Davide Aimar

Extraction, indexing and analysis of Ethereum smart contracts data

Master's Thesis in Computer Engineering
Supervisor: Prof.ssa Valentina Gatteschi
Co-supervisor: Prof. Mariusz Nowostawski
Academic Year 2022/2023

Politecnico di Torino

In collaboration with
Norwegian University of Science and Technology

ABSTRACT

Blockchain technology has gained popularity in the last decade. New protocols allow developers to build decentralized applications thanks to the usage of smart contracts. Ethereum is one of the most popular blockchain networks of this kind. Every twelve seconds, a new block is appended to this chain. Each block contains information that describes a market worth billions of dollars. Since Ethereum is a permissionless blockchain, this data is publicly available to anyone, but without proper tools, it is not easy to analyse.

This master's thesis focuses on extracting semantics from raw Ethereum data and making it easily available to users by indexing it with Dgraph, an open-source distributed graph database.

A review of the state-of-the-art tools showed that relevant work in this field has been done by private companies whose source code and methodology are not available. Many open-source and public projects resulted in being outdated or slow. This poses the risk of centralizing access to blockchain data in the hands of a few companies.

Part of this master's thesis was dedicated to analyzing the semantics that can be extracted from the blockchain and building a data schema around it that is optimized for graph databases. A custom software, called *eth2dgraph*, was developed to perform the extraction of data. It is an open-source tool written in *Rust* that maps Ethereum data to Dgraph format. It integrates a decompiler to extract and index the ABI of smart contracts. Eth2dgraph was developed with a focus on performance. This was done to scale the extraction process to the history of the Ethereum blockchain. At the end of the thesis, the data indexed in Dgraph has been analyzed to show the current state of the Ethereum blockchain.

This work provides a novel solution to the problem of blockchain data analysis. The open-source nature of the project allows other developers to build on top of it. Performing the actual extraction and indexing came close to hitting the limit of what can be done on a single machine. This highlights the fact that, in the future, distributed approaches will be the only possible way of handling the increasing amount of data that comes from the Ethereum blockchain. This is already evident with layer 2 protocols, which are generating data at a faster pace than Ethereum.

ACKNOWLEDGEMENT

I would like to express my deepest thanks to Prof. Nowostawski for giving me the opportunity to work on these topics and for his constant and valuable support. Additionally, I would like to thank NTNU for hosting me during the final year of my university studies.

My sincere thanks also go to Prof. Gatteschi for her precious availability to assist me in this thesis work.

Thanks to all the people who have supported me during these university years, including my classmates, colleagues from the Policumbent team, Erasmus friends, and lifelong friends.

A big thank you goes to my parents, who always believed in me and gave me all possible means to pursue my dreams.

Finally, thanks to Greta, with whom I shared this challenging yet beautiful university journey.

RINGRAZIAMENTI

Desidero esprimere un profondo ringraziamento al Prof. Nowostawski per avermi offerto l'opportunità di lavorare su questa tesi e per il suo costante e prezioso sostegno. Inoltre, ringrazio la NTNU per avermi accolto durante l'ultimo anno del mio percorso universitario.

Un sentito grazie anche alla Prof.ssa Gatteschi per la sua preziosa disponibilità nel seguirmi in questo lavoro di tesi.

Grazie di cuore a tutte le persone che mi hanno accompagnato in questi anni di università. I miei compagni di corso, i colleghi del team Policumbent, gli amici con cui ho condiviso l'Erasmus in Norvegia e gli amici di una vita.

Un grosso grazie va ai miei genitori, che hanno sempre creduto in me e mi hanno dato tutti i mezzi possibili per perseguire i miei sogni.

Infine, grazie a Greta, la mia compagna di viaggio in questi difficili, ma bellissimi, anni universitari.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

List of all abbreviations in alphabetic order:

- **ABI** Application Binary Interface
- **CFG** Control Flow Graph
- **CHF** Cryptographic hash function
- **CRHF** Cryptographic Resistant hash function
- **dApp** Decentralized Application
- **DQL** Dgraph Query Language
- **ECDSA** Elliptic Curve Digital Signature Algorithm
- **EIP** Ethereum Improvement Proposal
- **EOA** Externally Owned Account
- **ERC** Ethereum Request for Comment
- **EVM** Ethereum Virtual Machine
- **GDB** Graph Database
- **IPC** Inter-Process Communication
- **NFT** Non Fungible Token
- **OWHF** One Way Hash Function
- **p2p** Peer to Peer
- **SC** Smart Contract
- **SHA** Secure Hash Algorithm
- **RPC** Remote Procedure Call

CHAPTER

ONE

INTRODUCTION

1.1 Motivation

Since the publication of the Bitcoin [1] whitepaper by the pseudonym `Satoshi Nakamoto` in 2008, the concept of blockchain has gained widespread use in various domains. The technology has evolved and has become more sophisticated, allowing for more complex use cases than just preventing double spending in money transfers. This was allowed by the use of Smart Contracts, immutable[1] and deterministic pieces of code that rule the outcome of transactions based on the logic written in the code.

