This article has been accepted for publication, but has not been fully edited. Content may change prior to final publication.

Scalable Privacy-Preserving Query Processing Over Ethereum Blockchain

Shlomi Linoy, Hassan Mahdikhani, Suprio Ray, Rongxing Lu, Natalia Stakhanova, Ali Ghorbani
University of New Brunswick, Fredericton, Canada
{*slinoy, hmahdikh, sray, rlu1, natalia.stakhanova, ghorbani*}*@unb.ca*

***Abstract*—Blockchain technologies have recently received considerable attention, partly due to the success of crypto-currency applications such as Bitcoin and Ethereum. As the adoption of blockchain technologies by various sectors increases, there is a demand for tools that enable regulation enforcement which include monitoring, examining and ensuring compliance of the data stored by the blockchain systems, all in a privacy preserving way. Current blockchain solutions store transactions in an append only and immutable fashion without any indexing. Consequently, the querying support is limited and inefficient. Moreover, extending it could be cumbersome, additionally, there is no support for privacy-preserving query processing. To address these issues, in this paper, we propose a system that can provide auditors with efficient, scalable and richer blockchain query processing over Hadoop and synchronized Ethereum clients. The system additionally ensures the auditors' privacy by utilizing cryptography techniques over semi-trusted servers to protect the auditors' identities, queries and their results.**

***Keywords*-blockchain; privacy; bigdata; database;**

I. Introduction

With the advent of BitCoin [1], regarded as the first crypto-currency, multiple new forms of virtual currencies, such as Ethereum [2] and ZCash [3], have emerged. The underlying technology behind crypto-currencies, is the blockchain, which enables different parties, who do not trust each other to share information, without requiring any central coordinator through the use of a robust consensus protocol such as Proof-of-Work [1]. Due to its promise, blockchain can serve as a highly trustable solution for handling structured data as a distributed database. Consequently, blockchain has inspired research in the database and systems community [4]–[7]. Ethereum extends the blockchain technology to enable a distributed computing platform that supports execution of smart contracts. Such smart contracts can be used securely in many sectors, such as government and industries. For example, Ethereum smart contracts could be used to speed up claim processing, reduce operating costs in law enforcement sectors [8], enable online decentralized secure voting [9] and be used in insurance industries [10].

Multinational retail corporations face reoccurring supply chain management issues, such as the one related to the Walmart latest romaine lettuce E. coli outbreak in North America [11]. The exact source and extent of the contaminated lettuce could not be extracted from the supply chain system, which resulted in huge losses to all parties involved. In order to reduce such related issues companies are adopting solutions based on blockchain technologies at a high rate.

Private blockchains contain sensitive information which needs to be audited according to regulatory requirements. The auditors need to have access to the company's raw blockchain data, as opposed to an intermediate processed data repository where the data might be tampered with. In addition, to provide more efficient and rigid auditing procedures, the disclosure of queries should be prevented. As the amount of data stored in the blockchain increases, the immutability of every block and its transactions ensures an exponential data size increase. Due to the linked structure of the blockchain, only sequential pass over the entire blockchain data is possible, which limits querying capabilities. In order to enable richer and more performant querying, auditors need to fetch the raw data offline. This can be achieved by implementing a capable server with access to the company's blockchain nodes. An Ethereum node can be implemented in various languages, we chose to focus on Go-Ethereum (Geth) which stores data in a key-value store (LevelDB [12]). The Ethereum node uses a general JSON RPC protocol [13] which specifies limited querying capabilities of its internal storage. These define the retrieval of one block or transaction per request. In order to retrieve multiple blocks or transactions, multiple API calls per block/transaction need to be executed. This can be inefficient, especially as the blockchain consistently expands in volume. Some third party tools have been developed to address the scalability issues in the form of a centralized service. For example EtherQL [14] downloads the Ethereum blockchain data, stores it in MongoDB and exposes an API with predefined queries. However, custom queries and private information retrieval are not supported. As another study, vChain [15] provides a way to execute boolean ranged queries using cryptographic proofs, vital to enable query integrity. However, the use of cryptographic proofs incurs high processing times, which are orders of magnitude slower compared to our proposed system. Similar additional tools are described in the related work section.

