Ethereum Query Language

Santiago Bragagnolo
Inria Lille - Nord Europe
Villeneuve D'ascq, France
santiago.bragagnolo@inria.fr

Henrique Rocha
Inria Lille - Nord Europe
Villeneuve D'ascq, France
henrique.rocha@gmail.com

Marcus Denker
Inria Lille - Nord Europe
Villeneuve D'ascq, France
marcus.denker@inria.fr

Stéphane Ducasse
Inria Lille - Nord Europe
Villeneuve D'ascq, France
stephane.ducasse@inria.fr

ABSTRACT

Blockchains store a massive amount of heterogeneous data which will only grow in time. When searching for data on the Ethereum platform, one is required to either access the records (blocks) directly by using a unique identifier, or sequentially search several records to find the desired information. Therefore, we propose the Ethereum Query Language (EQL), a query language that allows users to retrieve information from the blockchain by writing SQL-like queries. The queries provide a rich syntax to specify data elements to search information scattered through several records. We claim that EQL makes it easier to search, acquire, format, and present information from the blockchain.

CCS CONCEPTS

• **Information systems → Query languages**; *Information retrieval query processing*; *Database query processing*; Computing platforms; Digital cash;

KEYWORDS

Ethereum, Blockchain, Query Language, SQL

ACM Reference Format:
Santiago Bragagnolo, Henrique Rocha, Marcus Denker, and Stéphane Ducasse. 2018. Ethereum Query Language. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB'18).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194113.3194123

1 INTRODUCTION

Blockchain was initially used as a distributed ledger allowing monetary interactions without the need of central trusted authority [2, 8, 10–12]. We prefer a more formal definition that blockchain is a globally shared transactional database managed by a peer-to-peer network [7]. Each peer in that network stores a complete copy of the blockchain database. The records are database transactions that append new information to the current state of the blockchain. Transactions are packed into blocks and the blocks are linked in a sequence forming a chain. Thus, the name blockchain.

Ethereum [5] is a popular blockchain platform that is capable of executing Turing-complete programs, called Smart Contracts [10]. Ethereum also has its own cryptocurrency for monetary interactions, called Ether [5].

Ethereum and other blockchains store a massive amount of heterogeneous data. Ethereum is estimated to have approximately 300GB of data [1]. Retrieving information from this massive data is not an easy task. Moreover, the Ethereum platform only allows direct access to its *first-class* data elements, which includes blocks, transactions, accounts, and contracts [5]. Therefore, if we search for a particular information inside a data element, we would need a unique identifier (i.e., either the number or its hash) to access the block containing such information. Another alternative would be to direct access one block and sequentially access its parents to search for a data. Moreover, the information returned by the Ethereum platform when we access its blocks is encoded into a JSON-like structure that we need to interpret to acquire a specific item. Therefore, the Ethereum platform does not provide a semantic way to search for information and neither an easy form to present such information.

For example, let's consider a blockchain application that manages a custom cryptocurrency controlled by a single person (i.e., an owner).[1] The owner needs to look for suspicious and possible malicious behavior in the history of one specific account. Currently, the owner would have two options to gather that account's data: (i) direct access each block with the specific information, which would require for the owner to know beforehand the blocks' hashes or numbers; or (ii) access the most recent block and sequentially search every parent block for any information related to the account. After the owner acquires the information he still needs to extract it from the stored representation and reformat it for a better visualization.

In a classical database, when we need to search for a particular information, we usually write a query to fetch, present, filter, and format such information. Database query languages like SQL provide a rich syntax to describe the data we want to acquire from the database. Since blockchain can be considered a database, it would be better if we could use a similar way to fetch information inside

WETSEB'18, May 27, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5726-5/18/05…$15.00
https://doi.org/10.1145/3194113.3194114

[1] There is an example in the Solidity documentation called Subcurrency [7]. We present a modified version of this same example in Section 2.2.

the blocks. In this paper, we propose the Ethereum Query Language (EQL), a query language that enables its users to acquire information in the blockchain by writing SQL-like queries. Our goal is to provide an easier way to fetch, format, and present information extracted from the blockchain database.

The remainder of this paper is organized as follows. Section 2 present basic concepts on Structured Query Language and Ethereum smart contracts. Section 3 describes the problems and challenges to search and retrieve information from the blockchain. Section 4 present EQL along with its syntax, examples, internal structure (indexes), and limitations. In Section 5, we show a preliminary evaluation describing the performance of EQL queries. In Section 6, we present related work in blockchain. Finally, Section 7 concludes the paper and outlines future work ideas.

2 BACKGROUND

In this section, we present basic concepts on Structured Query Language (Section 2.1), and Ethereum Smart Contracts (Section 2.2).

2.1 Structured Query Language

Structure Query Language (SQL) is considered the standard relational database language [4, 14]. Even though SQL is well established as a query language, it can also perform other operations such as manipulating data or specifying integrity constraints. The commercial success of relational databases is greatly due to SQL being a well-adopted standard [4].

