Scalable System for Indexing and Providing Access to Verifiable Blockchain Transaction Data

Scalable System for Indexing and Providing Access to Verifiable Blockchain Transaction Data

Lucas Huré-MacLaurin

A Thesis in the Field of Mathematics and Computation

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

July 2020

Abstract

Technological advancements in blockchain solutions have allowed for the development of applications that were not previously viable. Many of those applications will require querying potentially vast amounts of transaction data. With a potential transaction throughput orders of magnitude higher than was possible until recently, such applications would benefit from delegating those queries to third party services with the computing power and infrastructures allowing the quick return of accurate results.

The problem, however, with delegating queries, is that it creates a need for trust in a third party that blockchain technology was explicitly designed to eliminate. Such a third party query service should then be able to provide proof of the authenticity of the results it returns to its users.

In this project, we build an API that provides fast and scalable random access to transaction data on the Algorand blockchain and allows queries according to a number of transaction parameters. A mathematical proof based on the principle of Zero-Knowledge Proof allows the user of the API to verify the authenticity and completeness of the queried transaction data. This work is based on the vChain framework described in the paper "vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases" by Cheng Xu, Ce Zhang, and Jianliang Xu.

Acknowledgments

I want to thank my thesis director, Eric Gieseke, for his unwavering support throughout the process of researching, implementing, and documenting this project. His guidance was truly paramount to my being able to bring my thesis from conception to completion. I am very grateful for the time and energy he took to help me make this project a reality.

I would also like to thank my research advisor and program director Dr. Sylvain Jaume for his valuable support throughout the thesis process.

Finally, I want to thank my parents for showing me through their actions the love and beauty human beings are capable of. They have always set a standard of strength, integrity, and generosity that I admire and will aspire to for the rest of my life.

Contents

List of Figures

Chapter 1: Introduction

1.1 Blockchain Technology

A blockchain (Yaga et al., 2018) is a digital ledger that keeps track of all transactions and the time at which they occur between all the accounts on the blockchain (the distinction between a party or individual and an account is important since a single party could own several accounts on the same blockchain). As transactions are submitted to the blockchain, they are consolidated into blocks of fixed size that are then committed to the network based on a consensus protocol designed to ensure that only correct transaction information is stored. As each new block is committed, it is linked to the last block that was previously committed, in the sense that it contains a hash (Rogaway et al., 2004) of all the information contained in that previous block. This hash is a form of cryptographic fingerprinting mechanism which guarantees the integrity of the blockchain.

Digital records of transactions have existed for a long time in the form of different kinds of databases. What makes blockchain revolutionary is that it addresses a fundamental problem that has existed since the inception of bookkeeping of any kind, the problem of trust.

Whether it's in the case of financial transactions and banks, or medical records with hospitals and insurance companies, the problem people have always faced when it comes to keeping track of information is that of having to trust a single authority not to temper with the information stored in the database through malice of their own or that of an outside attacker.

Blockchain addresses the problem of trust using two main mechanisms. First, it achieves decentralization through being a distributed network. A typical database is centralized, meaning that information is stored by a single authority that grants access to it to other parties based on certain protocols. Even if that single authority keeps up-to-date copies of the database, that only solves the problem of potentially losing the information and of certain cases of tampering with it. In a distributed network, information is kept by several parties, or nodes, on the network. In the case of financial transactions, this means that all the parties (or those willing) participating in those transactions would be keeping that information, not just the specific bank or institution where someone happens to be a customer. This principle already exists in the case of peer-to-peer networks, in which each participating computer essentially acts as both a client and a server.

The other mechanism is immutability. Any information committed to the blockchain is permanent and cannot be subsequently altered in any way. Thanks to the hashing principle mentioned earlier, no block - and therefore no transaction either - can be changed in the blockchain. Any change in the information contained in any of the blocks will change that block's cryptographic fingerprint, which is stored in the next block.

Those two principles applied together all but guarantees that the information stored on the blockchain cannot be altered in any way. This immutability principle allows for traceability auditing. For example, products could be traced through the supply chain, and information recorded along the way couldn't be altered in any way.

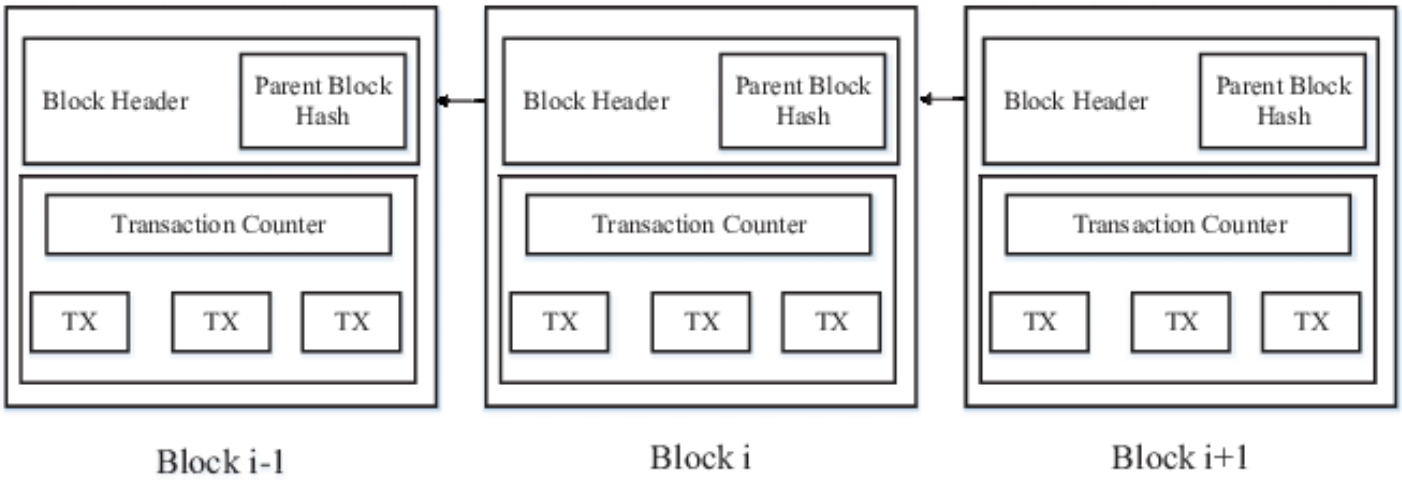


Figure 1: Basic Blockchain Structure (Zheng et al., 2017)

The figure above illustrates the basic structure of a blockchain and how blocks are cryptographically linked to one another through a hashing mechanism whereby each block stores the hash of the data contained in the previous block in the chain.

In most blockchain implementations, a Block Header contains a summary of the data contained in the body of the block, namely the hash of the previous block and the root of the Merkle tree (Beck, 2018), which encodes an aggregation of all the transactions contained in the block. The structure of the Merkle tree is introduced in section 1.4 and explained in detail in section 2.5.

1.2 The Algorand Blockchain

For this project, we have chosen to use the Algorand blockchain (Micali, 2016). blockchain implementations have so far suffered from one or more of a set of issues constituting what is known as the trilemma of blockchain; the idea that out of the three fundamental properties a blockchain

should have - security, scalability, and true decentralization - a blockchain will only have two of them. Several blockchain implementations have been criticized for claiming all the benefits of blockchain technology in the creation of private solutions that are effectively centralized networks, defeating one of the most fundamental aspects and advantages of the technology, the guarantee of immutability provided by true decentralization.

The Algorand network, a new implementation of blockchain technology, resolves the trilemma mentioned above and provides true decentralization despite achieving scalability of short and predictable block creation times (Chen et al., 2019). Algorand's consensus protocol is pure proof-of-stake, meaning that the probability for a participant in the network to be chosen to be part of the voting committee is proportional to the participant's stake in the network.

The Algorand network provides an interesting case for transaction indexing due to its support of an unusually high number of potential transactions per second. While Bitcoin (Nakamoto, S. 2009) and Ethereum (Buterin, V. 2014), currently the two largest blockchain platforms, allow for only around 7 and 15 transactions per second, respectively, the Algorand network allows for around 1,000 transactions per second. This provides an interesting challenge when it comes to the scaling ability of our Verifiable Query Service (VQS from now on). It also makes the network a viable option for many kinds of applications that other blockchains with much slower transaction throughput would not be able to support, making our VQS much more beneficial and impactful. Another reason we chose to use Algorand is that unlike other major blockchain implementations, Algorand never forks. Forking happens when several blocks are proposed at the same position in the blockchain, and it is not immediately clear which one is correct. When that happens, two chains are temporarily built-in parallel until the fork is eventually resolved.

The problem with this is that while the fork persists, it is impossible for the network's users to be certain that their transactions will be part of the final version of the blockchain when the fork eventually resolves, as it could choose to resolve using a maliciously built sub-chain.
Algorand's voting protocol guarantees with an overwhelming probability that the blockchain will never fork. As far as our project is concerned, this makes it possible to update our database in real-time as we listen for the transactions coming from the network.

Another key aspect of the Algorand blockchain that separates it from most other blockchain implementations is its distinction between archival and non-archival nodes.
Typically in a blockchain, a “full” node will store the entire ledger data, including all of the transactions in each block. Algorand’s archival nodes serve the same purpose and store the entire ledger data. However, most blockchain implementations also have what is typically called “light” nodes, which store only the headers of the blocks for the entire blockchain.

The headers contain a summary of the data contained in the block. For example, instead of storing all the transactions in a block, a block header stores only the root of the Merkle tree built from all the block transactions. The data present in the header still allow for important fundamental operations while hosting a “light” node is not prohibitive from a computing or storage standpoint. Algorand’s equivalent, a non-archival node, stores all the data contained in the last 1,000 blocks instead. Given the nature of the vChain framework, which relies on the assumption that the Query User hosts a light node and therefore has access to the block headers for the entire blockchain, Algorand’s design of light nodes proved to be a challenge that we addressed through the use of a Database of Transactions and describe in detail in section 2.3

1.3 Zero-Knowledge Proofs

A Zero-Knowledge Proof (Micali et al., 1989) is a process by which one party (the prover P) can show another party (the verifier V) that they have a certain piece of information without disclosing anything about that piece of information.

As an example, consider the following: Imagine that V has 2 diamonds in his possession sitting on a table. He knows one of them is real and the other fake, but doesn't know which one is which and cannot himself tell the difference. P, however, claims to be able to tell the difference between the two diamonds but refuses to reveal which one is real and which one is fake.

One way for V to verify P's claim would be to blindfold P, either switch the position of the diamonds or not, and then ask V whether the position of the diamonds has changed. If P truly can tell the difference between the diamonds, he should be able to say whether or not they have changed positions. Of course, even if he doesn't have that knowledge, P could decide to guess in an attempt to convince V he does have that knowledge.

The chance of guessing correctly would be 50%, which is not very strong evidence in favor of P's claim. However, if the experiment is repeated a second time, the probability that P would be able to guess correctly twice in a row falls to 25%. Each additional iteration of the experiment halves the probability that P is simply guessing correctly over and over. Mathematically, the probability can be expressed as $P(n) = 1/2^n$, where n is the number of iterations of the experiment. After only 10 iterations, the probability that P guessed correctly 10 times in a row is less than 1 in 1000. Since the probability converges to 0, the experiment can be repeated until a desired degree of certainty is achieved, without P ever disclosing which diamond is which.

