UNIVERSITÀ DEGLI STUDI DI CAGLIARI
FACOLTÀ DI SCIENZE
Corso di Laurea Magistrale in Informatica

A tool for Ethereum blockchain analysis

Relatore
Prof. Bartoletti Massimo

Candidato
Daniele Stefano Ferru
Matricola: 65036

Anno accademico 2016-2017

Contents

Chapter 1

Introduction

The last few years have witnessed a steady growth in interest on blockchains, driven by the success of Bitcoin and, more recently, of Ethereum [29]. This has fostered the research on several aspects of blockchain technologies, from their theoretical foundations both cryptographic [9, 10] and economic [17, 24] to their security and privacy [1, 6, 11, 14, 18].

This interest, in the most recent years, gave rise to a fusion of the crowdfunding concept and the cryptocurrency concept, giving rise to the concept of Initial Coin Offering, or ICO. First ICOs were launched in order to collect funds to create new cryptocurrencies, but current ICOs are used for any purpose. Generally, tokens are selled in order to collect money, with the token and his behaviour are defined nowadays using Ethereum Smart Contracts. Ethereum itself raised money with a token sale in 2014, raising 3,700 BTC in its first 12 hours, equal to approximately 2.3 million US dollars at the time. In May 2017, the ICO for a new web browser called Brave generated about 35 million US dollars in under 30 seconds.

Ethereum is (as of August 2017) the leading blockchain platform for ICOs with more than 50% market share. These tokens are called ERC20 (as described in chapter 2.2).

Among the research topics emerging from blockchain technologies, one that has received major interest is the analysis of the data stored in blockchains. Indeed, the two main blockchains contain several gigabytes of data (130GB for Bitcoin, 300GB for Ethereum), that only in part are related to currency transfers. Developing analytics on these data allows us to obtain several insights, as well as economic indicators that help to predict market trends.

Many works on data analytics have been recently published, addressing anonymity issues, e.g. by de-anonymising users [18, 22, 23], clustering transactions [26, 13], or evaluating anonymising services [20]. Other analyses have addressed criminal activities, e.g. by studying denial-of-service attacks [3, 27], ransomware [15], and various financial frauds [21]. Many statistics on Bitcoin and Ethereum exist, measuring e.g. economic indicators [16], transaction fees [19], the usage of metadata [5].

A common trait of these works is that they create views of the blockchain which contain all the data needed for the goals of the analysis. In many cases, this requires to combine data within the blockchain with data from the outside. These data are retrieved from a variety of sources, e.g. blockchain explorers, wikis, discussion forums, and dedicated sites. Despite such studies share several common operations, e.g., scanning all the blocks and the transactions in the blockchain, converting the value of a transaction from bitcoins to USD, etc., researchers so far tended to implement ad-hoc tools for their analyses, rather than reusing standard libraries. Further, most of the few available tools have limitations, e.g. they feature a fixed set of analytics, or they do not allow to combine blockchain data with external data, or they are not amenable to be updated.

The main contribution of this thesis is a framework to create general-purpose analytics on the Ethereum blockchain and on Initial Coin Offerings.

Chapter 2

Background

In order to full understand the topic of this thesis, in this chapter we discuss on background on Ethereum, highlighting the concept of: *Account*; *Message*; *Transaction*; *State Transiction Function* and *Smart Contract*. Furthermore, we discuss on background on Initial Coin Offerings, and how they work.

2.1 Background on Ethereum

Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference. These apps run on a custom built **blockchain**, an enormously powerful shared global infrastructure that can move value around and represent the ownership of property. This enables developers to create markets, store registries of debts or promises, move funds in accordance with instructions given long in the past (like a will or a futures contract) and many other things that have not been invented yet, all without a middleman or counterparty risk.

2.1.1 Ethereum accounts

In Ethereum, the state is made up of objects called ***accounts***, with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts. An Ethereum account contains four fields:

- The *nonce*, a counter used to make sure each transaction can only be processed once
- The account's current *ether balance*
- The account's *contract code*, if present
- The account's *storage* (empty by default)

Ether is the main internal crypto-fuel of Ethereum, and is used to pay transaction fees.
In general, there are two types of accounts: externally owned accounts, controlled by private keys, and contract accounts, controlled by their contract code.

2.1.2 Messages and Transactions

The term *transaction* is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account. Transactions contain:

- The *recipient* of the message
- A *signature* identifying the *sender*
- The *amount of ether* to transfer from the sender to the recipient
- An optional *data* field
- A *STARTGAS* value, representing the maximum number of computational steps the transaction execution is allowed to take
- A *GASPRICE* value, representing the fee the sender pays per computational step

The first three are standard fields expected in any cryptocurrency. The data field has no function by default, but the virtual machine has an opcode using which a contract can access the data; as an example use case, if a contract is functioning as an on-blockchain domain registration service, then it may wish to interpret the data being passed to it as containing two "fields", the first field being a domain to register and the second field being the IP address to register it to. The contract would read these values from the message data and appropriately place them in storage.
The STARTGAS and GASPRICE fields are crucial for Ethereum's anti-denial of service model. In order to prevent accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use.

2.1.3 Messages

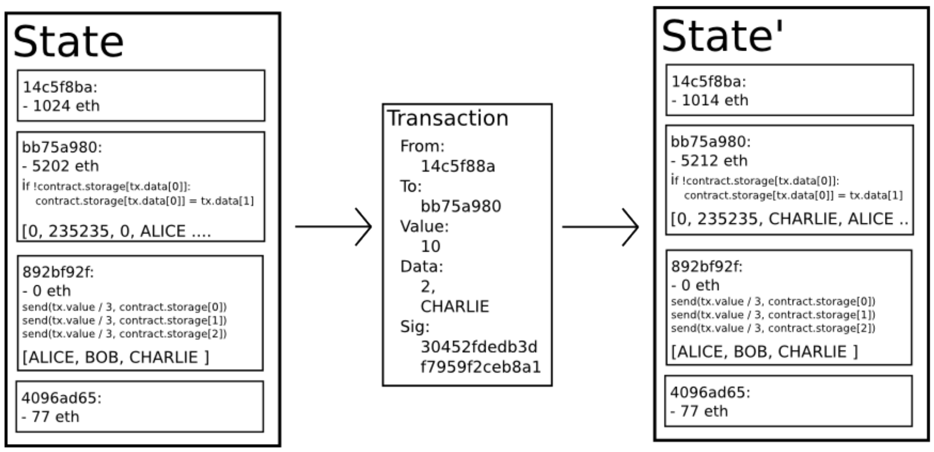
Contracts have the ability to send "messages" to other contracts. Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment. A message contains:

- The *sender* of the message (implicit)
- The *recipient* of the message
- The *amount of ether* to transfer alongside the message
- An optional *data* field
- *STARTGAS* value

Essentially, a message is like a transaction, except it is produced by a contract and not an external actor. A message is produced when a contract currently executing code executes the CALL opcode, which produces and executes a message.
Like a transaction, a message leads to the recipient account running its code. Thus, contracts can have relationships with other contracts in exactly the same way that external actors can.