In SQL, the "SELECT" statement is used to query data (i.e., retrieve information) [14]. Basically, a "SELECT" query consists of the clauses: *select*, *from*, *where*, *group by*, and *order by*. The *select* clause specifies what to show from the query results; the *from* clauses defines which tables to gather the data; the *where* clause defines a condition the information must satisfy to be returned as a result; the *group by* clause groups the gathered data based on a condition; and finally, the *order by* clause specifies how the query orders its results.

We based our proposed query language on SQL, due to its popularity and broad user base. More specifically, our queries use an adapted form of the "SELECT" statement.

2.2 Ethereum Smart Contracts

Smart contracts are programs that are executed in the blockchain platform [8, 10]. A smart contract is like a class; it contains attributes and functions, and it can use inheritance on other contracts [7]. The correct execution of smart contracts, as well as their resulting states, is ensured by a consensus protocol [10]. Solidity [7] is the primary language to specify smart contracts in the Ethereum blockchain. Since the Ethereum platform is Turing-complete, contracts can define any rules by using the Solidity language.

Solidity is a high-level language based on JavaScript, C++, and Phyton [7]. The contracts written in Solidity are compiled into a specific bytecode to run on the Ethereum Virtual Machine (EVM). A compiled contract can be deployed into the Ethereum blockchain by executing a special transaction that allocates an address to it [2]. This address is a 160-bit unique identifier that references the contract. Once deployed, a contract can be executed remotely by client applications.

Listing 1 shows a Solidity contract example (adapted from the subcurrency example on the Solidity documentation [7]) that manages a simple cryptocurrency controlled by a single person. The contract stores the balances of its accounts using a "mapping" (line 7) that works like a hash table by mapping blockchain addresses to unsigned integers. The constructor (lines 13-15) stores the address of who deployed the contract (i.e., the owner of this contract instance). Only the owner can call the function "mint" (lines 18-21), which creates new coins for a specific account. The function "send" (lines 24-30) allows the caller of this function to transfer his coins to another account, provided that the caller has sufficient funds for such operation.

Listing 1: Solidity Simple Cryptocurrency Contract, Adapted Example

```
1  pragma solidity ^0.4.20;
2
3  contract CustomCoin {
4     address private owner;
5
6     /* The keyword "public" makes it readable
          from outside */
7     mapping (address => uint) public balances;
8
9     /* Events allow light clients to react on
          changes efficiently */
10    event Sent(address from, address to, uint
          amount);
11
12    /* Constructor: only executed when the
          contract is deployed */
13    function CustomCoin() public {
14       owner = msg.sender;
15    }
16
17    /* Creates new coins */
18    function mint(address receiver, uint
          amount) public {
19       if (msg.sender != owner) return;
20       balances[receiver] += amount;
21    }
22
23    /* Allows the caller to send coins to
          another user/account */
24    function send(address receiver, uint
          amount) public {
25       if (balances[msg.sender] < amount)
26          revert(); // abort transaction
27       balances[msg.sender] -= amount;
28       balances[receiver] += amount;
29       Sent(msg.sender, receiver, amount);
30    }
31
32 } //end of contract
```

3 PROBLEM

In this section, we describe in more detail the problems and challenges when trying to acquire data from a blockchain. Although our research is focused on Ethereum, such problems are also present on other blockchain platforms as well.

3.1 Massive Data

Blockchain databases already possess a massive amount of data. For example, Ethereum is estimated to have approximately 300GB of data [1]. Moreover, Ethereum processed, on average, 876K transactions per day in December of 2017 (Figure 1).

Since the data in the blockchain cannot be deleted, the number of recorded transactions will only grow in time. In this context, older information could get overwhelmed by new transactions. Indeed, the common expression "looking for a needle in a haystack" could be updated to "looking for a hash in a blockchain".

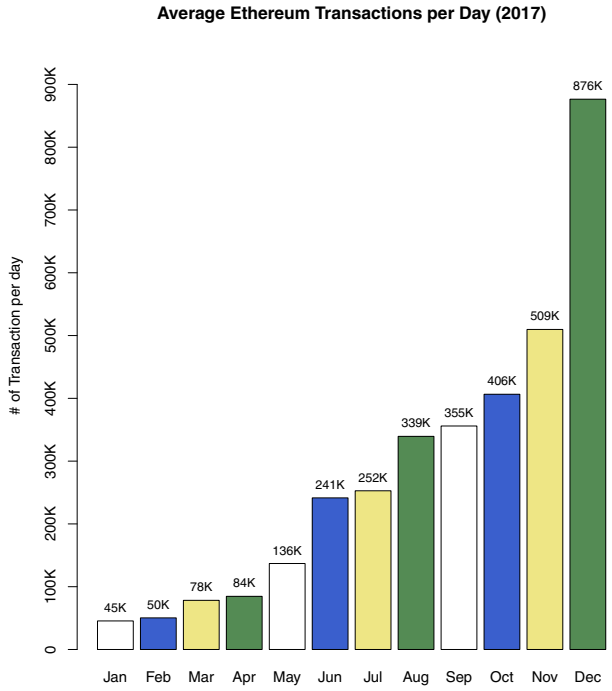

Figure 1: Average number of Ethereum transactions per day in 2017. Source: etherscan.io

3.2 Heterogeneous Data

Blockchain not only stores a great amount of data but also manages a mixture of *first-class* elements such as transactions, blocks, accounts, and smart contracts. Data elements are represented as transactions, which are bundled into a storage element (blocks). Smart contracts are programming elements that manage data and function, and accounts are elements to represent users.