In the case of the VQS, two things need to be accomplished in order to guarantee the validity of query results. One is to return the set of all transactions requested by the querier, but the validity of those transactions can be easily verified by the querier himself. The other thing, however, is to provide the querier with proof that the set of returned transactions is complete; in other words, that no transactions are missing from the results. The VQS needs to be able to prove to the querier that the set of transactions returned is complete, without having to show the querier the process the VQS goes through in order to retrieve all of those transactions. This is where Zero-Knowledge Proofs come in; we want to be able to demonstrate that we are in possession of certain information (the fact that the set of returned transactions is complete) without having to reveal anything about the information (walking the querier through the entire query fulfillment process).

The vChain framework, of which the VQS is an adaptation to the Algorand blockchain, allows for precisely this kind of proof to be returned to the query user.

1.4 vChain

vChain (Xu et al., 2019) is a framework that leverages the power of Zero-Knowledge proofs in order to provide users interested in querying transaction data on a blockchain network a cryptographic guarantee of the authenticity of the results returned.

With enough time and computing resources, a query user could theoretically scour the entire blockchain from the genesis block to the current last block, looking at every transaction within each block for transactions that fulfill their query parameters. This, however, given the rate of growth of a modern blockchain implementation such as Algorand's, is extremely impractical

and threatens the viability of any application relying on such a method of querying data. From the perspective of a single user potentially using a mobile device to run queries on transaction data, the memory cost of storing the entire ledger is prohibitive.

One possible solution is to delegate the work of querying transaction data to a powerful archival node. However, the archival node, while certainly orders of magnitude more powerful than a query user's mobile device, will still have to scour the entire blockchain for every single query, no matter how similar in its specified parameters to a previous query. For example, if two different Query Users specify the exact same query parameters just a few minutes apart, the archival node will scour the entire blockchain, looking at every single transaction and running their transaction data against the same exact query parameters twice in a row.

The key point here is that a lot of time and computing power is wasted going over every single transaction in every single block on the network in order to return query results. As in the case of dynamic programming, whereby previously computed data is stored to be re-used during subsequent computations, we can use a powerful database engine to store all the transaction data the VQS will need to run queries.

Given a query specifying a "to" or/and "from" address(es), as well as a time or block interval on the blockchain, a transaction within that time or block interval is either a match or a non-match, there is no other possibility and therefore those two sets together constitute the entirety or the results returned to the Query User.

If a transaction is a match, it is returned to the user and the verification process is very straight-forward; the user can recompute the root of the Merkle tree to which the transaction belongs, and confirm whether that root is identical to the one in their possession as a non-archival

node. If the reconstructed root is identical to the root in the block header hosted by the Query User, then the transaction is valid.

All we have done at this point though is to prove that the transactions returned are valid query results. This result is incomplete because it doesn't prove that the non-matching transactions are indeed non-matching. In other words, while we can easily prove that all returned transactions are valid, how do we know that certain valid transactions weren't omitted from the results? In order to guarantee not only the validity of the returned transactions but also the completeness of the query results as a whole, we need to prove that each transaction that wasn't returned to the Query User was indeed not a match for the query parameters.

In order to construct those proofs of non-membership, the vChain framework uses Authenticated Data Structures (ADS) that can be added to block headers and returned to the Query User, who can then verify the validity of the proofs and therefore the completeness of the query results. The ADS is computed for each transaction by the VQS and stored in the Database of transactions as "AttDigest". The implementation details of the ADS will be given in section 2.4.

1.5 The Importance of Demonstrably Correct Transaction Indexing and Querying

As mentioned earlier, a full node storing the entire blockchain could theoretically fulfill queries guaranteed to be accurate by simply scouring the entire blockchain, block by block and transaction by transaction, looking for transactions that fulfill its query parameters. However, without a structured storage mechanism, that full node would need to repeat the process of scouring the entire blockchain for every single query. This is without even considering the massive amount of memory required to store the entire blockchain. Algorand is already close to 10 million blocks

and is growing very quickly due to its high transaction throughput. Therefore, the true value of the VQS relative to manual queries is in its scalability and speed in returning results.

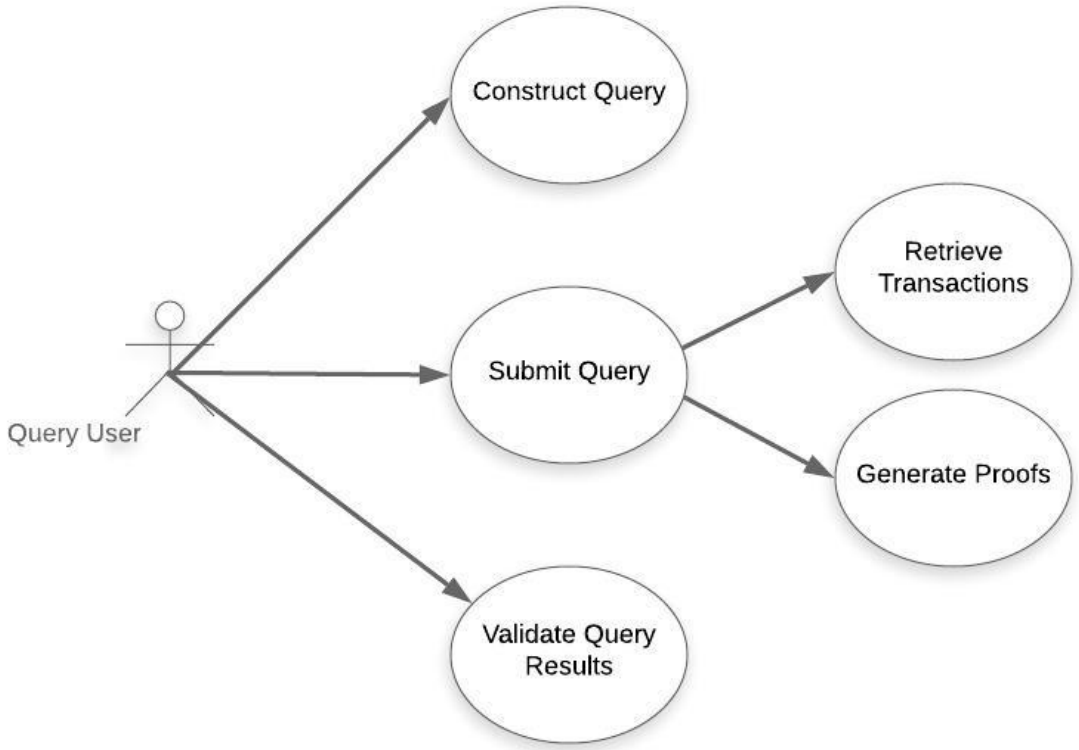
There are some use cases for which the integrity of query results would be paramount, for example, accounting and reporting for publicly traded companies, which are required to provide an accurate representation of their books at the end of the year. A company running blockchain transactions could use verifiable query results to prove that their accounting is accurate. The IRS could use a VQS to audit company records and prevent tax evasion. Machine learning algorithms could be used to detect fraud and money laundering on the blockchain, using the VQS to query specific data sets of transactions. Similarly, the VQS could be used for data analytics and the visualization of trends in transactions.

The note field of transactions could for example be used to contain information about products traveling along a supply chain. The VQS could return all the transactions containing information about products damaged at a specific point along the supply chain, providing an unprecedented degree of transparency to product transportation processes. In general, the public nature of blockchains such as Algorand has the potential to impose a new standard of transparency, and therefore accountability, in many industries, and the VQS could prove to be a key part of that process by providing a way to quickly retrieve transaction data guaranteed to be authentic.

There are however some challenges to providing verifiable query services. Algorand will grow at a rate of orders of magnitude greater than the current other major blockchains, which provides many challenges to the development of a transaction indexing system. Assuming a maximally functioning network at 1000 transactions per second, that comes out to about 30 billion transactions per year, reaching close to 1 trillion transactions in 30 years.

1.6 Requirements

The following use case diagram describes the high-level use cases of the VQS:



`

Diagram1: Verifiable Query Service Use Case Diagram

Query User: A user of the VQS interested in acquiring transaction data on the Algorand blockchain. The user is also interested in being able to verify the authenticity of the results, both the validity of the returned transactions and the completeness of the query results, i.e the guarantee that no transaction is missing from the set of returned transactions.

Construct query: The Query User constructs a query based on a set of parameters including time or block intervals as well as the addresses of the accounts involved in the transactions within that interval. Those parameters are turned using SQL format into a structured query that can be used as input into the VQS.

Submit query: The user-constructed query is imputed into the VQS API which will then process the query, generate the required proofs, and return the query results to the user.

Retrieve Transactions: The VQS API communicates with the Database of Transactions in order to retrieve all the transactions within the block or time interval specified by the Query User. After this initial filtering of all the transactions in the database, the remaining transactions are classified into matching and non-matching transactions. Matching transactions are appended onto the list of transactions returned to the user, along with the elements necessary to validate their authenticity, namely the root, branch, and navigation directions of the Merkle tree the transaction belongs to. Non-matching transactions are sent to the Proof Constructor.

Generate Proofs: For each non-matching transaction within the time or block interval specified by the Query User, the Proof Constructor computes the proof of non-membership necessary to guarantee the completeness of query results. Each proof is appended to a list of proofs returned to the Query User, along with the other cryptographic elements required for proof verification.

Validate query results: The authenticity of the query results returned to the Query User is established by two separate validation processes described in more detail later. One is the verification of the authenticity of the transactions returned to the user; the other is proof that no transaction is missing from the set of returned transactions. In other words validity and completeness of results.

1.6.1 Non-functional Requirements

Since a full node, or archival node in Algorand terms, could theoretically scour the entire blockchain and gather its own verified transaction data, the VQS' existence is motivated by the potential for very fast query results that don't require a systematic scouring of the entire blockchain for each query, as well as storage requirements orders of magnitude smaller than those for hosting a full, or archival, node.

Fast query results could allow for the existence of applications that wouldn't otherwise be viable due to slow query processing, while lower storage requirements will allow users not in possession of expensive hardware configurations, or even mobile device users, to query the blockchain for transaction information. Query results also need to be verifiable beyond doubt, at the level of mathematical proof, the way they would be if they were personally retrieved by the user traveling the blockchain from the genesis block to the current last block.

Finally, even with the authenticity of returned transactions being established, the user still needs to be certain that no transaction matching the query parameters was omitted from the results. As such, The VQS needs to be able to prove not only the authenticity of the returned transactions but also the completeness of the results. Given the potential high transaction throughput on the

Algorand blockchain, the database storing the transactions needs to be able to not only store a lot of transactions but also filter them efficiently based on different query parameters specified by the Query User.

The VQS' viability also depends on its ability to be up to date on the transaction data contained in the blockchain. This means that computationally expensive operations such as the creation of the ADS need to be processed fast enough to keep up with the growth rate of the blockchain.

1.6.2 Functional Requirements

The VQS needs to communicate with the Algorand blockchain in order to retrieve transaction data at a rate sufficient to keep up with the rate of growth of the network.
The VQS needs to store the transaction data in a database. It will then compute all the cryptographic elements that will be returned to the user in order to validate the authenticity of the query results and store those elements in the database as well.

