Easy and Efficient Querying of Smart Contract Data while Maintaining Data Integrity

Matthijs Kaandorp
m@tthijskaandorp.com

July 31, 2019, 43 pages

Research supervisor:	dr. Adam Belloum, A.S.Z.Belloum@uva.nl
Host/Daily supervisor:	Boaz Pat-El, b.pat-el@avanade.com
Host organisation/Research group:	Avanade, www.avanade.com/

Abstract

With blockchain becoming more popular, more and more application logic is moved to the blockchain in the form of Smart Contracts. Although the blockchain stores an immutable history of all interaction with- and resulting states of each Smart Contract, the way this data is stored makes it difficult and inefficient to access. We try to solve this problem by synchronizing the relevant data from a blockchain with a traditional database. We maintain a level of data integrity by constructing Merkle proofs for each data element, which allows the user and other parties to verify the resulting data. By decoupling our solution from the blockchain, we try to provide compatibility with existing platforms and Smart Contracts, and we try to make the system usable without extensive knowledge of blockchain technology. We evaluate both the functionality and performance of our solution by comparing it with a naive solution. Our solution supports a number of SQL queries, and can prove that the resulting data also exists on the blockchain. Our solution is faster than the naive solution when the entire blockchain is already indexed, or when only a small amount of transactions have been added since the last time a query was issued.

Contents

1 Introduction **4**
- 1.1 Problem statement . . . 5
  - 1.1.1 Research questions . . . 5
  - 1.1.2 Research method . . . 5
- 1.2 Contributions . . . 5
- 1.3 Outline . . . 5

2 Related work **7**
- 2.1 Query functionality . . . 7
- 2.2 Frameworks for static blockchain analysis . . . 7
- 2.3 Hosted services . . . 8

3 Background **9**
- 3.1 Blockchain . . . 9
- 3.2 Permissionless/permissioned blockchain . . . 9
- 3.3 Ethereum . . . 10
- 3.4 Tree structures in Ethereum . . . 13
  - 3.4.1 Merkle Tree . . . 13
  - 3.4.2 Inclusion proof [24] . . . 13
  - 3.4.3 Patricia Tree [28] . . . 13
  - 3.4.4 Modified Merkle Patricia Tree . . . 14
  - 3.4.5 Types of tries in Ethereum . . . 15
- 3.5 Querying data from a blockchain . . . 16

4 Implementation **18**
- 4.1 System requirements . . . 18
- 4.2 System design . . . 19
  - 4.2.1 Interaction between modules . . . 21
- 4.3 System implementation . . . 21
  - 4.3.1 Configuring the system . . . 21
  - 4.3.2 Relevant data . . . 22
  - 4.3.3 Storing the data . . . 22
  - 4.3.4 Synchronization . . . 23
  - 4.3.5 Creating proofs . . . 24
  - 4.3.6 Validating data . . . 25
  - 4.3.7 Sharing data . . . 27
- 4.4 System Usage . . . 28
- 4.5 Comparison to other solutions . . . 28
  - 4.5.1 Query directly off blockchain . . . 28
  - 4.5.2 Implement in smart contract . . . 28
  - 4.5.3 Comparing data instead of Merkle roots . . . 28
  - 4.5.4 Store only data identifiers . . . 29

5 Evaluation **30**
- 5.1 Use case . . . 30
  - 5.1.1 Green Energy Production Location . . . 30
  - 5.1.2 Green Energy Consumption Location . . . 30

5.1.3 Green Energy Contract . . . 31
5.2 Functional evaluation . . . 31
5.2.1 Scenario 1: Publishing valid data . . . 31
5.2.2 Scenario 2: Tampered database . . . 31
5.2.3 Scenario 3: Giving out false information . . . 32
5.3 Performance evaluation . . . 32
5.3.1 Static blockchain . . . 33
5.3.2 Dynamic blockchain . . . 33

6 Discussion **34**
6.1 Functional evaluation . . . 34
6.2 Performance evaluation . . . 34
6.2.1 Static blockchain . . . 34
6.2.2 Dynamic blockchain . . . 35

7 Conclusion **40**
7.1 Future work . . . 41

Bibliography **42**

Chapter 1

Introduction

Since blockchain was introduced by Satoshi Nakomoto as the foundation of Bitcoin[29], the technology has gained popularity during recent years in many different areas. Where it started out as a technology used in the financial sector, it has also found popularity in healthcare, manufacturing, security, and other sectors. [32] [6]

Useful characteristics of blockchain technology are practical immutability and a decentralized nature. These characteristics are due to all data on a blockchain being cryptographically linked, and consensus algorithms which cause all nodes on the network to have identical data without the need for a centralized party. Due to the practical immutability of the data on the blockchain, a level of data integrity is assured. [37]

Many solutions make use of Smart Contracts: application code stored on the blockchain. Users in the network can interact with these Smart Contracts. All interactions with- and resulting states of Smart Contracts are stored in blocks on the blockchain. [36]

Although an immutable history of all transactions and states relating to Smart Contracts is stored on the blockchain, the way this data is stored makes it difficult and inefficient to read and query this data. The blockchain usually allows data to be retrieved only by using the related unique identifier. However, this means that when one wants to query specific information, one has to know the unique identifier beforehand, or one has to perform sequential access, and go through all the data stored on the blockchain until the wanted data element is found. This problem has been identified in literature [32] [25] [26], and has been classified as a main usability challenge of blockchain technology [32]. The difficulty with accessing data of smart contracts specifically has been identified by Bragagnolo et al. [3].

With blockchain technology becoming more popular [31], and more application logic getting moved to the blockchain in the form of Smart Contracts [1], it becomes more important to be able to efficiently query historical and current Smart Contract application data. This allows users and maintainers to gain insight, analyze, and report about the behaviour of the system. However, as one of the main features of blockchain technology is the level of data integrity it provides due to its immutable nature, it is important for the resulting data from a query to also have a level of data integrity. If this level of data integrity would not be important for an application, blockchain technology might not be suitable as a whole. Furthermore, it should be possible for other parties to verify the resulting data. A blockchain network rarely consists of only one party, so resulting analyses and reports could be relevant to other parties.

At this moment there are a large number of non-interoperable blockchain implementations on the market. This fragmentation is often considered one of the main causes of the slow adoption of blockchain technology in industry [7]. Also, the immutable nature of blockchain makes it difficult to modify deployed Smart Contracts [8]. Furthermore, due to blockchain being a relatively young technology, the industry lacks expertise. Therefore, querying the blockchain would be most useful if it could be done without any change to the blockchain implementation or Smart Contracts, and without expertise in blockchain technology. In other words, the query application should be as decoupled from the blockchain as possible, so that a person without knowledge of blockchain technology would be able to query existing Smart Contracts on existing blockchain implementations.

Existing solutions for querying blockchain data can be found in literature. However, these solutions are not compatible with existing implementations [30], disregard data integrity [25] [32] [26], are only available as a hosted service [16] [17] [30], or do not support all relevant contract data [3] [2] [9] [25] [26] [32].

To illustrate the problem, a case from industry is used throughout this document. Avanade[1], a global company providing IT consulting and services focuses on the Microsoft platform, provided this case. In this case, an application is built using an Ethereum based blockchain to allocate green energy, in which Smart Contracts represent producers and consumers of green energy. Due to the characteristics of blockchain technology, there exists an immutable log of all energy produced and transferred. However, due to the way this data is stored, it becomes difficult to access this data. Users of this network want to be able to retrieve this information easily and in a timely fashion, but while maintaining its integrity. Furthermore, they want to be able to share this data with other parties, who should be able to easily validate the data themselves.

1.1 Problem statement

We have identified a problem with current blockchain technology, namely the difficulty and inefficiency of querying current and historical Smart Contract data. Although an immutable history of all data regarding Smart Contracts is stored, it is difficult an inefficient to access this data.

This problem has been identified both in literature and at the company Avanade.

A solution to this problem would be most useful when it preserves data integrity, is compatible with existing implementations, does not require extensive knowledge of blockchain technology, and allows other parties to verify the data.

1.1.1 Research questions

To find a suitable solution to the identified problem, we formulate the following research question:

> How can we easily and efficiently query Smart Contract data from a blockchain and publish the result while maintaining data integrity?

To answer this question, we deconstruct it into a number of sub-questions:

RQ1 What data stored on the blockchain classifies as relevant application data?
RQ2 How can we easily and efficiently query smart contract application data from a blockchain?
RQ3 How can we assure data integrity of off-chain data?
RQ4 How can we allow other parties to verify the resulting data?

1.1.2 Research method

To answer these questions we create a proof of concept application based on existing knowledge found in literature. This proof of concept application should be able to efficiently query application data from a blockchain while maintaining data integrity. We try to validate the proof of concept application by using it with a real world use case. We evaluate its functionalities and performance by looking at possible real word scenarios, and compare it to a naive solution.

1.2 Contributions

In this research, we present a proof of concept application which allows for efficient querying of historical and current Solidity Smart Contract application data. We provide possible validation of the results by utilizing Merkle proofs. We are able to construct proofs for the results, which can be used by other parties to validate the results. To the best of our knowledge, our system is the first to allow for the efficient querying of both storage data and transaction data regarding Smart Contracts, while maintaining a level of data integrity. Furthermore, we present the results of an experiment comparing the performance of our proof of concept against a naive implementation.

1.3 Outline

Chapter 2 explores research found in literature related to our own research, to show how close we are to a solution for the described problem and to explore solutions to similar problems. In Chapter 3

[1]`https://www.avanade.com/en`

we describe concepts and technical terminology related to the research, to give context to the rest of the thesis. Chapter 4 describes our proposed solution to the problem, and explains the choices made. Chapter 5 describes a way to evaluate our proposed solution to see if it really solves the problem. The results of these experiments are shown and discussed in Chapter 6. Finally, we present our concluding remarks in Chapter 7 together with future work.

Chapter 2

Related work

In this chapter we explore research found in literature relevant to our own research. We divide the literature into three categories based on functionality and use case.

2.1 Query functionality

Li et al. [25] introduce a query layer on top of Ethereum, EtherQL, which allows for complex queries. They synchronize the Ethereum blockchain with a MongoDB database, and perform the queries on the data in MongoDB. Queries can be performed on block, account and transaction data. However, this implementation gives only limited information on Smart Contracts, as the function input of transactions and the variables of contracts are not decoded or stored, and it is not possible to query these values. They conclude that their implementation is effective and efficient in querying blockchain data.

Pratama et al. [32] expand on the query functionalities of EtherQL by allowing for multiple search parameters and a number of analytic functions. They conclude that their implementation has a lower throughput than Ethereum.

Lin et al. [26] introduce HyperQL, a query layer similar to EtherQL for the Hyperledger Fabric blockchain. They use PostgreSQL instead of MongoDB, because MongoDB does not support JOIN operations. Query performance of PosgreSQL seems better than query performance of MongoDB.

Bragagnolo et al. [3] introduce SmartInspect, a tool to inspect Solidity smart contracts. SmartInspect allows users to easily access states of contract instances, in human readable format. They do this by using a local pluggable mirror-based reflection architecture, which is used to extract and structure the state of a Smart Contract from the blockchain.

Bragagnolo et al. [2] introduce the Ethereum Query Language, which allows users to retrieve information from the blockchain by writing SQL-like queries. Information on blocks, transactions and accounts can be easily retrieved. This implementation has limited support for querying information regarding Smart Contracts, as it does not allow for the querying of contract attributes or arguments in function calls. The Ethereum Query Language indexes the hashes of blockchain data to allow for improved query performance.

Ducasse et al. introduce SmartAnvil [9], an open-source suite for smart contract analysis, which includes SmartInspect and a new version of the Ethereum Query Language, called Ukulele. Ukulele now allows for the querying of attributes of contracts, but does not support querying the arguments in function calls. Furthermore, the suite allows for static analysis of smart contracts.

Helmer et al [19] take a different approach to providing query functionality on a blockchain. They introduce EthernityDB, which maps database functionality to Ethereum smart contracts to provide database functionality on a blockchain. They conclude that the costs of storing data on the Ethereum blockchain does not make it a viable option.

2.2 Frameworks for static blockchain analysis

Kalodner et al. [23] introduce Blocksci, an blockchain analysis platform which mainly focuses on Bitcoin. It does not support Ethereum. It uses an in-memory database to improve performance. Research shows that the application allows fast and complex analysis of blockchain data.

Bartoletti et al. [1] introduce a general framework for blockchain analysis, supporting both Bitcoin and Ethereum. Their approach allows for processing of external data besides the data already present on the blockchain, and can use both MySQL and MongoDB as underlying database. It does not support analysis of smart contracts.

2.3 Hosted services

Another solution is The Graph [17], which is a decentralized protocol which indexes data from blockchains, and provides query functionailities using the GraphQL API. Users can define subgraph manifests, which describe how contracts and data should be indexed. The Graph provides a hosted service that indexes blockchain data based on the subgraph descriptions. Indexed data is stored in a PostgreSQL database on IPFS, and can be accessed via the GraphQL protocol. Although promising, this project is still in its early stages and does not seem to support permissioned private blockchain implementations. Also, payment will be required for certain functionality [18]. Furthermore, it does not provide anything for users to validate the resulting data by comparing it to the blockchain.

Eth.events [16] provides a more centralized service to query blockchain data. The user does not have to run their own node in the network to query blockchain data. However, the service only provides limited support for private permissioned blockchains, and requires users to trust the data eth.events delivers. Furthermore, it does not provide a mechanism for users to validate the resulting data by comparing it to the blockchain.

Microsoft provides the Azure Blockchain Workbench [30], which supports Ethereum. The Blockchain Workbench provides an SQL-accessible replica of the data stored in the blockchain, which is stored in an Azure SQL database. This allows for easy reporting and analytics on blockchain data. The service suits enterprise use cases, but is only compatible with blockchains hosted by Microsoft.

Chapter 3

Background

In this chapter we explain a number of concepts and technical terms relevant to our research. We look at different types of blockchain, and extensively describe the Ethereum blockchain. Furthermore, we explain the tree structures used in Ethereum. Finally, we describe the difficulty in querying data from a blockchain.

3.1 Blockchain

In 2008, Satoshi Nakomoto [29] introduced blockchain, the technology on which his peer-to-peer electronic cash system, Bitcoin, was built.

Iansiti et al. [20] identify five core principles of blockchain technology:

Distributed database Each node on the blockchain holds the entire database.

Peer-to-peer transmission Communication goes directly from node to node, without the need for a central agent.

Transparency with pseudonimity Every transaction is visible to every node. Nodes have an address as identifier.

Irreversibility of records The blockchain is immutable. Transactions are linked to previous transactions, which ensures that they are permanent and chronologically ordered.

Computational logic Blockchain transactions can be tied to computational logic, and can be programmed.

Crosby et al. [6] define a blockchain as a public ledger of transactions shared among participants. Transactions are verified by consensus of the majority of participants, and, once entered, cannot be erased.

Although blockchain technology was initially mainly used in the financial sector, it has recently also found use in other sectors. Ekblaw et al. [10] introduce MedRec, a system which gives people agency over their own healthcare data, built on blockchain. Jain et al. [21] introduce Seguro, a system which provides digital storage of documents using blockchain.

3.2 Permissionless/permissioned blockchain

Current blockchain technologies can be categorized as either being permissionless or permissioned. Bitcoin [29], the first popular usage of blockchain, uses a permissionless blockchain. This means that everyone can join the network, send transactions, and verify transactions. A permissionless blockchain is truly decentralized, as anyone can participate. [35]

With a permissioned blockchain, a central party decides who can join the network, send transactions, and verify transactions. This results in a more centralized system. [35]

Brown[1] imagines a continuum, stretching from 100% decentralized to 100% centralized. He places Bitcoin-as-envisaged at fully decentralized, but argues that Bitcoin today shifts slightly more towards centralization due to mining centralization. Brown places Typical Core Banking System at fully centralized.

[1]`https://gendal.me/2014/11/14/the-unbundling-of-trust-how-to-identify-good-cryptocurrency-opportunities/`

Many permissionless blockchain systems such as Bitcoin use a proof-of-work consensus algorithm. Nodes in the network try to solve a mathematical puzzle, and the first node to solve this puzzle adds a block with valid transactions to the chain, and broadcasts this to other nodes. A problem with this proof-of-work consensus algorithm is the cost. The mathematical puzzles become harder and require more computational power to solve. This leads to transactions becoming more expensive.[35]

The big advantage of proof-of-work, as described by Nakamoto [29], is that it results in one-CPU-one-vote. It is costly to gain more votes, and thus difficult to gain a majority of the votes. This results in a consensus algorithm which works for a network of untrusted participants, as it protects against a Sybil attack. [35]

In trusted, permissioned systems in which the participants do not require anonymity, different consensus algorithms than proof-of-work can be used. As all participants are known and trusted, there is no way to create large amounts of pseudonymous identities, and thus no chance of a Sybil attack. An often used consensus algorithm used in permissioned systems is proof-of-authority, in which approved, authorized nodes can validate transactions. [35]

3.3 Ethereum

Ethereum [36] is a blockchain implementation which tries to extend blockchain technology to allow for more complex applications.