All of the *first-class* elements (blocks, transactions, etc.) are different but interrelated by hashes. Even though Ethereum allows access to any of its *first-class* elements, the heterogeneity of the elements (each one with different meaning and high-level representation) complicates the acquisition of information.

3.3 Direct Access

In general, blockchains only allow direct access to its elements by using a unique identifier. This identifier is a hash number that is generated for every *first-class* element stored in the blockchain [3].

The Ethereum platform, in particular, can also use a unique number (related to the order of the element) to access blocks and transactions [6]. In this scenario, a user only needs to provide a hash to the API to invoke the specific fetch method to acquire the desired element (e.g., block, transaction, contract). From the API point-of-view, this is a good solution because it is fast, simple, and possesses very little overhead. However, this direct access by hash can lead to the following issues when we consider the user's point-of-view:

- **Hash Storage.** Since the user needs the hash to fetch the information, he/she is also required to store the hash, someway, if he/she wants to acquire the same information later. Otherwise, the user will lose the key to access the desired information in the future. This problem increases when the user needs to "remember" multiple hashes for later use. In such case, a secondary private database is required to store all hashes with related meta-data to retrieve the information from the blockchain in the future.

- **Sequential Access.** If the user loses the hash, he/she can still find the relevant data by performing a sequential access in the blockchain. In Ethereum, it is possible to start at the most recent block and sequentially access the blocks parents. Since Ethereum blocks can also be fetched by their number (and not only the hash), a user could also access a sequence of blocks by just incrementing or decrementing the block number. Sequential access is also useful when a user needs to search for information scattered around multiple blocks. Searching for information in a sequential way is not efficient. Moreover, it is noteworthy that each block access is a remote procedure call (RPC), which may hinder the performance of any sequential search.

3.4 Data Opaqueness

In order to give users flexibility, Ethereum stores arbitrary information (e.g., contracts, transactions) with a generic representation. Therefore, the stored data is opaque, since there is no meta-data describing the information and neither a simple way to know what was recorded.

This opaqueness is useful and even necessary from the Ethereum standpoint because it reduces the data representation size and allows the storage of arbitrary structures and behaviors. On the other hand, the opaqueness overcomplicates searching for information, since a user needs to access generic representations without any knowledge of its content.

4 ETHEREUM QUERY LANGUAGE

In this section, we describe the Ethereum Query Language (EQL) version 0.8, a query language designed to acquire information from the blockchain.[2] EQL is publicly available on GitHub.[3]

We highlight the following benefits of using EQL to fetch information from blockchain: (i) describe structural and semantic filters to query for information; (ii) reformat and transform the acquired data; (iii) order the query results; and (iv) limit the amount of results returned.

4.1 Syntax

EQL syntax is based on the SQL language. The idea is to allow users to write queries as close to SQL as possible to facilitate the adoption of EQL since SQL is a very popular language [4, 14].

Listing 2 shows the main elements of the EQL syntax. The syntax is described using EBNF (Extended Backus-Naur Form), we did not format the terminals *identifier* and *number* in double-quotes to highlight that they are not literals. We omitted the Expression rule to not clutter the specification, but it follows a similar structure of SQL expressions.

Listing 2: EQL grammar in EBNF format

```
1  <SelectStatement> ::= <SelectClause>
2       <FromClause> [<WhereClause>]
3       [<OrderByClause>] [<LimitClause>]
4  <SelectClause> ::= "select" <Expression>
5       {"," <Expression> }
6  <FromClause> ::= "from" <SourceBind>
7       {"," <SourceBind> }
8  <WhereClause> ::= "where" <Expression>
9  <OrderByClause> ::= "order" "by" <Expression>
10      [ "asc" | "desc" ]
11 <LimitClause> ::= "limit" number
12 <SourceBind> ::= identifier "as" identifier
13 <Expression> ::= ...
```

As we can see from Listing 2, the syntax for EQL queries is very similar to the "Select" statement from the SQL language. The EQL "Select" statement consists of the following clauses: *select*, *from*, *where*, *order by*, and *limit*. Only the *select* and *from* clauses are required, the others are optional clauses. We also like to highlight that EQL similarly to SQL is also case-insensitive. Unlike SQL, the EQL "Select" statement does not have a *group by* clause.[4]

4.2 Examples

We present a few examples of queries written in EQL. In the current version of EQL, we are able to query blocks and transactions. Querying for information inside contracts (i.e., contract attributes or functions) is not yet supported by our implementation.

Listing 3 shows an example that query over blocks. In this query, the *from* clause (line 2) gathers data from all blocks (ethereum.blocks) using the alias "block" to reference them. The *select* clause (line 1) specifies the query results will show the block's parent number, its own hash, timestamp, number, and amount of transactions inside the block. The *where* clause (line 3) filters the blocks to fetch only those with a timestamp between 2016-01-01 and today, the block must also have more than ten transactions packed into it. The *order by* clause (line 4) arrange the results in descending order by how many transactions are inside each block. Finally, we use the *limit* clause (line 5) restricts our results to show only the first 100 blocks.