The VQS will need to compute the Merkle tree of all transactions within a block, as well as the branch of the tree required for each specific transaction to compute the root of the Merkle tree. In order to support the functionality we wish our VQS to have, we need to be able to process queries from the user. Structured Query Language (SQL) notation is a widely used way to express queries and the one the Query User will use to define queries and pass them to the VQS.

Once the VQS receives the user query as a SQL string, it needs to be able to communicate with the Database of Transactions and initially filter transaction data based on the time or block interval specified by the Query User. The VQS then needs to be able to determine for each

transaction whether that transaction is a match or not based on the query parameters specified by the user. If a transaction is a match, the VQS will return the transaction data to the user, as well as the cryptographic elements necessary to confirm that the transaction is a valid query result.

If the transaction is not a match, the VQS needs to generate a proof of non-membership using the cryptographic elements stored in the database with each transaction, and return the proof to the user along with the other cryptographic elements required for the user to verify the proof of non-membership and establish the completeness of the query results.

Finally, the Query User needs to be able to verify the validity of the query results returned by the VQS. This validation process requires two distinct, separate processes. In order to verify the validity of the returned matching transaction data, the user needs to compute the root of the corresponding Merkle tree using the cryptographic elements returned by the VQS, and then compare the reconstructed root to their own, locally stored root.

The user also needs to be able to establish the validity of the proofs of non-membership returned by the VQS in order to ultimately establish the completeness of the query results returned by the VQS.

Chapter 2: Design

This section describes the architecture and design of the VQS. It starts with a high-level overview of the different components of the VQS and then delves into the implementation details of each component.

2.1 Architecture

The VQS employs a modular architecture, in which most of the activity that takes place in each component is contained within that component. The following component diagram describes the components of the VQS:

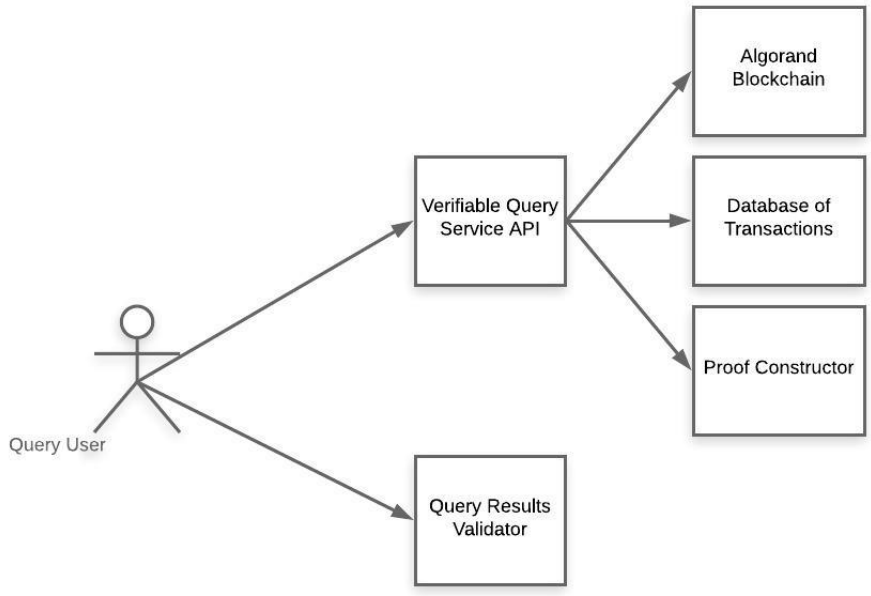


Diagram 2: Component Diagram

The Query User uses the VQS to process a query.

The Algorand blockchain contains the transaction information queried by the Query user. That information is continuously collected by the VQS and stored into the Database.

The Database stores all the transaction information relevant to the Query User, including the proofs and Merkle tree branches necessary to validate the authenticity of the Query Results.

The Proof Constructor computes the cryptographic proof necessary to establish the completeness of Query Results, in the case where a transaction is not a Query match.

The VQS API processes the query from the Query User and returns Query Results to the user in the form of transactions or proofs or non-membership, depending on whether or not a particular transaction is a Query match.

The Query Results Validator has two functions. The first is the reconstruction of the Merkle root using the Merkle tree branch returned by the VQS to the user. The second is the verification of the proof of non-membership returned by the VQS. Combined, those two functions establish the authenticity of each returned transaction as well as the completeness of the Query results.

2.1.1 Design and Technology Choices

We decided to use the Python programming language to implement our VQS for several reasons. Python natively supports integers of arbitrary size and operations between them with no loss of precision. This means that one can compute operations between extremely large numbers without having to use special libraries or having to worry about integer overflow. Compared to languages such as C/C++ and Java, which require the programmer to specify the type of integer to be used, Python abstracts that need, which makes dealing with large numbers much easier. Given that the numbers used to encode information into the AttDigest can reach upwards of 900 digits, this is a very useful feature of the language.

Python is the language used by the cryptographic accumulator library we ended up using, as it was the cleanest and most well-documented we could find. It is also the native language for Databricks notebooks, and the language we had the most experience and familiarity with.
The cryptographic accumulator library we used was designed by Oded Leiba in Python 3.

We chose Databricks as our database engine because of its ability to scale to a potentially very large number of simultaneous queries, as well as its integration with Spark and cluster-computing capabilities, which would allow for the computation in parallel of expensive operations such as the creation of the Att Digests. Other database engines such as SQLite, used internally by Algorand, are intended more for append-only type operations and wouldn't scale as well.

2.2 The Verifiable Query Service API

The process: The VQS API acts as an interface between the Query User and the components of the VQS. It takes in user-defined queries and fetches transaction data from the Database of Transactions based on the query parameters. The fundamental principle of the querying process is that a transaction either is or isn't a match. All transactions in the time or block interval specified by the query need to be processed one of two ways based on that binary classification.

If a transaction is a match for all the query parameters, it is returned to the user along with its associated validation elements. Those elements will allow the user to reconstruct the Merkle root and, by comparing it to the corresponding Merkle root in their own database, establish the authenticity of the returned transaction.

If a transaction isn't a match, the VQS sends it to the Proof Generator, where a proof of non-membership is created and returned to the user with its associated validation elements. The user will then be able to use those elements to verify the proof and establish the completeness of the query results. The VQS API also continuously collects block and transaction information from the Algorand blockchain in order to keep the Database of Transactions up to date.

The following diagram is a class diagram that describes the modular structure of the code used to implement the VQS. The color-coding shows which elements of the structure are native to the Algorand network, which are part of the VQS, and which are for use by the Query User.

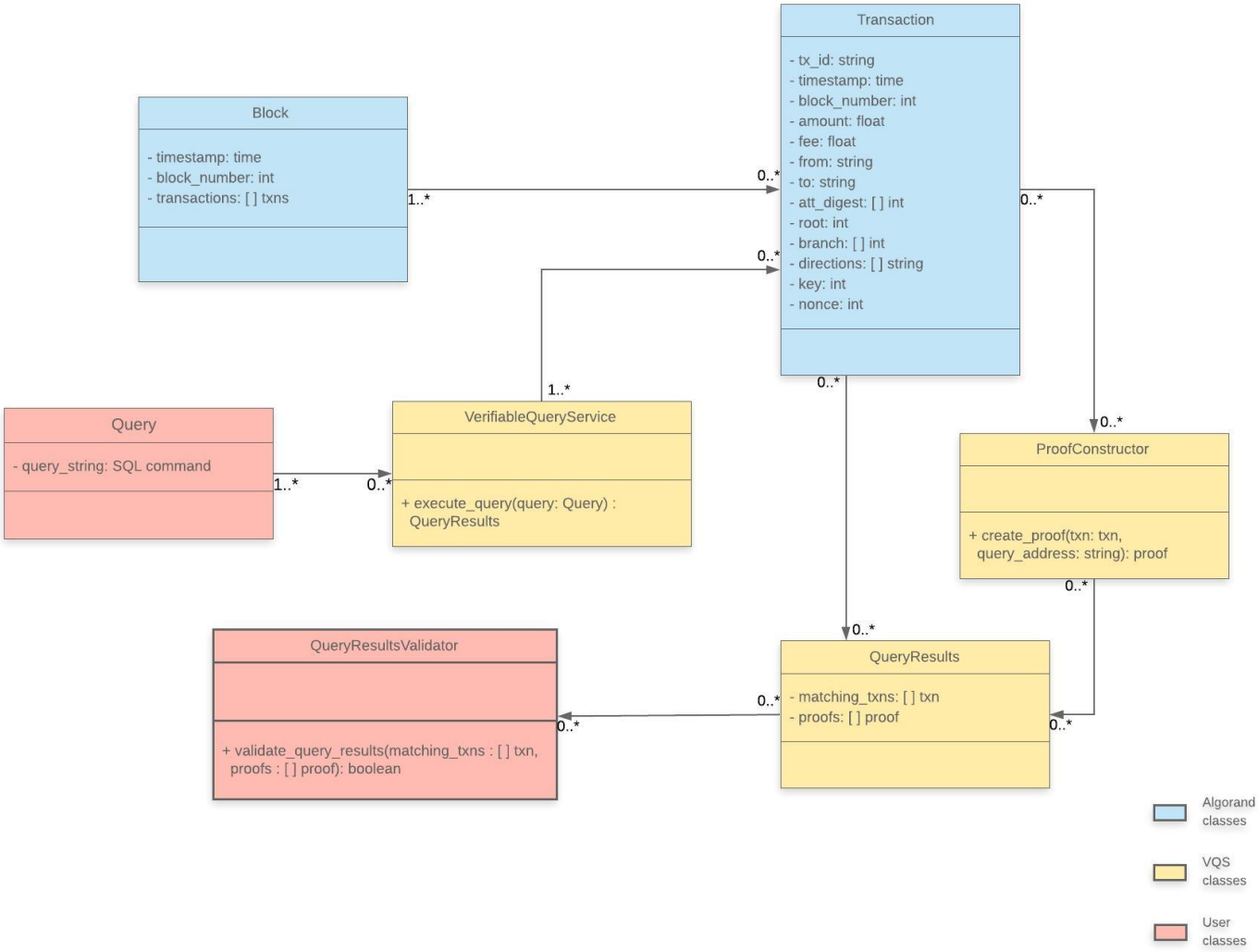


Diagram 3: Class diagram

2.2.1 Algorand Classes

Block:

-timestamp: the date and time at which this particular block was committed to the blockchain.

-block_number: a number that indicates the order of the block in the blockchain, relative to the first (genesis) block.

-transactions: a list of transaction objects, stored as Python dictionaries. Blocks make up the fundamental structure of the blockchain, linked together through mutual cryptographic encryption and containing all the information in the blockchain. In the Algorand blockchain, blocks contain additional information such as the block's hash, the previous block's hash, the current blockchain implementation's protocol, etc...however we will ignore those since they are not directly relevant to our application and use cases.

Transaction:

-tx_id: each transaction has a transaction ID associated with it. The transaction ID is generated by hashing all the contents (the attributes that follow) of the transaction.