In this paper we propose a system which enables multiple auditors to perform richer queries over blockchain data in an efficient and scalable way while supporting private information retrieval by utilizing cryptography techniques over semi-trusted servers to protect the auditors' identities, queries and their results. To handle the current and rapidly

increasing blockchain data volume, the system employs Hadoop [16], which is a scalable distributed processing solution for big data. Users submit SQL queries which are transformed into MapReduce tasks and are run on Hadoop. When missing data is required, MapReduce tasks are created and used to download the data from Ethereum clients in parallel and store them in the local HDFS. An in-memory B^+Tree-based index is used to index the downloaded data for efficient future access. The entire data fetching process utilizes privacy-preserving techniques. The client and the data server share a secret key. The client sends the server an encrypted and modified query, the server sends the client the encrypted results and the client continues to further refine the results according to the full SQL query. The communication between the client and the server involves an intermediary proxy which also serves to hide the identity of the client. Our contribution can be summarized as follows:

1) Propose support for SQL query language over blockchain data, which includes SELECT statements and aggregate functions, e.g. MIN, MAX and SUM, with WHERE clauses to fetch blockchain blocks and transactional information over specified ranges and additional filters.
2) Design and implement a scalable and robust system that executes queries in a parallel and distributed fashion by utilizing Hadoop's MapReduce infrastructure.
3) Design and implement a private information retrieval approach to ensure the client's (auditor) confidentiality in the submitted query and its results during the communication between the different parties, i.e. client, proxy and the data server.

II. The proposed system

The overall system architecture is shown in Fig. 1.

A. Main components

Client - Parses the user's (auditor) query, which includes a *block_number* range and extracts a *fetch query* and a *processing query*. The *fetch query* is used to fetch all blocks/transactions required by the *processing query* while enforcing privacy preservation. The *processing query* continues to run locally on the fetched data.

Proxy - Acts as a mediator between the client and the server. It hides the source of the client from the server and filters the fetched data from the server before sending it to the client in order to save network bandwidth and decryption resources.

Data Processing Server - Serves blocks or transactions requested by the client's modified *fetch query*. The data retrieval includes fetching missing data, if required, from Geth clients and saving them for future use. These are done using Hadoop MapReduce tasks. The results are encrypted and sent securely to the proxy.

B. Assumptions:

- The client and server share a secret key for data encryption/decryption.
- The proxy and the server are assumed to be honest-but-curious; i.e. the two follow the protocol but may try to extract additional information in the process. This assumption can be promised in practice since the companies should maintain their reputation and financial interests.
- No collusion between proxy and server. Since the server decodes the *fetch query* it knows only the extended range of the blocks/transactions from the client's query. The proxy contains knowledge of the actual range in the user's query. Collusion between the server and the proxy can reveal the exact range.

C. Privacy preserving query processing

To preserve the privacy of the client's query and to securely transmit the results, the client splits, modifies and encrypts the user's query by applying a secret key which is shared between both client and server, and sends it to the proxy. The proxy cannot decrypt the query and only propagates the encrypted query to the server, which can decrypt it by using the shared key. Since the proxy serves as an intermediary between the client and the server, the latter cannot identify the query's sender and, since the client extracted this partial query from the user's query and extended its boundaries, the server cannot know the exact user's query or even its exact range. The server then retrieves the requested blocks/transactions, encrypts them, and sends back the encrypted results to the proxy. The proxy cannot decrypt the results, but can filter some unnecessary results to improve the communication and decryption performance. The proxy then sends the filtered encrypted results to the client, which decrypts the results and continues to execute the complete query.

III. System Model

In this section we describe the design and implementation of the proposed system and explore in detail the processes involving all its components. We describe the processes in a flow that starts with the client receiving the user's (auditor) query and processes it, continues to the proxy and then to the server. The process continues with the server sending the results back to the proxy, and finally the proxy to the client.

SQL query parser

The client parses the user's query using a parser built with ANTLR4 [17] and supports SELECT statements that include aggregate functions, e.g. MIN, MAX, and SUM, with WHERE clauses including range specifications. Supported data sources are either block or transaction in the FROM clause. We demonstrate our system's capabilities with two example queries as presented in Table I. The queries are inspired by [6].

