ports and Kevin Grittner. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.*, 5(12):1850–1861, August 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367523. URL https://doi.org/10.14778/2367502.2367523. 6

[30] Redis. URL https://www.redis.io. 7

[31] RethinkDB. URL https://www.rethinkdb.com. 7

[32] Ripple. URL http://ripple.com/. 2

[33] Felix Schuhknecht, Ankur Sharma, Jens Dittrich, and Divya Agrawal. chainifyDB: How to get rid of your Blockchain and use your DBMS instead. *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*, January 2021. URL http://cidrdb.org/cidr2021/papers/cidr2021_paper04.pdf. 8

[34] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, and Divya Agrawal. ChainifyDB: How to Blockchainify any Data Management System, December 2019. URL https://arxiv.org/abs/1912.04820v1. ix, 8, 9

[35] Secure Hash Algorithm 2 (SHA-2). URL https://en.wikipedia.org/wiki/SHA-2. 12

[36] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: An Efficient Storage Engine for Blockchain and Forkable Applications. *Proc. VLDB Endow.*, 11(10):1137–1150, June 2018. ISSN 2150-8097. doi: 10.14778/3231751.3231762. URL https://doi.org/10.14778/3231751.3231762. ix, 9, 10

[37] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. URL https://gavwood.com/paper.pdf. 2