-timestamp: the timestamp of the block this particular transaction belongs to.

-block_number: the number of the block this particular transaction belongs to.

-amount: the amount sent in that transaction, in micro Algos.

-fee: each transaction has a transaction fee associated with it, at the point of writing this a minimum of 1,000 microAlgos.

-from: the address of the account from which the transaction originates.

-to: the address of the account receiving the transaction.

-att_digest: the ADS associated with this particular transaction. Used to compute the proof of non-membership if this transaction is not a query match.

-root: the root of the Merkle tree made of the transactions belonging to the same block.

-branch: the Merkle tree path needed to recompute the tree root starting from this single transaction. Used by the Query User to compare the reconstructed root to the original root and validate the authenticity of the returned transactions matching the query.

-directions: the directions needed by the user to navigate the Merkle tree path and reconstruct the root. Returned along with the set of matching transactions in the query results.

-key: the public key created in the setup of the ADS' accumulator. Required to compute the proof and verification of non-membership.

-nonce: a discardable single-use value associated with the "from" and "to" addresses, used to compute the cryptographic proof of non-membership.

Each transaction is implemented as a Python dictionary, a data structure that maps an arbitrary number of keys to corresponding values. The dictionary can contain other dictionaries, as for example in the case of "att_digest", which is itself a dictionary containing the original as well as final accumulator values, under the keys "A0" and "A_Final".

Of attributes relevant to our application and use cases, Algorand transactions originally contain only the transaction ID, amount, fee, and "from" and "to" addresses. We add a number of attributes to each transaction in order to facilitate the storage of transaction data necessary to process queries and return query results, as well as to validate query results. For example, the Query User can specify a search interval using either a starting and ending block number, or a starting and ending date. As such, we want transactions to store the timestamp and block number of the block they belong to.

We also want each transaction to contain its own AttDigest, key, and nonce, which can be stored in the database of transactions and returned to the user for validation of proof of non-membership.

Finally, we also want each transaction to contain the root of the Merkle tree this particular transaction belongs to, as well as the tree branch and directions necessary to reconstruct the root for validation purposes.

The following is a single JSON formatted transaction, as stored in the Database of Transactions:


```
    {
        "AttDigest": {
            "A0": 2014013358899255650040173772809720505445420964148016678366627567002921800948919132501248950473107889454623445134561927971620776914954633163133787284674904430406942956302422644212217093037683598410649254249432773995973867443674032309614386489001025189894019192685900314330416415349717885619599530453964872406417320324419574570077742220610713593402715250621053991699657695459815967993440012049885741860538125828387222991095415903128440312027800733596287845938604803887141190849362278007903232568881718716645030030110550724240142141681927350311063275608688028347179608719955887873729587903859443753216247724715091078251334296637605024423478385035431553206265754134004450315734638159208914100659607281840103945909087013143786625340335896377986227804084378322601691637793530214147944687529035362391335295789394169237932060495196863853243219887159740282344429260821316336005988163239238624298369345140442262206296641272039339539451,
            "A_Final": 284702526217798178816028167238820239743792485508741343806750888791968640589857933059559191915132908058221874133270467338219573711657859928653081128367288381180029013594294775098496825178129144306057092002051707306153719154758470466098743448303017701774412507263305785927226565457026052908764593875685298665117591366167470377292492491878161773437437372297513932327604799656022705932776209329986557958912127477090454819600796914856800144300789561310793984553966716097287985310028362794052723463609568356878746913707733893527453728797189405109041682258928426098332243492106780796767615504568091213248344306196811659333817650750891270390448627995941162440417521763098297257794238809799596892574081879913210012688260824422964146300157415234784029046469762567690731107437075727691319127575107041069285487490298935578610627975507027849623053041796971247096071530550966686174808310257088159744439708485755775840466850091714091424 33",
        },
        "S": {
            "751328189529295043834728856513719853540663324682592764330373752751487826433 91": 4,
            "107188517083251303240076816101522796925047428497470034036430390210549952110701": 65
        },
        "amount": 396,
        "branch": [
            "0d71008ed71a17bce27e110b0a84884b06193be927678cbd9dd2f8dbebf01d0b",
            "ab67cd3fc70032c14d37fb3a0a82d66149a6682caaabf49351194cc713145f72",
            "b102b77fe152bb91361d005e08010d63bfd96875be29adb492d0b4e810828e9c",
            "dce7913562ae6ad27fe85c4527e2738b068244d1a1aa0ca7d6c2c8626b4fe8c4",
            "159c372729d151330d4e38ed4080618d5e37ad713e2416e4fa6e3c10a446570c"
        ],
        "directions": [
            "right",
            "left",
            "left",
            "right",
            "left"
        ],
        "fee": 1199,
        "from": "IE4C3BNWT4EYKPUZXGWDOOKBTJFVOYAZKBCWFYRC37U7BJKBIUH6NEB7SQ",
        "key": 323015182271693099056040749975201423820665805865752797379948978874131994963674811261930769773717355925833073197603599540301900386997888652469341601131309464292507129709068582656156206402986396744141909149234428678517750339195690148407725497260563442763030480460613531599292096635727222358677165037368777249504878226245103696099532661950630542014511834991117787084201988155028448403982541531882160991025278014612547553124630113774033188600964791697882867075710136351240842214321385724892964546377905344675681002198491339834329713406915495440120511519103413698833160375845422749288186832391754343419206373934773722098776861819143992286204547075778312481635812707723107439074113887285921224028068636260882556528803775408008682908754398936768858797465850263521019341126633049908901097119412711565128883298380316557631283788578633288040161721506089722226434283222907881502996209957745638398042224729224714771277546069218424962076 7,
        "root": "4d2c63530654bddf98ff016ccc934e25e7e872a016ac4554111ba32ca35a572c",
        "round": 2502854,
        "timestamp": 1571342817,
        "to": "KU7E2CPCNKQBK57FQQLGS4RCXSTJLL2WAMXS4DMYPINMT3HT3FOHACWNNA",
        "tx_id": "CQIT3Y7I26HTSKUNWC4TKSRGCTRIG7A4MCD5NWTCWQ653PJLD3EA"
    }
]
```


Figure 2: JSON Formatted Transaction

2.2.2 VQS Classes

VerifiableQueryService:

-execute_query(query: Query): This method takes as input a query in the format of a SQL command and returns the results of the query as two separate lists. It first filters all the transactions in the Database of Transactions by time or block interval.

The Query User can specify either a start and end block or a start and end date. If neither is specified, the query is assumed to apply to the entire blockchain. Similarly, if only one end of the interval is specified by the query, the other end is assumed to be unbounded. For example, for a query specifying only a starting block number of 2,596,035, the VQS API would assume a block interval starting from block 2,596,035 up to the last block currently committed to the blockchain.

The execute_query method then goes on to check each transaction in the set of filtered transactions for the address(es) specified by the query. The Query User can specify either a "from" or "to" address or both. As explained at the beginning of this chapter, once all the query parameters are taken into account, a transaction is either a match or not. If the transaction in the database, once filtered for all other query parameters, matches the address(es) specified by the query, it is a match and is appended to a list of query matching transactions. If the transaction isn't a match, execute_query(query: Query) calls the create_proof(txn: txn, query_address: string) function, which takes as input the transaction and the first address ("from" or/and "to" in the query parameters) to not match the transaction, and returns the proof

of non-membership necessary for the user to establish the completeness of the query results.

The first list returned by the execute_query method is a list of all the transactions matching the query parameters. Each transaction is a Python dictionary with the attributes shown above in JSON format. Additionally to the original transaction data found in the Algorand Transaction class

(fee, amount, ‘from’ and “to” addresses, tx_id, etc...), the transaction contains the information necessary to validate its authenticity, namely the root and branch of the Merkle tree to which the transaction belongs, as well as the instructions required to navigate the reconstruction process.

The second list returned by the execute_query function is a list of proofs of non-membership for each transaction that didn’t match the query parameters. The proof is also a Python dictionary (shown below in JSON format) that contains the other elements necessary for proof verification; namely, the transaction’s AttDigest, associated key, and the query address’ cryptographic nonce. The proof dictionary also stores the transaction ID and the first of the query addresses that didn’t match the transaction’s “from” or/and “to” address(es).

The following is a single JSON formatted proof dictionary, as returned to the Query User for non-matching transactions:


```
    {
        "AttDigest": {
            "A0": 105165831173777016116742898543999044818337723364260173623776915202249489129319818732536981309991414274456672385730670888576511075912599595111670086935268336676779619225
700821615209978769987072778091156374832451677967684375145399429174761183320450562992344282739327856434639808548021148407426448008300839156921133002620502615082302597735424982979693850793217446639421772910842468683520230979934077238490542735143435208428931647760042335290733468085353109933025231148558197409624224157951023710462541315482333634623405939044475358550852271635960929728909572857864336438777221384757793517188557024740221708091030715494526335794148338462649997515333988623536083314580484152617311765244671580666625454712356773304003242735040171022189376583297519292685367431417627875517760645848088636535554823913119846084778075131187407089404865152272761349479913668518503531770634436961035205185871279629167331033896636082556196964294779782680 6,
            "A_Final": 17455023802589289601668747037745255262030104673573517630930669521797709089351600144558155031185026783971321985666111185506821236088335313621307474070707241739534485689137351971848376144908337266193702575743889338505671416845644266164750493382668669513111400473500433230037086137912272379348698468632232931150441402206637567046082297246965694639327154318796931086847708724022146731144732925149471134257425952661951834921679937043544813556233575508394930104028835062100773328967973318124573696615540425164688361994005260980994660768422106010918535451192486291938245354480250474986616518489377449969067816792450986797245145521533663510466514479412949198536344027828081377367789316266966778716325884280694404895213293606038664972626941936799909309939825235015937879539571346929793284161870516901639783489115519563944103655956266952014267328033544748878892286412531692191173801525997844663501712927957660179786223426571558427495 9
            },
            "n": 18041249215477296390135231008522197223493097327132659128906015134652737182950172295696035123029804448444324460970982925834555950558631033712447162109083270295502211445924208819755072979222968441063151617593345674710592354131168809403381947557155388102282561583947942224322808836899103388093667839983836947596647260871567069305370080748907185149143584629537717269625762042364446423022580159353205524230968378432593456850956575847327258417657858798450195734513255225070784343136489101152688046041615831707491490067548768432191528084939266594343043084247398912270422028418450600805140701915762505000379770878488485784148055988423525483587398703711883496240203343183619520445168782883179358272325501671836910933109742147639636181129580079690958676180089822102837080071344191248606287279482459432402978199703399727949571323547807098086596359376340864145914638799238279338034797695345641054181436905413841466890312325626043201063,
            "proof": [
                139676575450278355407443794293085816734066561274063043795790900040760915823327903723508202284271465430463068397945353808072102873513565872441720755648507483450663660310231585185852997743315673204419718126718827761107309237097163570514524824261441669463481593093462058939142156861562436416338873144308336204606294713785470130547652152814149338834345005765461937338746909959599698880473395280545839956807642575903849756447258905403591001972916648511753947525559866120953929693772866717645390599287750123124682713165017895660446672643645940553901338632766181293438570953331660044426202380217583207682386684484136260282750643107552405915953241950154520876717149480202316193060419716938983166214726564394090221882884899300205165773800600458467958324723909820804546712208459891578607795481996226097265065629654183002559281782816798895216154357046810288025109399047254435167036277820576789863580861235006074954143713623521610370109108,
                596883905631632621710903248752050354 8
            ],
            "query_address": "KU7E2CPCNKQBK57FQQLGS4RCXSTJLL2WAMXS4DMYPINMT3HT3FOHACWNNA",
            "query_address_nonce": 4,
            "tx_id": "W22AWFOQU5JZBMY77NCBSUXNRUXZL7AIYPDIPB2HRNREUR4VRZAQ"
    },
```


Figure 3: JSON Formatted Proof Dictionary

ProofConstructor:

The create_proof(txn: txn, query_address: string) method takes as input a transaction that doesn't match the query parameters, as well as the first of the non-matching query addresses, and returns a proof of non-membership as a Python dictionary (shown above). The proof contains all the elements necessary for subsequent proof verification, namely the cryptographic proof of non-membership itself and its associated key, nonce, and AttDigest. Details about the inner-workings of the proof creation are given in section 3.4.