Listing 3: EQL Block Query Example

```
1 SELECT block.parent.number, block.hash,
      block.timestamp, block.number,
      block.amountOfTransactions
2 FROM ethereum.blocks AS block
3 WHERE block.timestamp BETWEEN date('2016-01-01')
      AND now() AND block.transactions.size >10
4 ORDER BY block.transactions.size
5 LIMIT 100;
```

Figure 2 shows the top-20 results formatted for a better visualization from the query (Listing 3). Figure 3 shows a raw representation of the same top-20 results.

block.parent().number	block.hash	block.timestamp	block.number	block.amo
779523	8969218	2016-01-01T05:53:07+01:00	779524	11
779112	104199555	2016-01-01T03:51:04+01:00	779113	11
779505	226022799	2016-01-01T05:48:01+01:00	779506	11
782753	96179627	2016-01-01T21:48:43+01:00	782754	11
780348	239565626	2016-01-01T10:10:40+01:00	780349	11
779113	101468812	2016-01-01T03:51:20+01:00	779114	11
782218	39336575	2016-01-01T19:18:16+01:00	782219	11
778657	129555478	2016-01-01T01:48:18+01:00	778658	11
779281	249003683	2016-01-01T04:38:40+01:00	779282	11
780351	103525535	2016-01-01T10:11:05+01:00	780352	11
781319	259274786	2016-01-01T14:56:03+01:00	781320	11
779829	186089684	2016-01-01T07:21:49+01:00	779830	12
781563	192796997	2016-01-01T16:11:22+01:00	781564	12
778286	148395039	2016-01-01T00:05:16+01:00	778287	12
781498	245360810	2016-01-01T10:11:23+01:00	781499	12
780917	167190109	2016-01-01T15:52:34+01:00	780918	12
780917	180827207	2016-01-01T13:04:17+01:00	780918	12
781459	255862972	2016-01-01T15:40:24+01:00	781460	12
780687	118747668	2016-01-01T11:48:46+01:00	780688	12
778514	20851823	2016-01-01T01:07:21+01:00	778515	13

Figure 2: EQL Block Query Example Results (Formatted Representation)

Listing 4 shows a query example that searches for transactions. The *from* clause (line 2) determinates that the data will come from every Ethereum transaction (ethereum.transactions) using the alias "transaction". The *select* clause (line 1) will show as result the transaction timestamp, the hash from the account that initiated the transaction, the hash to the account that received this transaction, and the amount moved by this transaction multiplied by 0.1. We used the multiplication to show that EQL can transform the resulting data on its *select* clause by using simple expressions. The *where* clause (line 3) stipulates that only the transactions with a timestamp greater than 2018-01-01; the clause also restricts the

[2] Currently, EQL works only on the Ethereum blockchain platform, even though, we are working to extend its reach to other platforms as well. Therefore, when we refer to "blockchain" in this section, we are in fact referring specifically to the Ethereum blockchain platform.
[3] https://github.com/sbragagnolo/UQLL, verified 2018-03-08.
[4] EQL is a project under development. Although we have plans to support a *group by* clause in the future, the current version does not allow the usage of such clause.