Ethereum is permissionless and uses a proof-of-work consensus algorithm. It provides the ability to execute instructions in a virtual state machine, the Ethereum Virtual Machine (EVM). The instructions are stored in smart contracts on the blockchain. These instructions can be executed by referencing the address at which the contract is stored on the blockchain in a transaction. Computational effort costs gas, which is related to Ethereums intrinsic currency, Ether.

The Ethereum blockchain consists of a number of blocks which are chained together using a cryptographic hash. These blocks contain transactions and a world state. Figure 3.1 shows the relationships between the different parts of the system.

In this section we explore the anatomy of the Ethereum blockchain, as well as some ways to use Ethereum and relevant terms.

Ethereum block [36]

A block in Ethereum consists of three parts: a block header, a number of transactions, and a set of other block headers. The block header contains, amongst others, the following items:

- **stateRoot** Keccak 256-bit hash of the root node of the state trie.
- **transactionsRoot** Keccak 256-bit hash of the root of the transaction trie.
- **number** Number of the block, genesis block has number zero.

Each individual block contains a reference to a specific state trie and a specific transaction trie.

Ethereum State [36]

The world state is a mapping between addresses and account states, stored as a Merkle Patricia tree. There are two types of accounts: contract accounts and non-contract accounts. The former means that the address is linked to a smart contract, while the latter isn't. Each account state contains, amongst others, the following fields:

- **storageRoot** Keccak 256-bit Hash of the root node of the storage contents of this account.
- **codeHash** Hash of the EVM code of this account.

Ethereum Storage [36] [3]

Each account is associated with a possible empty Storage State. This Storage State contains information regarding the account. This information can be changed by running the EVM code associated with the account. The information in the Storage State is encoded into slots accessible through contiguous indices. Each slot can store up to 32 bytes.

If an account is a Smart Contract, the Storage State contains the attributes declared in the code. However, how these attributes are stored depends on the type and the order in which they are defined in the Smart Contract. Figure 3.2 shows how the variables of an example contract are stored.

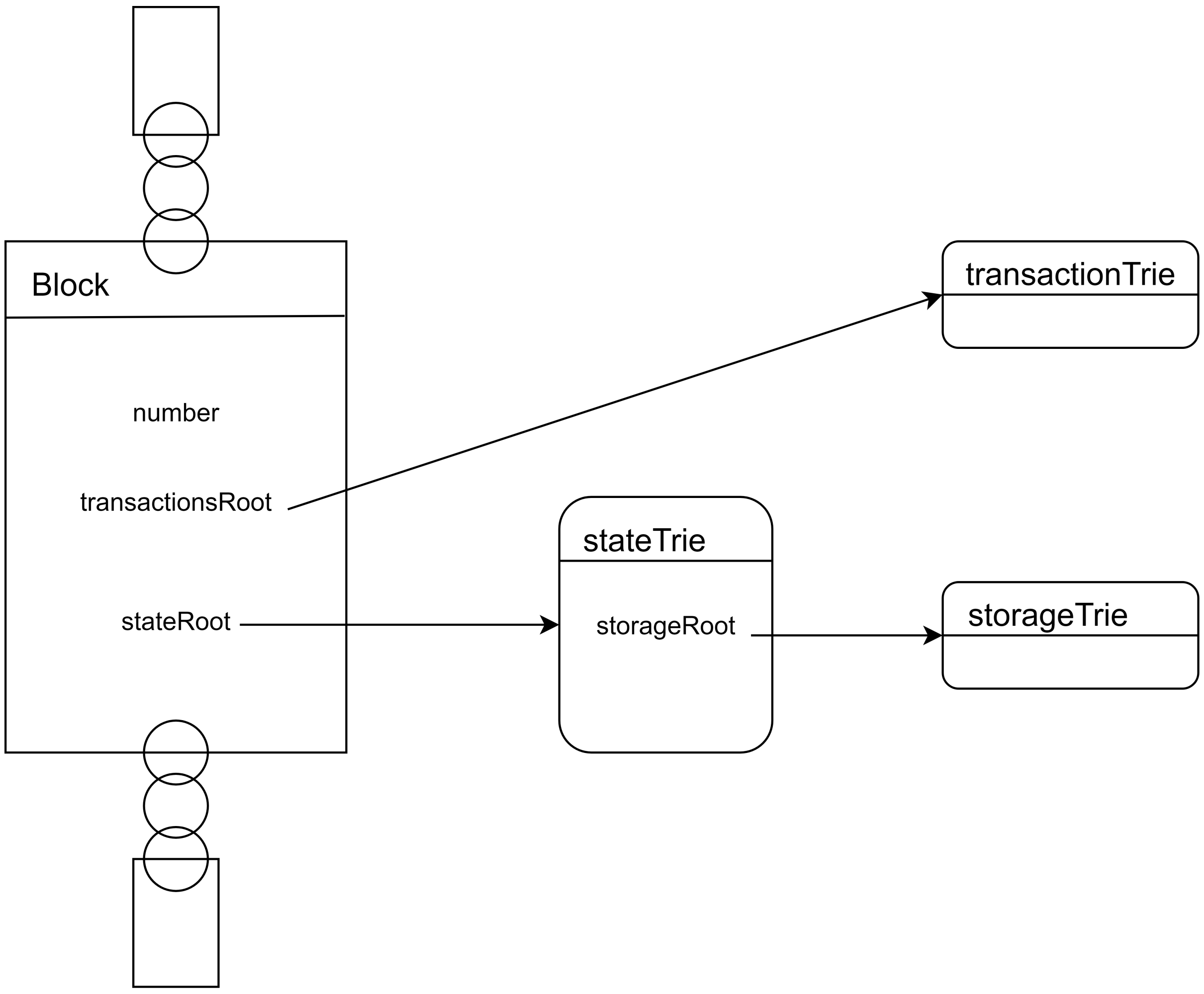


Figure 3.1: Relationships between parts of the Ethereum system.

Static types such as integers use up as much space as needed. For instance, one `int128` attribute uses 16 bytes. The compiler tries to optimize the space used, so it will try to fit the next variable into the same slot, if possible. For instance, if the next attribute defined in the Smart Contract is also of the type `int128`, it will use the remaining 16 bytes in the same slot, and the slot will contain two variables. The next attribute will not fit into the slot, and will be placed in the next slot. [34]

Structs and static arrays always start in a new slot, and their elements are stored as if they were given explicitly. [34]

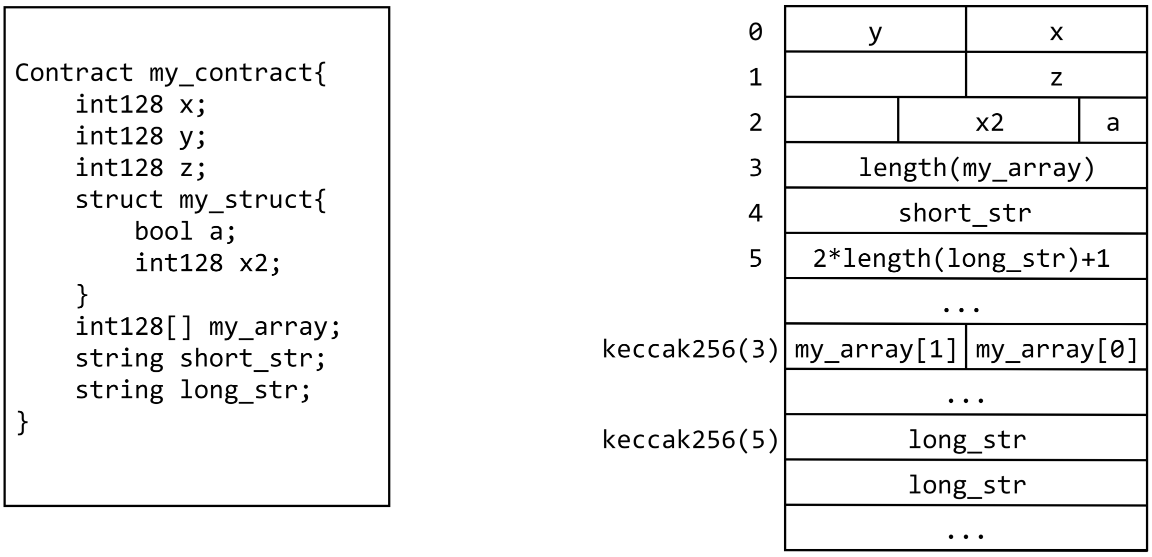
Dynamic types occupy one slot at position `p` in the same way as static types. In the case of dynamic arrays, slot `p` stores the number of elements in the array. For mappings, slot `p` is empty. The actual values of which the dynamic type is composed are stored at another location, which can be found at `keccak256(p)`, in which `keccak256()` is the Keccak-256 hash function. [34]

Byte arrays and strings are stored differently. Short byte arrays and strings, of which the data is at most 31 bytes long, are stored at the usual position `p`. The last byte stores the length of the byte array or string times two. If the data is larger than 31 bytes, slot `p` stores the length times two plus one. If this is the case, the actual data can be found at `keccak256(p)`, in which `keccak256()` is the Keccak-256 hash function. [34]

Ethereum transaction [36]

There are two types of transactions in Ethereum. The first type results in message calls, while the second type results in the creation of new accounts and contracts. They have a number of common fields:

- **nonce** Value equal to number of transactions sent by sender.
- **gasPrice** Value equal to the number of Wei to be paid per unit of gas.



```
Contract my_contract{
    int128 x;
    int128 y;
    int128 z;
    struct my_struct{
        bool a;
        int128 x2;
    }
    int128[] my_array;
    string short_str;
    string long_str;
}
```

Figure 3.2: Example Solidity contract with variables and how they are stored.

- **gasLimit** Value equal to the maximum amount of gas that can be used in this transaction.
- **to** Address of recipient.
- **value** Value equal to number of Wei to be transferred.
- **v,r,s** Values corresponding to signature of transaction and sender.
- **init/data** init for contract creation, data for message call. Unlimited size byte array specifying EVM-code or input data.

A transaction is a type of permanent data. Once recorded on the chain, it is never altered.

Full client and light client

The Ethereum network consists of different types of nodes/clients. Full clients store a local copy of all data stored on the blockchain. However, because data is only ever added to a blockchain, and never removed, this data set can become very large. Therefore, Ethereum supports light clients. These light clients only download the headers of blocks by default, and verify only a small part of what needs to be verified. Because the headers contain the roots of the Patricia Merkle trees in which all data is stored, light clients are able to verify any additional data relevant to them. This makes light clients suitable for low-capacity environments, such as personal computers or smartphones. Most full clients eventually become partially light clients, which only store the full data of the last couple of thousand blocks. [13]

Solidity Smart Contract

Solidity [34] is a popular language used to program smart contracts on the Ethereum platform. Solidity code can be compiled to executable bytecode, which can be deployed on the blockchain to create instances of the contract. A Solidity smart contract consists of two main components: attributes and functions. The attributes store the current state of the smart contract. The functions can modify the values of the attributes of the smart contract, return the values of the attributes, call functions of other contracts, and deploy instances of smart contracts. Participants on the network and deployed smart contracts can call an instance's functions by sending a transaction containing the function call to the relevant instance.

Interaction with contracs is usually done via the ABI, the Application Binary Interface, of the contract. The ABI describes, amongst other things, the functions and variables of the contract.

RLP [15]

Ethereum uses RLP (Recursive Length Prefix) to encode arbitrarily nested arrays of binary data. It is used to serialize objects. RLP encodes structure, but does not provide data types. The RLP encoding function takes as input an item, which is either a string or a list of items. How an item is encoded depends on the length of either the string or the list. The resulting encoding includes the length of the item. To decode, subsets of data are taken from the front of the encoded value sequentially. The size of these subsets can be found in the encoded value.

Quorum

Quorum is a permissioned blockchain implementation based on Ethereum, which focuses on use in enterprises [33]. It is developed by J.P. Morgan. J.P. Morgan recently announced a partnership with Microsoft to drive enterprise adoption of Quorum [22]. Quorum is one of the most popular permissioned blockchain implementations based on Ethereum, and one of the most popular permissioned blockchain implementations in general. It uses a fork of the Go Ethereum Client. It provides transaction and contract privacy, different consensus algorithms, network management, and a higher performance than Ethereum. [33]

Geth

Geth, or Go Ethereum, is the official Golang implementation of the Ethereum protocol. It allows users to run an Ethereum node, which allows users to, amongst other things, mine ether, transfer funds, create contracts, and send transactions. The user can interact with Geth either through the Javascript Console, or the JSON-RPC server. [11]

JSON-RPC is a light-weight remote procedure call protocol for data in JSON format. It provides a number of methods which can be used to interact with the Ethereum blockchain.[12]

3.4 Tree structures in Ethereum

Ethereum uses tree structures to store much of its data, such as transactions and account storage. Ethereum makes use of the Modified Merkle Patricia trie, which contains characteristics of both the Merkle tree and the Patricia tree. In this section we explain these trees and how they are used in Ethereum.

3.4.1 Merkle Tree

Merkle tree [27] is a tree structure. Every leaf contains a hash of data, and every internal node contains a hash of its children's hashes. When data in the tree is added or modified, the hashes from the modified part of the tree to the top hash are recalculated. As a result, a Merkle tree provides a single value which identifies specific data. By comparing the top hash, Merkle trees provide efficient verification of large amounts of data. Proof of inclusion can be done with Olog(n) complexity. An example of a Merkle tree can be found in figure 3.3, in which `p,q,r,s` are data elements, and `H()` is a hash function.

3.4.2 Inclusion proof [24]

Inclusion proofs allow one to prove that a Merkle tree with a specific root hash contains specific data. A Merkle Proof consists of the node hashes necessary to compute the root hash, in the order of deepest node to node closest to root.

To verify that a specific Merkle tree contains a specific value, one needs a trusted root hash of this Merkle tree, the value one wants to verify, and a Merkle proof. By first hashing the mentioned value, and thereafter continually hashing the resulting hash with the next hash in the Merkle proof, one gets a new root hash. One can then compare this new root hash with the trusted root hash. If they are equal, one has proven that the Merkle tree with the trusted root hash contains the specified value.

As an example, we can look at figure 3.4, which shows the hashes included in the proof for `r`. As in the picture, this proof exists of `H(s)` and `H(H(p) + H(q)`. If we want to verify that a Merkle tree with a trusted root hash `X` includes data element `r`, we hash `r` to create `H(r)`. Then we hash this with `H(s)`, which is included in our proof, to create `H(H(r) + H(s))`. Next, we hash this result with `H(H(p) + H(q)`, which is also included in our proof, to create the root `H(H(H(p) + H(q)) + H(H(r) + H(s)))`. If this calculated root is equal to the trusted root hash `x`, the Merkle tree with root hash `x` includes data element `r`.

3.4.3 Patricia Tree [28]

Patricia Trie is a tree structure related to the prefix tree and radix tree.

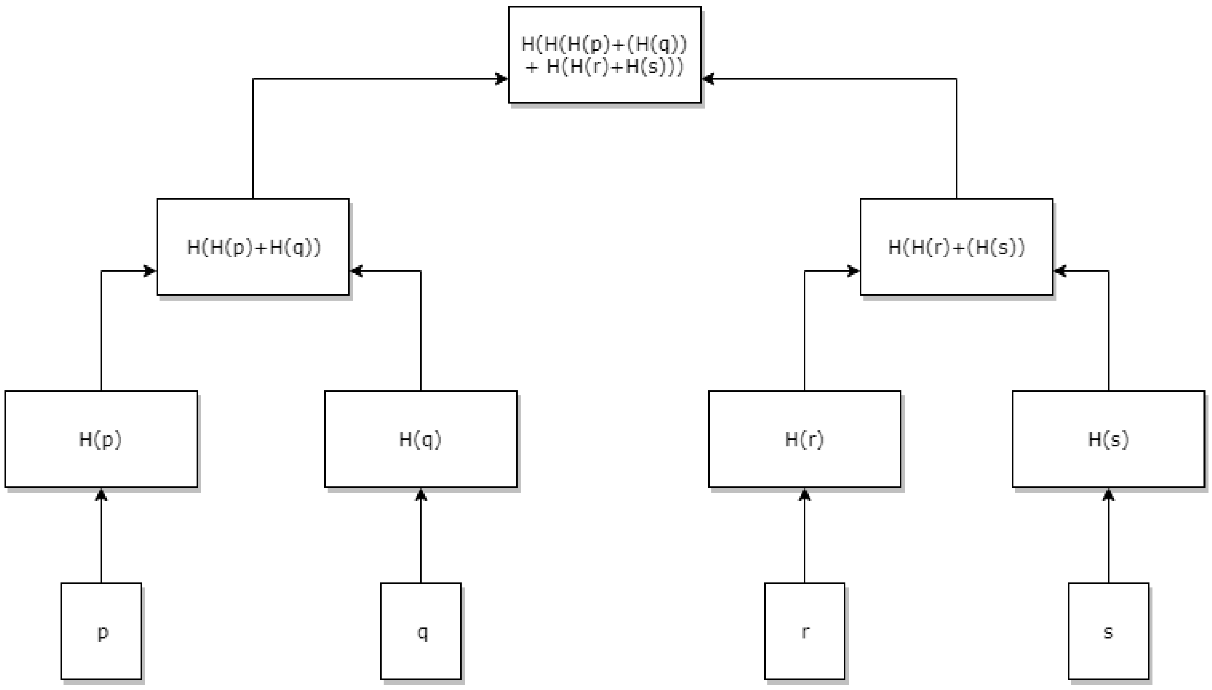


Figure 3.3: Example Merkle Tree in which p,q,r,s are data elements, and H is a hash function.