QueryResults:

Query results are returned by the execute_query method as two separate lists, one list of transactions matching the user query, and another list of proofs, one for each non-matching transaction. Those two lists are returned by the VQS API to the Query User for validation.

2.2.3 User Classes

Query:

The queryString attribute is a string in SQL command format defined by the Query User and passed to the VQS API's execute_query method. The string specifies the query parameters chosen by the user, namely the time or block interval over which they wish to retrieve transactions, as well as the "from" or/and "to" account addresses associated with those transactions.

Any combination of account addresses can be specified. If both "from" and "to" addresses are specified, only the transactions fulfilling both constraints will be returned. If only a "from" or "to" address is specified, all transactions fulfilling that constraint will be returned. In the case in

which no addresses are specified by the query parameters, all transactions over the time or block interval specified will be returned, regardless of provenance or destination accounts.

QueryResultsValidator:

The validate_query_results(matching_txns: [ ]txn, proofs: [ ]proof) method takes as input the two lists returned as query results by the VQS API; the list of transactions matching the query parameters and the list of proofs for every non-matching transaction.

For every matching transaction, validate_query_results uses the Merkle tree branch and navigation directions provided to reconstruct the Merkle root the transaction belongs to and validate its authenticity.

For every non-matching transaction, validate_query_restults uses the Att_Digest, proof, and associated elements to verify the validity of the proof of non-membership. If the validation process fails for any root reconstruction or proof verification, an error message is delivered to the Query User with the transaction ID associated with the failed validation. The details of the validation process are given in section 3.4

The following is a sequence diagram showing the flow of functionality between the different components of the VQS:

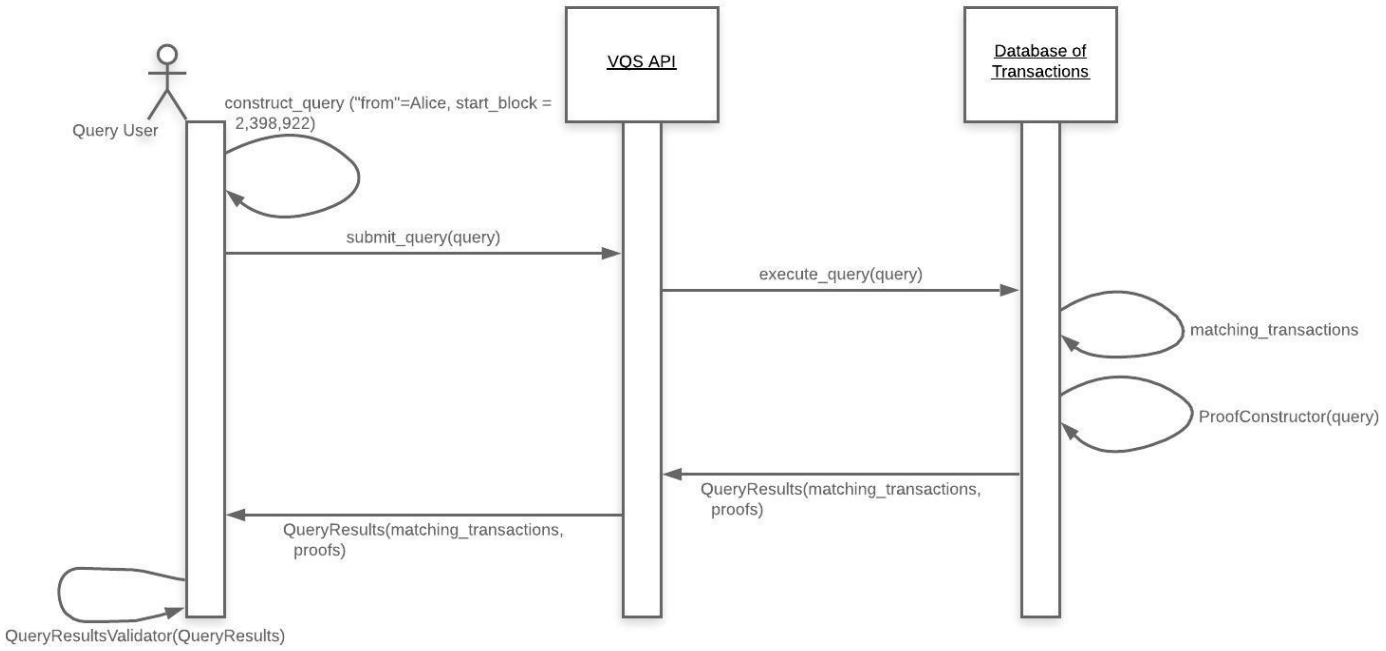


Diagram 4: Sequence Diagram

The Query User constructs a query, formatted as a SQL command string. The query specifies a time or block interval over which to look for matching transactions, and either a “from” or “to” address or both. The query is then submitted to the VQS API, which sends it through the execute_query method to the Query Engine to be processed. The Query Engine fetches the matching and non-matching transactions in the Database of Transactions and constructs the appropriate proofs for the non-matching transactions, before returning the query results to the VQS API, which in turn returns them to the Query User. The Query User then uses the validate_query_results method contained in the QueryResultsValidator class to authenticate the returned transactions matching the query parameters and to verify the returned proofs of non-membership for each non-matching transaction. Those two validation processes combined guarantee the authenticity as well as the completeness of the returned query results.

2.3 The Database of Transactions

	round	timestamp	AttDigest	S	amount	root	branch	directions	fee	key	from	to	tx_id
2	2502853	1571342813	{'A0': '495237446351920109(	{10718851708325130324	784	1e82bedb2b43f83e9(	['192cc523f9badb26ac	['right', 'right', 'right', '	1956	7212874470410	IE4C3BNWT4EYKF	MDLLZKI2EK4JO4	7AFXQAI47KAKSNDOO27
3	2502853	1571342813	{'A0': '3862904775428162172	{10718851708325130324	58	1e82bedb2b43f83e9(	['3aa2abe6ac71182e1(	['left', 'right', 'right', 'ri	1923	1201879950464	IE4C3BNWT4EYKF	TSJN53HFFMWRC	DY2XYUQJ7MEFEQKS7T6
4	2502853	1571342813	{'A0': '6155231772577229671	{10718851708325130324	77	1e82bedb2b43f83e9(	['ea61408c6d35da40fc	['right', 'left', 'right', 'ri	1850	1100860093387	IE4C3BNWT4EYKF	TDXQNTHX4VQ3N	W22AWFOQU5JZBMY77N
5	2502853	1571342813	{'A0': '7493199212596476910	{10718851708325130324	782	1e82bedb2b43f83e9(	['7c41288ccbf22d1a76	['left', 'left', 'right', 'rig	1847	1308151288845	IE4C3BNWT4EYKF	DH4VKOAIYHOM2	WRXX6J3O4EKJBLNK3J7(
6	2502853	1571342813	{'A0': '2607908157560376929	{10718851708325130324	988	1e82bedb2b43f83e9(	['37b81d6fbc300ae89(	['right', 'right', 'left', 'ri	1789	5462653991064	IE4C3BNWT4EYKF	RB3K2ZHU26AHG	5YLHYMVHBVK5KGMHTJ\
7	2502853	1571342813	{'A0': '2983335512759350105	{10718851708325130324	309	1e82bedb2b43f83e9(	['d498e9ebc3f20cab62	['left', 'right', 'left', 'rig	1716	3647282333664	IE4C3BNWT4EYKF	NSN2OLLIT3KDLK	YZPWPCPUSGJBJXRONN
8	2502853	1571342813	{'A0': '1374677050855425546	{10718851708325130324	849	1e82bedb2b43f83e9(	['a5fa50e323af59bbc8	['right', 'left', 'left', 'rig	1673	2245139382982	IE4C3BNWT4EYKF	KU7E2CPCNKQBH	TRAOZFT4GESSH62D6E7\
9	2502853	1571342813	{'A0': '5947428519923509271	{10718851708325130324	876	1e82bedb2b43f83e9(	['400940ae2c3c9e439:	['left', 'left', 'left', 'right	1636	1985021417467	IE4C3BNWT4EYKF	6NPT3PE4TEO6XH	Z3RCMJOYXZMHE6QTHL.
10	2502853	1571342813	{'A0': '8654836045497075286	{10718851708325130324	821	1e82bedb2b43f83e9(	['e1a465219a66691fce	['right', 'right', 'right', '	1607	5033997866780	IE4C3BNWT4EYKF	A2PZKQSYOQ24D	LZOIZBPX3YMPBKXCA3Y:
11	2502853	1571342813	{'A0': '4650993193977501503	{10718851708325130324	305	1e82bedb2b43f83e9(	['bebbe314a6ed9954d	['left', 'right', 'right', 'le	1587	7579557342749	IE4C3BNWT4EYKF	7GEIFHY46PKZ23	VIU4UCKQS7EHR4NZQVS
12	2502853	1571342813	{'A0': '7378715128898836755	{10718851708325130324	156	1e82bedb2b43f83e9(	['ed46390db4319085e	['right', 'left', 'right', 'le	1524	2314503894288	IE4C3BNWT4EYKF	6QGZIIB4GRXXYV	OZQ5IYWYZTUAQQR6HY)
13	2502853	1571342813	{'A0': '1340223068180182515	{10718851708325130324	332	1e82bedb2b43f83e9(	['5805f028db1dc6f6b2	['left', 'left', 'right', 'left	1437	1521615809393	IE4C3BNWT4EYKF	YOHNUI2L74EPAE	Q7HM3MBYV6QULV3CCI2
14	2502853	1571342813	{'A0': '112959671256971219-	{10718851708325130324	75	1e82bedb2b43f83e9(	['edd43fdb5375a3f6aa	['right', 'right', 'left', 'le	1410	1239361563695	IE4C3BNWT4EYKF	V2STB7J3KOY4C2	QBGVYY5ISYOQJGJQTGI(
15	2502853	1571342813	{'A0': '1958460163208904549	{10718851708325130324	252	1e82bedb2b43f83e9(	['13000b27ae8b534d0	['left', 'right', 'left', 'left	1374	2683245123305	IE4C3BNWT4EYKF	Y7VCJ5IEUMRFMI	C7ASEXBK5ZLUV4CI2WIA
16	2502853	1571342813	{'A0': '3137101804331553103	{10718851708325130324	484	1e82bedb2b43f83e9(	['43645e06a2bcb68b2	['right', 'left', 'left', 'left	1232	8272498293181	IE4C3BNWT4EYKF	RB3K2ZHU26AHG	YPDTVZ6QS7HV4P2SSRG
17	2502853	1571342813	{'A0': '4764435517083169162	{10718851708325130324	905	1e82bedb2b43f83e9(	['816a84c1e5481f6695	['left', 'left', 'left', 'left',	1113	1810746411980	IE4C3BNWT4EYKF	XAGS7BL6YBKU5	3ND546G2H5QQSOCJYW
18	2502853	1571342813	{'A0': '1313176980842772872	{10718851708325130324	923	1e82bedb2b43f83e9(	['8ff7d7b29fe812439d	['right', 'right', 'right', '	1099	2892208147209	IE4C3BNWT4EYKF	RI5AYDLQLWSHK	O2EVSYKX5OV3AANU4RC
19	2502853	1571342813	{'A0': '1333503532090819007	{10718851708325130324	526	1e82bedb2b43f83e9(	['0c8e065c0b0675286	['left', 'right', 'right', 'ri	1061	1827620615152	IE4C3BNWT4EYKF	WYPBSJQ7UUXU3	TSNCK2KXQYIQL34B7FPE
20	2502853	1571342813	{'A0': '232451135976067849!	{10718851708325130324	808	1e82bedb2b43f83e9(	['c7089ca2b1a0c1c17	['right', 'left', 'right', 'ri	1055	8755734738424	IE4C3BNWT4EYKF	D2NFPT76J7AXBS	KTTKWHLDDIBRD3RZ4EA
21	2502853	1571342813	{'A0': '1106512158205733229	{10718851708325130324	422	1e82bedb2b43f83e9(	['44c9f3d35094fb65f32	['left', 'left', 'right', 'rig	1035	1621031247132	IE4C3BNWT4EYKF	R6TZCRR25ZQPA4	IBJOWF4W6PPQD3R7O3C
22	2502853	1571342813	{'A0': '9744046444451395927	{10718851708325130324	985	1e82bedb2b43f83e9(	['75b4e6d206a17d255	['right', 'right', 'left', 'ri	1014	1388493027443	IE4C3BNWT4EYKF	A2PZKQSYOQ24D	A4KIYKSYCH3ENN6D2YYI