2.1.4 Ethereum State Transition Function

A step of Ethereum State Transition Function can be displayed as follows:


State
14c5f8ba:
- 1024 eth
bb75a980:
- 5202 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, 0, ALICE ....
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth
Transaction
From:
14c5f88a
To:
bb75a980
Value:
10
Data:
2,
CHARLIE
Sig:
30452fdedb3d
f7959f2ceb8a1
State'
14c5f8ba:
- 1014 eth
bb75a980:
- 5212 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, CHARLIE, ALICE ..
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth

This function, `APPLY(S,TX) -> S'` can be defined as follows:

1. Check if the transaction is well-formed (i.e. has the right number of values), the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.

2. Calculate the transaction fee as STARTGAS * GASPRICE, and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.

3. Initialize GAS = STARTGAS, and take off a certain quantity of gas per byte to pay for the bytes in the transaction.

4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.

5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.

6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

2.1.5 Ethereum Smart Contracts

A smart contract [8] is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts allow the performance of credible transactions without third parties. These transactions are trackable and irreversible. Smart contracts were first proposed by Nick Szabo, who coined the term, in 1994.

There are, basing on applications, four types of smart conntract:

- *Smart Legal Contract*: smart contract combined with legal contract templates;
- *Decentralized Autonomous Organizations (DAO)*: multiple smart contracts combined with governance mechanism;
- *Distributet Applications (DApps)*: Combination of smart contract codes;
- *Smart Contracting Devices*: combined with devices (IoT).

Proponents of smart contracts claim that many kinds of contractual clauses may be made partially or fully self-executing, self-enforcing, or both. The aim of smart contracts is to provide security that is superior to traditional contract law and to reduce other transaction costs associated with contracting. Various cryptocurrencies have implemented types of smart contracts.

A smart contract can be deployed using three different languages:

- *Solidity*: A Javascript-like Object Oriented language;
- *Serpent*: A Python-like Object Oriented language;
- *LLL (Low-level Lisp-like Language)*: A Lisp-like Functional language.

2.1.6 Applications in Token Systems

On-blockchain token systems have many applications ranging from sub-currencies representing assets such as USD or gold to company stocks, individual tokens representing smart property, secure unforgeable coupons, and even token systems with no ties to conventional value at all, used as point systems for incentivization.
Token systems are surprisingly easy to implement in Ethereum. The key point to understand is that all a currency, or token system, fundamentally is a database with one operation: subtract `X` units from `A` and give `X` units to `B`, with the proviso that

1. `A` had at least `X` units before the transaction;
2. The transaction is approved by `A`.

All that it takes to implement a token system is to implement this logic into a contract. The basic code for implementing a token system in Serpent looks as follows:

```
def send(to, value):
    if self.storage[msg.sender] >= value:
        self.storage[msg.sender] = self.storage[msg.sender] - value
        self.storage[to] = self.storage[to] + value
```

Theoretically, Ethereum-based token systems acting as sub-currencies can potentially include another important feature that on-chain Bitcoin-based meta-currencies lack: the ability to pay transaction fees directly in that currency. The way this would be implemented is that the contract would maintain an ether balance with which it would refund ether used to pay fees to the sender, and it would refill this balance by collecting the internal currency units that it takes in fees and reselling them in a constant running auction. Users would thus need to "activate" their accounts with ether, but once the ether is there it would be reusable because the contract would refund it each time.

2.2 Background on ICOs

2.2.1 Initial Coin Offering

An **initial coin offering** (ICO) is a controversial means of crowdfunding centered around cryptocurrency [7, 25] which can be a source of capital for startup companies.

In an ICO, a quantity of the crowdfunded cryptocurrency is preallocated to investors in the form of *tokens*, in exchange for legal tender or other cryptocurrencies such as **Bitcoin** or **Ethereum**. These tokens supposedly become functional units of currency if or when the ICO's funding goal is met and the project launches.

ICOs provide a means by which startups avoid costs of regulatory compliance and intermediaries, such as venture capitalists, bank and stock exchanges while increasing risk for investors. ICOs may fall outside existing regulations or may need to be regulated depending on the nature of the project, or are banned altogether in some jurisdictions, such as China and South Korea

2.2.2 ERC20 Tokens

The ERC20 token standard [28] describes the functions and events that an Ethereum token contract has to implement.
Following is an interface contract declaring the required functions and events to meet the ERC20 standard:

```
contract ERC20Interface {
    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint balance);
    function allowance(address tokenOwner, address spender) public constant returns
        (uint remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns (
        bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner,address indexed spender, uint tokens);
}
```

Some of the tokens include further information describing the token contract:

```
    string public constant name = "Token Name";
    string public constant symbol = "SYM";
    uint8 public constant decimals = 18;
```

2.2.3 How a Token Contract Works

Following is a fragment of a token contract to demonstrate how a token contract maintains the token balance of Ethereum accounts:

```
contract TokenContractFragment {

    // Balances for each account
    mapping(address => uint256) balances;

    // Owner of account approves the transfer of an amount to another account
    mapping(address => mapping (address => uint256)) allowed;

    // Get the token balance for account `tokenOwner`
    function balanceOf(address tokenOwner)
    public constant returns (uint balance) {
        return balances[tokenOwner];
    }

    // Transfer the balance from owner's account to another account
    function transfer(address to, uint tokens)
    public returns (bool success) {
    balances[msg.sender] = balances[msg.sender].sub(tokens);
    balances[to] = balances[to].add(tokens);
    Transfer(msg.sender, to, tokens);
    return true;
    }

    // Send `tokens` amount of tokens from address `from` to address `to`
    // The transferFrom method is used for a withdraw workflow, allowing contracts
        to send
    // tokens on your behalf, for example to "deposit" to a contract address and/or
        to charge
    // fees in sub-currencies; the command should fail unless the _from account has
    // deliberately authorized the sender of the message via some mechanism; we
        propose
    // these standardized APIs for approval:
```

```
    function transferFrom(address from, address to, uint tokens)
    public returns (bool success) {
        balances[from] = balances[from].sub(tokens);
        allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
        balances[to] = balances[to].add(tokens);
        Transfer(from, to, tokens);
        return true;
    }

    // Allow `spender` to withdraw from your account, multiple times, up to the `
        tokens` amount.
    // If this function is called again it overwrites the current allowance with
        _value.
    function approve(address spender, uint tokens)
    public returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        Approval(msg.sender, spender, tokens);
        return true;
    }
}
```