A prefix trie is a search tree, which is often used to store strings. The position of each node in the trie specifies which key it is associated with. When storing strings, each node contains a character. Starting from the root of the trie, which is associated with an empty string, one can create a path to form a word, and find the associated value.

A radix trie is a compact, space optimized, prefix tree. Each node that is an only child is merged with its parent. Therefore, each node has a maximum number of children that is equal to its radix.

A Patricia trie is a radix trie with a radix equal to 2, making it a binary radix trie. Patricia stands for Practical Algorithm To Retrieve Information Coded In Alphanumeric.

3.4.4 Modified Merkle Patricia Tree

Ethereum's data structure combines characteristics of both the Merkle Tree and the Patricia Tree. Merkle Trees are great to efficiently authenticate data, but quite inefficient to edit. This does not matter when it stores transactions, as transaction trees are immutable, but it becomes a problem when storing the states, as the state tree is frequently updated. The modified tree used by Ethereum, which incorporates characteristics of the Patricia tree, allows us to quickly calculate a new root when editing the tree. Furthermore, the modified tree has a bounded depth, which prevents denial of service attacks by making the tree so deep that updates become very slow. Furthermore, the modified tree has a root which depends only on the data, and not on the order of the data. [5]

To make the tree cryptographically secure, each node is referenced by its hashed value. This hashed value depends on the children of the node. This value is usually used to look up the actual data in a leveldb database. [4]

The tree consists of four types of nodes: Blank, Leaf, Extension and Branch. Blank nodes are empty. A Leaf node consists of a key and a value. An Extension node also contains a key and a value, but the value corresponds with a hash of another node. Branch nodes are lists with a length of seventeen, in which the first sixteen elements are the 16 possible hex characters in a key. The seventeenth element can hold a value. [4]

They key under which a value is stored is encoded into the path one has to take. This key is represented as a string of hexadecimal characters, or nibbles. When one wants to find the value which corresponds to a key, one starts at the root node, and at each branch chooses the path corresponding with the next nibble in the key. [4]

An example of the Modified Merkle Patricia Tree can be found in figure 3.5, in which the following key-

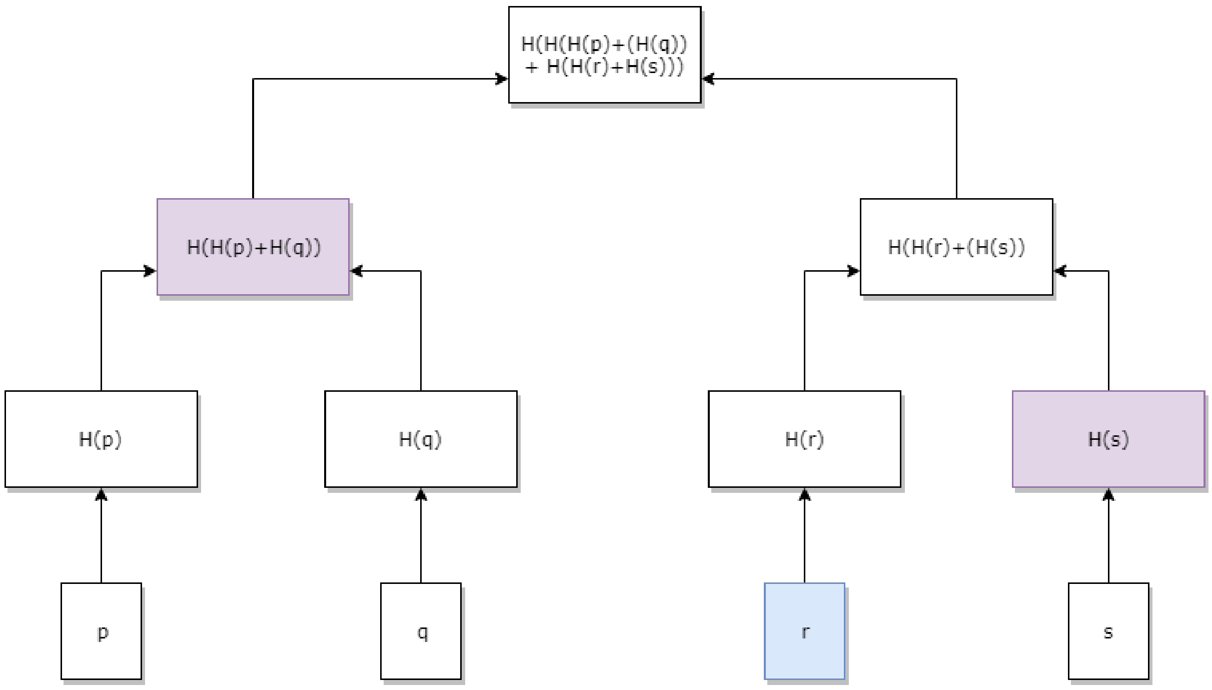


Figure 3.4: Example Merkle tree in which the hashes included in the proof for r are marked.

value pairs are stored: `(do,verb)`,`(dog,puppy)`,`(doge,coin)`, and `(horse,stallion)`. This example is taken from the Ethereum Wiki [14].

Merkle Proofs in Ethereum Trie

A Merkle proof for the Modified Merkle Patricia tree used by Ethereum is different from a Merkle proof for a regular Merkle Tree. Because the root hash is still dependent on all data in the tree, many of the same concepts still apply.

A Merkle proof for a value in the modified tree consists of a key, the value found at this key in the tree, and all nodes in the path dictated by the key, starting from the root. The proof shows that the tree has the given value at the end point of the given path. It is possible to verify that the value is at said position in the key, because the hashes of all nodes going up are dependent of one another. It is impossible to provide a proof for a path and value that does not exist, because the root hash is based on the hashes of its children. [14]

An example of a Merkle proof for a value in the modified tree can be found in figure 3.6, in which the nodes included in the proof for the value `puppy` are highlighted. The proof consists of these nodes, the value `puppy`, and the key `64 6f 67`, which is the string `dog` converted to bytes, presented in hexadecimal format.

3.4.5 Types of tries in Ethereum

Ethereum uses a number of different tries which store different kinds of data. All tries are Modifed Merkle Patricia Tries. The header of each block stores the roots of three of these tries, namely the State trie, the Transaction trie, and the Receipts trie. [14] Figure 3.1 shows how these tries are related.

State Trie

There is one State trie, which maps addresses to accounts. A path in this trie is formed by `sha3(address)`, and a value is formed by `rlp(account)`. Each accounts stores the root of a Storage trie. [14]

Storage Trie

Each account is linked to a Storage trie, which contains all contract data. The path is `sha3(sha3(variablePosition))`, and a value is formed by `rlp(variableValue)`. [14]

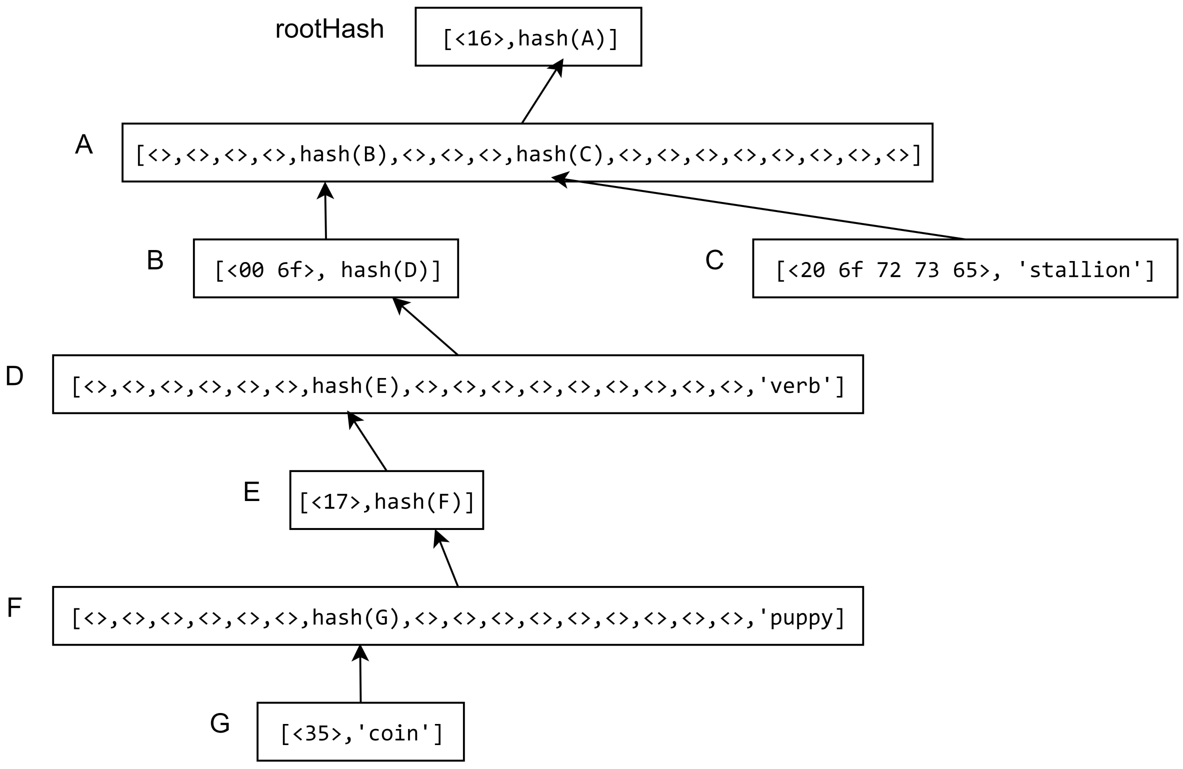

rootHash
[<16>,hash(A)]
A
[<>,<>,<>,<>,hash(B),<>,<>,<>,hash(C),<>,<>,<>,<>,<>,<>,<>,<>]
B
[<00 6f>, hash(D)]
C
[<20 6f 72 73 65>, 'stallion']
D
[<>,<>,<>,<>,<>,<>,hash(E),<>,<>,<>,<>,<>,<>,<>,<>,<>,'verb']
E
[<17>,hash(F)]
F
[<>,<>,<>,<>,<>,<>,hash(G),<>,<>,<>,<>,<>,<>,<>,<>,<>,'puppy]
G
[<35>,'coin']

Figure 3.5: Example of Modified Merkle Patricia Tree.

Transactions Trie

Each block has a Transactions trie, which stores all transactions stored in this block. In this trie, the path is `rlp(transactionIndex)`, and the value is `rlp(transaction)`. [14]

3.5 Querying data from a blockchain

It is possible to retrieve data from a blockchain, but there are some limitations. For instance, using the API Geth provides, one can retrieve all kinds of data directly from the Ethereum blockchain, like information on blocks, transactions and accounts. However, to retrieve this data directly, one often needs to know the unique identifier of the element. If one does not know this identifier, the only option is to go through the blockchain sequentially, block by block, and search for the wanted data element. [2]

Storing the unique identifiers in a second database, and using these to query data, can make this process more efficient, as an element can be accessed directly. However, the unique identifier of a data element does not give any information on the type of data, or the data itself. In other words, it does not provide any context, which makes storing only identifiers quite useless when querying specific data. [2]

Walking through the blockchain sequentially is, due to the many RPC calls, quite inefficient. The time it takes to walk through the blockchain becomes larger as the blockchain grows. Due to the immutable nature of the blockchain, blocks can never be removed, and a blockchain can only become larger with use. This means that, whenever a transaction is sent over the network and is appended to the blockchain, walking through the blockchain will take longer, and querying information this way will take longer. [2]

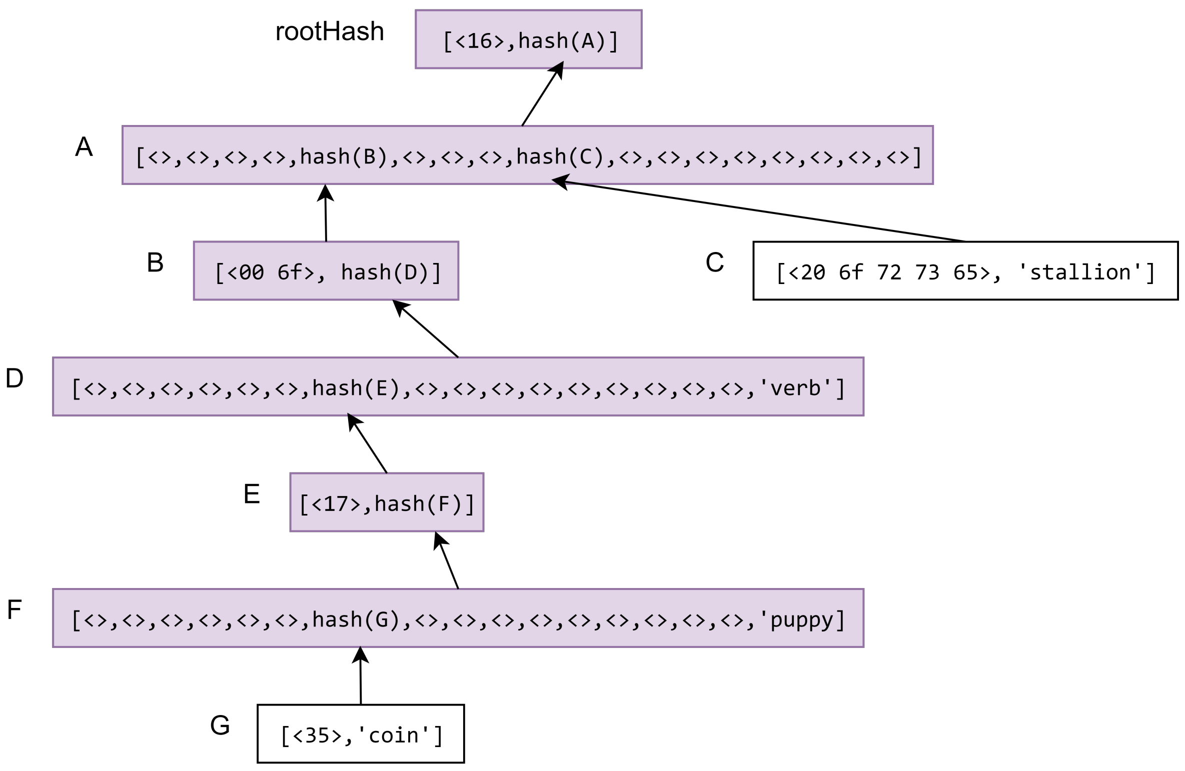

rootHash
[<16>,hash(A)]
A
[<>,<>,<>,<>,hash(B),<>,<>,<>,hash(C),<>,<>,<>,<>,<>,<>,<>,<>]
B
[<00 6f>, hash(D)]
C
[<20 6f 72 73 65>, 'stallion']
D
[<>,<>,<>,<>,<>,<>,hash(E),<>,<>,<>,<>,<>,<>,<>,<>,<>,'verb']
E
[<17>,hash(F)]
F
[<>,<>,<>,<>,<>,<>,hash(G),<>,<>,<>,<>,<>,<>,<>,<>,<>,'puppy]
G
[<35>,'coin']

Figure 3.6: Example of Modified Merkle Patricia Tree in which the nodes included in the proof for the value "puppy" are marked.

Chapter 4

Implementation

In this chapter we introduce our proof of concept application and explain our choices in the implementation. We first define the requirements based on the problem found in Chapter 1. After this, we explain the high-level design of our application, after which the actual implementation is explained. Lastly, we explain how the system can be used, and we compare our solution to other possible solutions.

4.1 System requirements

We identified a problem with blockchain technology, namely the fact that querying historic and current data from Smart Contracts is difficult and not time efficient. To try to solve this problem, we create a proof of concept application.

We design an application which facilitates easy and efficient querying of application data of existing Smart Contracts on existing blockchain implementations, while preserving data integrity for the querying party and any parties the data is shared with. By looking at this goal and the current state of blockchain technology, we formulate a number of requirements.

- **Query functionality**

The application should provide the user with complete information regarding the application data of selected Smart Contracts. This means that the user should be able to query all interactions with the selected Smart Contracts, as well as the resulting states of the Smart Contracts.

The user should be able to retrieve specific information about specific Smart Contract instances, as well as information which adheres specified conditions.