Diagram 5: Database of Transactions

The above diagram shows the contents of the Database of Transactions (columns resized to fit the page) as stored in Databricks. Each row represents a different transaction and each column a transaction feature. Due to the size of the numbers used to encode information into the AttDigest, they are converted to strings so that the database can store them accurately instead of having to replace them with a value of “infinity”. The same applies to the cryptographic key. Those strings are converted back to integers when transaction data is retrieved from the database by the VQS and returned to the Query User.

In this particular implementation of the VQS, we chose to store the Att Digests in our Database of Transactions to address the challenge of dealing with Algorand’s design of light nodes, or non-archival nodes. The vChain framework requires that the user have access to the Att Digest for each transaction in order to verify the proof of non-membership for transactions not matching the query parameters, thereby ensuring not only the authenticity of the query results but also their

completeness. However, since Algorand's code is open source, a new instance of the blockchain could be created to have non-archival nodes store block headers, or at least the cryptographic elements needed by the user to validate query results returned by the VQS.

The database also stores the root of the Merkle tree corresponding to the transaction in that row, the branch necessary to recompute the root for verification purposes, and the directions required to navigate the branch all the way up to the root. All those elements need to be returned to the user so that they can recompute the Merkle root and compare it to the one in the header that they store as a light node. Similarly to the challenge posed by the design of the non-archival nodes, we store the Merkle root in the Database of Transactions even though in an implementation of the Algorand blockchain better suited to the use of the VQS, the user would be in possession of the data by virtue of having access to the entire ledger's block headers as a non-archival node.

2.4 The Proof Constructor

This section describes how proofs of non-membership are constructed and used to establish the completeness of query results.

2.4.1 Authenticated Data Structures and Proofs

The Authenticated Data Structure, or AttDigest, is the most fundamental cryptographic element of the VQS. It is what allows the Query User to verify the proofs and guarantee the completeness of the query results returned by the VQS.

The challenge here is not to prove that an element is a member of a particular set, but instead to prove that it is not a member of that set. This will allow us to prove that a transaction is not part of the set of transactions that fulfill a certain set of constraints, in our case the set of parameters specified by a query.

Following is the list of cryptographic elements necessary to the construction and verification of the proofs of non-membership.

-Cryptographic Hash Function: A cryptographic hash function takes a string of any length as input and outputs a string of fixed length. Its most important properties are that it is practically non-invertible, i.e that it is infeasible to figure out the input string from a certain output string, and that it is collision-resistant, meaning that the probability of two unique input strings mapping to the same output string is overwhelmingly small.

-Bilinear Pairing: Let $\mathbb{G}$ and $\mathbb{H}$ be two cyclic multiplicative groups with the same prime order p. Let g be the generator of $\mathbb{G}$. A bilinear mapping is a function $e : \mathbb{G}X\mathbb{G} \rightarrow \mathbb{H}$ with the following properties:

1) Bilinearity: if $u, v \in \mathbb{G}$ and $e(u, v) \in \mathbb{H}$, then $e(u^a, v^b) = e(u, v)^{ab}$ for any u, v
2) Non-degeneracy: $e(g, g) \neq 1$

-q-Strong Diffie-Hellman (q-SDH) Assumption (Boneh et al., 2004):

Let $pub = (p, \mathbb{G}, \mathbb{H}, e, g)$ be a bilinear pairing as described above. It states that for all polynomials q and for all probabilistic polynomial-time adversaries Adv,

$Pr[s \leftarrow \mathbb{Z}_p; \sigma = (pub, g^s, , g^{sq}); (c, h) \leftarrow Adv(\sigma) : h = e(g, g)^{\frac{1}{c+s}}] \approx 0$

-q-Diffie-Hellman Exponent (q-DHE) Assumption (Camenisch et al., 2009):

Let $pub = (p, \mathbb{G}, g)$ as described above. It states that for all polynomials q and for all probabilistic polynomial-time adversaries Adv,

$Pr[s \leftarrow \mathbb{Z}_p; \sigma = (pub, g^s, , g^{sq-1}, g^{sq+1}, , g^{s2q-2}); h \leftarrow Adv(\sigma) : h = g^{sq}] \approx$
0

-Cryptographic Multiset Accumulator: A multiset is a set to which an element can belong multiple times. A cryptographic multiset accumulator is a function $acc(x)$ which maps a multiset to an element in some cyclic multiplicative group in a collision resistant fashion (Xu et al., 2017).

The cryptographic multiset accumulator has the following algorithms:

1) $KeyGen(1^\lambda) \rightarrow (sk, pk)$: On input a security parameter (1^λ), it generates a secret key sk and a public key pk.
2) $Setup(X, pk) \rightarrow acc(X)$: On input a multiset X and the public key pk, it computes the accumulative value $acc(X)$.
3) $ProveDisjoint(X_1, X_2, pk) \rightarrow \pi$: On input two multisets X_1, X_2 where $X_1 \cap X_2 = \emptyset$, and the public key pk, it outputs a proof π.
4) $VerifyDisjoint(acc(X_1), acc(X_2), \pi, pk) \rightarrow 0, 1$: On input the accumulative values $acc(X_1), acc(X_2)$, a proof \pi, and the public key pk, it outputs 1 if and only if $X_1 \cap X_2 = \emptyset$

The properties of the multiset that allow to prove set disjointedness are what we will use to prove that certain transactions do not belong to the set of transactions matching the query parameters.

Using the ProveDisjoint algorithm, the VQS will output a proof of non-membership for each transaction not matching the query parameters and return that proof to the Query User, who will then use the VerifyDisjoint algorithm to verify the validity of the proof returned by the VQS.

2.5 Validation of Query Results

The following diagram shows how the Merkle root is recomputed by the Query User in order to verify the validity of the returned matching transactions.

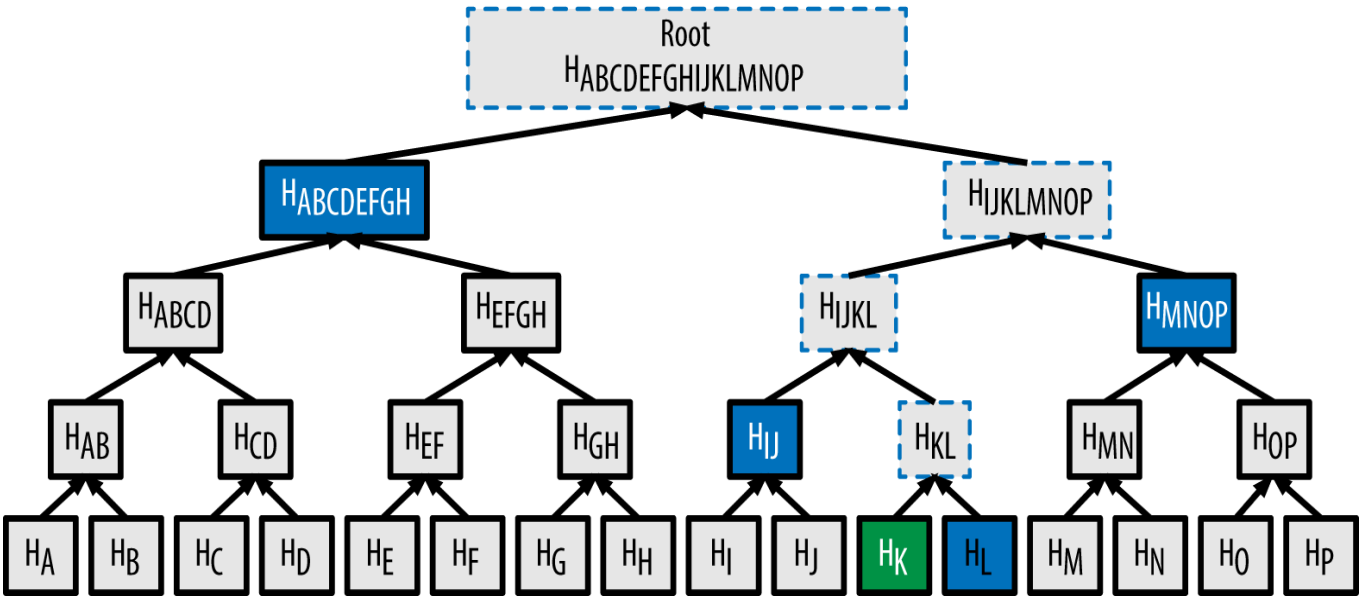


Diagram 6: Merkle Root Computation

The above diagram shows the structure of a Merkle tree and describes the process of using a single transaction as well as a specific path in the Merkle tree, its branch, in order to compute the root of the tree. This process is the one that allows the user to verify the authenticity of the transactions matching their query parameters returned by the VQS.

