RESEARCH   **Open Access**

Leveraging blockchain for immutable logging and querying across multiple sites

Mustafa Safa Ozdayi[1*], Murat Kantarcioglu[1] and Bradley Malin[2]

Abstract

Background: Blockchain has emerged as a decentralized and distributed framework that enables tamper-resilience and, thus, practical immutability for stored data. This immutability property is important in scenarios where auditability is desired, such as in maintaining access logs for sensitive healthcare and biomedical data. However, the underlying data structure of blockchain, by default, does not provide capabilities to efficiently query the stored data. In this investigation, we show that it is possible to efficiently run complex audit queries over the access log data stored on blockchains by using additional key-value stores. This paper specifically reports on the approach we designed for the blockchain track of iDASH Privacy & Security Workshop 2018 competition. In this track, participants were asked to devise an efficient way to run conjunctive equality and range queries on a genomic dataset access log trail after storing it in a permissioned blockchain network consisting of 4 identical nodes, each representing a different site, created with the Multichain platform.

Methods: Multichain duplicates and indexes blockchain data locally at each node in a key-value store to support retrieval requests at a later point in time. To efficiently leverage the key-value storage mechanism, we applied various techniques and optimizations, such as bucketization, simple data duplication and batch loading by accounting for the required query types of the competition and the interface provided by Multichain. Particularly, we implemented our solution and compared its loading and query-response performance with SQLite, a commonly used relational database, using the data provided by the iDASH 2018 organizers.

Results: Depending on the query type and the data size, the run time difference between blockchain based query-response and SQLite based query-response ranged from 0.2 seconds to 6 seconds. A deeper inspection revealed that range queries were the bottleneck of our solution which, nevertheless, scales up linearly.

Conclusions: This investigation demonstrates that blockchain-based systems can provide reasonable query-response times to complex queries even if they only use simple key-value stores to manage their data. Consequently, we show that blockchains may be useful for maintaining data with auditability and immutability requirements across multiple sites.

Keywords: Blockchain, Multichain, Query-response, Cross-site data sharing

*Correspondence: mustafa.ozdayi@utdallas.edu
[1]Department of Computer Science, Erik Jonsson School of Engineering and
Computer Science, The University of Texas at Dallas, 800 W. Campbell Road,
Richardson 75080, TX, USA
Full list of author information is available at the end of the article

Ozdayi *et al. BMC Medical Genomics* 2020, **13**(Suppl 7):82

Page 2 of 7

Background

Blockchains allow a set of parties to collaboratively maintain a collection of data on a tamper-resilient and decentralized ledger. This provides numerous benefits compared to traditional data storage models, where the administration of a shared database is delegated to either one or more trusted entities.

One particularly notable benefit is that a blockchain-based data storage solution mitigates security issues that can arise from malicious administrators. Moreover, it eliminates the potential for a single point of failure because data is replicated across multiple entities. Due to their immutability and auditability guarantees, blockchains are useful for storing access logs from multiple sites to different datasets (e.g., genomic research data). This is because access logs require strong auditability (e.g., auditing accesses to genomic information), transparency (e.g., publicly verifying that certain data is not misused by checking the logs) and tamper-resistance (e.g., preventing an attacker from manipulating the stored logs).

Furthermore, blockchains provide access to an uniform view of data that is logged from different sites. This, in conjunction with its other properties, can be beneficial in various contexts as demonstrated by previous investigations, such as those in the healthcare domain [1–3].

However, the underlying data structure of a blockchain, by default, does not provide for an efficient technique to query the stored data. To overcome this limitation, most existing blockchain implementations, such as Bitcoin [4] and Ethereum [5], provide support for key-value stores to duplicate and store data on the blockchain. These key-value stores are then leveraged to support simple key-based retrieval queries that return the associated values.

In this paper, we show how to leverage the key-value store provided by a blockchain to run complex queries efficiently (e.g., range queries) on the blockchain data. We specifically report on the approach we designed for the blockchain track of iDASH Privacy & Security Workshop 2018 competition [6]. Our findings show that our approach induces reasonable overhead, in terms query-response time, in comparison to a traditional relational database management tool.