Token Balance

For an example, assume that this token contract has two token holders:
`0x1111111111111111111111111111111111111111` with a balance of 100 units
`0x2222222222222222222222222222222222222222` with a balance of 200 units
The token contract's balances data structure will contain the following information:
`balances[0x11111...] = 100`
`balances0x22222...] = 200`
The `balanceOf(...)` function will return the following values:
`tokenContract.balanceOf(0x11111...)` will return 100
`tokenContract.balanceOf(0x22222...)` will return 200

Transfer token balance

If `0x11111...` wants to transfer 10 tokens to `0x22222...`, `0x11111...` will execute the function:

```
    tokenContract.transfer(0x22222..., 10)
```

The token contract's `transfer(...)` function will alter the balances data structure to contain the following information:
`balances[0x11111...] = 90`
`balances[0x22222...] = 210`
The `balanceOf(...)` function will now return the following values:
`tokenContract.balanceOf(0x11111...)` will return 90
`tokenContract.balanceOf(0x22222...)` will return 210

Approve And TransferFrom Token Balance

If `0x11111...` wants to authorise `0x22222...` to transfer some tokens to `0x22222...`, `0x11111...` will execute the function:

```
    tokenContract.approve(0x22222..., 30)
```

The approve data structure will now contain the following information:
    `tokenContract.allowed[0x11111...][0x22222...] = 30`
If`0x22222...` wants to later transfer some tokens from `0x11111...` to itself, `0x22222...` executes the `transferFrom(...)` function:

```
    tokenContract.transferFrom(0x11111..., 0x22222..., 20)
```

The balances data structure will be altered to contain the following information:
    `balances[0x11111...] = 70`
    `balances[0x22222...] = 230`
And the approve data structure will now contain the following information:
    `tokenContract.allowed[0x11111...][0x22222...] = 10`
0x22222... can still spend 10 tokens from 0x11111.... The `balanceOf(...)` function will now return the following values:
    `tokenContract.balanceOf(0x11111...)` will return 70
    `tokenContract.balanceOf(0x22222...)` will return 230

2.3 Related works

This work is based on a previous work. We first developed a tool for Ethereum blockchain analytics, improving an existing tool, which is intended to analyse Bitcoin blockchain [4]. It extrapolates Bitcoin blockchain data, such as block and transactions, and combine them with external data, such as exchange rates or addresses tags. With this data, it constructs a view using either a SQL (MySQL) or a NoSQL (MongoDB) DBMS.

EtherScan offers a REST api to retrieve blockchain, doing what we have done with Web3J. They achieve this with a GETH/Parity proxy from the REST api to a direct JSON-RPC request sent to either a GETH or a Parity client installed in their servers. But it is not as complete as our tool, because it does not offer a way to combine Ethereum blockchain data with external data.
As we will see in section 3.2.5, Etherscan offers a way to retrieve also external date through REST API, and we use this API to retrieve external data useful to combine it with blockchain in ICOs analysis.

Chapter 3

A tool for Ethereum analytics

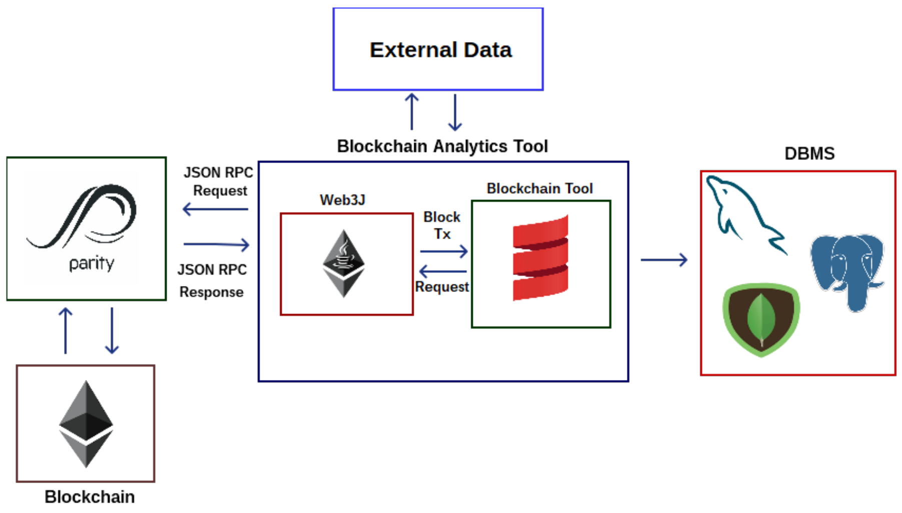
In this chapter we describe in details how our created tool works. First of all, we describe its architecture in details, highlighting every separated part, in particular we talk about *Parity*, *Web3J* and *JSON-RPC*. Further, we discuss some case studies using our tool, retrieving data from Ethereum blockchain, and then combining it with external data. Furthermore, we discuss about tool implementation, and how it works internally.

3.1 Tool architecture

In this section we will describe how this tool works. This tool is build for create a custom view of either the Ethereum blockchain or the Bitcoin blockchain.
Since we do not use Bitcoin blockchain, we will describe only how the Ethereum part works. We said the the view is *custom* because this tool allows the user to extract whatever he needs from the blockchain, such as:

- Blocks information, e.g. block number, block hash;
- Transactions information, e.g. transaction hash, sender, receiver, created contract (if exists);
- Contract internal transactions information, e.g. transaction hash, sender, receiver, father transaction hash (the transaction that calls the defined contract)

Furthermore, the view *customization* is possible because our tool is able to retrieve **external data** (for external we mean data not stored directly inside blockchain) and combine it with blockchain's internal data. This external data could be of any nature, e.g. exchange rates, ICOs data, *etc.*...
The image in the next page shows how out tool works.



As you can see, in order to retrieve blockchain data, our tool makes a request through Web3J (described in 3.1.2), which in turn makes a request directly to the Ethereum client, in our case Parity (described in 3.1.1). This requests are made using the JSON-RPC protocol (described in 3.1.3).

When Parity returns the wanted result, Web3J serialize it into a collection of objects, then returns them to our tool. Since these classes do not contains all information, we decided to wrap them in another classes, retrieving all missing information, i.e. contract internal trasactions.

In the next subsections, we briefly describe how Parity, Web3J and JSON RPC work. Then, in the next sections, we show some view examples, using *MongoDB* as DBMS.