Client to proxy

The client receives the query from the user, parses it using the SQL query parser and extracts two separate queries: a *fetch query* and a *processing query*. The *fetch query* is used to fetch all blocks in the extended range which is

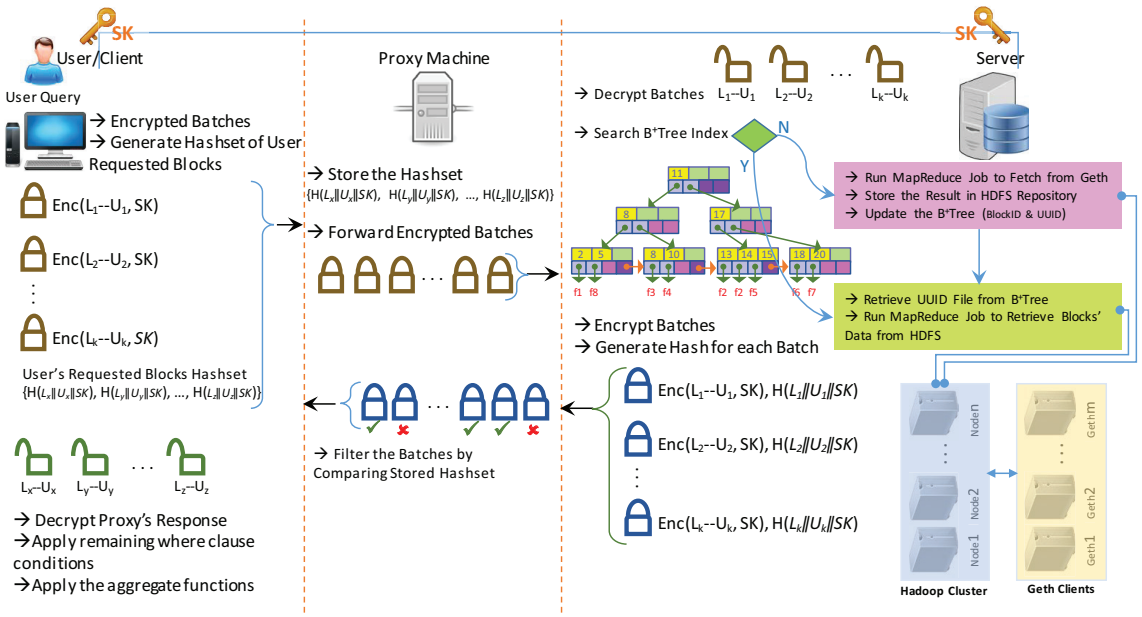


Figure 1: System Components and Model

extracted from the user query from the server. The *fetch query* is used to fetch all blocks in the extended range extracted from the user query (see Algorithm 1 for the query preparation) from the server. Once the fetched data are received by the client, the *processing query* processes the data locally. For example, the query Q1 in Table I is split into a *fetch query* part (FQ in Table II) whose results are stored locally in *fetched_results* and to a processing part (PQ in Table II). To generate the fetch query the client extracts the *block_number*'s lower/upper bounds (*LB*/*UB*) from the query. For each bound, a random value in a user specified range (*LBR*/*UBR*) is generated. One random value is subtracted from the *LB* and the second random value is added to the *UB* to extend the range. This extended range is then split into a set of ranges with a user defined batch size which is encrypted individually using a key shared with the server Sk. In addition, for each block range that intersects the range of the the original *LB*/*UB* range which is appended with Sk, the client generates a hash. The client then sends the encrypted batches and their hashes to the proxy. The proxy receives the encrypted range splits and their hashes and saves the hashes for later processing.

Table I: Query examples

Q1	SELECT MAX(value) FROM transactions WHERE block_number BETWEEN <LB> AND <UB>
Q2	SELECT * FROM transactions WHERE block_number BETWEEN <LB> AND <UB> AND account_address=<address>

Table II: Fetch/Processing query extraction example

FQ	SELECT * FROM transactions WHERE block_number BETWEEN <ELB>AND <EUB>
PQ	SELECT MAX(value) FROM fetched_results WHERE account_address = <address>

Index

The server uses an in-memory B^+Tree-based index, which is indexed on block number to retrieve the corresponding transactions data file paths in HDFS. The index serves two purposes when used on a range of block numbers:

1) It finds the HDFS repository file paths that contain the transaction/block in the provided range.
2) It finds block numbers which do not exist in the HDFS repository in the provided range.