In the diagram, “H” stands for “Hash” and the subscripts A through P stand for transaction IDs.

A Merkle tree is a binary tree in which the parent node at any level of the tree is created by concatenating its two children nodes and hashing the resulting string. The process is then repeated

until there is only a single node left, the root of the tree. The principle behind this data structure is that each Hash, whether the initial hash of the transactions making up the tree or the root of the tree is of fixed size by the definition of a hashing function. The root of the tree can then be used as a cryptographic encoding of all of the transactions in a block. By the very nature of hashing functions, only a tree built using the same exact list of transactions will yield the same root. As such, the root of the Merkle tree can be used to verify whether a certain transaction belongs to the block associated with the root.

The idea is that since the size of the output of a hash function is always constant regardless of the size of its input, the root of the Merkle tree can be used as a record of all the transactions in a block and requires only a fraction of the space required to store a list of all the transactions. Specifically, in a block containing 10,000 transactions, the size of the root of the Merkle tree would be 1/10,000th of the size of the list of all transactions. Over billions, or even possibly trillions of transactions eventually, the storage space saved through this process is very significant. In most blockchain implementations, the root of the Merkle tree is stored in the block headers so that light nodes, or non-archival nodes in the case of Algorand, contain the transaction data necessary to verify that a transaction, in fact, belongs to a specific block.

Three elements are required in order to verify a transaction's membership to a certain block. The transaction itself, the root of the Merkle tree associated with the block the transaction belongs to, and the path required to recompute the root. The path, or branch, is a list of the transactions needed at each level in the tree in order to recompute that specific hash and move to a higher level in the tree. In order to recompute the root of a Merkle tree, we only need a single transaction per level in the tree.

The diagram above provides an example. We start with a transaction whose authenticity we wish to verify. “K” is the transaction ID, and “H_K” (in green in the diagram) is the hash of that transaction ID. Given “K” initially, we wish to recompute the root of the entire Merkle tree in the most efficient way possible, both in terms of storage and computation overhead.

Looking at the diagram, we can see that if we have “H_L”, the hash of the transaction ID of transaction L, we can apply the construction principle of the Merkle tree and compute the parent of “H_K” and “H_L”. Namely, we can concatenate “H_K” and “H_L” and hash the resulting string, which will output “H_KL”. At that level in the tree, we can see that in order to get to yet a higher level we need the sibling node of “H_KL”, “H_IJ”. Given “H_IJ”, we can concatenate “H_IJ” and “H_KL“ and hash the resulting string, which will output “H_IJKL” and take us to the above level in the tree. By repeating this process as many times as necessary we will eventually get to the root of the Merkle tree, in this example “H_ABCDEFGHIJKLMNOP”.

The key point to notice in this reconstruction process is that we only need a single hash per level in the tree in order to recompute the root, regardless of how many transactions are in the block. This is where the true value of Merkle trees lies. Since only a single hash per level of the tree is necessary to recompute the root starting with a certain transaction ID, the size of the path in the tree, or branch, is logarithmic to the number of transactions in the block. Specifically, for n transactions in the block, the size of the branch will be $log_2(n)$ (rounded down to the nearest integer).

In the example above for a block containing 16 transactions, we only need a list of 4 transactions to recompute the root of the Merkle tree. However to get a better sense of how efficient this process is, let us consider the case of a block of 10,000 transactions. Without the process previously described, we would need 5,000 hashing operations to get to the second level in the

tree, then 2,500 hashing operations to get to the level above that, etc...however using the Merkle branch associated with our starting transaction, we would only need $log_2(n) \approx 13$ operations.

In the algorithm we implemented in order to recompute the root of the Merkle tree given a starting transaction and its path throughout the tree, directions are also needed to navigate that path. The reason is that given the nature of hashing functions, the order of concatenation preceding the hashing of the resulting string is going to matter. In other words, H(H_K + H_L) is not equivalent to H(H_L + H_K), so at each level in the tree, we need to know the order in which to concatenate the two strings and get the correct output hash.

Our implementation of the Merkle tree algorithm keeps track not only of the path needed throughout to eventually recompute the root but also of the directions needed to navigate that path. Both of those data are stored in the database and returned to the user with every transaction matching their query parameters. The user can then use all of those elements to recompute the root of the Merkle tree the transaction belongs to. The resulting root is then compared to the root in the block header stored on the blockchain itself in the header of the block associated with that transaction. If the two roots are identical, the transaction is cryptographically guaranteed by the definition of hashing functions to belong to the block in question. Repeating this process allows the user to verify the authenticity of the transactions returned by the VQS.

2.6 Risk

The principal risk with this implementation of the VQS lies in the assumption that it will be able to scale to the rate of growth of the Algorand blockchain. At maximum usage, Algorand could reach a throughput of 10,000 transactions per second. While proof construction and

verification are fast and inexpensive operations, the computation of the Att Digest is significantly more expensive, taking approximately 2 seconds per transaction on a latest generation laptop.

This means that on any single computer, the VQS would not be able to keep up with Algorand's eventual transaction throughput, falling increasingly behind the network's growth over time and therefore becoming useless as a query system, which needs to be kept constantly up-to-date with the current state of the network in order to be a viable option as an application.

The computation of a particular Att Digest however is completely independent from that of another, meaning the task is highly parallelizable. Using the parallel computing abilities of Apache Spark clusters, we should theoretically be able to reach a rate of Att Digest computation sufficient to stay up-to-date with the current state of the Algorand network. This however hasn't been tested in this current version of the VQS.

2.7 Testing

We tested the functionality of the VQS using a JSON file containing 1000 blocks from the Algorand blockchain with an approximate average of 20 transactions per block. In order to compare the results returned by the VQS to the data contained in the blocks, we ran each test query both with the VQS and by scouring the blocks in the JSON file and sequentially going over each transaction within each block, running the transaction data against the parameters defined by the test query, determining whether the transaction was a match or not and accordingly either rebuilding the Merkle root and comparing it to the root stored in Database of Transactions, or constructing the proof of non-membership and verifying it using the functionality provided to the Query User.

Across a large sample of tests including every possible combination of “from” or/and “to” addresses, as well as starting or/and ending block or time intervals, our results running the queries “manually” by sequentially going over every transaction within each block one by one, were identical to the results returned to the Query User by the VQS.

Chapter 3: Results

In this project, we successfully implemented a transaction data query service for the Algorand blockchain whose query results are cryptographically guaranteed to be authentic.
A query user is able to submit a query in SQL format to retrieve transaction data from the Algorand blockchain. The VQS allows the query user to specify any combination of a starting or/and ending block or date. The user then specifies any combination of 'from' or/and 'to' addresses. This flexibility allows, for example, a user to retrieve all the transactions originating from a certain account A to another account B over a specific time interval, such as January 1st, 2019 to June 15th, 2020. It would also allow a user to retrieve all the transactions originating from the same account A to any other account on the network over a specific block interval.

Once given a query with those parameters, the VQS is able to retrieve transaction data from the Database of Transactions and return the results to the user. The VQS successfully returns a set of transactions matching the query parameters, as well as a set of proof of non-membership for each non-matching transaction.

The verification functionality provided to the Query User allows them to validate the results returned by the VQS. The first verification function successfully recomputes the root of the Merkle tree the matching transaction belongs to, compares it to the Merkle root in the user's possession, and confirms or denies their equivalence. The second verification function successfully uses the proof of non-membership returned by the VQS to confirm that the corresponding transaction doesn't belong to the set of matching transactions.

These two verification processes successfully guarantee to the user both the authenticity and completeness of the results returned by the VQS, therefore fulfilling the most important and fundamental requirements of the VQS.

Certain aspects of our initial requirements however were not met. Namely, we were not able to set up the VQS to gather a constant stream of transaction data from the Algorand network in order to stay up-to-date with the current state of the blockchain. This is because the time required to compute the Att Digests is such that the single computer we used to test this version of the VQS could not keep up with the transaction throughput of the Algorand network. This could be addressed in a future version of the VQS by using parallel computing as explained in chapter 4.

While more performance testing should be done on the VQS' ability to stay up-to-date with the transaction data on the blockchain, we strongly believe that its algorithmic foundation is sound and that given the capabilities of a cluster-computing framework such as Apache Spark's, the VQS would be able to stay up-to-date with the current state of the blockchain and therefore be shown to be viable as a stand-alone application or as support for third-party applications.

One of the greatest challenges we had to deal with was to deal with the absence of typical "light" nodes in Algorand's implementation of a blockchain. Algorand's non-archival nodes store all the block information for the last 1,000 blocks only. However, the Att Digests, as well as the other cryptographic data needed to guarantee the authenticity of the query results need to be stored on a light node hosted by the Query User. Typically this is done by having light nodes, non-archival nodes on the Algorand network, store the headers of the block.

The vChain framework assumes that the ADS are stored in the block headers, where they can be retrieved by the user and used to verify the proofs of non-membership. We proceeded with the knowledge that the open-source code of the Algorand blockchain could be modified such that

another instance of the network could use a different protocol whereby non-archival nodes store the block headers for the entire ledger instead of all the block data for the last 1,000 blocks. Our solution was to store the Att Digests in our Database of Transactions, where they are provided to the user for validation of query results.

Chapter 4: Future Work and Applications

There are two assumptions of the vChain framework that the Algorand blockchain doesn't fulfill and that we worked around for this project. The first is that the ADS are meant to be integrated within the block header during the creation and submission of a block to the blockchain since once committed a block cannot by definition be modified. This issue would have presented itself with any currently active implementation of a blockchain network. A potential workaround for Algorand would be to instantiate a copy of the blockchain that would recreate the contents of the original blockchain and to change the block creation protocol to include the computation and integration of the ADS to the block headers. Any future instantiation of the Algorand protocol could be implemented in such a way as to include the
ADS computation and integration process.

The second assumption is that an Algorand non-archival node, typically labeled a light node in most blockchain networks, contains the block headers from each block in the blockchain, but light nodes in the Algorand network store only the last 1000 blocks. However, the headers are crucial to the query results validation process since the Att Digests are needed to compute the proofs and proof verifications. This could also be implemented in a different instantiation of the Algorand blockchain with some changes to the source code, which is open source and freely modifiable.

Those are the reasons that we store in our Database of Transactions the Att Digests, which the Query User would have access to in the block headers. Despite that assumption being violated in this specific implementation of the Algorand blockchain, the many advantages it has relative to other blockchains made it a very good platform to build and test the VQS on top of; and as

mentioned, those assumptions could be fulfilled in future implementations of the network with some straightforward modifications to the source code.

For this version of the VQS, we implemented a naive strategy for intra and inter-block data aggregation. We apply the original process of checking a transaction for a match with the query parameters and repeat that process for each transaction within the time or block interval specified by the Query User. However, optimizations of query performance are possible through batch verification, and the vChain paper proposes two different intra and inter-block batch processes. One of those optimization methods could be used in the future to speed up the verification process and efficiency of the VQS.

Algorand transactions contain a note field that could be used for query purposes. The words in the note field could be viewed as set-valued attributes and added to the cryptographic accumulator, such that a proof of non-membership for transactions that wouldn't match the queried terms could be computed in the same exact way that it is in the VQS for transaction addresses.