Index	Item
1	an OrderedCollection [5 items] ('block..parent.()'->'Block- 779523 - 4 transaction(s) ' 'block.hash.'->8969218 'block.timestamp.'->2016-01-01T05:53:07+01:00 'block.number.'->779524 'block..amountOfTransactions.()'->11)
2	an OrderedCollection [5 items] ('block..parent.()'->'Block- 779112 - 1 transaction(s) ' 'block.hash.'->104199555 'block.timestamp.'->2016-01-01T03:51:04+01:00 'block.number.'->779113 'block..amountOfTransactions.()'->11)
3	an OrderedCollection [5 items] ('block..parent.()'->'Block- 779505 - 0 transaction(s) ' 'block.hash.'->226022799 'block.timestamp.'->2016-01-01T05:48:01+01:00 'block.number.'->779506 'block..amountOfTransactions.()'->11)
4	an OrderedCollection [5 items] ('block..parent.()'->'Block- 782753 - 0 transaction(s) ' 'block.hash.'->96179627 'block.timestamp.'->2016-01-01T21:48:43+01:00 'block.number.'->782754 'block..amountOfTransactions.()'->11)
5	an OrderedCollection [5 items] ('block..parent.()'->'Block- 780348 - 8 transaction(s) ' 'block.hash.'->239565626 'block.timestamp.'->2016-01-01T10:10:40+01:00 'block.number.'->780349 'block..amountOfTransactions.()'->11)
6	an OrderedCollection [5 items] ('block..parent.()'->'Block- 779113 - 11 transaction(s) ' 'block.hash.'->101468812 'block.timestamp.'->2016-01-01T03:51:20+01:00 'block.number.'->779114 'block..amountOfTransactions.()'->11
7	an OrderedCollection [5 items] ('block..parent.()'->'Block- 782218 - 0 transaction(s) ' 'block.hash.'->39336575 'block.timestamp.'->2016-01-01T19:18:16+01:00 'block.number.'->782219 'block..amountOfTransactions.()'->11)
8	an OrderedCollection [5 items] ('block..parent.()'->'Block- 778657 - 2 transaction(s) ' 'block.hash.'->129555478 'block.timestamp.'->2016-01-01T01:48:18+01:00 'block.number.'->778658 'block..amountOfTransactions.()'->11)
9	an OrderedCollection [5 items] ('block..parent.()'->'Block- 779281 - 3 transaction(s) ' 'block.hash.'->249003683 'block.timestamp.'->2016-01-01T04:38:40+01:00 'block.number.'->779282 'block..amountOfTransactions.()'->11)
10	an OrderedCollection [5 items] ('block..parent.()'->'Block- 780351 - 8 transaction(s) ' 'block.hash.'->103525535 'block.timestamp.'->2016-01-01T10:11:05+01:00 'block.number.'->780352 'block..amountOfTransactions.()'->11)
11	an OrderedCollection [5 items] ('block..parent.()'->'Block- 781319 - 0 transaction(s) ' 'block.hash.'->259274786 'block.timestamp.'->2016-01-01T14:56:03+01:00 'block.number.'->781320 'block..amountOfTransactions.()'->11)
12	an OrderedCollection [5 items] ('block..parent.()'->'Block- 779829 - 1 transaction(s) ' 'block.hash.'->186089684 'block.timestamp.'->2016-01-01T07:21:49+01:00 'block.number.'->779830 'block..amountOfTransactions.()'->12)
13	an OrderedCollection [5 items] ('block..parent.()'->'Block- 781563 - 1 transaction(s) ' 'block.hash.'->192796997 'block.timestamp.'->2016-01-01T16:11:22+01:00 'block.number.'->781564 'block..amountOfTransactions.()'->12)
14	an OrderedCollection [5 items] ('block..parent.()'->'Block- 778286 - 0 transaction(s) ' 'block.hash.'->148395039 'block.timestamp.'->2016-01-01T00:05:16+01:00 'block.number.'->778287 'block..amountOfTransactions.()'->12)
15	an OrderedCollection [5 items] ('block..parent.()'->'Block- 780353 - 3 transaction(s) ' 'block.hash.'->245360810 'block.timestamp.'->2016-01-01T10:11:23+01:00 'block.number.'->780354 'block..amountOfTransactions.()'->12)
16	an OrderedCollection [5 items] ('block..parent.()'->'Block- 781498 - 1 transaction(s) ' 'block.hash.'->167190109 'block.timestamp.'->2016-01-01T15:52:34+01:00 'block.number.'->781499 'block..amountOfTransactions.()'->12)
17	an OrderedCollection [5 items] ('block..parent.()'->'Block- 780917 - 3 transaction(s) ' 'block.hash.'->180827207 'block.timestamp.'->2016-01-01T13:04:17+01:00 'block.number.'->780918 'block..amountOfTransactions.()'->12)
18	an OrderedCollection [5 items] ('block..parent.()'->'Block- 781459 - 1 transaction(s) ' 'block.hash.'->255862972 'block.timestamp.'->2016-01-01T15:40:24+01:00 'block.number.'->781460 'block..amountOfTransactions.()'->12)
19	an OrderedCollection [5 items] ('block..parent.()'->'Block- 780687 - 5 transaction(s) ' 'block.hash.'->118747668 'block.timestamp.'->2016-01-01T11:48:46+01:00 'block.number.'->780688 'block..amountOfTransactions.()'->12)
20	an OrderedCollection [5 items] ('block..parent.()'->'Block- 778514 - 0 transaction(s) ' 'block.hash.'->20851823 'block.timestamp.'->2016-01-01T01:07:21+01:00 'block.number.'->778515 'block..amountOfTransactions.()'->13)

Figure 3: EQL Block Query Example Results (Raw Representation)

results to the accounts with a balance greater than 100 ether that initiated the transaction. The *order by* clause (line 4) arrange the results in descending order by the balance on the account receiving the transaction. Finally, the *limit* clause (line 5) limits the results to fetch only the first ten transactions.

Listing 4: EQL Transaction Query Example

```
1 SELECT transaction.timestamp,
      transaction.from.hash,
      transaction.to.hash,
      transaction.amount*0.1
2 FROM ethereum.transactions AS transaction
3 WHERE transaction.timestamp > date('2018-01-01')
      AND transaction.from.balance()>100 ether
4 ORDER BY transaction.to.balance() DESC
5 LIMIT 10;
```

Figure 4 shows the results formatted for a better visualization of the query described in Listing 4.