3.1.1 Parity

Parity [2] is an Ethereum client, written from the ground-up for correctness-verifiability, modularisation, low-footprint and high-performance.
To this end it utilises the Rust language, a hybrid imperative/OO/functional language with an emphasis on efficiency.
Parity comes with an extensive, in-built Ethereum Wallet and DApp environment. It includes:

- Account, address-book and multi-sig management.
- Key creation, importing and exporting.
- Web3 app browser.
- Hardware and electronic cold-wallet support.

- Name registry support.
- Contract development, deployment and interaction environment.

3.1.2 Web3J

Web3J is a highly modular, reactive, type safe Java and Android library for working with Smart Contracts and integrating with clients (nodes) on the Ethereum network. This allows to work with the Ethereum blockchain, without the additional overhead of having to write an integration code for the platform.

Web3J is capable to connect to a local blockchain, downloaded using Parity, Geth or another Ethereum client, or a remote one, like Infura. In order to retrieve blockchain data, Web3J uses a protocol called JSON-RPC, which will be described in section 3.1.3.

3.1.3 JSON-RPC

JSON-RPC [12] is a remote procedure call protocol encoded in JSON. It is a very simple protocol (and very similar to XML-RPC), defining only a few data types and commands. JSON-RPC allows for notifications (data sent to the server that does not require a response) and for multiple calls to be sent to the server which may be answered out of order.

JSON-RPC works by sending a request to a server implementing this protocol. The client in that case is typically software intending to call a single method of a remote system. Multiple input parameters can be passed to the remote method as an array or object, whereas the method itself can return multiple output data as well. (This depends on the implemented version).

All transfer types are single objects, serialized using JSON. A request is a call to a specific method provided by a remote system. It must contain three certain properties:

- *Method*: A String with the name of the method to be invoked;
- *Params*: An Object or Array of values to be passed as parameters to the defined method;
- *Id*: A value of any type used to match the response with the request that it is replying to.

The receiver of the request must reply with a valid response to all received requests. A response must contain the following properties:

- *Result*: The data returned by the invoked method. If an error occurred while invoking the method, this value must be null;
- *Error*: A specified error code if there was an error invoking the method, otherwise null;
- *Id*: The id of the request it is responding to.

Since there are situations where no response is needed or even desired, notifications were introduced. A notification is similar to a request except for the id, which is not needed because no response will be returned. In this case the id property should be omitted (Version 2.0) or be null (Version 1.0).

Examples

In this subsection, we will show one example of *Request and Response* and one of *Notification* using JSON-RPC version 2.0.
Request and response:

```
Request
{
    "jsonrpc": "2.0",
    "method": "subtract",
    "params": {
                "minuend": 42,
                "subtrahend": 23
              },
    "id": 3
}
Response
{
    "jsonrpc": "2.0",
    "result": 19,
    "id": 3
}
```

Notification, with no response

```
{
    "jsonrpc": "2.0",
    "method": "update",
    "params": [1,2,3,4,5]
}
```

3.2 External Information Sources

3.2.1 ICO-Rating

Available Data

ICO-Rating contains a list of active ICOs. It rates them relying on they stability and risk on investment. ICORating rates an ICO using three scores:

- ***HypeScore***: Hype-score shows investor level of interest in the project. The higher the score, the more people may consider the project for future investments. That is a numeric score between 0 and 5;

- ***RiskScore***: Risk score is aimed at assessing the risk of potentially fraudulent activities. The higher the risk score, the less information there is on the details of the ICO campaign, product development, the team and the documentation, which calls into question the possibility for success of the start-up and the ICO/Token sale. That is a numeric score between 0 and 5;

- ***Investment Rating***: The metrics of this parameter is divided into 10 levels: *Positive+, Positive, Stable+, Stable, Risky+, Risky, Risky-, Negative, Negative-, Default.* The higher the rate assigned to the project, the better the overall quality of the projects documentation, and the lower the number of risks for future investors.

Extracted API Methods

In this subsection we describe the API internal methods that retrieve data listed in 3.2.1.
In the following table is written that the methods used to retrieve all the data described above take only the token name as input.
This is because ICORating does not have a REST API, or any other API to call, but we had to do a scraping on the HTML page at the address `icorating.com/ico/<token-name>/`.

In order to retrieve all this data, we created a class called **ICORatingAPI**, following are the methods in this class.

Field Name	*Description*	*Method Signature*
Hype Score	ICORating's hype score for the given ICO	`getHypeScore(icoName)`
Risk Score	ICORating's risk score for the given ICO	`getRiskScore(icoName)`
Investment Rating	ICORating's investment rating for the given ICO	`getInvestmentRating(icoName)`

3.2.2 ICOBench

Available Data

ICOBench is a free ICO rating platform and a blockchain community supported by a wide range of experts that provides analytical, legal, and technical insights to the investors. ICOBench also gives a rating for each Token. Rating is given in combination of:

- Their assessment algorithm that uses more than 20 different criteria on which each ICO can earn more than 30 points and

- The rating the independent experts give to the ICO following our rating methodology suggestions.

The rating is a float value beetween 0 and 5.0 and is splitted in four subratings, which are combined (with a simple arithmetic average) to retrieve the general ICO rating:

- ICO Profile
- Team
- Vision
- Product

ICOBench gives an useful REST API to retrieve all data they have about 1787 ICO from 164 different countries.
Like ICORating in 3.2.1, in order to retrieve all needed data with rest APIs, we have to use either the token name or the token symbol. We have to do this because ICOBench does not store (or does not make available) the contract address, which we believe is more unambiguous then token name. In the next subsection we will talk about all the data that we extracted using this API.

Extracted API Methods

In order to extract useful data using this API, we have created a specific class, called **ICOBenchAPI**, following are the methods in this class.

Field Name	*Description*	*Method Signature*
ICO Symbol	ICO's symbol	`getSymbol(icoName)`
Exchange Details	Details about the Exchanges that trade this token	`getExchanges(icoName/icoSymbol)`
Market Cap	Actual Market Capitalization of the given ICO	`getMarketCap(icoName/icoSymbol)`
General Rating	ICOBench's general rating for the given Token	`getGeneralRating(icoName/icoSymbol)`
Profile Rating	ICOBench's profile rating for the given Token	`getProfileRating(icoName/icoSymbol)`
Team Rating	ICOBench's team rating for the given Token	`getTeamRating(icoName/icoSymbol)`
Vision Rating	ICOBench's vision rating for the given Token ICO	`getVisionRating(icoName/icoSymbol)`
Product Rating	ICOBench's product rating for the given Token	`getProductRating(icoName/icoSymbol)`

3.2.3 TokenWhoIs

Available Data

TokenWhoIs is a website that contains a wide list of ICOs, and for eeach of them, it contains information like Market Capitalization, used blockchain, unit price in various currencies.
Like ICOBench in 3.2.2 in order to retrieve all needed data with rest APIs, we have to use either the token name or the token symbol.