Blockchain

Blockchain was first introduced by Nakamato as the underlying ledger of the now famous Bitcoin cryptocurrency [4]. Briefly, a blockchain is an append-only, distributed and replicated database. It allows the participants of a network to collectively maintain a sequence of data in a tamper-resilient way. More importantly, it does so without a requirement for a trusted third party by invoking a consensus mechanism.

Informally, a blockchain network operates as follows: participants broadcast their data and certain nodes called *miners* gather and store the data they receive in wrapper structures called *blocks*. Through a consensus mechanism, the network elects a leader miner in a decentralized fashion for a sequence of epochs. The epoch leader broadcast his block to the network and, having received the leaders block, other nodes store it in their local memory where each block maintains a hash-link to the previous block.

The consensus algorithm that the blockchain network deploys may depend on whether or not the network is *permissionless*. For example, Bitcoin operates on a permissionless network, where anyone is free to join and there is no uniform view of the network across participants. It utilizes a cryptographic puzzle called Proof-of-Work [7] to achieve consensus. This makes tampering with the order of blocks computationally infeasible when the majority of the computation power in the network follow the protocol honestly.

In *permissioned* networks however, participants can employ more efficient consensus algorithms, such as PBFT [8]. This is because the identity and number of participants are known to every party.

Multichain

Multichain is a platform to deploy permissioned blockchains [9]. In this context, permissioned means that access to the blockchain network can be arbitrarily restricted. Such networks are usually initialized by a single party who, at a later point in time, allocates permissions to other nodes to join the network and participate in the consensus protocol. For consensus, Multichain deploys a variant of a classical Byzantine fault tolerance algorithm whose exact details are provided in the *Mining in MultiChain* section of the corresponding whitepaper [10].

To handle queries efficiently, Multichain provides a module called *streams*, which uses an abstraction of a dictionary (i.e., key-value store) on top of a blockchain [11]. The streams module allows a node to store an arbitrary datum and an associated key by submitting a key-value pair in a transaction to the blockchain. Multichain duplicates and indexes the data stored on the blockchain in LevelDB (a key-value store [12]), which is locally maintained by each node to serve queries submitted to the blockchain efficiently [13]. In other words, the streams module allows a node to interact with the underlying key-value store. It is possible to store multiple values with the same key, such that query results can be returned as lists.

The streams module supports the following methods (among others) on top of a blockchain.

- **createDictionary**(dictionary-name): Creates a dictionary with the specified name.
- **insert**(dictionary-name, value, key): Inserts the key-value pair to the specified dictionary.

Ozdayi *et al. BMC Medical Genomics* 2020, **13**(Suppl 7):82

Page 3 of 7

- **retrieve**(dictionary-name, key): Retrieves the value(s) corresponding to the given key from the specified dictionary.

We note that it is possible to create an arbitrary number of dictionaries on top of a blockchain, each of which stores data independently.

Overview of the task

The blockchain track of the iDASH Privacy & Security Workshop 2018 competition provided a genomic dataset access log trail [14] in which each log consists of 7 fields: *timestamp, node, ID, ref-ID, user, activity* and *resource*. The activity and resource fields can take on arbitrary string values. The other fields can take on arbitrary positive integer values.

This trail is stored on a blockchain, where it is assumed that the trail arrives as a data stream (i.e., one log at a time). The competition rules dictated that the blockchain must be created using Multichain version 1.0.4 and the network must consist of four identical nodes, each representing a different site, initialized with default parameters per Multichain specifications. It should be possible to insert and query the data using any node. Also, the rules prohibited the use of using any off-chain mechanisms to handle the data other than what Multichain provides.

The goal is to develop a system that can query the blockchain efficiently under this setting while minimizing loading times and storage space. A viable solution must support two types of queries:

- Conjunctive equality queries on selected fields.
- A range query on the timestamp field.

Furthermore, the system should support the reporting of results in ascending or descending order on any field.

Methods

We now describe the techniques and the optimizations we deployed to handle queries efficiently in our system. We note that although our system is explicitly tuned for the blockchain track of iDASH 2018, our methods can be applied to support more general queries.

In what follows, we first describe how to handle conjunctive equality queries. Next, we describe how to handle range queries. We then explain how query response times can be further improved via batch loading.