transaction.timestamp	transaction.from.hash	transaction.to.hash	transaction.amount*0.1
2018-01-01T04:03:57+01:00	0xca35b7d915458…	0x627306090abaB3…	1.3
2018-01-01T21:01:07+01:00	0x14723a09acff6d…	0x4b0897b0513fdc7…	0.2
2018-01-01T16:00:05+01:00	0x4b0897b0513fdc…	0xdd870fa1b7c4700f…	0.0
2018-01-01T01:03:47+01:00	0xf17f52151EbEF6…	0xC5fdf4076b8F3A5…	0.5
2018-01-01T03:56:14+01:00	0x627306090abaB…	0x821aEa9a577a9b…	0.0
2018-01-01T16:19:18+01:00	0x0d1d4e623D10F…	0x2932b7A2355D6fe…	0.0
2018-01-01T10:09:37+01:00	0x2191eF87E3923…	0x0F4F2Ac550A1b4…	0.3
2018-01-01T19:43:32+01:00	0x6330A553Fc937…	0x6330A553Fc9376…	0.0
2018-01-01T07:49:21+01:00	0x5AEDA56215b1…	0xE44c4cf797505AF…	0.1
2018-01-01T13:01:07+01:00	0xF014343BDFFb…	0x0E79EDbD6A727…	0.1

Figure 4: EQL Transaction Query Example Results

4.3 Collections and Objects

In EQL, a collection is a semantic representation of queried data. More specifically, collections are used in the *from* clause as they represent the data we are searching. The EQL language have four predefined collections: *ethereum.blocks*, *ethereum.transactions*, *ethereum.accounts*, and *ethereum.contracts*. Each one of those collections represents all first-class elements from a particular type (blocks, transactions, accounts, or contracts). It is possible to create custom collections from a subset of another one. EQL also supports "views" similar to SQL.

When we retrieve information from a collection of blocks, the result will be presented as block objects. As we shown in Listing 3, Ethereum blocks have attributes related to their storage structure. Blocks queried by EQL have the following attributes:

- **number:** integer, the block's number.
- **hash:** hash (binary 32 bytes), the block's hash.
- **parentHash:** hash, the parent's block hash.
- **parent:** block, the parent's block (EQL automatically fetches the parent information using the hash).
- **nonce:** integer, the proof-of-work nonce value.
- **timestamp:** timestamp, the unix timestamp when the block was created.
- **size:** integer, the size of the block in bytes.
- **miner:** address (binary 20 bytes), the account who mined this block.
- **difficulty:** integer, the proof-of-work difficulty for this block.
- **totalDifficulty:** integer, the total difficulty of the chain until this block.
- **gasLimit:** integer, the maximum gas allowed in the block.
- **gasUsed:** integer, the total gas used by all transactions in this block.
- **extraData:** binary variable size, a general field to contain extra data for the block.
- **transactionsRoot:** hash, the first transaction on this block.
- **transactionsHashes:** set of hashes, the hashes from the transactions on this block.

- **transactions:** set of transactions, the transactions in this block.
- **amountOfTransactions:** integer, the number of transactions in this block.
- **uncleHashes:** set of hashes, the uncles[5] hashes.
- **uncles:** set of blocks, the uncles' blocks.

In the above attributes, the ones that are a set (e.g., transactions) can be called as a function passing the index as parameter. In this case, the function returns only the element at the specified index (or null if the index is out of bounds). For example, *block.transactions(3)* would return the third transaction in the current block.

Transaction objects have a different representation. It is noteworthy to mention that while a block contains many transactions, a transaction object is contained inside a single block. In EQL, transaction objects have the following attributes:

- **hash:** hash, the transaction's hash.
- **nonce:** integer, the nonce of the sender account (i.e., for accounts the nonce counts the number of transactions it created; it is a form to avoid *double spending*).
- **blockHash:** hash, the block's hash.
- **blockNumber:** integer, the block's number.
- **block:** block, the block that contains this transaction.
- **transactionIndex:** integer, the transaction's index position in the block.
- **fromHash:** address, the address from the account that initiated this transaction (i.e., the sender).
- **from:** account, the sender's account.
- **toHash:** address, the address of the receiver of this transaction.
- **to:** account, the account of the receiver of this transaction.
- **value:** integer, the amount transfered in Wei.
- **gasPrice:** integer, the gas price set by the sender in Wei.
- **gas:** integer, gas consumed by this transaction.
- **input:** binary variable size, the data send along with the transaction.
- **timestamp:** timestamp, the unix timestamp when the transaction was created.

Accounts are a simple object that have the following attributes in EQL:

- **address:** address, the account's address.
- **name:** string, the account's name.
- **balance:** integer, the amount of cryptocurrency (in Wei) on this account.

EQL have limitations when dealing with contracts (described in more detail at Section 4.5). A contract object is a special type of an account. In the current version, contracts have the following attributes:

- **address:** address, the contract's address.
- **name:** string, the contract's name.
- **balance:** integer, the amount of cryptocurrency (in Wei) on this contract.
- **bytecode:** binary variable size, the contract code.

[5] In Ethereum, uncle blocks are valid blocks that were mined but rejected. The uncles are orphaned blocks that contribute to the security of the main chain Uncle blocks do not contain transactions.

4.4 Indexes

Internally, our implementation of EQL relies on indexes to increase its performance when querying data. Since each access to the blockchain to fetch data is a remote call, any form of optimization can improve the time required to acquire and present data.