Extracted API Methods

In order to extract useful data using this API, we have created a specific class, called **TokenWhoIsAPI**, following are the methods in this class.

Field Name	*Description*	*Method Signature*
Used Blockchain	Blockchain used by the given token	`getUsedBlockchain(icoName)`
Market Cap	Actual Market Capitalization of the given ICO	`getMarketCap(icoName)`
Token Unit Price (USD)	Token current unit price (USD)	`getUSDUnitPrice(icoName)`
Token Unit Price (ETH)	Token current unit price (ETH)	`getETHUnitPrice(icoName, icoSymbol)`
Token Unit Price (BTC)	Token current unit price (BTC)	`getBTCUnitPrice(icoName)`
Exchange Names	Name of Exchanges that trade this token ICO	`getExchangeNames(icoName)`
Token Supply (USD)	Token total supply in USD	`getUSDSupply(icoName,icoSymbol)`

3.2.4 CoinMarketCap

CoinMarketCap is a website that provides market data about cryptocurrencies and ICO tokens. It provides data like actual and historical market capitalization and actual and historical cryptocurrency unit price in various currencies.

Available Data

CoinMarketCap provides an useful REST API, which prvides the following data for each cryptocurrency:

- *Name*: Token Name;
- *Symbol*: token Symbol;
- *Price USD*: Current currency unit price in USD;
- *Price BTC*: Current currency unit price in BTC;
- *24h volume USD*: Volume of currency transferred in the last 24 hours, in USD;
- *Market Cap USD*: Coin Market Capitalization in USD;
- *Available Supply*: Coin available supply. 'Available' means 'available to buy';
- *Total Supply*: Coin total supply;
- *Percent Change 1h*: Change of coin price in the last hour;
- *Percent Change 24h*: Change of coin price in the 24 hours;
- *percent Change 7d*: Change of coin price in the seven days;
- *Last Updated*: Last time these information were updated.

Extracted API Methods

In order to extract useful data using this API, we have created a specific class, called **CoinMarketCapAPI**, following are the methods in this class.

Field Name	*Description*	*Method Signature*
Market Cap	Actual Market Capitalization of the given token	`getTokenMarketCap(icoName, icoSymbol)`
Token Unit Price (USD)	Token current unit price (USD)	`getTokenUSDPrice(icoName, icoSymbol)`
Token Unit Price (BTC)	Token current unit price (BTC)	`getTokenBTCPrice(icoName, icoSymbol)`
Exchange Names	Name of Exchanges that trade this token ICO	`getExchangeNames(icoName)`
Token Supply (USD)	Token total supply in USD	`getUSDSupply(icoName,icoSymbol)`

3.2.5 EtherScan

Available Data

EtherScan is a Block Explorer and Analytics Platform for Ethereum. It provides a view of all blockchain containing:

- Block information such as block number, hash, miner;
- Accounts information;
- Transaction information such as hash, receiver, sender, amount;
- Internal Transaction information such as hash, receiver, sender, amount, father transaction's hash;
- Tokens information such as token name, symbol, owners;

Etherscan also provides an useful REST API to retrieve all data they have. In the next section we'll describe all data extracted from Etherscan

Extracted API Methods

In order to extract useful data using this API, we have created a specific class, called **EtherScanAPI**, following are the methods in this class.

Field Name	*Description*	*Method Signature*
Token total Supply	Token total supply	`getTotalSupplyByAddress(icoAddress)`
Token balance by adderess	Given an account address, it returns its token balance	`getTokenAccountBalance(icoAddress, accountAddress)`

3.2.6 Ethplorer

Available Data

Ethplorer is a token blobkchain explorer. For each token, it provides information such as:

- Basic token information (Name, Symbol, Price, Total Supply);
- Basic token contract information (Address, creator's address, balance (ETH), number of transactions);
- Token Operations details inside a time interval (Transaction hash, sender, receiver, amount of token transferred);

Extracted API Methods

In order to extract useful data using this API, we have created a specific class, called **EthplorerAPI**, following are the methods in this class.

Field Name	*Description*	*Method Signature*
ICO Name	ICO's name	`getTokenName(contractAddress)`
ICO Symbol	ICO's Symbol	`getTokenSymbol(contractAddress)`
Token Price	Token current price (USD)	`getTokenPrice(contractAddress)`

3.3 Implementation

In this section we will explain in details the implementation of the tool, built to retrieve blockchain data and other external data and combine them.

This tool is written in Scala Programming Language and built with Scala Build Tool (SBT). Scala has been chosen because, since it's compiled in bytecode that runs on Java Virtual Machine (JVM), it can be combined with Java external libraries without any problem.
Thanks to this property, it was possible to use Web3J (explained in 3.1.2) library to retrieve blockchain data. It is a Java and Android library for working with Smart Contracts and integrating with clients (nodes) on the Ethereum network.

The core class, used to access both *Ethereum* and *Bitcoin* blockchain is `BlockchainLib`. It has two methods:

- `getBitcoinBlockchain(settings: BitcoinSettings)`
- `getEthereumBlockchain(url: String)`

Since for this thesis we used only `getEthereumBlockchain`, we will describe only this class.
The `getEthereumBlockchain` method takes one parameter that is the url where the blockchain is stored. For example, if you are using Parity locally, the url should be like `http://localhost:8545`, since the `8545` port is the default port where Parity listens to JSON-RPC requests.

This method returns an istance of `EthereumBlockchain` which extends a `Traversable`, so it can be looped with a simple `foreach`, setting first the following parameters:

- *startBlock*: the block from which to start. It is modifiable with the `setStart` method. If not setted, the default is 0;
- *endBlock*: the block from which to end. It is modifiable with the `setEnd` method. If not setted, the default is the last block;
- *step*: the interval between each block visited. It is modifiable with the `setStep` method. If not setted, the default is 1.

This `foreach` loops over all requested block. Each block is an instance of the `EthereumBlock` class. An `EthereumBlock` object is retrieved from an `EthBlock.Block` object of the *Web3J* library, retrieved calling the `Web3J.getBlockByNumber` method. This *Web3J*'s method does internally a JSON-RPC request to the *getBlockByNumber* method, directly to Parity, at the previously defined url.
The difference between the `EthereumBlock` object and the `EthBlock.Block` object is that the first one contains also the internal transactions information, not normally returned using an implemented method in *Web3J*.
In order to retrieve all information about Contract Internal Transaction, we have to do anther JSON-RPC request calling the *trace_block* method. This request allows to see all the transactions and internal transactions contained in this block. From this request, we extract only the internal transactions, filtering the normal transactions, already known.
Each `EthereumBlock` object contains the following information:

- *number*: block number inside blockchain;
- *hash*: block hash;
- *parentHash*: hash of the parent block;
- *miner*: address of the account that mined this block;
- *size*: size of block in bytes;
- *timeStamp*: the unix timestamp for when the block was collated;
- *transactions*: Array of transaction objects (this data structure will be described below);
- *internalTransactions*: Array of contract transaction objects (this data structure will be described blow).