Conjunctive equality queries

For each field, we create a dictionary that uses field values as keys and logs as values. For example, in the *user* dictionary at key 1, we have the logs whose user field's value is 1. Similarly, in the *ID* dictionary at key 2, we have the logs whose ID is 2. Note that the logs are duplicated for each field.

When processing such a query, we first find the most restrictive field key and retrieve the logs from the corresponding dictionary with that key. Next, we filter the retrieved logs with other field keys. Finding the most restrictive field can be achieved efficiently. This is because Multichain keeps track of how many items are stored at a key in a dictionary, which can be accessed by a **getCount**(dictionary-name, key) method.

As an example, consider a query that requests logs with $user = 1 \wedge ID = 2$. We first compute $x = $ **getCount**(user-dictionary, 1) and $y = $ **getCount**(id-dictionary, 2). Then if $x > y$, we retrieve the logs from the id-dictionary, via **retrieve**(id-dictionary, 2) and discard the logs whose user field is not equal to 1.

Range queries on a single field

To handle range queries, we designed a *bucketization* technique. That is, we create intervals of a fixed size and assign each log to exactly one of those intervals depending on the queried field's value. Each interval is referred as a *bucket* and identified by an unique value. Particularly, our bucketization technique works as follows: first, we create a separate dictionary, *range-dictionary*, in which we assign each log the key $= \lfloor$Timestamp of log$/N\rfloor$, where N is a predefined bucket size.

Next, given a range query $[x, y]$, which requests logs whose timestamps are between x and y (inclusive), we retrieve all of the logs with keys $\lfloor x/N \rfloor + 1, \lfloor x/N \rfloor + 2, \ldots, \lfloor y/N \rfloor - 1$. Finally, we retrieve and perform a linear scan of the logs at keys $\lfloor x/N \rfloor$ and $\lfloor y/N \rfloor$ and discard logs whose timestamps are not in $[x, y]$.

Note that, for each log with a key in range $[\lfloor x/N \rfloor + 1, \lfloor y/N \rfloor - 1]$, it is guaranteed that the log is in $[x, y]$. As a result, we do not need to scan the logs stored at these keys.

Improving retrieval speed via batch loading

We observed that in the Multichain platform loading logs in batches can substantially improve retrieval speed. Here batch loading means that, instead of inserting one log in each transaction, we buffer and insert several logs in a single transaction.

We observed that if we load logs as batches of size k, then retrieving these logs would be roughly k times faster than storing them one at a time. The competition rules required the solution to be crash-resistant, so a straightforward way of buffering would have failed to meet this requirement. For example, if our buffer size is 4, then we load logs to the blockchain in batches of size 4. Yet if the system crashes after the first 2 logs arrive, then both of these logs would be lost due to the fact that the contents of the buffer was not loaded to the blockchain at the crash time.

To overcome this problem, we extend our solution to maintain two dictionaries per field, a *batch dictionary* and

Ozdayi *et al. BMC Medical Genomics* 2020, **13**(Suppl 7):82

Page 4 of 7

a *regular dictionary*. In the batch dictionary, we load logs in batches, while in a regular dictionary, we load the logs one at a time (i.e., a buffer size of 1). When retrieving data, we select from the batch dictionary, compare the size of the retrieved list with the corresponding list in the regular dictionary and execute a crash recovery (if needed).

For example, imagine a query that attempts to retrieve logs with *node* = 1. To support this query, we first retrieve the logs from the batch dictionary via *batchLogs* = **retrieve**(batch-node-dictionary, 1). We then compute the length of the corresponding list in the regular dictionary via *logListSize* = **getCount**(regular-node-dictionary, 1). Then, if the size of *batchLogs* is equal to *logListSize*, we simply return *batchLogs* as the result.

Otherwise, it becomes evident that we lost some data from the batch dictionary due to a crash. To recover, we compute the difference between the size of *batchLogs* and *logListSize*, i.e., *x = logListSize* - **size**(batchLogs). We then retrieve the last *x* items from the regular dictionary and append these to both *batchLogs* and the batch dictionary. Finally, we return *batchLogs* as the result.

Results

In this section, we report on a set of experiments designed to characterize the performance of our system.