An index is a summarization of data stored into a structure that improves the performance of retrieval operations. The basic idea is to allow a more efficient search into the database. In our particular case, we index blockchain data (e.g., blocks, transactions) with its related hash to speed up fetching data.

For EQL, we implemented a Binary Search Tree (BST) to serve as the index structure. The BST employs a two-dimensional array where the first dimension of each entry is used for storing the property value, and the second dimension is used for storing a set of hashes to the elements that correspond to this specific value. We chose this implementation because of its simplicity for selecting an interval of indexes in any comparison operation, such as "greater than", "lesser than". We acknowledge that BST has a high storage requirement. However, we wanted a simple and fast solution to our first implementation of EQL. Moreover, EQL builds its indexes automatically, without the need for user interaction.

4.5 Limitation

The current version of the EQL implementation (version 0.8) has some limitations.

First, we are not able to search inside contract attributes when querying. We are still able to query blocks and transactions that were created to record a smart contract attribute change. However, we would need to use the information on blocks and transactions to find information related to contracts. We are working to circumvent this limitation and offer contract querying on the next release of EQL.

A second limitation is also related to smart contracts. We are unable to use smart contract functions on EQL expressions. Since, smart contracts store not only data but also functional behavior, it might be necessary to execute a contract function to properly query for contract information. We acknowledge that allowing users to call any contract function could lead to performance bottlenecks, security issues, and re-triggering the contract. Therefore, we plan to allow "read-only" type of smart contract functions in EQL expressions.

Another limitation is that we placed a maximum upper bound on the number of results returned by EQL. Even though, the *limit* clause is optional, our implementation will always return at maximum 1000 results because of memory constraints. Although we have no plans to allow unlimited results being returned, we are working to use an "offset" definition on the query so that a user can retrieve bigger results in installments.

The lack of *group by* clause in EQL is also another limitation. This hinders the expressiveness of the EQL language as we cannot perform aggregation queries. We are working to add *group by* and aggregate functions in a future version of EQL.

5 PRELIMINARY EVALUATION

As a preliminary form of evaluation, we tested the performance of EQL on retrieving information. We compared EQL against using a driver[6] to direct access the information inside the blockchain. We are using the direct access as a baseline for comparison since it is not possible for EQL (or any approach) to perform better than direct access.

For the evaluation, we employed direct access to fetch 100 randomly selected blocks. Then, we used the EQL query shown in Listing 3 (Section 4.2) to fetch 100 blocks. First, we executed both retrieval operations with an empty cache. Second, we repeated the same operations to verify how both would perform when the searched information is already in the cache. It is noteworthy that the driver we used for direct access maintains a cache to improve its performance, and it is not a standard feature of direct accessing Ethereum.

Table 1 shows the performance comparison between both approaches without and with information on the cache. Table 1 measures the time (average, standard deviation, mode, maximum, and minimum) to fetch one block in milliseconds. We were not able to calculate the standard deviation for the cached execution because the numbers were too small.

Table 1: Performance Tests to Fetch Blocks measured in Milliseconds

	Avg	St.Dev.	Mode	Max	Min
Direct without cache	5.88	12.36	1.60	127.2	0.30
EQL without cache	159.69	13.65	107.37	225.64	88.05
Direct with cache	0.04	–	0.03	0.05	0.03
EQL with cache	0.04	–	0.03	0.05	0.03

As we can see from Table 1, when both approaches do not use cached information, direct access is much faster than EQL. This is expected because direct access is fetching blocks directly using their hashes, while EQL is searching for information inside the blocks (i.e., the timestamp and number of transactions as defined in the query in Listing 3) to see which ones will return as the query result. When the information being searched is already cached then both approaches reach similar results.

6 RELATED WORK

Porru et al. [13] acknowledge the need to create and adapt tools and techniques for blockchain-oriented software (BOS). The authors define the term BOS as a software that interacts with blockchain. Basically, they discuss the software engineering issues when dealing with BOS. The authors first present challenges on the state-of-art BOS; second they analyse 1184 GitHub projects using blockchain; and finally, they propose ideas for research on BOS. Even though the authors present very interesting research possibilities, they did not foresee research in query languages for blockchain.

Bartoletti et al. [1] propose a framework for blockchain analytics coded as a Scala library. Their framework works on both Ethereum and BitCoin platform and it employs a general-purpose abstraction layer to promote reuse. One great feature of the authors' framework is the ability to integrate data from other sources besides blockchain, such as a NoSQL database. The authors contrast their framework features against five other tools, but they do not conduct a performance comparison. This work is interesting because the authors combine blockchain data with a secondary database. We have plans to incorporate in EQL the capabilities to join blockchain data and a secondary database on the query.

Kalodner et al. [9] implement an open-source blockchain analysis platform, called BlockSci. Their platform comes with many tools and features to better help with the analysis. For instance, the authors' claim it is 15 to 600 times faster than other tools. They support the following blockchains: BitCoin, LiteCoin, Namecoin, and Zcash. Similarly to EQL, BlockSci also uses indexes in its implementation. Unlike EQL, BlockSci indexes are stored in SQLite database. Moreover, the authors claimed that many analyses do not require indexes at all. This contrasts with EQL, which indexes are essential to speed up the retrieval performance. Although BlockSci provides a better visualization and navigation on blockchain data, it does not provides an easy way to search or filter information.