Proxy to server

The proxy propagates the encrypted range splits to the server. The server uses Hadoop HDFS to store blocks/transactions and to run queries using MapReduce tasks

Algorithm 1: client_prepare_fetch_query

Input: *SQL_like_query* - user query, *Sk* - client/server shared key, *LBR*/*UBR* - Lower/Upper Bound Range to calculate random value, *blocks_per_batch* - number of blocks per range split
Output: *enc_range_splits_list* - list of encrypted ranges, *ranges_hash_set* - hashes of ranges intersecting query range

```
fetch_query, process_query ← parse(SQL_like_query)
LB, UB ← parse(fetch_query) // query Lower/Upper Bounds
rand_lower ← get_random_in_range(LBR)
rand_upper ← get_random_in_range(UBR)
ELB ← min(1, LB - rand_lower) // Extended Lower Bound
EUB ← UB + rand_upper // Extended Upper Bound
ranges_hash_set ← {}
enc_range_splits_list ← list()
extended_range_size ← (EUB - ELB + 1)
splits_count ← extended_range_size/blocks_per_batch
for i ← 0 to splits_count by 1 do
    split_start ← ELB + i · blocks_per_batch
    split_end ← split_start + blocks_per_batch
    add_range_split(IN LB, IN UB, IN split_start, IN split_end, IN
      Sk, OUT enc_range_splits_list, OUT ranges_hash_set)
// last remaining range split will be addressed similarly
```

Algorithm 2: server_get_data

Input: *enc_range_splits_list*, *Sk* - client/server shared key, *blocks_per_task*, *Geth_client_ip_addresses*, *threads_per_task*, *HDFS_output_dir*, *local_results_dir*, *block_number_pos* - position in results line
Output: *enc_ranged_res_hash_map* - all blocks in range, split by ranges, encrypted and hashed

```
range_splits_list ← list()
for enc_range in enc_range_splits_list do
    range ← AES.decrypt(enc_range, Sk)
    range_splits_list.add(range)
ELB ← range_splits_list.first().LB // Extended Lower Bound
EUB ← range_splits_list.last().UB // Extended Upper Bound
missing_blocks_list ← Index.get_missing_blocks(ELB, EUB)
if missing_blocks_list ≠ ∅ then
    server_fetch_data_from_Geth(IN missing_blocks_list, IN
      blocks_per_task, IN Geth_client_ip_addresses, IN
      threads_per_task)
server_retrieve_data_from_Hadoop(IN ELB, IN EUB, IN
  range_splits_list, IN HDFS_output_dir, IN local_results_dir, IN
  block_number_pos, OUT enc_ranged_res_hash_map)
```

Algorithm 3: server_retrieve_data_from_Hadoop

Input: *ELB/EUB* - Extended Lower/Upper Bound, *range_splits_list* - range splits, *HDFS_output_dir*, *local_results_dir*, *block_number_pos* - position in results line
Output: *enc_ranged_res_hash_map* - all blocks in range, split by ranges, encrypted and hashed

```
index_file_name_list ← Index.get_indexed_file_names(ELB, EUB)
update retrieve_map_reduce template source code to fetch only data
  in the block numbers range between ELB and EUB
compile the updated map_reduce code into
  updated_retrieve_map_reduce
updated_retrieve_map_reduce.task(index_file_name_list,
  HDFS_output_dir)
copy all mapppers results files from HDFS_output_dir to
  local_results_dir
results ← merge all mappers results files in local_results_dir
ranged_results_hash_map ← {}
for range in range_splits_list do
    ranged_results_hash_map.add(range, list())
sorted_range_splits_list ← sort range_splits_list on LB
for res in results do
    block_number ← res.get(block_number_pos)
    range ← sorted_range_splits_list.get_range(block_number)
    // get_range function searches sorted_range_splits_list for a
    // range with the largest LB that is smaller or equal to block
    // number. Works in O(log n)
    ranged_results_hash_map.get(range).add(res)
enc_ranged_res_hash_map ← {}
for range in ranged_results_hash_map.keys do
    range_hash ← SHA256(range|Sk)
    batch ← ranged_results_hash_map.get(range)
    enc_batch ← AES.encrypt(batch, Sk)
    enc_ranged_res_hash_map.add(range_hash, enc_batch)
```

To serve queries that contain a block number range, the server executes two MapReduce tasks (Algorithm 2):

1) A MapReduce task to fetch missing blocks/transactions that are not already stored in the HDFS repository from Geth [18] clients using Web3j [19].
2) A MapReduce task to retrieve existing blocks/transactions from the HDFS repository (Algorithm 3).

in order to retrieve existing data from HDFS repository and to fetch missing data from Geth clients.