Implementation details and experiment setting

We implemented our solution using Python 3.5.2, Multichain 1.0.4 and used the Savoir wrapper to interact with Multichain API [15]. Our code is available at [16]. Our test setup consisted of four identical virtual machines with the following specifications: 2-Core CPU (2.6. GHz Intel Xeon E5), 7.5 GB of RAM and 50 GB of storage with the Ubuntu 14.04 LTS operating system.

We used the dataset supplied by the competition organizers, which consisted of four files, one per node, in which each file has 10^5 logs.

From the previous discussion regarding batch loading, it is evident that the larger the buffer size, the faster the retrieval speed. However, Multichain imposes a size limit on each transaction, such that it is not possible to increase buffer size arbitrarily. We observed that, for the given dataset, the transaction size limit is reached for a buffer size around 10^4. As a result, we set the buffer size to 10^4.

Now, it can be seen that the number of buckets we have to retrieve decreases with the increasing bucket size. However, the number of individual logs we have to scan may increase. This is because it depends on the distribution of logs over the buckets and the query. Note that if one chooses bucket size poorly, it could be the case that all the logs would go to the same bucket.

To determine an appropriate bucket size, we ran several range queries of varying sizes on the given data and measured the average running time. During our empirical
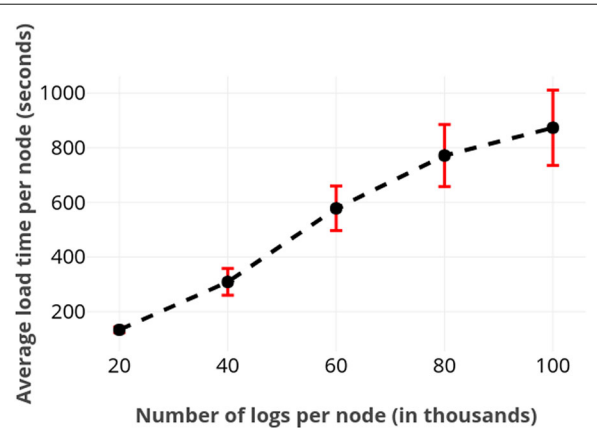

Fig. 1 Average time required for a node to load logs of various sizes. The large standard deviation is likely due to network latency. As expected, load times scale linearly with the number of the logs

analysis, we observed an increase in the average running time as bucket size increased from 10^1 to 10^7. After that point however, average running time started to decrease with the increasing bucket size. As a result, we selected a bucket size of 10^7.

Finally, we compared our solution's performance with a traditional relational database, namely SQLite 3.22 which we ran in one of the virtual machines. We report on the average measures over 10 runs and illustrate standard deviations by error bars in our plots. Also we note that

Experiments

First, we measured the load time by using logs of various sizes concurrently at each node. Figure 1 depicts the average load time of a node with respect to number of logs loaded by it. We further plot the influence of file size on total load time, which corresponds to the slowest node. Those results are provided in Fig. 2. These figures do not include the results from SQLite because
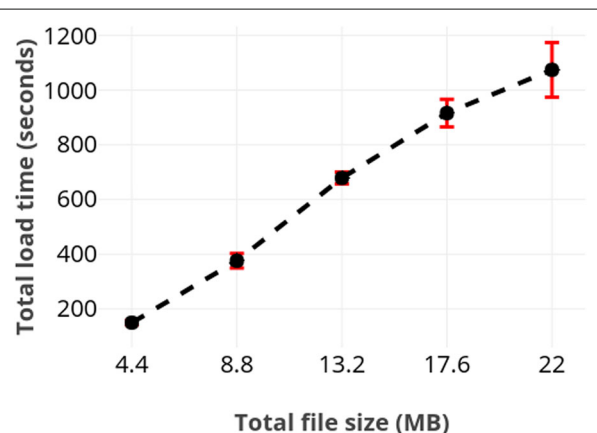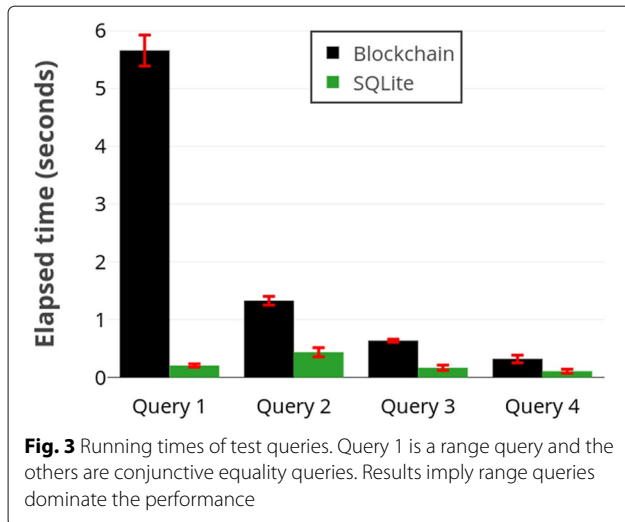