- **Performance**

Querying application data from a blockchain directly is often not time efficient. Due to the way this information is stored, querying application data often requires going through every block on the blockchain, decode the data, and check if its relevant, needing many JSON-RPC calls [25] [2]. Preliminary experiments show that querying smart contract data on a blockchain on which 10,000 transactions are stored can take up to 20 minutes. However, when querying data, the user usually expects the resulting information as fast as possible, preferably in a matter of minutes. Our application should give the results of a query faster than simply reading out the whole blockchain would.

- **Data integrity**

One of the key features of blockchain technology is the practical immutability of data stored on the blockchain. This practical immutability assures a level of data integrity. This level of data integrity is often an important reason for choosing to use blockchain technology for an application.

Because this data integrity is often an importance aspect of the application, our application should preserve this data integrity. In other words, the resulting data from a query should still have a level of data integrity.

Also, the data should be complete. The user should be able to query all relevant data currently stored on the blockchain.

- **Compatibility**

At this moment there are a large number of non-interoperable blockchain implementations on the market. This fragmentation is often considered one of the main causes of the slow adoption of blockchain technology in industry [7].

Creating a whole new blockchain implementation, or requiring fundamental changes to existing implementations to serve our goal, would cause more fragmentation. Furthermore, this would make the application very impractical, as it would require users to make a possibly costly switch to a new system.

To prevent further fragmentation and prevent costly switches to new systems, our application would be most useful if it could be used with existing popular blockchain implementations, without any changes to these implementations.

Also, the immutable nature of blockchain makes it difficult to modify deployed Smart Contracts [8]. Therefore, our application would be most useful if it could be used with already deployed Smart Contracts.

To make the application both work with existing blockchain implementations and deployed Smart Contracts, we should make our application as decoupled from the blockchain as possible.

- **Usable without extensive knowledge of blockchain technology**

To understand the application data of an application deployed on a blockchain, one does not need extensive knowledge of blockchain technology. A Smart Contract deployed on a blockchain is in many ways similar to local or internet applications, in that it has functions and variables, and often represents a real word entity. Therefore, it would be useful if retrieving this application data from a blockchain would also not require extensive knowledge of blockchain technology.

Furthermore, to make the application easier to use, querying the data should be similar to how one usually interacts with a traditional database.

- **Possibility to share data while maintaining integrity**

When one analyzes application data, one might want to share this information with other parties. To make this information more useful, data integrity should be preserved.

Therefore, our application should make it possible for other parties to validate given application data.

4.2 System design

We base the design of our implementation on the work of Li et al. [25] and Lin et al. [26]. Like those implementations, we synchronize the blockchain with a traditional, off-chain database. This provides two advantages. First, as shown by Li et al. and Lin et al., querying a local database is much faster than querying the blockchain. By limiting the amount of RPC calls to the blockchain, in other words limiting the interaction with the blockchain, we can expect higher performance when querying data. Second, this way we can easily utilize the powerful query functionalities of a traditional database with data off a blockchain.

Where Li et al. [25] and Lin et al. [26] focus on indexing accounts, transactions and blocks, our application focuses on indexing decoded information regarding the current and historic state of smart contracts, and the interactions with smart contracts. Furthermore, as opposed to the work by Li et al. and Lin et al., we provide a way to ensure a level of data integrity over the indexed data, and we provide a way to share queried data while maintaining a level of data integrity.

The only blockchain implementations the system currently supports are Ethereum and platforms based on Ethereum, such as Quorum. It could possibly be extended to other blockchain implementations. The reason Ethereum is chosen first is because it allows for application logic on the blockchain in the form of Smart Contracts.Furthermore, there is already some research done on efficient querying on the Ethereum blockchain [25] [2], and the platform is very popular, public and in enterprise. Currently the system only supports the Go Ethereum (geth) client[1], which is a popular Golang implementation of the Ethereum protocol.

The only database the system currently supports is PostgreSQL, but it could possibly be extended to support other databases. The choice for PostgreSQL was made based on the research by Lin et al.

[1] `https://github.com/ethereum/go-ethereum`

[26], which states that PostgreSQL performs better when querying blockchain data than MySQL and MongoDB.

The system is built with Python 3. This choice was made due to the fact that it allows us to make a local application using mature libraries to interact with the Ethereum blockchain. It makes extensive use of the Web3.py library[2] to communicate with the blockchain and manipulate data from the blockchain. The system uses the pyrlp library[3] to encode data to- and decode data from Recursive Length Prefix encoding. It also uses the py-trie library[4] to create local Merkle tries. Furthermore, it makes use of parts of the Web3.py fork from Ouvrard[5] to retrieve and verify certain Merkle proofs. Ouvrard was the first to implement support for this functionality in Web3.py. The system uses the Psycopg2 library[6] to communicate with the PostgreSQL database.

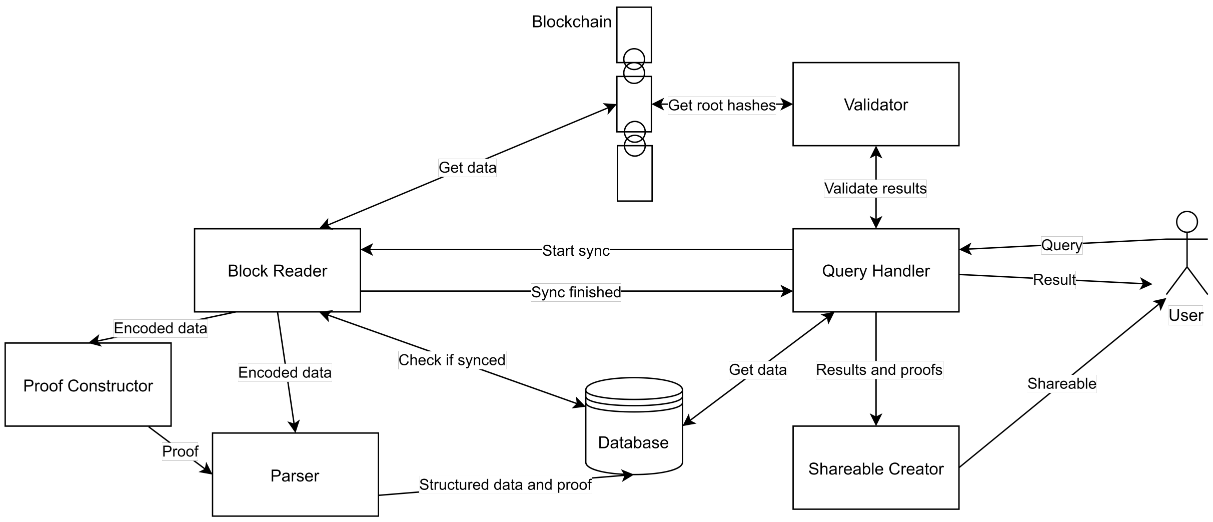


Figure 4.1: Interaction between modules in system.

The functionalities have been divided into six modules, which are explained in the next sections. Figure 4.1 shows the interaction between the different modules, and is explained below.

- **Block Reader**

The Block Reader synchronizes our local database with the blockchain. Every time a user issues a new query, the module checks if the local database is up to date. If it is not, the missing data is retrieved from the blockchain and sent to the Parser module and the Proof Constructor module.

Another approach would be to actively listen for new blocks, so the local database is synchronized with the blockchain at all times. This approach has been explored by Li et al. [25]. Although this could improve the performance of queries, it would add complexity and overhead to the application, as a service listening for new blocks would have to run at all times. Which approach is better depends on the use case and priorities. If a machine is available to constantly run this service and query speed is prioritized, actively listening for new blocks might be more suitable. If the application runs on a personal computer, it could be beneficial to let the user choose when the application synchronizes. However, the concept of synchronizing a traditional database with a blockchain while maintaining data integrity works the same for both approaches, as the concept does not depend on when the data is synchronized. Because of the reduced complexity, the fact that continuous synchronization has already been explored, and the fact that the concept does not depend on the timing of the synchronization, we chose to only synchronize when the user issues a query. However, we are still able to measure query performance when actively listening for new blocks by measuring query performance when the application is already fully synchronized, in other words, when no blocks have been added since the last query.

[2]`https://github.com/ethereum/web3.py`
[3]`https://github.com/ethereum/pyrlp`
[4]`https://github.com/ethereum/py-trie`
[5]`https://github.com/paouvrard/web3.py/tree/EIP-1186-eth_getProof`
[6]`http://initd.org/psycopg/`

- **Parser**

The Parser module receives information about a transaction on the blockchain or a storage entry on the blockchain from the Block Reader. This module parses the information into a human readable format, and specifies the database table and columns that should be updated. It also receives a proof for the data from the Proof Constructor module. It adds the parsed information with proofs into the database.

- **Proof Constructor**

The Proof Constructor receives data from the Block Reader, and constructs a Merkle inclusion proof for this data. This proof is then sent to the Parser Module.

- **Query Handler**

The Query Handler module receives queries from the user, and forwards these to the database. It sends the resulting data combined with proofs to the Validator module to be validated. When the Validator returns a positive response, the resulting data is returned to the user. If the Validator returns a negative response, the module gives the user an informative error message.

- **Validator**

The Validator module receives data and accompanying proofs from the Query Handler. It uses these proofs to test the validity of the data. If all information is deemed valid, a positive response is returned. If the data seems to be invalid, a negative response is returned.

- **Shareable Creator**

The Shareable Creator module receives the result of a query from the Query Handler module, and creates a machine readable, verifiable copy of the data. This copy is stored in a JSON file, which can be shared with other parties.

4.2.1 Interaction between modules

The interaction between the modules is shown in Figure 3.1.

When the user issues a query, this query is received by the Query Handler module. This module tells the Block Reader module to start synchronizing. The Block Reader module checks if the database is already synchronized. If it isn't, the Block Reader module requests the data of the next block from the blockchain, and sends this data to the Proof Constructor and Parser modules. The proof constructor sends a constructed proof to the Parser module. The Parser module stores the decoded and structured data into the database. Once the database is fully synchronized, the Block Reader module sends a message to the Query Handler Module. The Query Handler module sends the query from the user to the database, and receives the resulting data. This data is checked by the Validator module, which compares the data with the data on the blockchain. If the data is valid, the result is sent to the user. Optionally, the Query Handler module sends the data to the Shareable Creator to create a shareable version of the result.

4.3 System implementation

4.3.1 Configuring the system

By using the Configuration File, the user can specify which contracts they want the system to index. To help the application read the relevant information from the blockchain, the user has to provide some information about each contract. The information the user has to provide is kept to a minimum, to make the system easier to use. However, some needed information can not be read from the blockchain, so has to be provided by the user. The following data should be provided for each contract:

- **name** An arbitrary, unique name to identify the contract.
- **address** An address or multiple addresses to which the contract is deployed. Can be left empty if the contract is not yet deployed but can be deployed by other contracts.

- **creator_functions** A list of function names corresponding to a subset of functions of this smart contract which, when called, deploy new contracts.
- **variables_order** A list of the variables of this contract in the same order they appear in the Solidity code, which corresponds to the order in which they are stored in the memory of the Ethereum client.
- **abi** The abi of the contract.
- **deployed_bytecode** The deployed bytecode of the contract.

4.3.2 Relevant data

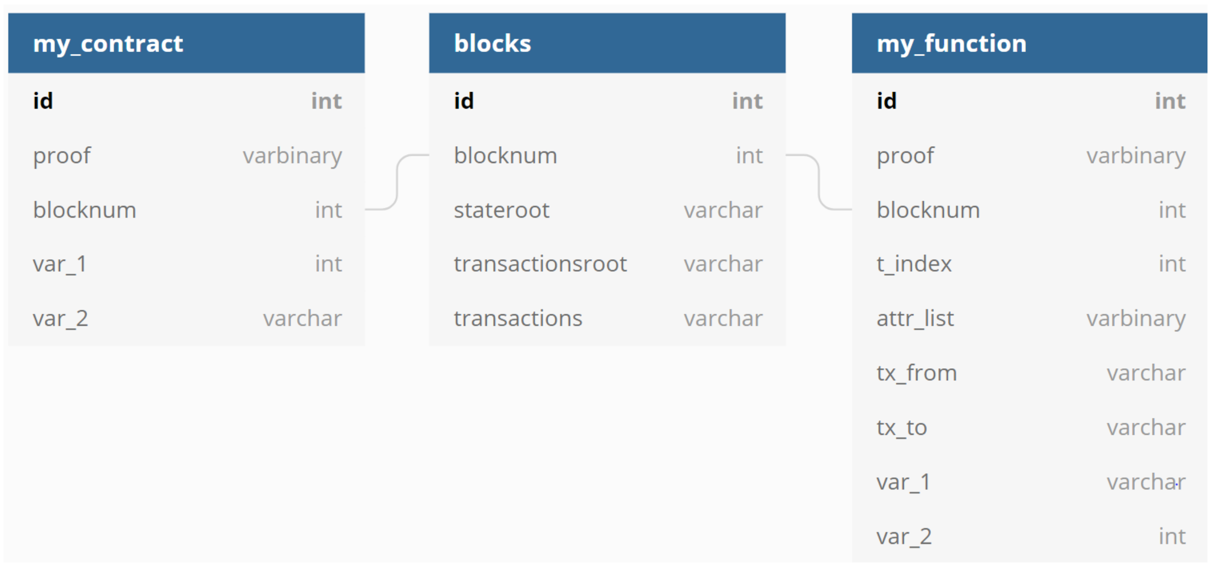
We need to know exactly what data needs to be indexed to provide users with a complete view of the data of their smart contracts. We do this by looking at what smart contract data is stored and how this is stored, as explained in Chapter 3.

End users want to know about historical and current states of smart contracts. Also, they want to know how this state was attained. We can find the current state, the value of the attributes of the instances, in the storage trie of each relevant contract in each block. Contract functions can change these values, so to provide a complete overview, we also index the function calls to each contract, which can be found in the transaction trie of each block.

We are interested in all data stored in the storage trie, because all information in here represents values of contract variables. However, we are not interested in all transaction data, as some of this data does not tell us anything about how the relevant state of the instance was attained. For example, the relevance of fields like `gasLimit` and `gasPrice` are limited to the blockchain domain, and play no role in how the state of the contract instance was attained. The only fields relevant for our purpose are the `data` field, which stores the arguments used in the function call, the `to` field, which allows us to identify function calls to different instances of the same Smart Contract by their address, and the `from` field, which allows us to see who sent the function call, thus who is responsible for the state change. Other information is stored to assist in creating proofs.

By also storing the relevant block number, and the index of each transaction, we are able to present the data in chronological order. This also allows the user to query specific ranges of states.

4.3.3 Storing the data



my_contract	
id	**int**
proof	varbinary
blocknum	int
var_1	int
var_2	varchar

blocks	
id	**int**
blocknum	int
stateroot	varchar
transactionsroot	varchar
transactions	varchar

my_function	
id	**int**
proof	varbinary
blocknum	int
t_index	int
attr_list	varbinary
tx_from	varchar
tx_to	varchar
var_1	varchar
var_2	int

Figure 4.2: Database Tables

At first start, the system reads the Configuration File. Python objects representing contracts and contract instances are created. After the file is read, database tables are created based on the gathered information. The following types of tables are created:

Blocks Table

This table contains information on all the blocks read by the system. There is only one version of this table and it always exists. The name of this table is simply "blocks". The following information is stored:

- **blocknum** Blocknumber, as a way to identify the block.
- **stateroot** The root of the State Trie of this particular block.
- **transactionsroot** The root of the Transaction Trie of this particular block.

Contract Table

This type of table contains the variables of each instance of a specific contract at each block. In other words, it contains the states of all instances of a contract at all blocks. Multiple instances of a contract can be stored in the same table as they contain the same variables. There are a number of default columns present in each table of this type. Other columns are specified by the variables in the contract. Tables of this type are named after the name of their respective contract, as dictated in the Configuration File. This type of table contains the following default columns:

- **proof** A binary string representing a Python dictionary containing a Merkle Proof for the respective data.
- **blocknum** The block at which the respective state was valid.

Function Table

This type of table contains the arguments of each call of a specific function. There are a number of default columns present in each table of this type. The other columns are specified by the arguments of the function. Tables of this type are named in the following way: $contractname + \_ + functionname$. This type of table contains the following default columns:

- **proof** A binary string representing a Python dictionary containing a Merkle Proof for the respective data.
- **blocknum** The block which contains the transaction containing the function call.
- **t_index** The index at which the transaction containing this function call is positioned in the block.
- **attr_list** A binary string representing a Python list containing a number of attributes of the transaction which contains the function call. Contains the following attributes: nonce, gasPrice, gas, to, value, v, r, and s.
- **tx_from** The address of the sender of the transaction containing the function call.
- **tx_to** The address of the receiver of the transaction containing the function call.

After the initially empty tables are created, the system starts the synchronization process.

4.3.4 Synchronization

To give complete results, the system should try to synchronize the blockchain with the traditional database, to make sure both contain the same data.