There are many researches towards security on blockchains and smart contracts. Luu et al. [10] analyses the security flaws in Ethereum smart contracts. The authors identified security problems and possible ways of attack by exploiting smart contracts. Then, the authors formalize solutions for the identified security issues. They also implement a tool that checks for problematic code on a smart contract. Their tool processed over 19K Ethereum contracts and found unsecured coding practices on approximately 8K of them.

Juels et al. [8] investigate what they called, criminal smart contracts (CSC). CSC is a contract that facilitates illicit activities and rewards those interacting with it. The authors create their own CSC as a proof of concept. They also propose countermeasures against CSC which could help the blockchain community prevent CSC proliferation. Some of the described criminal activities could be more easily caught by querying blockchain data.

7 CONCLUSION

The amount of data stored in blockchain is massive and that data is also heterogeneous and opaque. Moreover, the Ethereum platform only allows direct or sequential access to its blocks. In this context, searching for information inside the blockchain is a challenging task because we must sequentially access a huge amount of opaque data. To help in this challenge, we proposed EQL, a query language that allows users to retrieve information by writing SQL-like queries. Our implementation of EQL automates the task of sequentially searching into generic opaque structures, and provides an easier way to specify the information we want to acquire in a higher abstraction level. Although the current implementation (version 0.8) still shows limitations when dealing with smart contract information, we are able to fetch records related to contracts but only in the form of blocks or transactions.

We tested the performance of EQL against direct accessing blocks for a baseline comparison. As expected, EQL is much slower when the information is not cached (average of 159 milliseconds to fetch a block) than direct access (average 5.88 milliseconds to fetch a block). However, when the information being searched is already

[6] The driver used for direct accessing the blockchain is also publicly available on GitHub at https://github.com/sbragagnolo/Fog (verified 2018-03-08).

cached, then EQL reach similar results to direct access (average of 0.04 milliseconds to fetch a block). Even though using EQL can take longer than direct access, the goal is to help users who cannot directly access their information and need to search for it in the blockchain. Since EQL automates the searching task, it simplifies information retrieval for Ethereum.

For future work, our first priority is to tackle the limitations of the implemented version of EQL, especially to support smart contract querying. Moreover, we plan to add support to *group by* clauses and aggregate functions as well. We are also planning to create a tool to write queries and show results based on SQL database tools (e.g., MySQL Workbench). Another future work idea is to perform a more in-depth performance evaluation, and also a user feedback evaluation on EQL. Moreover, we plan to allow EQL to merge blockchain data and data from other sources (e.g., NoSQL database, relational database) when presenting results.

ACKNOWLEDGMENT

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. This research was also supported by UTOCAT.

REFERENCES

[1] Massimo Bartoletti, Stefano Lande, Livio Pompianu, and Andrea Bracciali. 2017. A General Framework for Blockchain Analytics. In *1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL '17)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3152824.3152831

[2] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611

[3] BitCoin.org. 2018. Bitcoin Developer Reference. Bitcoin Core APIs. (2018). https://bitcoin.org/en/developer-reference#opcodes Bitcoin Project 2009-2018.

[4] Ramez Elmasri and Shamkant Navathe. 2010. *Fundamentals of Database Systems* (6th ed.). Addison-Wesley Publishing Company, USA.

[5] Ethereum Foundation. 2014. Ethereum's white paper. (2014). https://en.wikibooks.org/wiki/LaTeX/Bibliography_Management

[6] Ethereum Foundation. 2018. JSON RPC. (2018). https://github.com/ethereum/wiki/wiki/JSON-RPC

[7] Ethereum Foundation. 2018. Solidity Documentation Release 0.4.20. (2018). https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf

[8] Ari Juels, Ahmed Kosba, and Elaine Shi. 2016. The Ring of Gyges: Investigating the Future of Criminal Smart Contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 283–295. https://doi.org/10.1145/2976749.2978362

[9] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan. 2017. BlockSci: Design and applications of a blockchain analysis platform. *ArXiv e-prints* (Sept. 2017). arXiv:cs.CR/1709.02489

[10] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS'2016 (ACM Conference on Computer and Communications Security)*.

[11] Satoshi Nakamoto. 2009. BitCoin: A peer-to-peer electronic cash system. (2009). bitcoin.org

[12] Russell O'Connor. 2017. Simplicity: A New Language for Blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security (PLAS '17)*. ACM, New York, NY, USA, 107–120. https://doi.org/10.1145/3139337.3139340

[13] Simone Porru, Andrea Pinna, Michele Marchesi, and Roberto Tonelli. 2017. Blockchain-oriented Software Engineering: Challenges and New Directions. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 169–171. https://doi.org/10.1109/ICSE-C.2017.142

[14] Abraham Silberschatz, Henry Korth, and S. Sudarshan. 2011. *Database Systems Concepts* (6 ed.). McGraw-Hill, Inc., New York, NY, USA.