Fig. 2 Total load time required as a function of file size. As expected, the total load time scaled linearly in the size of the file

Ozdayi *et al. BMC Medical Genomics* 2020, **13**(Suppl 7):82

Page 5 of 7

Fig. 3 Running times of test queries. Query 1 is a range query and the others are conjunctive equality queries. Results imply range queries dominate the performance

our solution is substantially slower. For instance, loading all 400.000 logs required merely about 4 seconds in SQLite.

Next, we investigated query-response times. We ran the test queries supplied by the competition organizers. The queries and the number of records returned by were as follows:

- Query 1: SELECT * FROM Chain WHERE user = 7 AND (timestamp BETWEEN 1522257730000 AND 1522449160000). Returns 30489 records.

- Query 2: SELECT * FROM Chain WHERE resource = 'MOD_WormBase'. Returns 67462 records.

- Query 3: SELECT * FROM Chain WHERE user = 1 AND resource = 'TOPMed'. Returns 17098 records.

- Query 4: SELECT * FROM Chain WHERE node = 3 AND ref−ID = 40345 ORDER BY timestamp ASC. Returns 5983 records.

The running times for these queries are shown Fig. 3 As the results indicate, the main bottleneck of our solution is the range query. We also note that SQLite internal representation and processing scheme is quite different than our method. As such, the SQLite running time is not always highly correlated with the blockchain time.

In Fig. 4, we compare the range query performance of our solution with respect to SQLite's performance. We observe both methods scale linearly where the difference is between 5-6 seconds.

Finally, Fig. 5 illustrates how the number of records retrieved influences the query-response time. In this experiment, we ran queries without any restrictions (i.e., query returns every stored log) after loading appropriate number of logs. Given how our approach handles conjunctive equality queries, this plot also represents the performance of conjunctive equality queries. This is due to the fact that a conjunctive equality query makes a call
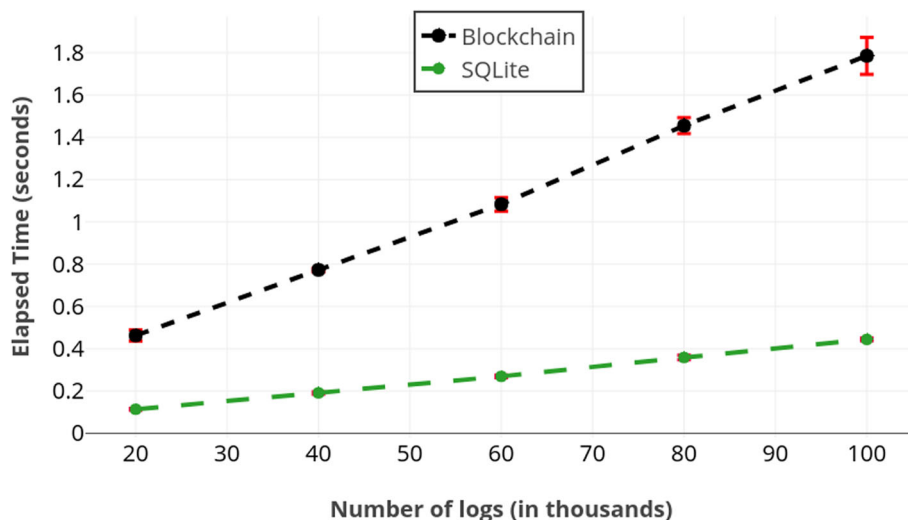