At each query, the system compares the highest blocknumber found in the traditional database with the highest blocknumber found on the blockchain. If the highest blocknumber found in the traditional database is smaller, the system starts reading blocks until they are equal.

From each block, the transactions and storage values are read. Reading a block is done by using a JSON-RPC call.

Transactions

The block contains an attribute *transactions* which contains a list of the hashes of all transactions stored in this block. We use a JSON-RPC call to get the details of each transaction. We also look at the receipt of the transaction by using a JSON-RPC call. This receipt contains an attribute *status*, which tells us if the transaction succeeded, and the function call was executed. We only index the transaction if it succeeded.

We use the function selector, which consists of the first 4 bytes of the *input* attribute of the transaction, to determine which function was called.

We use the associated Web3.Contract object to decode the function input. This gives us a list of the arguments used to call the function. This list, accompanied by a number of different attributes of this transaction and a computed proof, are then sent to the parser.

There is a special case where the called function is defined as a creator function in the Configuration File. This means that the function deploys a new contract instance on the blockchain, which we would also like to index. To do this we create a new local Contract Instance object. However, to index this new instance we need to know the address of this new instance. This address cannot be found in the transaction which contains the function call. But, because computing the address of the new contract is deterministic, we can find the new address. The address of the new instance can be computed from the address of the creator and the amount of transactions the creator has sent. Therefore, we can calculate a new possible address the following way.

```
sha3(rlp(senderAddress,nonce))
```

In the above, `sha3` stands for the SHA3 hash function, `rlp` for the RLP encode function, `senderAddress` for the address of the sender of the transaction, and `nonce` for the number of transactions the sender has sent, or the number of contracts a contract has deployed. We get the `senderAddress` from the transaction details, and keep count locally of the created contracts.

To get this `nonce`, we use a counter to count every time a contract instance deploys a new instance. However, because the user might not have specified all creator functions in the configuration file, we can not be sure we count every time a contract deploys a new contract. Therefore, we cannot be sure our value for `nonce` is correct. To solve this problem, we check if the calculated address contains the expected code. We start at our calculated `nonce`, calculate an address, and check if the bytecode at the address is equal to the bytecode belonging to the deployed contract. If it is not, we add one to the `nonce` and try again. If it is equal, we have found the address of our deployed instance. We add the address to the new Contract Instance object, so we can index this new instance.

Storage

For every contract instance we have stored, we want to find the value of every variable at each block. To find these, we simply use message calls to the used Ethereum client to get the value of each variable.

A proof is computed for the found values, and the values and proof are sent to the parser module.

4.3.5 Creating proofs

We want to be able to validate the data a query returns. We do this by constructing Merkle proofs for all data we index. This is done differently for the two types of data we index, transactions and storage. Our implementation is based on the implementation by Mitton[7].

We use Ethereums modified Merkle Patricia Tree for our proofs. This allows us to construct the same root hashes as those stored on the blockchain, which allows for easy comparison.

Transactions

We construct a Merkle proof for a given transaction. This Merkle proof should consist of a list of hashes necessary to calculate the root hash. We do this by creating a new local Merkle tree using the Ethereum pytrie library. We go through all transactions on the relevant block, RLP encode these transactions, and add these transactions to our local tree. When the tree is complete, we move down the tree as dictated by the key, and add each visited node to the proof. The key is the RLP encoded transaction index of our given transaction. This gives us a list with the root hash, the hashes necessary to calculate the root hash, and the relevant transaction hash.

Storage

To be able to validate variables of a contract, we construct a Merkle proof for the storage values of a given contract. We do this by simply using the RPC method `getProof(address,indices,blocknum)`, introduced by Ethereum Improvement Proposal 1186[8]. `address` is the address of the relevant contract, `indices` is a list of storage keys we want included, and `blocknum` is the number of the relevant block.

[7]`https://github.com/zmitton/eth-proof`
[8]`https://github.com/ethereum/EIPs/issues/1186`

This returns a dictionary containing a proof for the account, the hash of the root of the storage trie, and a storage proof. This storage proof consists of a value and a proof for each requested storage key. This proof consists of the hashes needed to calculate the root of the storage trie.

4.3.6 Validating data

When a query is issued, we want to make sure the resulting data is valid, and has not been tampered with. We do this by validating the relevant Merkle proofs.

When an user issues a query to the system, such as `SELECT col1, col2 FROM table1 WHERE col1 = 1`, the query is modified to include the proofs and data needed to validate the proofs in the query, and becomes `SELECT * FROM table1 WHERE col1 = 1`. This query is send to the database, and returns the resulting data combined with the proof for each data element. The system tries to validate this data using the proofs. Because proofs and data for both types are returned in different formats, this is done differently for transactions and storage values.

Transactions

To validate that a certain function was called with certain arguments at a specific block, we start with the arguments, the name of the function, and the other attributes of the function. These other attributes consist of the nonce, gasPrice, gas, to, value, v, r and s attributes of the transaction. Using the relevant Merkle proof, we can prove that a Merkle tree with a specific root hash includes a certain transaction hash. So, to mark the data from our database as valid, two conditions have to be met:

1. A transaction hash made with the data from our database should be equal to the transaction hash in our proof at the expected key.
2. A trusted root hash should be equal to the root hash in our proof.

Figure 4.3 shows schematically how transactions stored in the database are validated by comparing data from the database, the proof, and the blockchain.

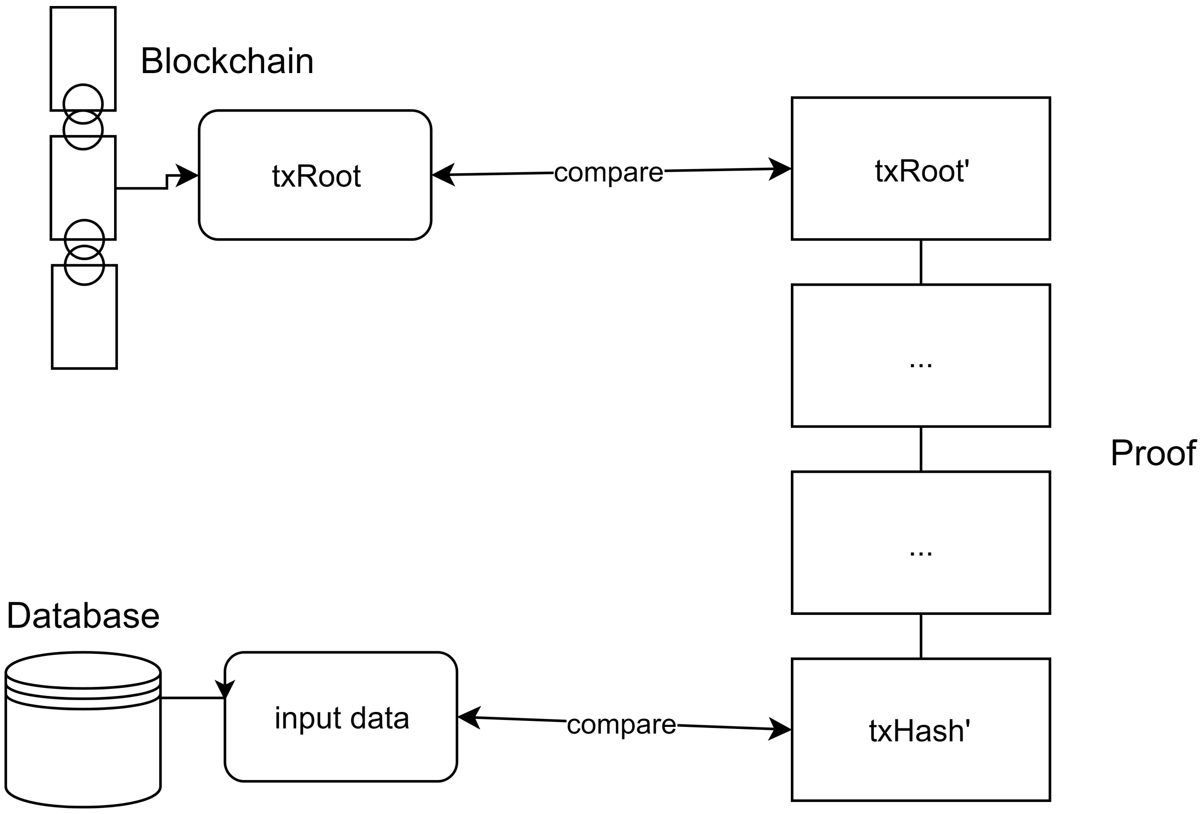


Figure 4.3: Validation of transaction.

• **Condition 1** For the first condition, we can calculate the transaction hash of the data from our database by RLP encoding a serialized list of the attributes of the transaction. Most of the attributes are already present, except for the input attribute, which is made up by the details of the function call. Because we know which function was called by looking at the database table queried, we know the ABI of the relevant contract by looking at the Configuration File, and we have received the arguments of the function call from our database, we can construct this input attribute. We do this by using the ABI. This gives us the input attribute, so we can calculate the hash of the transaction with the values stored in our database.

To get the transaction hash from our proof, we need our proof, our local root hash, and the index at which we expect the transaction hash. We walk through the tree in our proof as dictated by the key, which is the RLP encoded index, to get the transaction hash from our proof.

When the transaction hash we calculated with the data from our database is equal to the transaction hash we got from our proof, the first condition is met, and we know the Merkle tree with our local root hash contains the data from our database.

• **Condition 2** For the second condition, we compare the root hash from our proof to a trusted root hash.

We can find the root hash from our proof simply as the first element in our proof. We can find a trusted root hash by asking our Ethereum client with help from the Web3.py library.

If the root hash from our proof is equal to the trusted root hash, we know that the local Merkle tree we use for our proof is equal to the Merkle tree stored on our blockchain client.

Storage

To validate that the variables of a specific contract instance at a specific block had certain values, we start with the values from our database, the contract, and the proof. To mark the data from our database as valid, three conditions have to be met:

1. The values of the variables in the proof should be equal to the values of the variables from our database.
2. The state trie in our proof should contain an account at the expected key containing the root of the storage trie from our proof.
3. The root of the state trie in our proof should be equal to a trusted root of the state trie.

Figure 4.4 shows schematically how storage values stored in the database are validated by comparing data from the database, the proof, and the blockchain.

• **Condition 1** The data from our database is in a different format than the data from our proof. The data from our database is in human readable format, and has a type as specified by the ABI. This means an integer is stored as an integer equivalent, a string as a string equivalent, etcetera. The data from our proof is copied directly from the storage slots of our Ethereum client. Because the smart contract compiler dictates how this data is stored, decoding this data is non-trivial.

A first problem is that variables are stored in the same order as they are defined in the smart contract. However, this order is not always preserved when creating the ABI. Therefore, it is required that the user specifies the order of the variables in the configuration file.

Furthermore, how the variables are stored in the storage slots depends on the order and types of the data, as explained in Chapter 3. This makes it difficult to compare the data from our database with the data from our proof.

Fortunately, the ABI tells us the type of each variable. Because we know the type of each variable, the presumed value of each variable, and the order of the variables, we can find out where each variable should be stored in the storage slots.

We extract each variable from the presumed location in the storage slot, and compare it to the relevant variable we received from the database. Because the data is stored in the storage slots as bytes, we have to convert this to the relevant data type. When the compared data is not equal, we return an error.

If all data is equal, we know that the data in our database is equal to the data in the storage trie from our proof.

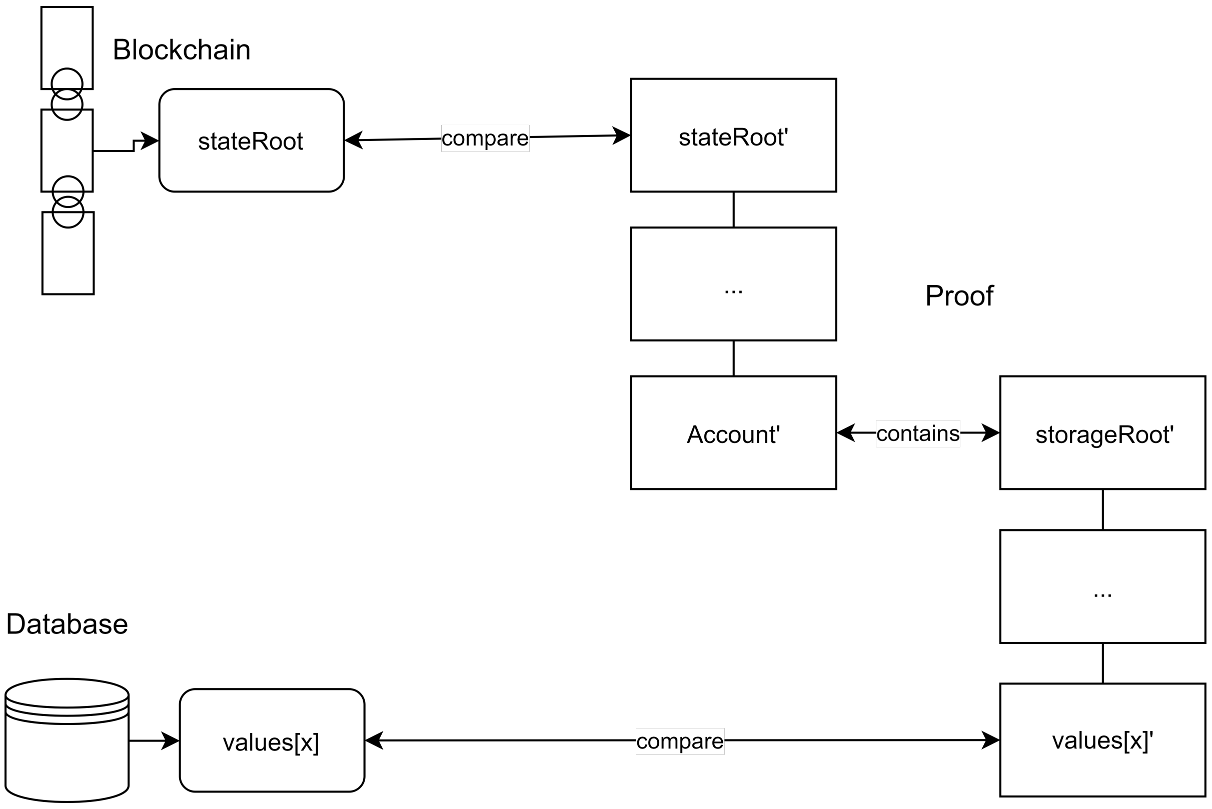


Figure 4.4: Validation of storage values.

• **Condition 2** We base our implementation on the implementation of Ouvrard[9].

We build an account object by RLP encoding a list containing the nonce, balance, storageRoot, and codeHash we find in the proof. We find the key at which it should be stored in the trie by hashing the address of the account found in the proof. Then, we walk through the trie as dictated by the key. We compare this result with the account object we just build.

If the account object we created is equal to the account object found in the proof, we know that the storage trie from our proof is valid if the state trie from our proof is valid

• **Condition 3** For the third condition, we compare the root hash from the state trie from our proof to a trusted root hash.

We can find the root hash from the state trie from our proof simply as the first element in the proof for the account. We can find a trusted root hash by asking our Ethereum client with help from the Web3.py library.

If the root hash from the state trie from our proof is equal to the trusted root hash, we know that the local state trie we use for our proof is equal to the state trie stored on our blockchain client.

4.3.7 Sharing data

We want to be able to share the results of our query with other parties, while allowing these other parties to verify the data themselves.

We do this by creating a JSON file which can accompany the result of a query. This JSON file contains all the information necessary to verify the data. A second application can be used to verify this data. The process of verification in this second application is similar to the process of verification in the main application. The only difference is that it does not read the necessary information from the traditional database synchronized with the blockchain, but from the provided JSON file.

To verify this data, this second application only needs the relevant Merkle roots, which are stored on the blockchain. This allows three types of parties to validate the data themselves:

- Full nodes in the same network.

[9]`https://github.com/paouvrard/web3.py/tree/EIP-1186-eth_getProof`

- Light nodes in the same network.
- Parties in a trusted relationship with either a full node or a light node in the same network.

4.4 System Usage

Once the configuration file has been filled out, the user can simply start the application from a terminal. A required argument when starting the application is the actual query, which should be expressed in SQL, similar to how queries are usually issued in PostgreSQL. Only `SELECT` queries are supported, with an optional conditional in the form of a `WHERE` clause. Optional flags are `--publishable`, which creates a JSON file which allows other parties to validate the data themselves, and `--reload`, which forces the application to re-synchronize the blockchain with the traditional database.

Once a query has been issued, the application synchronizes the traditional database with the blockchain. Once they are fully synchronized, the application retrieves the requested data from the traditional database. This data is then verified, after which it is returned to the user in the form of a list of lists. If the system detects an error in the data, an error message is returned.

After the configuration file has been filled out, the system allows people who do not have extensive knowledge of blockchain, but do have some experience with SQL, to retrieve application data from the blockchain.

4.5 Comparison to other solutions

There are a number of other solutions for querying smart contract data from a blockchain. However, these all have their own disadvantages regarding compatibility and performance.