Upon receiving the encrypted query from the proxy containing the extended block numbers range splits, the server decrypts the ranges and extracts the extended lower bound (*ELB*) and extended upper bound (*EUB*) of the range. It then searches the index for the block numbers that are missing in the index, and consequently, in HDFS. If there are such blocks, a MapReduce task is run to fetch the missing blocks/transactions from the Geth clients. To do so the server prepares an input file for the MapReduce task. The MapReduce task is configured to consume the input file one line at a time. Each input file line contains a different iterating Geth client IP address in order to distribute the fetching load between all tasks in addition to a user defined *threads_per_task* count to use in each node to fetch the blocks concurrently. The number of input file lines and, hence, the number of tasks is calculated by *missing_blocks_list*.size()/*blocks_per_task*. The *blocks_per_task* parameter should be determined to maximize memory consumption in each node in order to utilize the nodes efficiently. The MapReduce task is configured to not use reducers in order to prevent memory issues when downloading a large amount of data and to maximize resources usage (more nodes are used as mappers) when fetching the data from Geth clients. Each mapper communicates with a synchronized Geth client using Web3j to fetch the missing data and produces a result file in the *HDFS_output_dir* directory. When all MapReduce tasks are complete, the mappers' result files are downloaded locally to the server from HDFS and merged into one file that contains all blocks/transactions delimited by a new line. The file is then uploaded to the HDFS repository to be used in future queries. The index is updated to point to the uploaded file from all the missing block numbers. An offline process is used to split this file into smaller files and update the index accordingly for more efficient retrievals in future queries. In the next step, the server retrieves all the blocks requested by the query from the HDFS repository, encrypts them, generates their hashes, and returns both to the proxy. To do so (See Algorithm 3) the server searches the index for all the HDFS repository indexed file paths that contain all block numbers in the extended query range. The resulting list of file paths is used as input to the MapReduce task that retrieves blocks/transactions from HDFS. To prepare this MapReduce task, the server uses a Java template code that provides the functionality for a task to get each line from its input file and output the line only if it passes a filter statement. The filter statement is a placeholder for an if-condition that is generated by the server according to the extended lower and upper bounds (*ELB*/*EUB*) of the query. The code is then compiled to produce the mapper task binaries. This MapReduce task is also configured to not use reducers to preserve memory and maximize node resources. Each mapper uses a text file input split with a default split size of 64M. The task is then run with the indexed file paths retrieved from the indexer and the mappers' results files are copied to the server locally and merged into a single file

whose lines are delimited by a new line. The merged results file are then partitioned into batches. Each batch contains block lines with block numbers in the batch's specific range. The ranges are extracted from the range splits provided in the query. Each batch is encrypted using the secret key shared with the client and the hash of the batch range plus the secret key is calculated.

Server to proxy

The server sends the encrypted results and their hashes to the proxy. The results are composed of tuples. Each tuple contains an encoded batch with block/transaction lines, and the second item in the tuple is the hash of the batch range plus the shared key. The proxy filters the received results by retaining only the batches whose range hash is contained in hashes provided by the client.

Proxy to client

The proxy sends the filtered results to the client. The client decrypts each batch with the shared key and merges it into a *fetch_results* list. Since a batch may contain information on block numbers not in the full user's query range, the client filters these lines. The client continues to locally execute the process query on the *fetch_results*, which includes the aggregate functions and other WHERE clause filters.

IV. Evaluation

We used a dataset from live Ethereum feed provided by Geth clients. The experiments were conducted on a cluster of four machines, each having an Intel(R) Xeon(R) CPU E5472 @ 3.00GHz and 8GB of memory. Two additional machines were used for running Ethereum Geth clients. Table I contains the queries used for evaluation. Table III shows the varied block numbers and their ranges values used in our experiments. Each block range configuration was run with either 1 or 4 nodes per cluster. See Table IV for the configuration details. "Blocks per batch" indicates the number of blocks each task downloads. "Threads per task" indicates the number of threads each task uses to download the transactions/blocks concurrently from the Geth clients.