The *transaction* field is a List of `EthereumTransaction` objects. These objects are created using a factory method that takes as input a object of `EthBlock.TransactionObject` *Web3J* class.
Each `EthereumTransaction` object contains the following information:

- *hash*: transaction hash;
- *blockHash*: hash of the block that contains this transaction;
- *transactionIndex*: index of the transaction inside its block;
- *from*: transaction sender;
- *to*: transaction receiver;
- *value*: value transferred in this transaction in Wei;

- *gasPrice*: gas price provided by the sender in Wei;
- *gas*: gas provided by the sender;
- *input*: the data send along with the transaction;
- *creates*: creates contract hash. This field is not null if and only if this transaction creates a contract, which address is displayed here;

The *internalTransactions* field is a List of `EthereumInternalTransaction` objects. These objects are created unmarshalling the result of the *trace_block* JSON-RPC request.
Each `EthereumInternalTransaction` object contains the following information:

- *parentTxHash*: hash of the parent transaction;
- *txType*: transaction type (create, suicide, call);
- *from*: transaction sender;
- *to*: transaction receiver;
- *value*: value transferred in this transaction in Wei.

These classes envelop all the Ethereum Blockchain's useful data. In order to retrieve external data (e.g. ICO data), we have to create one new class inside the `custom` package.
Let's consider the section 3.5 as an example. In order to retrieve the exchange rate information, we created a new class called `custom.PriceHistorical` with one method, called `getPrice` which takes one argument as input that is the date timestamp, and returns the ETH/USD exchange rate in that timestamp.

When all the data is prepared to be gathered, you must choose what kind of database you will use. This tool supports both SQL (MySQL, PostgreSQL) and NoSQL (MongoDB) databases. If you want to use a **SQL** database, you have to use our `db` package, designed to be used in these cases. More precisely, you have to instantiate an object of the `Table` class, that is a table in the SQL database.
Here's an exhaustive example:

```
import tcs.db.sql.Table
import tcs.db.{DatabaseSettings, PostgreSQL}
val blockTable = new Table(
  sql"""
      SQL COMMAND TO CREATE TABLE
     """,
  sql"""
       SQL COMMAND TO INSERT ELEMENT IN TABLE
     """,
  new DatabaseSettings(dbName, PostgreSQL)
)
```

If you want to use a **NoSQL** database, you have to use our `mongo` and `db` packages, designed to be used in these cases. More precisely, you have to instantiate an object of the `Collection` class, then call this `append` method to add elements inside the collection.
Here's an example:

```
import mongo.Collection
import db.{DatabaseSettings, MongoDB}
val collection = new Collection(
    "collectionName",
    new DatabaseSettings("dbName", MongoDB)
)
//Data Gathering
collection.append(gatheredData)
```

3.4 Case study: a basic view of the blockchain

This view contains data about all transactions (and, contract internal transactions) that has been done inside Ethereum blockchain. The fields in this view are the following:

- ***txHash***: transaction hash;
- ***blockHeight***: block number of the block that contains this transaction;
- ***txIndex***: transaction progressive number inside its block;
- ***date***: the date when block is mined;
- ***from***: transaction sender (the address of who is transferring money or creating a contract);
- ***to***: transaction receiver (the address of who is receiving money, this field is empty if this transaction creates a contract);
- ***value***: how much (in ETH) is transferred;
- ***creates***: the address of the newly created contract (empty if this transaction does not create a contract);
- ***internalTransactions***: contains all the internal transactions generated by this transaction, it's a list of objects containing the following fields:

  parentTxHash: hash of the father transaction (the transaction that generates this one);

  txType: internal transaction type (call, suicide, create);

  from: internal transaction sender;

  to: internal transaction receiver;

  value: how much (in ETH) in transferred.

Here's the code snippet used to create this view:

```
val blockchain = BlockchainLib.getEthereumBlockchain("http://localhost:8545")
  val mongo = new DatabaseSettings("myDatabase")
  val myBlockchain = new Collection("myBlockchain", mongo)

  blockchain.foreach(block => {
    if(block.number % 1000 == 0){
      println("Current block ->" + block.number)
    }
    val date = new Date(block.timeStamp.longValue()*1000)
    block.transactions.foreach(tx => {
      val internalTransactions = block.internalTransactions.filter(itx => itx.
          parentTxHash.equals(tx.hash))
      val creates = if(tx.creates == null) "" else tx.creates
      val to = if(tx.to == null) "" else tx.to
      val list = List(
        ("txHash", tx.hash),
        ("blockHeight", tx.blockNumber.toString()),
        ("txIndex", tx.transactionIndex),
        ("date", date),
        ("from", tx.from),
        ("to", to),
        ("value", tx.value.doubleValue()),
        ("creates", creates),
        ("internalTransactions", internalTransactions)
      )
      myBlockchain.append(list)})
```

3.4.1 Querying view in MongoDB

Ethereum per day

This query calculates the total amount of Ether transacted and its mean per day.

```
db.myBlockchain.aggregate([
   { $group : {
       _id: {
           year : { $year : "$date" },
           month : { $month : "$date" },
           day : { $dayOfMonth : "$date" },
       },
       sumValues: { $sum: "$value"},
       avgValues: { $avg: "$value"}
   }},
   { $sort : { avgValues : 1, sumValues: 1}}
]);
```

Contract creation

This query search all transactions that create a smart contract. It returns only the created smart contract address.

```
db.myBlockchain.find({
   creates: {$ne: ""}
},{
   _id: 0, creates: 1
});
```

3.5 Case study: Exchange rates

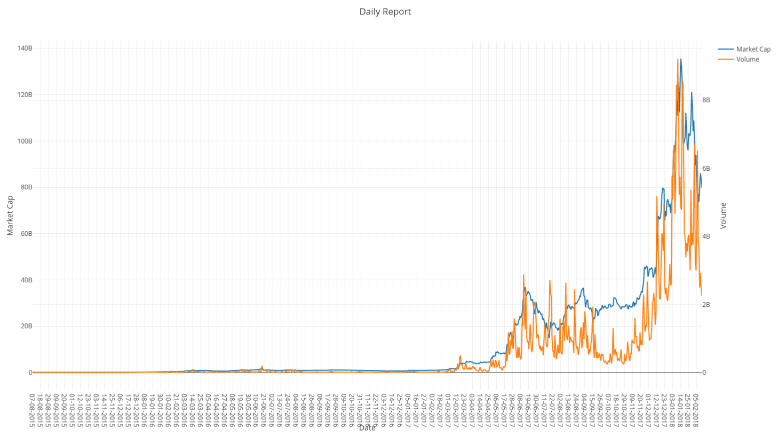
This view contains data about all transactions, combined with the Ethereum conversion price in USD in that specific day. The fields in this view are the following:

- ***txHash***: transaction hash;
- ***blockHeight***: block number of the block that contains this transaction;
- ***txIndex***: transaction progressive number inside its block;
- ***date***: the date when block is mined;
- ***from***: transaction sender (the address of who is transferring money or creating a contract);
- ***to***: transaction receiver (the address of who is receiving money, this field is empty if this transaction creates a contract);
- ***value***: how much (in ETH) is transferred;
- ***creates***: the address of the newly created contract (empty if this transaction does not create a contract);
- ***rate***: Ethereum-USD conversion price.

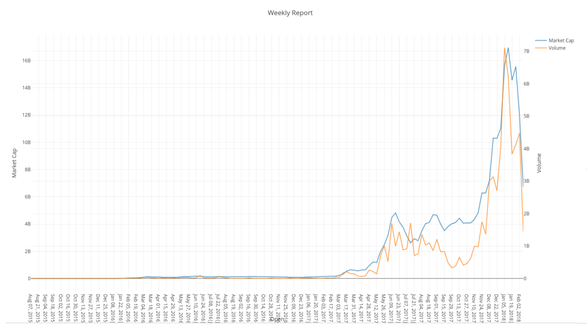
Here's the code snippet used to create this view:

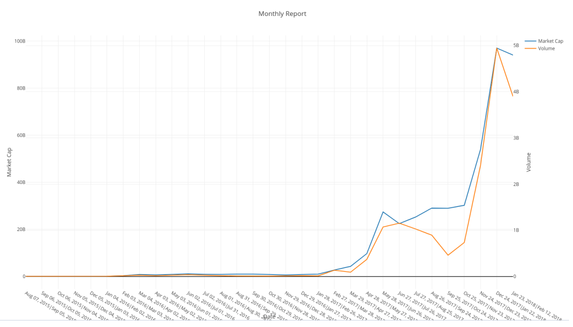
```
val blockchain = BlockchainLib.getEthereumBlockchain("http://localhost:8545")
val mongo = new DatabaseSettings("myDatabase")
val weiIntoEth = BigInt("1000000000000000000")
val txWithRates = new Collection("txWithRates", mongo)
val format = new SimpleDateFormat("yyyy-MM-dd")
val priceHistorical = PriceHistorical.getPriceHistorical()

blockchain.foreach(block => {
  if(block.number % 1000 == 0){
    println("Current block ->" + block.number)
  }
  val date = new Date(block.timeStamp.longValue()*1000)
  val dateFormatted = format.format(date)
  block.transactions.foreach(tx => {
    val creates = if(tx.creates == null) "" else tx.creates
    val to = if(tx.to == null) "" else tx.to
    val list = List(
      ("txHash", tx.hash),
      ("blockHeight", tx.blockNumber.toString()),
      ("txIndex", tx.transactionIndex),
      ("date", date),
      ("from", tx.from),
      ("to", to),
      ("value", tx.value.doubleValue()/weiIntoEth.doubleValue()),
      ("creates", creates),
      ("rate", if(block.timeStamp.longValue() < 1438905600) 0 else
          priceHistorical.price_usd(dateFormatted))
    )
    txWithRates.append(list)})
```

If we combine this view with the previous one, which contains the amount of ether transacted per day, we can plot a graphic containing the volume per day for Ethereum. In the next page, we show the extracted graphic.



If we set the query in order to retrieve the weekly and monthly mean of this information, we have the following graphics:





3.6 Case study: Levenshtein Distance between contract

In information theory, Linguistics and computer science, the Levenshtein distance is a string metric for measuring the difference between two sequences. In order to calculate Levenshtein distance between contracs EVM, first of all we create a view in MongoDB or SQL that contains only useful information retrieved from blockchain. The useful fields are only two:

- ***contractAddress***: The contract address inside blockchain (to uniquely identify a contract inside view);
- ***contractCode***: The contract EVM code

The distance calculation is done using this formula:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

3.7 Case Study: Collection of ICOs data

In order to store blockchain data combined with the external data retrieved using functions described in section 3.2, we chose to use SQL, for the sake of precision, we used PostgreSQL. In the following section we will explain in details this view, and how we used it.

3.7.1 ICO's View

The view is composed of four SQL tables:

- *Block*: contains all information about blocks retrieved directly from blockchain. In particularly, it contains the following columns:
  - *Hash*: the block hash inside blockchain, which is an unambiguous 32-Byte value and is used as table primary key;
  - *Number*: represents the block's order inside blockchain;
  - *Parent Hash*: contains the hash of the previous block, considered as its father inside blockchain;
  - *Timestamp*: the date and time when this block is mined;
  - *Miner*: the account that has mined this block;
- *Transaction*: contains all information about transaction contained inside every block, retrieved directly from blockchain. In particularly, it contains the following columns:
  - *Hash*: the transaction hash inside blockchain, which is unambiguous like block ones and is used as table primary key;
  - *nonce*: the number of transactions made by the sender prior to this one;
  - *transaction index*: the number of this transaction inside its block;
  - *from*: the transaction sender's address;
  - *to*: the transaction receiver's address;
  - *value*: value transferred in this transaction, in Wei;
  - *creates*: this field is filled i and only if this transaction creates a contract. It contains the address of the newly created contract;
  - *gas*: gas provided by the sender;
  - *gasprice*: gas price provided by the sender in Wei;
  - *blockhash*: the hash of the block that contains this transaction, it is used as the foreign key for the *Block* table.
- *Internal-Transaction*
  - *Id*: this field is used as primary key in this table;
  - *Parent Tx Hash*: the hash of the father transaction. that is the transaction that generates this internal transaction;
  - *Transaction type*: the internal transaction type (a value between call, suicide, create);
  - *From*: the internal transaction sender's address;
  - *To*: the internal transaction receiver's address;
  - *Value*: alue transferred in this transaction, in Wei;

- *ICO*
  - *Id*: this field is used as primary key in this table;
  - *Token Name*: The token name;
  - *Token Symbol*: The token Symbol;
  - *Contract Address*: the address of the ERC20-contract that has created this token;
  - *Market Cap*: The token market capitalization;
  - *Total Supply*: The token total supply;
  - *Price USD*: The current token price (USD);
  - *Price BTC*: The current token price (BTC);
  - *Hype Score*: The token hype score, given by ICORating;
  - *Risk Score*: The token risk score, given by ICORating;
  - *Investment Rating*: The token investment rating, given by ICORating;
  - *Tx Creator Hash*: The address of the transaction that has created the ERC20-contract, which address is in the *Contract Address* field.