4.5.1 Query directly off blockchain

A possible solution to query smart contract data from the blockchain is directly querying the blockchain. Our implementation makes use of this tactic, but serves as a middleman to provide better performance and more complex query functionality.

However, one could get this information by querying the blockchain client directly. For instance, if one wanted to know the current and past values of a specific variable of a specific contract instance, and the address of the instance is known, one could make multiple calls to the relevant getter function using the JSON-RPC. Retrieving this information would require as many JSON-RPC calls as indexing the information with our application would.

However, by not indexing this information, issuing this query twice would require two times as many JSON-RPC calls. Also, when the query includes a condition, unnecessary JSON-RPC calls are made, because we don't know the value before querying.

4.5.2 Implement in smart contract

One could implement the query functionality directly in the smart contract. If one knows what queries will be relevant before deploying the contract, functions can be implemented which perform the queries. By doing this, a simple call to the smart contract can return the results of the query.

Another solution would be implementing events, which fire when specific functions are called, and provide listeners with information about the function call. An off-chain application can listen to these events, and index the provided information in a similar fashion as our application.

A problem with both of these solutions is the fact that they require one to know which queries will be relevant before deploying the smart contract. This is because of the immutable nature of blockchains, which does not allow deployed smart contracts to be modified. Solutions to make smart contracts upgradeable exist [8], but these solutions still require one to implement the functionality to upgrade the contract before deploying it. Because of this, these solutions do not allow one to query already deployed contracts without these functionalities.

4.5.3 Comparing data instead of Merkle roots

To verify data queried from the traditional database, our solution asks the blockchain for the relevant Merkle roots. Another solution would be verifying the data received from the traditional database by querying the same data directly from the blockchain.

However, this solution requires us to issue a JSON-RPC call for every transaction, while in our solution, multiple transactions could require only one JSON-RPC call, because they could be stored in the same block, thus having the same Merkle root. This difference in amount of necessary JSON-RPC calls is even more apparent when querying for storage values, as they require a JSON-RPC call per variable, while our implementation can verify all variables of a contract in a block with only the relevant Merkle root, thus requiring only one JSON-RPC call.

Furthermore, this solution would not allow other parties that are light nodes to verify the resulting data, as they only store the block headers, and not all transactions and storage values in every block.

Therefore, comparing the actual data from the blockchain and traditional database instead of just the Merkle roots would be less efficient, and make it harder to verify the resulting data.

4.5.4 Store only data identifiers

A possible solution for efficient querying of blockchain data while maintaining a level of data integrity would be storing only identifiers locally. This is the approach the Ethereum Query Language uses [2]. Because most data elements such as blocks, transactions and accounts can be identified by relevant hashes, one could only store these hashes, and use these hashes to query the related data from the blockchain. The Ethereum Query Language stores the data in a Binary Search Tree, in which property values are linked to a set of hashes of elements corresponding to this value.

A problem with this approach is that by storing only the hashes, information and context is lost, which limits query functionality or performance. For instance, let's say one has stored a number of hashes and classified these as referring to transactions to a certain contract instance. If one wants to query all transactions to this instance in which a specific function was called with a specific value as input parameter, this information is missing, and the user would either be unable to issue this query, or issuing this query would require extra RPC calls, depending on the implementation. One could add context, such as the Binary Search Tree in the Ethereum Query Language. However, up until a point where all data is stored locally, this has a negative impact on query functionality or performance, due to the missing information. When one stores all data in a traditional database, a situation identical to the one described in subsection 4.5.3 emerges.

Chapter 5

Evaluation

We evaluate our proof of concept by testing it with a use case from industry. We evaluate it both on functionality and performance.

5.1 Use case

Avanade uses a number of Smart Contracts build with Solidity on an Ethereum based blockchain to manage the allocation of green energy. The company values the immutable and decentralized nature of blockchain technology which provides a level of data integrity for the current and past states of the system.

Avanade wants to keep a close eye on how the system is used, and they want to be able to share their findings with others. They want this analysis to be fast, and they want the results to be verifiable.

The system consists of three main components: Green Energy Production Location, Green Energy Consumer, and Green Energy Contract.

5.1.1 Green Energy Production Location

The Green Energy Producer Smart Contract represents a producer of green energy, such as a solar farm. The contract has a number of variables and functions.

The contract has the following variables:

- **owner** The address of the owner of the production location.
- **locationId** An integer used to identify the production location.
- **locationName** A string representing the name of the production location.
- **energyCount** An integer representing the amount of energy stored.
- **energySetsPerProductionDate** A mapping which maps energy sets to production dates.
- **lastDepositDate** The last date at which energy was deposited to this production location.
- **totalDepositsCount** The total number of deposits to this production location.

The contract has, among others, the following functions:

- **depositEnergy(_quantity,_productionDate)** Deposits _quantity energy with _productiondate to this production location.
- **consumeEnergy(energySetProductionDate,quantity)** Consumes quantity energy.

5.1.2 Green Energy Consumption Location

The Green Energy Consumption Location Contract represents a consumer of green energy. The contract has a number of variables and functions.

The contract has the following public variables:

- **owner** The address of the owner of the consumption location.
- **locationID** An integer used to identify the consumption location.
- **locationName** A string representing the name of the consumption location.
- **energyCount** An integer representing the amount of energy stored.

The contract has the following function:

- **allocateEnergy(quantity)** Allocates quantity energy to the Consumption Location.

5.1.3 Green Energy Contract

The Green Energy Contract serves as a form of manager, and is able to create new production locations and consumption locations. Furthermore, this contract can deposit and transfer energy between locations.

This contract has the following public variables:

- **owner** The address of the owner of this contract.
- **bankOwner** The address of the owner who can deposit energy.
- **allocationOwner** The address of the owner who can transfer energy.

The contract has, among others, the following functions:

- **createProductionLocation(_productionLocationName)** Creates a new production location with name _productionLocationName.
- **createConsumptionLocation(_consumptionLocationName)** Creates a new consumption location with name _consumptionLocationName.
- **depositEnergy(_productionLocationId, _quantity, _productionDate)** Deposits an amount of _quantity energy with date _productionDate to production location with id _productionLocationId.
- **transferEnergy(_productionLocationId, _consumptionLocationId, _energyBankProductionDates, _quantities, _transferDate)** Transfers an amount of _quantities energy from _productionLocationId to _consumptionLocationId.

5.2 Functional evaluation

We evaluate the functional requirements of the proof of concept by looking at a number of different scenarios.

5.2.1 Scenario 1: Publishing valid data

Avanade has deployed the Smart Contracts on a private Ethereum network. They want to know how much energy has been transferred per block between a production location with ID `0` and a consumption location with ID `0`. They want to publish this analysis, and they want other parties who read the publication to be able to verify the data, so that there is no doubt the data is correct. The database has not been tampered with.

Another company, Company X, is interested in this analysis. They are a participant on the same Ethereum network, but are a light node, so only store the block headers. Company X does not trust Avanade.

To perform this analysis, Avanade uses our proof of concept. They simply fill out the Configuration File with the required information, and link the application to their Ethereum client and a local PostgreSQL database. They start the system, and issue the following query:

```
SELECT blocknum, _quantities FROM gecontract_transferenergy WHERE
_productionlocationid = 0 AND _consumptionlocationid = 0
```

This query returns a list of lists with two elements, namely block number and quantity transferred. The application has verified this data, and also returns the relevant proofs.

Avanade publishes this analysis on their website, accompanied by the relevant proofs. Company X reads the data, but, as it has a trustless relationship with Avanade, does not trust the data Avanade provided to be valid. Therefore, Company X verifies the data themselves by loading the data into the proof of concept application, and letting the application verify the data by checking it with Company X's Ethereum client. It turns out the data is valid.

5.2.2 Scenario 2: Tampered database

Avanade wants to perform the same analysis as in Scenario 1, but, unbeknownst to the company, a hacker broke into the machine they use for their analysis, and changed some data. The hacker made it look like smaller quantities of energy have been transfered between the production location with ID `0` and the consumption location with ID `0`.

Avanade issues the same query as in Scenario 1, but it returns an error message saying that it could not validate the data. Avanade now knows that the queried data is invalid, and can take steps to fix the problem. They delete their local database, and synchronize a new database with their blockchain.

5.2.3 Scenario 3: Giving out false information

Company X performs the same analysis as in Scenario 1 and 2. The application marks the resulting data as valid. Company X wants to publish this data, but they want to change some data, so that it looks like the consumption location with ID `0` was allocated smaller quantities of energy.

Company X publishes the invalid data on their website, accompanied by the valid proofs. Avanade reads the data, and decides to verify the data themselves using the proof of concept application. The proof of concept application returns an error message indicating that the provided data is invalid. Avanade now knows that the provided analysis is not correct, and can ask Company X for an explanation.

5.3 Performance evaluation

To evaluate the effect storing blockchain data in a traditional database has on the performance of queries, we compare the proof of concept application, which stores blockchain data in a traditional database, with a naive implementation, which does not store blockchain data in a traditional database, but reads the data directly off the blockchain each time a query is issued.

To perform this comparison we implemented a naive version of the application. This naive implementation is similar to our proof of concept application in that it steps through the blockchain block by block, decoding all found transactions and storage values. The difference between the implementations is that the naive implementation does not store the data in a traditional database. Instead, the naive implementation checks for each decoded value if it adheres to the conditions issued in the query. If it does, the value is temporarily stored in memory. After all blocks have been read, the temporarily stored values are returned to the user. This allows for a high level of data integrity, as the values are read straight off the blockchain, but a possible lower performance, as it has to read all blocks for each query issued. Furthermore, because this naive implementation is not linked to a traditional database, it does not support SQL queries, so queries have to be hardcoded into the application.

The number of transactions and blocks added to a blockchain per unit of time largely depend on the use case. The number of transactions and blocks currently stored on a blockchain depends on the number of transactions and blocks added per unit of time, and the amount of time the blockchain has been in use. Some use cases might require only a couple of transactions per day, while others might require multiple transactions per second.

To make our test results relevant to a broad range of use cases, we perform the experiment with different numbers of stored transactions and blocks, and different numbers of transactions and blocks stored since the last query.

We experiment on querying both transaction data as well as storage data, as both types of data are handled differently by the implementations.

We deploy the aforementioned Green Energy Contracts on a Quorum network in a Vagrant[1] environment. This setup gives us a private permissioned blockchain we have full control over, which allows us to easily create blockchains suitable for our experiments. We use a number of transactions to create Production Locations and Consumption Locations. The other transactions deposit energy to Production Locations, and transfer energy from Production Locations to Consumption Locations. Both implementations run on a Dell Latitude 7390 with an Intel Core i5-8350u CPU with 16GB RAM running Windows 10, and communicate with a node in the Quorum network using websockets. Each block contains 1.43 transactions on average.

We measure the time between the query being issued and the result returned using the Python `datetime` module. Each individual test is performed three times, after which the average is computed.

We chose the queries used in the experiments based on the fact that we want the application to support complex queries, and based on the fact that we want the test results to relate to real world use. For testing the querying of transaction data, we use `select blocknum, _consumptionlocationid, _quantities from gecontract_transferenergy where _productionlocationid = 0`. The result of this query shows

[1]`https://www.vagrantup.com/`

us to which Consumption Locations, in what order, and with which quantities, energy was transfered from the Production Location with id `0`. For testing the querying of storage data, we use `select blocknum, energycount from geconloc where locationid = 0`. The result of this query shows us how much energy is stored at the Consumption Location with id `0` at each block.

5.3.1 Static blockchain

We measure the performance of querying Smart Contract data from a blockchain. We measure this both for an implementation which stores the data in a traditional database, and an implementation which reads the data straight off the blockchain. In this experiment, there are no transactions or blocks added to the blockchain between issuing the first and the second query.

As many use cases will require the same query to be issued more than once, especially after new transactions have been performed, or require multiple different queries to be issued, we perform the experiment for a first and a second time a query is issued. This shows us the impact indexing the database has on the query performance. All subsequent times a query is issued are disregarded, as the operations performed by both implementations will not change after the second time the query is issued.

The measurement of the second time a query is issued shows the query performance when the database is already indexed. This also shows us how the application would perform if it would actively synchronize the blockchain, as opposed to only synchronizing when the user issues a query.

5.3.2 Dynamic blockchain

In this experiment, we look at a more realistic scenario, in which transactions and blocks have been added to the blockchain between the last and the new query. In reality, the number of added transactions and blocks depends on the usage of the blockchain and the time between the two queries. In this experiment, we start with a number of indexed transactions, and compare the time it takes to complete a query after a number of transactions have been added to the blockchain. We measure the time between the moment the query is issued and the moment the results are returned for both implementations. We also calculate the speedup the proposed solution gives us, by dividing the query time of the naive solution with the query time of the proposed solution. This shows us how much times faster the proposed solution is than the naive solution.

This experiment shows us in which situations our current implementation is faster than the naive implementation.

Chapter 6

Discussion

In this chapter we describe and discuss the results from the experiments described in chapter 5.

6.1 Functional evaluation

The scenarios in 5.2 show that our current implementation satisfies a number of our set requirements.

The application allows for somewhat complex queries in SQL format over complete smart contract data. This includes current and historic states of contract instances, as well as all function calls which could have altered the state. Supported queries are `SELECT` queries, in which columns can be specified, and `SELECT` queries with an additional `WHERE` clause, to only retrieve information that adheres to specified conditions. Aggregate queries such as `SUM` do not yet work, as our application requires the retrieval of individual data elements to validate the data with the accompanying proofs. By supporting SQL queries and providing the data in human readable format, our application does not require extensive knowledge of blockchain to use, apart from setting up the configuration file.

Furthermore, our application shows to be compatible with existing smart contracts, such as those in the aforementioned use case, without requiring any modification to the used blockchain.

Also, our application maintains a level of data integrity, as the system can detect when data from the traditional database is not stored at the given location in the blockchain. Furthermore, our system allows for other parties to verify data resulting from a query.

6.2 Performance evaluation

6.2.1 Static blockchain

Figure 6.1 shows us the difference in time to complete a query for our proposed solution and a naive solution when querying transaction data on blockchains of different sizes. We ran the experiment a first and second time to measure the difference indexing the database has on the performance of our proposed solution. Figure 6.2 shows the results of a similar experiment in which storage data was queried.

The results of these experiments in Figure 6.1 and Figure 6.2 show us that the time it takes to complete the queries grows linearly as more transactions are added to the chain. Both when querying transactions and storage, the proposed solution is slower than the naive implementation when the data has not yet been indexed, but is faster when the data has been indexed. The difference in time to complete queries grows with the number of transactions added to the chain.

A probable explanation for the difference in query time between the solutions is that indexing the data and constructing the proofs is very slow. However, once the data is stored in the traditional database, retrieving the data and validating the data is much faster than reading the data straight from the blockchain, possibly because less RPC calls are needed.

Querying storage values seems to be slower than querying transactions for all tests. A possible explanation for this is the fact that reading the storage of one instance from the blokchain requires multiple RPC calls, and validating the storage values requires us to compare them one by one. This is different from transaction data, where reading from the blockchain requires only one RPC call, and to validate the data we only compare the hash value.

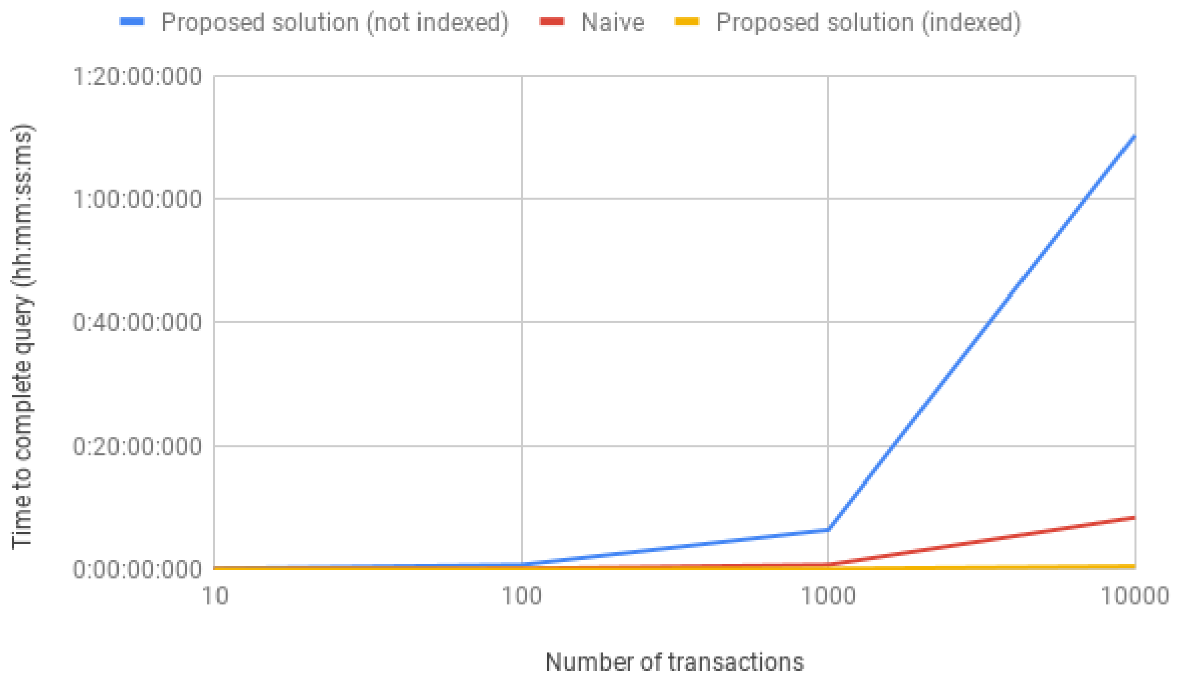


Figure 6.1: Comparing time to complete querying transactions between proposed solution and naive solution.