Transactions matching the queried terms would be returned to the Query User in the same way, with the addition of the note field to the transaction dictionary, and with the combination of the set of matching transactions and the set of proofs of non-memberships for non-matching transactions, authenticity, and completeness of query results could be established in the same way they are for account addresses. Applications of term queries have vast potential across many industries. For example, a real estate company could run verifiable queries on a large catalog classified in different categories (number of rooms, square meters, etc…). A law firm could search the note field of data committed to the blockchain for certain legal terms.

Summary

For the first time in human history, a solution to the fundamental problem of trust has been discovered and shown to be viable, practical, and sustainable in the form of blockchain technology. Thanks to its cryptographically guaranteed immutability and decentralized nature, blockchain technology promises to eliminate the need for trust in a third party.

While the first blockchain implementations suffered from an inability to resolve all three constraints of security, decentralization, and scalability, the Algorand network has solved this "trilemma of blockchain" and offers a platform that not only fulfills the promise of blockchain technology but also whose scalability and transaction throughput can rival those of currently ubiquitous financial institutions.

The advent of such technology is all but guaranteed to accelerate the already steady growth of blockchain adoption. In the context of such a development, the need to query transaction data will grow as well, creating the need for fast and powerful query services.

The use of such a service however reintroduces the need for trust in a third party, the very need that blockchain was invented to eliminate. Using the cryptographic mechanism of a Zero-Knowledge Proof, the vChain framework offers a solution to this problem by returning to the query user not only the set of transactions matching the query parameters but also a set of proofs of non-membership for non-matching transactions. This guarantees the completeness of query results additionally to their authenticity.

In this project, we successfully implemented a functional version of the vChain framework for the Algorand blockchain. We believe it is fundamentally sound and can be

straightforwardly integrated into the Algorand consensus protocol and become a viable and valuable service for any application requiring fast and verifiable transaction data queries.

References

Beck, R. (2018). Beyond bitcoin: The rise of blockchain world. *Computer, 51*(2), 54-58.

Boneh, D., Boyen, X., & Shacham, H. (2004, August). Short group signatures. In *Annual International Cryptology Conference* (pp. 41-55). Springer, Berlin, Heidelberg.

Buterin, V. (2013). Ethereum white paper. *GitHub repository, 1*, 22-23.

Camenisch, J., Kohlweiss, M., & Soriente, C. (2009, March). An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *International workshop on public key cryptography* (pp. 481-500). Springer, Berlin, Heidelberg.

Chen, J., & Micali, S. (2019). Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science, 777*, 155-183.

Goldwasser, S., Micali, S., & Rackoff, C. (1989). The knowledge complexity of interactive proof systems. *SIAM Journal on computing, 18*(1), 186-208.

Micali, S. (2016). ALGORAND: the efficient and democratic ledger. *CoRR, abs/1607.01341, 3*(3), 3-3.

Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system Bitcoin: A Peer-to-Peer Electronic Cash System. *Bitcoin. org. Disponible en https://bitcoin. org/en/bitcoin-paper*.

Rogaway, P., & Shrimpton, T. (2004, February). Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption* (pp. 371-388). Springer, Berlin, Heidelberg.

Xu, C., Zhang, C., & Xu, J. (2019, June). vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data* (pp. 141-158).

Xu, C., Chen, Q., Hu, H., Xu, J., & Hei, X. (2017). Authenticating aggregate queries over set-valued data with confidentiality. *IEEE Transactions on Knowledge and Data Engineering, 30*(4), 630-644.

Yaga, D., Mell, P., Roby, N., & Scarfone, K. (2018). Blockchain Technology Overview (NISTIR-8202). *NIST: National Institute of Standards and Technology*.

Zheng, Z., Xie, S., Dai, H., Chen, X., & Wang, H. (2017, June). An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)* (pp. 557-564). IEEE.

Appendix A: VQS Code

A.1. Python, RSA Accumulator Library by Oded Leiba

Listing A.1: Extract Transactions

```
# extract txns from n blocks and returns list of txns as dicts
# with txn data including Merkle root, branch and directions
def extract_txns(data, n):
    txn_list = []
    i = 0
    for b in range(n):
        block_txns = []
        for txn in data[b]['txns']['transactions']:
            txn_list.append({'tx_id':'', 'timestamp':0, 'fee':0, 'from':'', 'to':'', 'amount':0, 'AttDigest':{'A0': 0, 'A_Final': 0}, \
                             'key':0, 'S':dict(), 'root':'', 'branch':[], 'directions':[], 'round':0})
            txn_list[i]['timestamp'] = data[b]['timestamp']
            txn_list[i]['round'] = data[b]['round']
            txn_list[i]['fee'] = txn['fee']
            txn_list[i]['from'] = txn['from']
            txn_list[i]['to'] = txn['payment']['to']
            txn_list[i]['amount'] = txn['payment']['amount']
            txn_list[i]['tx_id'] = txn['tx']

            block_txns.append(txn_list[i])

            i += 1

        txn_branches = merkle_root_and_branch(block_txns)

        for k in range(len(block_txns)):
            block_txns[k]['root'] = txn_branches[k]['root']
            block_txns[k]['branch'] = txn_branches[k]['branch']
            block_txns[k]['directions'] = txn_branches[k]['directions']

    return txn_list
```

Listing A.2: Construct Merkle Tree. Return Root, Branch and Directions

```
# given a list of tx_ids compute the Merkle tree and return the root
# as well as the branch and directions required for each txn to recompute the root
def merkle_root_and_branch(txns):
    # copy tx_ids to new list
    txid_list = []
    for txn in txns:
        txid_list.append(txn['tx_id'])

    num_nodes = len(txid_list)

    # hash all txn ids in list, duplicating the last one if number of txns is odd
    for i in range(num_nodes):
        txid_list[i] = sha_256(txid_list[i])
    if (num_nodes % 2) != 0:
        txid_list.append(txid_list[num_nodes-1])
        num_nodes += 1

    # find out number of levels to Merkle tree
    levels = math.ceil(math.log2(num_nodes))

    txn_branches = []

    for i in range(num_nodes):
        txn_branches.append({'opposite to node':txid_list[i], 'branch':[],
'directions':[], 'root':''})

    # set up counter for directions
    k = 1

    for l in range(levels):
        if (num_nodes % 2) != 0:
            txid_list[num_nodes] = txid_list[num_nodes-1]
            num_nodes += 1

        combinations = num_nodes//2

        for t in txn_branches:
            for i in range(num_nodes):
                if txid_list[i] == t['opposite to node']:

                    if i % 2 == 0:
                        t['branch'].append(txid_list[i+1])
                    else:
                        t['branch'].append(txid_list[i-1])
                    break

```

```
        # update directions to follow for root reconstruction from branch
        for i in range(len(txn_branches)):
            if (i//k)%2 == 0:
                txn_branches[i]['directions'].append('right')
            else:
                txn_branches[i]['directions'].append('left')

        for i in range(combinations):
            # concatenate and hash [0,1], [2, 3], etc...and store back in list in order
            combined_hash = sha_256(txid_list[i*2] + txid_list[(i*2)+1])
            txid_list[i] = combined_hash

        for t in range(len(txn_branches)):
            txn_branches[t]['opposite to node'] = txid_list[t//(k*2)]

        num_nodes = num_nodes//2

        k = k*2

    root = txid_list[0]

    # update root value in txn_branches list
    for i in range(len(txn_branches)):
        txn_branches[i]['root'] = root

    return txn_branches
```

Listing A.3: Reconstruct Merkle Root and Compare With User Root.

```
1.  # recomputes the Merkle root using a txn, its Merkle branch and directions
2.  # and returns True or False based on whether the txn is valid
3.  def root_from_branch(tx_id, branch, directions, root):
4.      # hash each tx_id
5.      tx_id = sha_256(tx_id)
6.      computed_root = tx_id
7.
8.      for i in range(len(branch)):
9.          if (directions[i] == 'right'):
10.             computed_root = sha_256(computed_root + branch[i])
11.         else:
12.             computed_root = sha_256(branch[i] + computed_root)
13.
14.     return computed_root == root
```

Listing A.4: Query Results

```
# takes a query dict as input and checks all txns within the time or block interval
# if txn is a match for from or/and to addresses, returns transaction with Merkle root and branch
# if txn isn't a match, computes and return nonce + proof + key (n) + AttDigest + query_address
def query_results(query, txns):
    # filter by date or block interval first and store filtered txns in list
    if query['dates']:
        filtered_txns = time_interval_txns(query['dates'], txns)
    elif query['rounds']:
        filtered_txns = block_interval_txns(query['rounds'], txns)
    # if no date or block constraints, use every transaction in the database
    else:
        filtered_txns = txns

    matching_txns = []
    proofs = []

    # check each txn for a query match and append it to the correct list
    for txn in txns:
        address = check_query_addresses(query['addresses'], txn)
        if address:
            proof, query_address_nonce = create_proof(txn, address)
            proofs.append({'proof':proof, 'query_address': address,'query_address_nonce': query_address_nonce, \
                           'n':txn['key'], 'AttDigest':txn['AttDigest'], 'tx_id':txn['tx_id']})
        else:
            matching_txns.append(txn)

    return matching_txns, proofs
```

Listing A.5: Create AttDigest. Construct and Verify Proof of Non-Membership

```
# populates the AttDigest, key and S fields of the txn dict
def create_AttDigest(txn):
    n, A0, S = setup()
    # add from and to addresses to accumulator
    A1 = add(A0, S, address_to_int(txn['from']), n)
    A2 = add(A1, S, address_to_int(txn['to']), n)
    # populate relevant txn fields in dictionary
    txn['AttDigest']['A0'] = A0
    txn['AttDigest']['A_Final'] = A2
    txn['key'] = n
    txn['S'] = S
```

```
def create_proof(txn, query_address):
    query_address = address_to_int(query_address)
    prime, query_address_nonce = hash_to_prime(query_address)
    proof = prove_non_membership(txn['AttDigest']['A0'], txn['S'], query_address,
query_address_nonce, txn['key'])
    return proof, query_address_nonce


def verify_proof(AttDigest, proof, query_address, query_address_nonce, n):
    query_address = address_to_int(query_address)
    verified = verify_non_membership(AttDigest['A0'], AttDigest['A_Final'], proof[0],
proof[1], query_address,query_address_nonce, n)
    return verified
```

Listing A.6: Query Results User Validation.

```
1.  # Query user function. Checks the validity of returned txns and verifies returned proofs
2.  def user_check_results(matching_txns, proofs):
3.      # check each returned matching txn by reconstructing the Merkel root from the branch
4.      for txn in matching_txns:
5.          if root_from_branch(txn['tx_id'], txn['branch'], txn['directions'], txn['root'])
    == False:
6.              return "Merkle root of transaction" + txn['tx_id'] + "couldn't be
    reconstructed"
7.
8.      # verify each returned proof of non-membership to confirm completeness of query
    results
9.      for proof in proofs:
10.         if verify_proof(proof['AttDigest'], proof['proof'], proof['query_address'], \
11.                         proof['query_address_nonce'], proof['n']) == False:
12.             return "proof for transaction" + proof['tx_id'] + "couldn't be verified"
13.
14.     return True
```