Fig. 4 Running times of range queries

Ozdayi *et al. BMC Medical Genomics* 2020, **13**(Suppl 7):82

Page 6 of 7
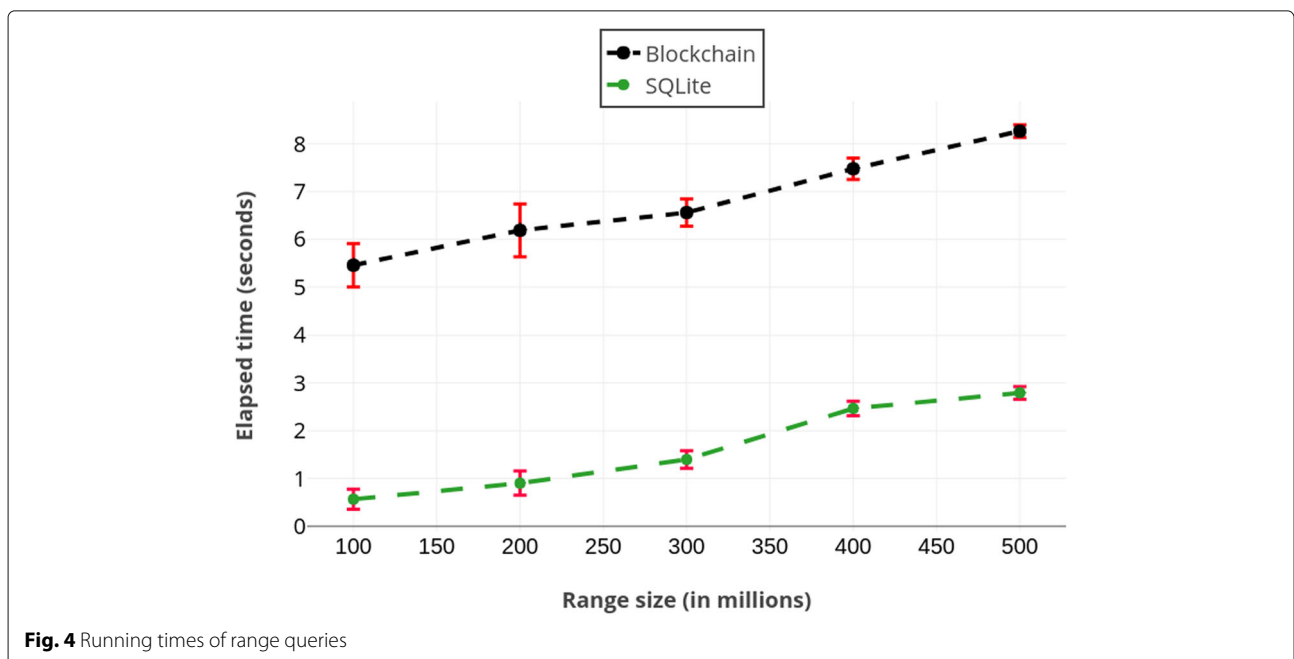
Fig. 5 Processing time required as a function of the number of retrieved records

to **getCount**(.) per field given in the query in addition to retrieving the data. This only adds a negligible overhead.

Finally, we considered the storage requirements. After loading all logs, the size of the blockchain was about 3 GBs per node, whereas the size of the SQLite database was several orders of magnitude smaller at 20 MBs.

Discussion

In this section, we discuss certain limitations and highlight opportunities for improvement of our approach.

First, as mentioned earlier, bucket and buffer size was based on an empirical investigation. We did not conduct extensive studies on these parameters to optimize them. It might be possible to improve query-response times by fine-tuning these parameters.

Second, it is possible to map string fields (i.e., resource and activity) to the integer values to reduce the size of logs. This may improve the loading and query-response times.

Third, we did not consider parallelization. Although Multichain platform imposes some limitations on the parallelization (e.g., concurrently reading different parts of a stream is not possible) workarounds might exist [17].

Further, per the rules of the competition, we were not permitted to modify the blockchain parameters. A straightforward way of improving performance might be to optimize these parameters. For example, the *target-block-time* parameter controls the average number of seconds between two blocks whose default value is 15. It might be possible to decrease loading times by letting the blockchain generates blocks more often.