The experiments evaluate two independent processes:

1) Server fetch time, which consists of fetching missing blocks, index update, storing of newly fetched data into HDFS, and fetching all query data from HDFS.
2) All steps of the query processing time after the server's data fetch. This includes server results encryption, proxy filtering, client results decryption, applying WHERE clause filtering and aggregate function.

The server's total fetch time is dependent only on the query's extended range size and not on any other part of the query. This means that different queries with the same extended range should present similar server total fetch times. Fig. 2 demonstrates the significant improvement of our solution due to the parallel downloading and fetching of all the query blocks/transactions.

Table III: Block numbers and ranges used

Blocks	LB	UB	Blocks	LB	UB
100	3000000	3000100	200K	3000000	3200000
1K	3000000	3001000	300K	3000000	3300000
10K	3000000	3010000	400K	3000000	3400000
100K	3000000	3100000	500K	3000000	3500000

Table IV: Single and multiple nodes configuration

Nodes	Geth clients	Blocks per batch	Threads per task
4	2	5000	4
1	1	5000	8

5.1 Fetching missing data from Geth clients

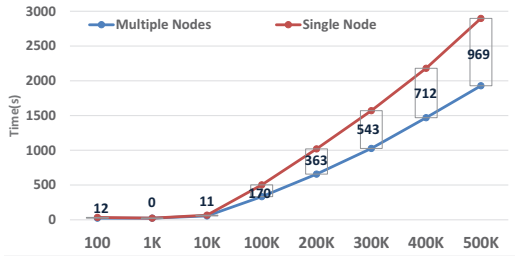
When the system is initialized (first time), all Ethereum blocks generated until that point are needed to be fetched from Geth client(s) and the index needs to be built. After that, a background process is run periodically to fetch newer blocks from Geth clients. Fig. 2(a) shows the time improvement when all blocks are fetched from Geth client(s) by using multiple Hadoop nodes in comparison to that with a single node. The average speedup achieved is 1.52×. In this configuration, 2 Geth clients are used with the 4 Hadoop nodes. It may seem that the speedup should be at least 2. However, since all 4 nodes send requests to the same 2 Geth clients, the consequent load on each Geth client lowers the speedup gain. Using at least 2 more Geth clients would significantly increase the speedup.

5.2 Fetching data from HDFS repository

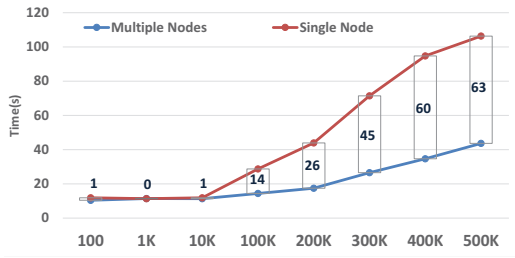
For most queries, it is expected that the blocks specified by the query ranges are already fetched from Geth clients. Fig. 2(b) shows the time improvement for the data retrieval from HDFS, when all blocks are already fetched from Geth clients and the index is updated accordingly. Multiple nodes configuration is compared to the single node configuration. The average speedup achieved is 2.47×. This can be attributed to our use of the default text file split size of 64M. We believe that the speedup can be increased by reducing default text file split size or alternatively, by retrieving a bigger range of blocks resulting in bigger data files. Fetching from HDFS is the typical use case where most data is already indexed and occasionally few blocks from live data are fetched from Geth clients.

5.3 Steps following server data fetching

Following the fetch of missing/existing data, the server encrypts and calculates the range hash of each batch and sends these results to the proxy, which filters the relevant batches according to the hashes it received from the client and sends them to the client. The client then decrypts the results and filters the data that are in the user query range (before continuing to execute the *processing query*). The summary and breakdown of the processing times for these steps are shown in Fig. 3 for the two types of queries in Table I. Fig. 3(a) summarizes the processing times for query Q2 in Table I and Fig. 3(b) summarizes the processing times for query Q1 in Table I. As can be seen, the relative processing times of the different steps are similar for 100k and above