6.2.2 Dynamic blockchain

Figure 6.3 shows the difference in time to complete a query for our proposed solution and a naive solution when querying transaction data on a blockchain with an initial size of 100 transactions, to which a number of transactions have been added since the last time a query was issued. In other words, our proposed solution has not yet indexed the entire blockchain. It also shows the speedup our proposed solution gives us by dividing the time it takes the naive solution to complete a query by the time it takes our proposed solution to complete a query.

Figure 6.5 and Figure 6.7 shows the results of a similar experiment, but on blockchains with initial sizes of 1000 and 10000 transactions respectively.

Figure 6.4, Figure 6.6 and Figure 6.8 show the results of similar experiments, in which storage data is queried instead of transaction data. The initial sizes of the blockchains are 100, 1000 and 10000 respectively.

Figure 6.9 shows a comparison of the speedup values found on blockchains of different sizes and with a different number of transactions added since the last query.

The results of these experiments show that our current implementation returns the results of a query faster when the number of unindexed transactions is much lower than the complete number of transactions stored in the blockchain. The larger the blockchain, and the less transactions are added since the last query, the higher this speedup is. Once the number of new transactions becomes large enough, the naive implementation becomes more efficient. The naive implementation becomes more efficient when the number of newly added transactions is around 1/10th of the total number of transactions on the blockchain when querying transactions. When querying storage, this point seems to be around 1/5th.

A probable explanation for these results is the fact that querying data that has already been indexed by our proposed solution is faster than reading the data directly from the blockchain, but reading data from the blockchain while indexing this data and creating proofs is slower than only reading the data.

The results also show that the speedup when querying storage data is usually larger than the speedup when querying transaction data. A possible explanation is that querying storage data directly off the blockchain with the naive implementation is slower than querying transaction data this way, due to the multiple RPC calls that are required.

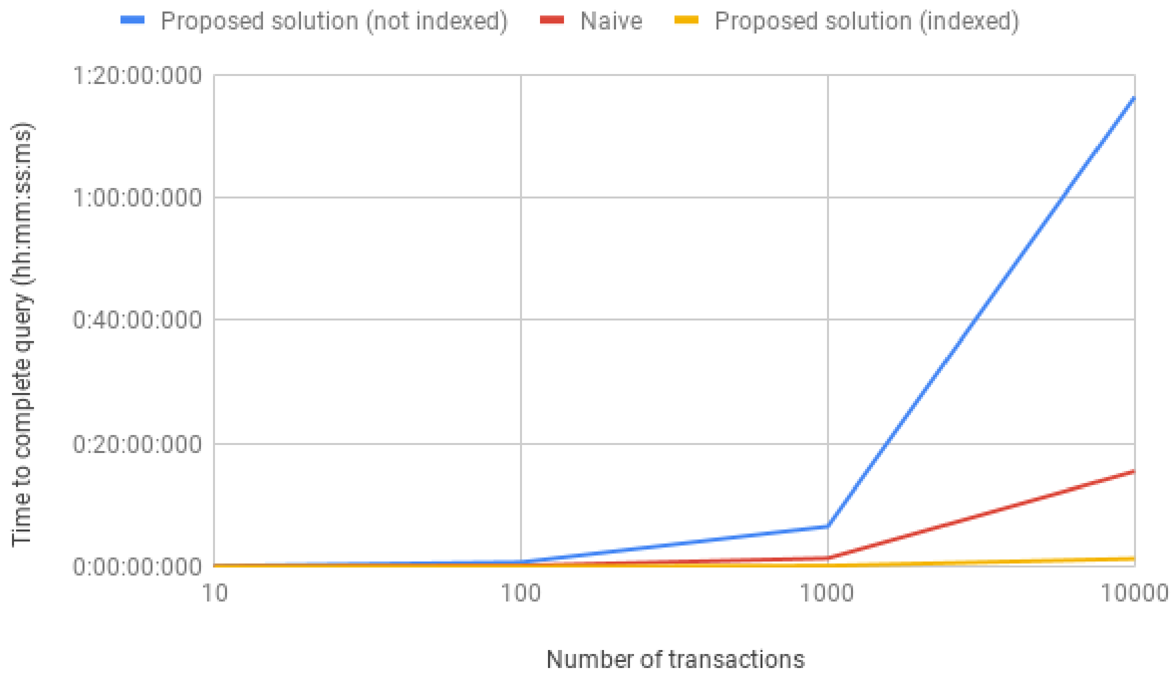


Figure 6.2: Comparing time to complete querying storage between proposed solution and naive solution.

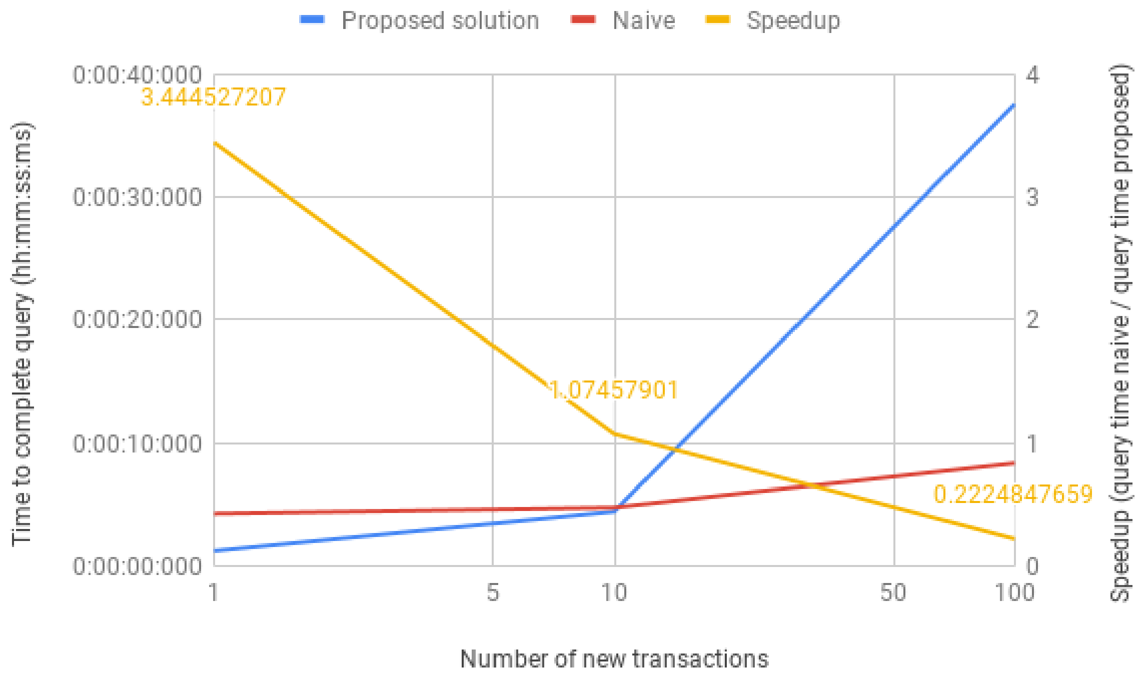


Figure 6.3: Comparing time to complete querying transactions between proposed solution and naive solution on a blockchain initially consisting of 100 transactions to which a number have been added since the last query.

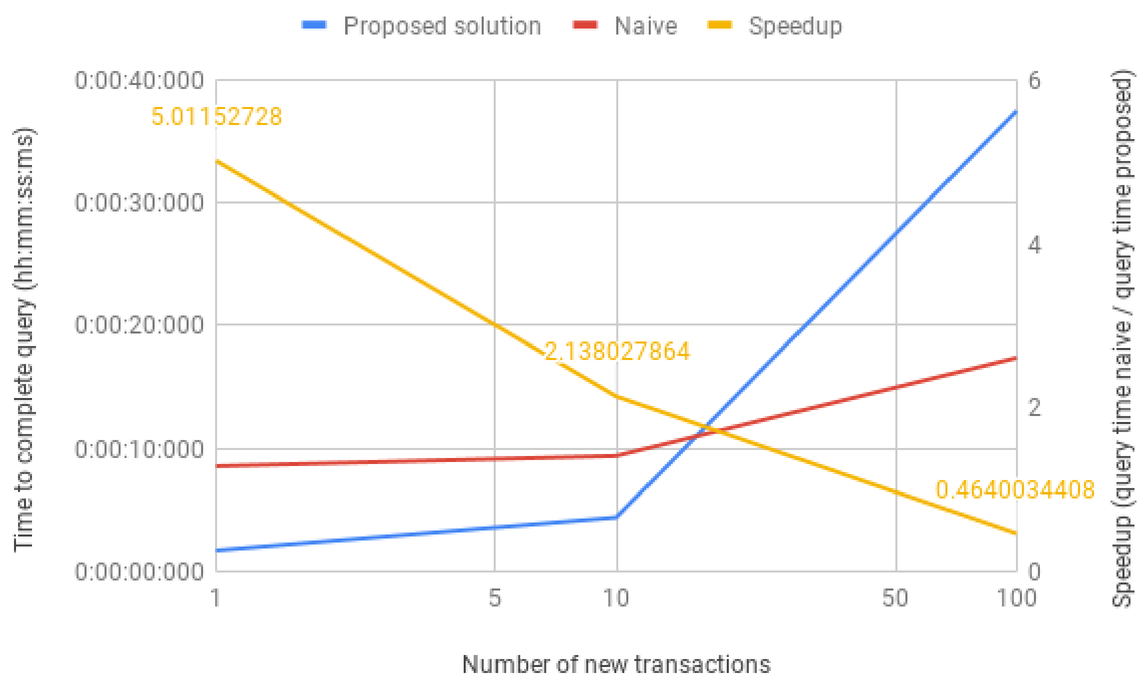


Figure 6.4: Comparing time to complete querying storage between proposed solution and naive solution on a blockchain initially consisting of 100 transactions to which a number have been added since the last query.

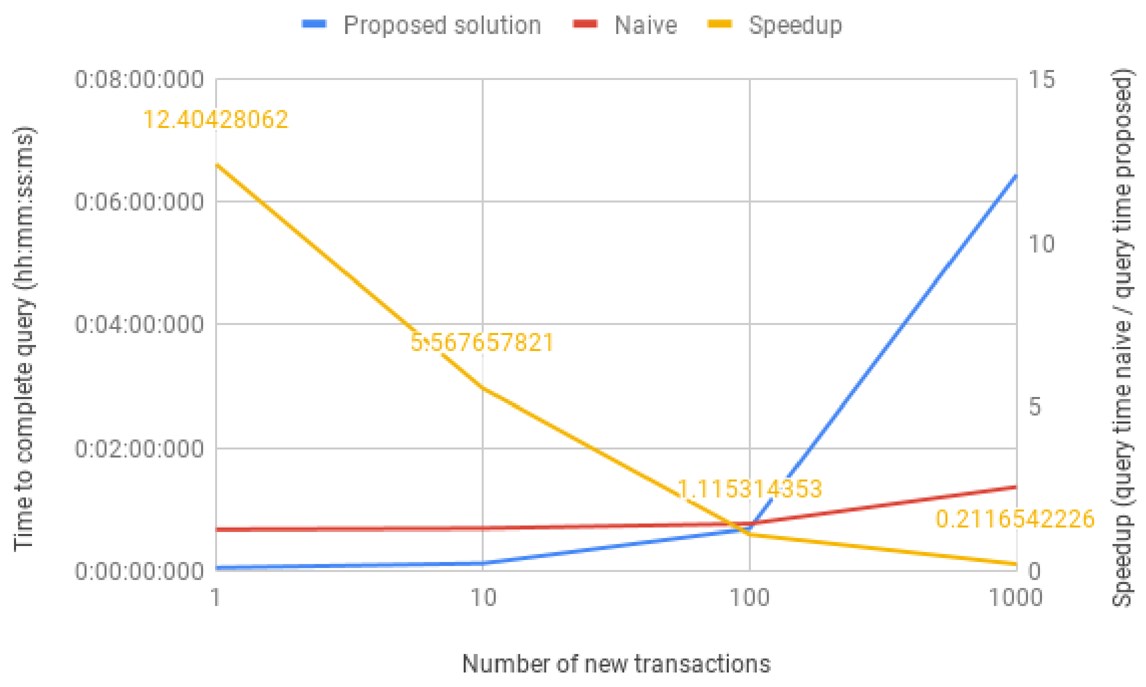


Figure 6.5: Comparing time to complete querying transactions between proposed solution and naive solution on a blockchain initially consisting of 1000 transactions to which a number have been added since the last query.

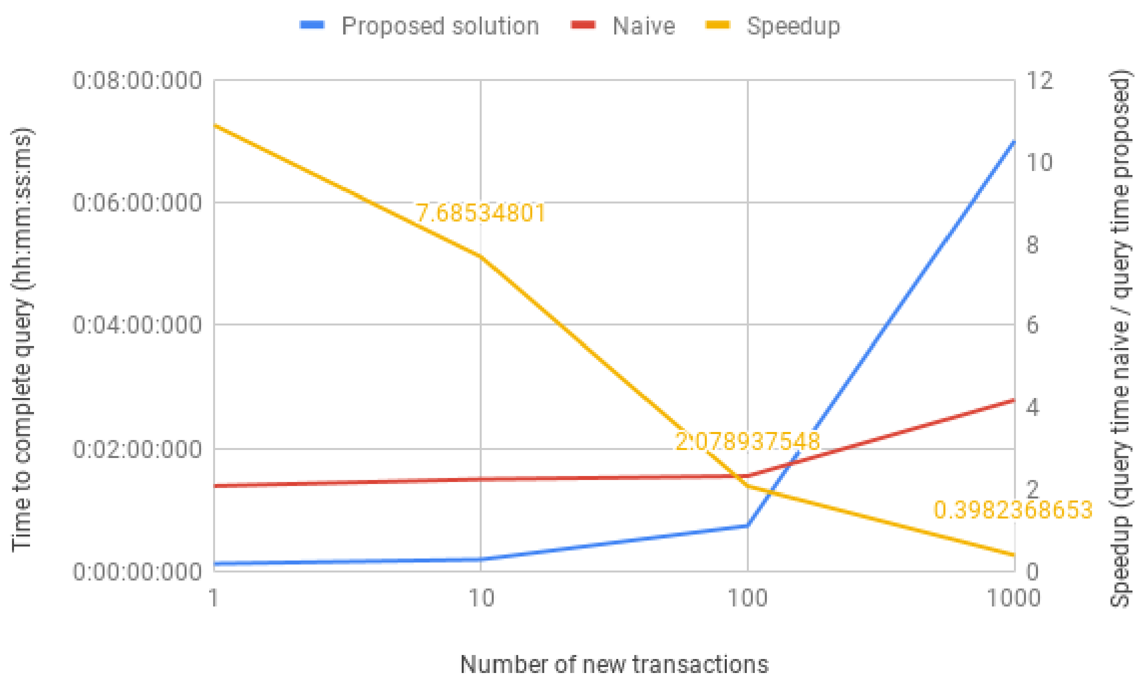


Figure 6.6: Comparing time to complete querying storage between proposed solution and naive solution on a blockchain initially consisting of 1000 transactions to which a number have been added since the last query.