3.8 Examples of tool usage on ICOs

In this view we stored all data contained beetween block 3000000 and block 4100000, these are the block mined in the first seven month of 2017. We collect data about:

- 1.100.000. Blocks;
- About 22.000.000 Transactions;
- 4.000.000 Internal transactions;
- 1.500 ICOs

In this section, we discuss some studies done with all this data, stored in a SQL database.

3.8.1 MarketCap and Volume with Exchange Rates

With all information retrieved on *CoinMarketCap* and *CryptoCompare*, we can extract the historical exchange rate of all tokens, and combine them with transaction data in order to retrieve Market Capitalization and Volume of each token.
In this example, we examinate the historical Volume and Historical Market Capitalization of one ICO: 0x (ZRX).
Insert here query to retrieve transaction data about 0x.

3.9 Performance

3.10 Conclusions and future works

We have presented a framework for developing general-purpose analytics on the Ethereum blockchain and his ICOs. Its main component is a Scala library which can be used to construct views of the blockchain, possibly integrating blockchain data with data retrieved from external sources. Blockchain views can be stored as SQL or NoSQL databases, and can be analysed by using their query languages. Our experiments confirmed the effectiveness and generality of our approach, which uniformly comprises in a single framework several use cases addressed by various ad-hoc approaches in literature. Indeed, the expressiveness of our framework overcomes that of the closer proposals in the built-in support for external data, and the support of different kinds of databases and blockchains. Importantly, coming in the form of an open source library for a mainstream language, our framework is amenable of being validated and extended by a community effort, following reuse best practices.

On the comparison of SQL vs NoSQL, our experiments did not highlight significant differences in the complexity of writing and executing queries in the two languages. Instead, we observed that the schema-less nature of NoSQL databases simplifies the Scala scripts.

A more accurate analysis, carried over a larger benchmark, is scope for future work. Anyway, it is worth recalling that the goal of our proposal is provide to the final user the flexibility to choose the preferred database, rather than ascertain an idea of best-fit-for-all in the choice.

Bibliography

[1] Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 34–51. Springer, 2013.

[2] Parity Authors. Ethereum rust client, 2017.

[3] Khaled Baqer, Danny Yuxing Huang, Damon McCoy, and Nicholas Weaver. Stressing out: Bitcoin stress testing. In *International Conference on Financial Cryptography and Data Security*, pages 3–18. Springer, 2016.

[4] Massimo Bartoletti, Andrea Bracciali, Stefano Lande, and Livio Pompianu. A general framework for bitcoin analytics. *arXiv preprint arXiv:1707.01021*, 2017.

[5] Massimo Bartoletti and Livio Pompianu. An analysis of bitcoin op_return metadata. In *International Conference on Financial Cryptography and Data Security*, pages 218–230. Springer, 2017.

[6] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A Kroll, and Edward W Felten. Mixcoin: Anonymity for bitcoin with accountable mixes. In *International Conference on Financial Cryptography and Data Security*, pages 486–504. Springer, 2014.

[7] Usman Chohan. Initial coin offerings (icos): Risks, regulation, and accountability. 2017.

[8] Christopher D Clack, Vikram A Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 2016.

[9] Joseph Bonneau Andrew Miller Jeremy Clark, Arvind Narayanan Joshua A Kroll Edward, and W Felten. Research perspectives and challenges for bitcoin and cryptocurrencies. *url: https://eprint. iacr. org/2015/261. pdf.*

[10] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[11] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings*

of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 3–16. ACM, 2016.

[12] JSON-RPC Working Group et al. Json-rpc 2.0 specification, 2012.

[13] Martin Harrigan and Christoph Fretter. The unreasonable effectiveness of address clustering. In *Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences*, pages 368–373. IEEE, 2016.

[14] Ghassan O Karame, Elli Androulaki, Marc Roeschlin, Arthur Gervais, and Srdjan Čapkun. Misbehavior in bitcoin: A study of double-spending and accountability. *ACM Transactions on Information and System Security (TISSEC)*, 18(1):2, 2015.

[15] Kevin Liao, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Behind closed doors: measurement and analysis of cryptolocker ransoms in bitcoin. In *Electronic Crime Research (eCrime), 2016 APWG Symposium on*, pages 1–13. IEEE, 2016.

[16] Matthias Lischke and Benjamin Fabian. Analyzing the bitcoin network: The first four years. *Future Internet*, 8(1):7, 2016.

[17] Loi Luu, Ratul Saha, Inian Parameshwaran, Prateek Saxena, and Aquinas Hobor. On power splitting games in distributed computation: The case of bitcoin pooled mining. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 397–411. IEEE, 2015.

[18] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[19] Malte Möser and Rainer Böhme. Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In *International Conference on Financial Cryptography and Data Security*, pages 19–33. Springer, 2015.

[20] Malte Möser and Rainer Böhme. Anonymous alone? measuring bitcoins second-generation anonymization techniques. In *Security and Privacy Workshops (EuroS&PW), 2017 IEEE European Symposium on*, pages 32–41. IEEE, 2017.

[21] Malte Moser, Rainer Bohme, and Dominic Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *eCrime Researchers Summit (eCRS), 2013*, pages 1–14. IEEE, 2013.

[22] Micha Ober, Stefan Katzenbeisser, and Kay Hamacher. Structure and anonymity of the bitcoin transaction graph. *Future internet*, 5(2):237–250, 2013.

[23] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[24] Okke Schrijvers, Joseph Bonneau, Dan Boneh, and Tim Roughgarden. Incentive compatibility of bitcoin mining pool reward functions. In *International Conference on Financial Cryptography and Data Security*, pages 477–498. Springer, 2016.

[25] Jason Teutsch, Vitalik Buterin, and Christopher Brown. Interactive coin offerings. *URl: https://people. cs. uchicago. edu/˜ teutsch/papers/ico. pdf (visited on 11/16/2017)*, 2017.

[26] Marie Vasek and Tyler Moore. Theres no free lunch, even using bitcoin: Tracking the popularity and profits of virtual currency scams. In *International conference on financial cryptography and data security*, pages 44–61. Springer, 2015.

[27] Marie Vasek, Micah Thornton, and Tyler Moore. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In *International Conference on Financial Cryptography and Data Security*, pages 57–71. Springer, 2014.

[28] F Vogelsteller. Erc 20 token standard, 2015.

[29] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.