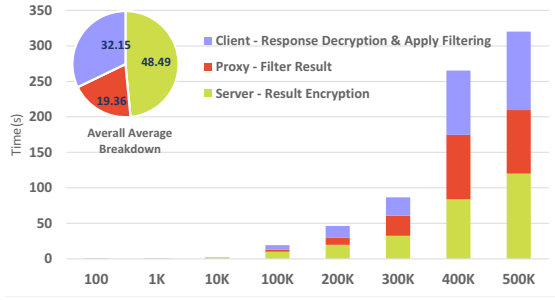


(a) Fetching from Ethereum Geth clients

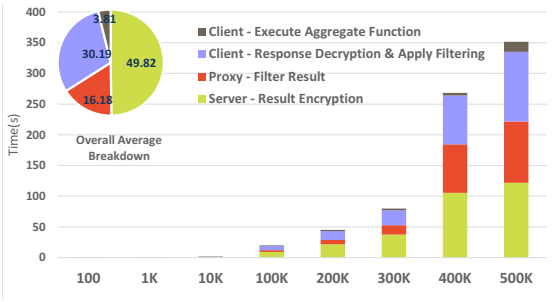


(b) Fetching from HDFS

Figure 2: (a) Fetching all data blocks from Ethereum Geth clients (system initialization). (b) Fetching of all requested data from HDFS (already indexed).



(a) Query Q2 in Table I



(b) Query Q1 in Table I

Figure 3: Performance breakdown following data fetch

ranges in both query types. The average breakdown of the different processing parts can be seen in the pie charts. Here about 50% of the processing time is attributed to the results encryption by the server, followed by about 30% of the processing time for the decryption of the filtered results by the client, and about 20% of the processing time to filter the results by the proxy. In Query Q1 there is an additional use of an aggregation function, which is not specified in Q2. This adds the aggregation time to the client side, which is relatively negligible in comparison to the other parts.

V. Related Work

Ethereum block explorers are useful tools for block and transaction queries, as they allow to follow transactions and diagnose possible problems. Some usage examples include: finding all the information about a specific block/transaction, all transactions in a specific block, what transactions were made to/from specific account-address, etc. Some implementations of Ethereum blockchain explorer include: ERC20-Exporter [20] - a lightweight explorer that looks into all information on-the-fly from a back-end Ethereum node. It was developed with Node.js, Express.js and Parity. ERC20 [21] provides a common list of rules for Ethereum tokens to follow within the larger Ethereum ecosystem, allowing developers to accurately predict interaction between tokens. These rules include how the tokens are passed between addresses and how data within each token are accessed. ERC20-Exporter is used to explore the ERC20-based Ethereum tokens and supports Parity back-end node (the authors state it also supports Geth client although this was not tested yet [20]). Initial data export for large tokens takes up to 30 minutes, as it tries to scrape the blocks info like Ethereum Scraper [22] that exports the blockchain data by indicating start and end block number. EthExplorer [23] - is a work in progress explorer developed with Node.js. EtherScan [24], ETCExplorer (Ethereum Classic Blockchain Explorer) [25], and Ethplorer [26] provide web-based UI and supports mostly related RESTful APIs, such as *getTopTokens* and *getTokenHistory*. They are implemented by calling basic methods from Ethereum clients and each implementation enforces its own limitation. For example, in EthersScan the API requests are limited to 5 requests/sec.

Privacy-preserving concerns or even simple logical combinations in the user requests are not supported by any of the existing systems. Etherchain Light [27], another lightweight blockchain explorer built with Node.js, Express.js and Parity, retrieves information on the fly from a back-end Parity node. It has extended the Ethereum Web3 API to provide some statistical measures such as transaction count and is still under development. Ethereum Explorer [28] is a decentralized client for Ethereum that interacts with the Ethereum blockchain via the Ethereum Web3 API, and provides users with basic interfaces to explore blocks. EtherQL [14] implements a query layer for Ethereum that supports some powerful APIs e.g., range query and top-k queries and is backed by MongoDB as persistence layer to store blockchain data. vChain [15] proposes a solution to produce boolean query results in blockchains. The query result is paired with a cryptographic proof to guaranty its integrity. To support this verifiable query processing, vChain requires