Finally, we note that Multichain is expected to deploy some new features to support data handling more efficiently in future versions. For instance, in version 2, blockchain stores just the hashes of data [18]. Since

transactions will be shortened, this will likely reduce loading and response times. One can simply compare the hashes of data after fetching them from the accompanying key-value store with the hash on blockchain to ensure immutability in this model.

Conclusion

In this paper, we demonstrated that blockchain technology can overcome inherent limitations on querying and, thus, can be a useful tool for managing data accross multiple sites, particularly in scenarios that require strong immutability and auditability. We showed how bucketization, simple data duplication and batch loading can be utilized to run complex queries efficiently over blockchains that provide support for only simple key-value stores. Particularly, we implemented these notions in the submission to the blockchain track of iDASH 2018 competition that supports efficient conjunctive equality and range queries over blockchains created with Multichain platform. We illustrated that our approach induced reasonable overhead, in terms query-response time, in comparison to a traditional relational database management tool.

Ozdayi *et al. BMC Medical Genomics* 2020, **13**(Suppl 7):82

Page 7 of 7

Author details
[1]Department of Computer Science, Erik Jonsson School of Engineering and Computer Science, The University of Texas at Dallas, 800 W. Campbell Road, Richardson 75080, TX, USA. [2]Department of Biomedical Informatics, Vanderbilt University Medical Center, 2525 West End Avenue, Nashville 37203, TN, USA.

References
1. Ozercan HI, Ileri AM, Ayday E, Alkan C. Realizing the potential of blockchain technologies in genomics. Genome Res. 2018;28(9):1255–63.
2. Choudhury O, Sarker H, Rudolph N, Foreman M, Fay N, Dhuliawala M, Sylla I, Fairoza N, Das AK. Enforcing human subject regulations using blockchain and smart contracts. Blockchain Healthc Today. 2018;1. https://www.blockchainhealthcaretoday.com/index.php/journal/article/view/10.
3. Li P, Nelson SD, Malin BA, Chen Y. DMMS: A decentralized blockchain ledger for the management of medication histories. Blockchain Healthc Today. 2019;2. https://blockchainhealthcaretoday.com/index.php/journal/article/view/38.
4. Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. 2008. https://bitcoin.org/bitcoin.pdf.
5. Wood G. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Proj Yellow Paper. 2014;151:1–32.
6. iDASH Secure Genome Analysis Competition 2018, GMC Medical Genomics. 2019.
7. Jakobsson M, Juels A. In: Preneel B, editor. Proofs of Work and Bread Pudding Protocols(Extended Abstract). Boston, MA: Springer; 1999, pp. 258–72.
8. Castro M, Liskov B. Practical byzantine fault tolerance and proactive recovery. ACM Trans Comput Syst. 2002;20(4):398–461. https://doi.org/10.1145/571637.571640.
9. Multichain: An Open Platform for Building Blockchains. https://www.multichain.com/. Accessed 2 June 2019.
10. MultiChain Private Blockchain — White Paper. https://www.multichain.com/download/MultiChain-White-Paper.pdf. Accessed 2 June 2019.
11. Introducing MultiChain Streams. https://www.multichain.com/blog/2016/09/introducing-multichain-streams/. Accessed 2 June 2019.
12. LevelDB. http://leveldb.org/. Accessed 2 June 2019.
13. How do Streams Work Under the Hood? https://www.multichain.com/qa/9635/how-do-streams-work-under-the-hood. Accessed 2 June 2019.
14. iDASH Privacy and Security Workshop 2018 Competition Tracks. www.humangenomeprivacy.org/2018/competition-tasks.html. Accessed 2 June 2019.
15. Savoir:A Python Wrapper for Multichain Json-RPC API. https://github.com/DXMarkets/Savoir. Accessed 2 June 2019.
16. https://github.com/TinfoilHat0/idash2018BlockchainTrack. Accessed 2 June 2019.
17. Multiprocessing and Streams. https://www.multichain.com/qa/10947/multiprocessing-and-streams. Accessed 2 June 2019.
18. Second MultiChain 2.0 Preview Release. https://www.multichain.com/blog/2018/01/second-multichain-2-0-preview-release/. Accessed 2 June 2019.

Publisher's Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.