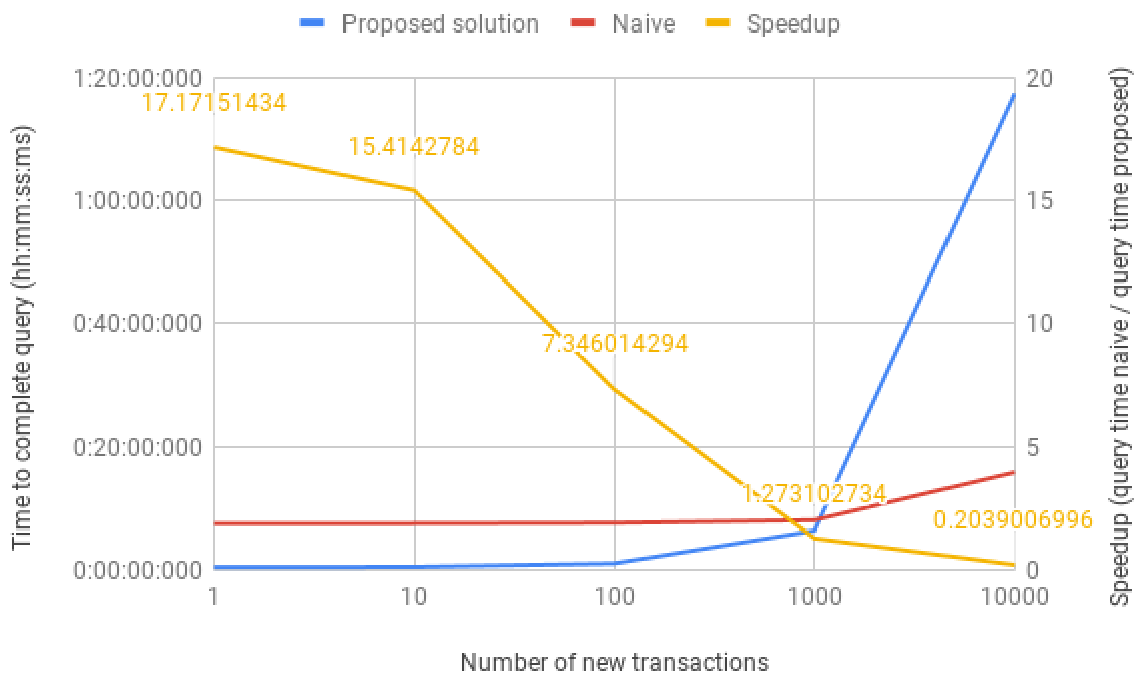


Figure 6.7: Comparing time to complete querying transactions between proposed solution and naive solution on a blockchain initially consisting of 10000 transactions to which a number have been added since the last query.

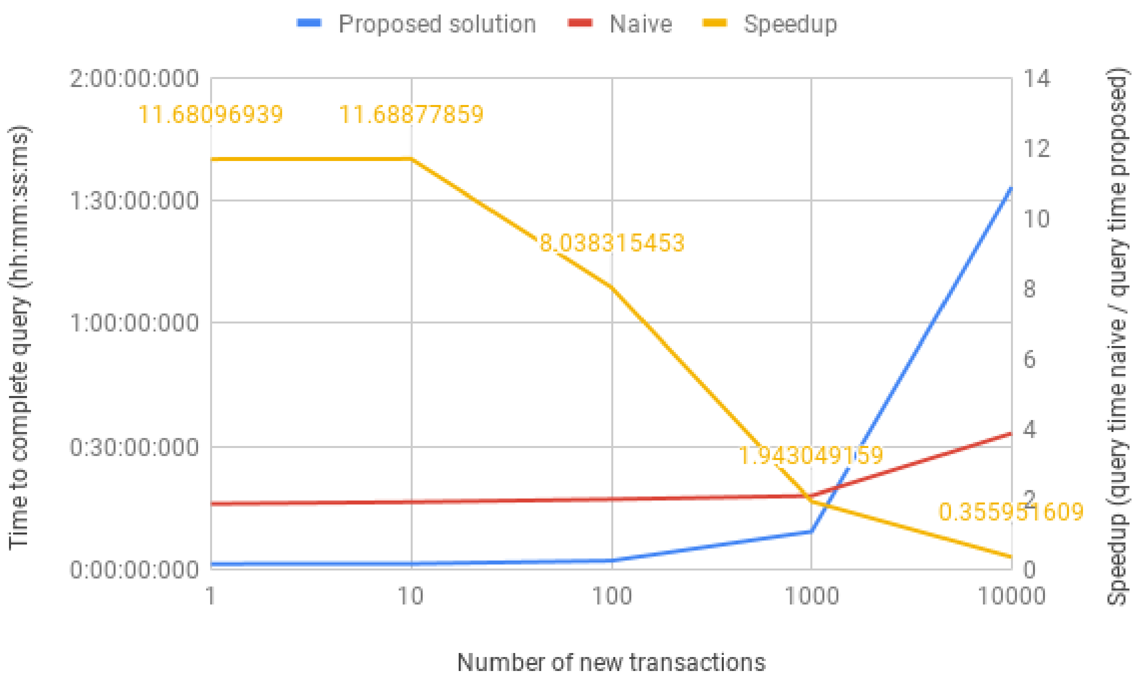


Figure 6.8: Comparing time to complete querying storage between proposed solution and naive solution on a blockchain initially consisting of 10000 transactions to which a number have been added since the last query.

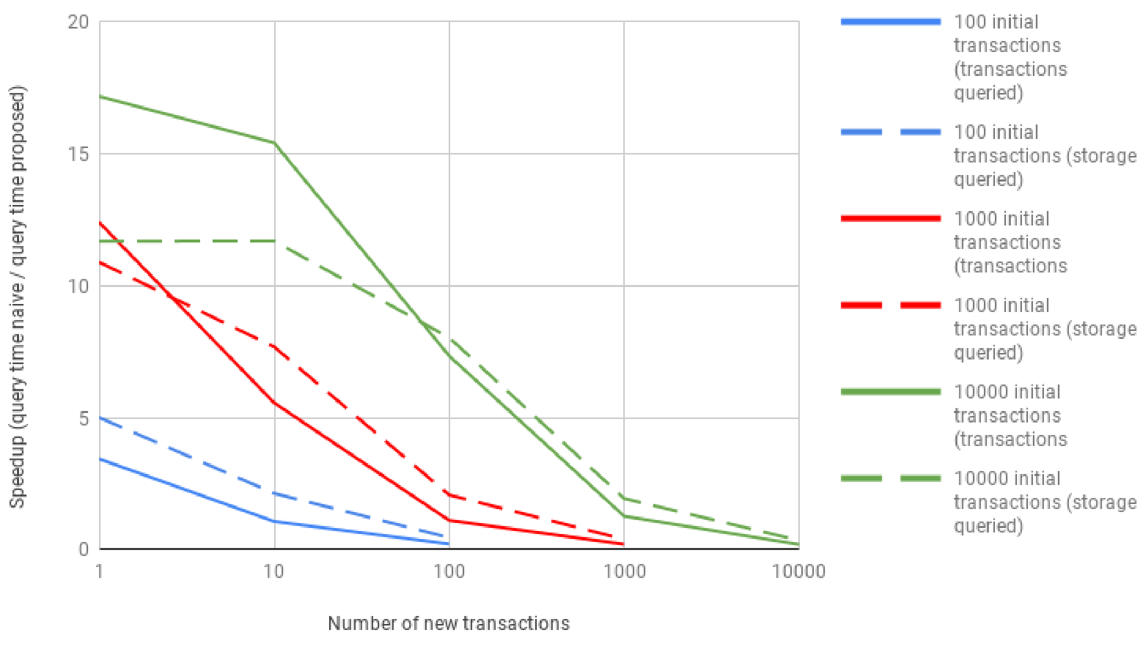


Figure 6.9: Comparing speedup proposed solution gives on blockchains of different sizes with different number of transactions added since last query.

Chapter 7

Conclusion

The goal of this study was to explore a solution to efficiently and passively query application data from a blockchain while maintaining data integrity. We did this by developing a proof of concept application based on existing knowledge, and evaluating this proof of concept.

By evaluating the proof of concept, we are able to answer the research questions.

RQ1 What data stored on the blockchain classifies as relevant application data?

By looking at the way Smart Contracts work and how they are stored on the blockchain in section 4.3.2, we were able to find out what data is relevant. We found that the transaction trie and storage trie found in each block contain the relevant information. The relevant information consists of all communication of participants on the network with smart contract instances, which are all function calls to contract instances. These are stored in the transaction trie. Furthermore, the relevant information consists of the state of each contract instance on each block, which are the values of its variables, which are stored in the storage trie.‘The combination of these two sources of information shows us the state of each contract instance on every block, and how this state was attained.

RQ2 How can we easily and efficiently query application data from a blockchain?

Chapter 6 shows that we were able to efficiently and passively query application data from a blockchain by using an off-chain, traditional database. By synchronizing this database with the blockchain, and storing the application data in a structure that suits queries, the end user is able to efficiently perform complex queries by using the build-in functionalities of the database.

However, we should note that our approach is not always more efficient than reading the data straight off the blockchain. Our experiments show that our implementation is more efficient when the number of added transactions since the last query is smaller than 1/10th of the total number of transactions initially stored in the blockchain. Thus, whether our implementation is more efficient depends on the use case. However, our experiments showed that our proposed solution is always faster when all data is indexed. Thus, if an implementation would continually synchronize the system with the blockchain, our proposed solution would be faster in any use case.

Because the end-user only communicates with the traditional database, and the data is stored in human readable format, stripped from most blockchain-specific information, there is no need for extensive knowledge of blockchain technology. Also, decoupling the query functionality from the blockchain allows the application to be compatible with existing blockchain implementations and deployed smart contracts.

RQ3 How can we assure data integrity of off-chain data?

Section 6 shows that we are able to assure a level of data integrity of off-chain data. By generating a Merkle proof for each data element while synchronizing the traditional database with the blockchain, of which the root corresponds to a root stored on the blockchain, we are able to verify the data stored in the traditional database. These Merkle proofs allow any party with trusted block headers to validate the data, which assures a level of data integrity.

However, we should note that we can only prove inclusion. This means that we can prove that a data element exists at a given position on the blockchain. However, we can not prove that a dataset is complete, in other words, contains all relevant data stored on the blockchain.

RQ4 How can we allow other parties to verify the resulting data?

Section 6 shows that by including the relevant Merkle proofs when publishing the data, other parties are able to easily verify the data by processing the Merkle proofs with trusted block headers. However, we should note that these parties are required to provide their own trusted block headers, so they should either be a full or light node in the same network, or they should be in a trusted relationship with a full or light node in the same network. Furthermore, we can only prove inclusion of data, and not that a dataset is complete.

We conclude that we are able to query smart contract data from a blockchain easily and efficiently while maintaining a level of data integrity by synchronizing the relevant data to a traditional database and create proofs for all data elements. This concept does not require any modification to the blockchain or smart contracts, and allows users to use SQL to query data. By providing these proofs, other parties are able to verify the resulting data.

7.1 Future work

Our proof of concept could be improved in a number of ways.

A logical improvement would be to support continuous synchronization, and explore its use cases and overall performance. Although our experiments show that querying the data while the application is fully synchronized is very fast, it would be interesting to research the impact on the system and different approaches to synchronize continually.

Furthermore, the current implementation has only been tested with one system of smart contracts. Although the application is designed to be compatible with existing smart contracts, it would be interesting to explore the compatibility of the concept and application with different systems of smart contracts.

It would also be interesting to extend the implementation to support other blockchain implementations. Other blockchain implementations use their own languages for smart contracts and their own ways of storing data, which might come with their own challenges. Ethereum is very popular at the moment for usage of smart contracts, but there are a number of other noteworthy implementations which also support this functionality, such as Hyperledger.

Another interesting topic to research could be not only proving the inclusion of data, but also proving the completeness of a dataset, while still providing efficiency. In other words, prove that the resulting dataset from a query contains all relevant data.

Also, it would be interesting to look at ways to support more kinds of queries. Due to the fact that we need all stored data of each transaction or storage value to validate the transaction or storage value, our implementation only supports a limited range of queries.

Other, smaller possible improvements are automatically repairing the database when it finds errors, and supporting more data types when validating storage data.

Bibliography

[1] M. Bartoletti, S. Lande, L. Pompianu, and A. Bracciali, "A general framework for blockchain analytics", in *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, ACM, 2017, p. 7.

[2] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Ethereum query language", in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ACM, 2018, pp. 1–8.

[3] ——, "Smartinspect: Solidity smart contract inspector", in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, IEEE, 2018, pp. 9–18.

[4] E. Buchman, *Understanding the ethereum trie*, Jun. 2014. [Online]. Available: `https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie/`.

[5] V. Buterin, *Merkling in ethereum*, Nov. 2015. [Online]. Available: `https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/`.

[6] M. Crosby, P. Pattanayak, S. Verma, V. Kalyanaraman, *et al.*, "Blockchain technology: Beyond bitcoin", *Applied Innovation*, vol. 2, no. 6-10, p. 71, 2016.

[7] A. Deshpande, K. Stewart, L. Lepetit, and S. Gunashekar, "Distributed ledger technologies/blockchain: Challenges, opportunities and the prospects for standards", *Overview report The British Standards Institution (BSI)*, 2017.

[8] E. Dimitrova, *Writing upgradable contracts in solidity*, Jun. 2016. [Online]. Available: `https://blog.colony.io/writing-upgradeable-contracts-in-solidity-6743f0eecc88/`.

[9] S. Ducasse, H. Rocha, S. Bragagnolo, M. Denker, and C. Francomme, *Smartanvil: Open-source tool suite for smart contract analysis*, 2019.

[10] A. Ekblaw, A. Azaria, J. D. Halamka, and A. Lippman, "A case study for blockchain in healthcare:"medrec" prototype for electronic health records and medical research data", in *Proceedings of IEEE open & big data conference*, vol. 13, 2016, p. 13.

[11] Ethereum, *Geth - go-ethereum wiki.* [Online]. Available: `https://github.com/ethereum/go-ethereum/wiki/geth`.

[12] ——, *Json-rpc - ethereum wiki.* [Online]. Available: `https://github.com/ethereum/wiki/wiki/JSON-RPC`.

[13] ——, *Light client protocol - ethereum wiki.* [Online]. Available: `https://github.com/ethereum/wiki/wiki/Light-client-protocol`.

[14] ——, *Patricia tree.* [Online]. Available: `https://github.com/ethereum/wiki/wiki/Patricia-Tree`.

[15] ——, *Rlp - ethereum wiki.* [Online]. Available: `https://github.com/ethereum/wiki/wiki/RLP`.

[16] *Eth.events's documentation.* [Online]. Available: `https://docs.eth.events/en/latest/`.

[17] *Graph docs.* [Online]. Available: `https://thegraph.com/docs/quick-start`.

[18] Graphprotocol, *Graph protocol - mechanism design.* [Online]. Available: `https://github.com/graphprotocol/research/tree/master/specs/graph-protocol-hybrid-network/mechanism-design`.

[19] S. Helmer, M. Roggia, N. El Ioini, and C. Pahl, "Ethernitydb–integrating database functionality into a blockchain", in *European Conference on Advances in Databases and Information Systems*, Springer, 2018, pp. 37–44.

[20] M. Iansiti and K. R. Lakhani, “The truth about blockchain”, *Harvard Business Review*, vol. 95, no. 1, pp. 118–127, 2017.

[21] A. Jain, A. Jain, N. Chauhan, V. Singh, and N. Thakur, “Seguro digital storage of documents using blockchain”, 2018.

[22] *J.p. morgan and microsoft announce strategic partnership to drive enterprise adoption of quorum*, May 2019. [Online]. Available: `https://news.microsoft.com/2019/05/02/j-p-morgan-and-microsoft-announce-strategic-partnership-to-drive-enterprise-adoption-of-quorum/`.

[23] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan, “Blocksci: Design and applications of a blockchain analysis platform”, *arXiv preprint arXiv:1709.02489*, 2017.

[24] E. Kozliner, *Merkle tree introduction*, Sep. 2017. [Online]. Available: `https://hackernoon.com/merkle-tree-introduction-4c44250e2da7`.

[25] Y. Li, K. Zheng, Y. Yan, Q. Liu, and X. Zhou, “Etherql: A query layer for blockchain system”, in *International Conference on Database Systems for Advanced Applications*, Springer, 2017, pp. 556–567.

[26] Y.-J. Lin *et al.*, “Efficient blockchain query”, PhD thesis, National Central University, 2018.

[27] R. C. Merkle, “A digital signature based on a conventional encryption function”, in *Conference on the theory and application of cryptographic techniques*, Springer, 1987, pp. 369–378.

[28] D. R. Morrison, “Patricia—practical algorithm to retrieve information coded in alphanumeric”, *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[29] S. Nakamoto *et al.*, “Bitcoin: A peer-to-peer electronic cash system”, 2008.

[30] PatAltimore, *Azure blockchain workbench architecture.* [Online]. Available: `https://docs.microsoft.com/en-us/azure/blockchain/workbench/architecture`.

[31] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, “Blockchain-oriented software engineering: Challenges and new directions”, in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2017, pp. 169–171.

[32] F. A. Pratama and K. Mutijarsa, “Query support for data processing and analysis on ethereum blockchain”, in *2018 International Symposium on Electronics and Smart Devices (ISESD)*, IEEE, 2018, pp. 1–5.

[33] Quorum, *Enterprise ethereum client.* [Online]. Available: `https://docs.goquorum.com/en/latest/`.

[34] *Solidity.* [Online]. Available: `https://solidity.readthedocs.io/en/v0.5.3/`.

[35] T. Swanson, “Consensus-as-a-service: A brief report on the emergence of permissioned, distributed ledger systems”, *Report, available online, Apr*, 2015.

[36] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger. ethereum project yellow paper (2014)*, 2014.

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