to modify the block structure to incorporate an authenticated data structure. To optimize query efficiency, inter-block and intra-block indexes are implemented. Our solution differs by not requiring any modifications to the blockchain. Where vChain supports a query in a specific format our solution supports an SQL query. vChain uses homomorphic encryption techniques which are costly compared to our solution's use of relatively lightweight symmetric encryption AES and SHA256. In addition vChain performs all cryptographic calculations on the server containing the full node, in our solution the query initiates the encryption and download of the ranged data from a blockchain node and eventually continues with the processing of the query's main logic on the client side. This results in server (and client) processing times which are orders of magnitude lower than in vChain. Finally, The SQL query in our system provides more capabilities (e.g. aggregation), which can be easily extended to support more complex features which are not restricted by the homomorphic encryption constraints as in vChain.

VI. Conclusion

As an increasing number of sectors are integrating blockchain technologies, it is important to have an efficient and secure auditing system to help monitor and analyze blockchain repositories while preserving the auditors' privacy. To this end, our proposed system uses big data processing techniques to support all the above requirements. Our system provides a secure, robust, and scalable way to process SQL queries over any blockchain. It enables multiple auditors to execute queries in an efficient and scalable way, while preserving the privacy of auditors' identities and prevent the disclosure of the queries being used and their results. The system supports SQL queries with range and aggregate functions which are transformed into MapReduce tasks to be run on Hadoop. The system uses Hadoop's MapReduce tasks to efficiently fetch missing blocks from Ethereum clients. In addition, an in-memory B+Tree-based index is utilized to index previously downloaded and stored Ethereum blocks. We conducted a systematic performance study, which suggests that the system's performance can improve by adding more Hadoop nodes and more synchronized Ethereum clients.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform, 2013."

[3] "ZCash, 2016."

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *EuroSys, 2018*.

[5] Z. Xu, S. Han, and L. Chen, "Cub, a consensus unit-based storage scheme for blockchain system," in *ICDE ,2018*.

[6] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *SIGMOD, 2017*.

[7] D. Puthal, N. Malik, S. P. Mohanty, E. Kougianos, and G. Das, "Everything you wanted to know about the blockchain: Its promise, components, processes, and problems," *IEEE Consumer Electronics Magazine*, vol. 7.

[8] "Blockchain has grabbed the attention of investors," https://www.cnbc.com/2018/04/02/blockchain-has-grabbed-theattention-of-investors.html, 2018.

[9] A. K. Koç and U. C. Çabuk, "Towards secure e-voting using ethereum blockchain."

[10] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, and V. Santamaría, "Blockchain and smart contracts for insurance: Is the technology mature enough?" *Future Internet*, vol. 10.

[11] "From farm to blockchain: Walmart tracks its lettuce," https://www.nytimes.com/2018/09/24/business/walmart-blockchain-lettuce.html.

[12] "Leveldb, 2014."

[13] "Ethereum json rpc," https://github.com/ethereum/wiki/wiki/JSON-RPC, 2014.

[14] Y. Li, K. Zheng, Y. Yan, Q. Liu, and X. Zhou, "EtherQL: a query layer for blockchain system," in *DASFAA, 2017*.

[15] C. Xu, C. Zhang, and J. Xu, "vchain: Enabling verifiable boolean range queries over blockchain databases," *SIGMOD, 2019*.

[16] "Apache hadoop, 2009."

[17] "Antlr," http://www.antlr.org/, 1995.

[18] "Go ethereum, 2014."

[19] "Web3j , 2016."

[20] "Erc20exporter," https://github.com/gobitfly/erc20-explorer, 2017.

[21] "Erc20 token standard," https://theethereum.wiki/w/index.php/ERC20_Token_Standard#The_ERC20_Token_Standard_Interface.

[22] "Ethereumscraper," https://github.com/medvedev1088/ethereum-scraper, 2018.

[23] "Ethexplorer," https://github.com/etherparty/explorer, 2015.

[24] "Etherscan," https://github.com/sebs/etherscan-api, 2016.

[25] "Etcexplorer," https://github.com/ethereumproject/explorer, 2016.

[26] "Ethplorer," https://ethplorer.io/, 2016.

[27] "Etherchain light," https://github.com/gobitfly/etherchain-light, 2017.

[28] "Ethereum explorer," https://github.com/mix-blockchain/ethereum-explorer, 2017.