Smart Contracts were first implemented in 2015 by the Ethereum blockchain [2], the first of many *Blockchain 2.0* that allowed developers to build decentralized applications.

Ethereum is a permissionless blockchain, it can be seen as a digital public ledger. It can be modified just through append operations and is immutable. One of the main strengths of this kind of blockchain is the fact that the public ledger is transparent, so everyone can independently download and access and verify the data.

Reading and understanding this public ledger can bring huge value, since it describes the entire history of a market that, as of now, is valued hundreds of billions of dollars. Apart from the economical aspects, the knowledge of on-chain data is an essential building block for Decentralised Applications (dApps), that are, by definition, applications which interact with Smart Contracts.

Unfortunately, as noted also in other works [3, 4, 5], extracting and analyzing data from the Ethereum blockchain is not an easy task. The reason is that the amount of data is huge and Ethereum nodes store it for optimizing storage instead of the ease of access.

It is possible to query the Ethereum nodes by just a few parameters, such as transaction hashes, block numbers and indexed log topics. Without an external index, it is impossible to search data based on any other attribute.

My work aims to ease access to on-chain data and describes a way to do it using *Dgraph* [6], an open-source distributed database.

[1] The theoretical immutability of smart contracts is analyzed more in-depth in chapter 6.

1.2 Research questions

Two research questions were defined:

- **RQ1**: What kind of information is possible to extract from EVM blockchains without relying on centralized services?
- **RQ2**: What computational resources are required to independently extract and index the entire history of the Ethereum blockchain in August 2023?

1.3 Contribution

This thesis provides multiple contributions to the topic of data mining from the Ethereum blockchain:

- A research on the main state-of-the-art tools, highlighting their strengths and shortcomings.
- The definition of a schema for Ethereum data optimized for graph databases. This schema includes both raw data and the semantics that can be built from it.
- The release of **eth2dgraph**, an open-source software written in *Rust* that efficiently extracts data from the Ethereum blockchain to be indexed and queried with *Dgraph*, a distributed graph database.
- An analysis of the Ethereum data, both in terms of infrastructure and time needed to perform extraction and indexing and in terms of what semantics can be extracted.

1.4 Outline

The next chapters of this thesis as structured as follows:

- **Chapter 2 - Background**: this chapter introduces the technical details of how a blockchain works, with a focus on Ethereum. It also includes a section about Dgraph.
- **Chapter 3 - Previous work**: in this chapter, each section describes a work done in the field of data extraction and/or indexing of blockchain data.
- **Chapter 4 - Methods**: here it is described in details how eth2dgraph works. It is shown how each piece of information has been extracted.
- **Chapter 5 - Results**: this chapter shows the outcome of running eth2dgraph using a local Ethereum node to extract and index all the history of the chain.
- **Chapter 6 - Analysis of data**: in this chapter there are six independent analysis on the data extracted that show the state of the chain.

- **Chapter 7 - Discussion**: in this chapter are discussed the results of this research, the future work and more generally the upcoming challenges of the sector.
- **Chapter 8 - Conclusions**: the last chapter summarizes the main findings and gives answers to the research questions

CHAPTER

TWO

BACKGROUND

2.1 Cryptographic background

2.1.1 Hash Functions

A *Hash Function* is a deterministic function that maps an input message m into an output $O = H(m)$ that has a fixed length. O is often called *digest* or simply the *hash of* m.

Cryptographic hash functions (CHF) is a family of hash functions used in many information security applications, such as digital signatures. They are defined as hash functions that satisfy the following properties:

1. Efficiency: given a message m it is computationally quick to calculate its digest $H(m)$.

2. Pre-image resistance: given the digest $H(m)$ it is computationally infeasible to find m. H is a one way function.

3. Second pre-image resistance: given a message m and its digest $H(m)$ it is computationally infeasible to find another message m' such that $H(m) = H(m')$.

This is also the formal definition of One Way Hash Function (OWHF) given by Merkle [7].

In addition to these three properties, a CHF can also be *collision resistant*. This last property implies that it is computationally infeasible to find two messages (m, m') – with $m \neq m'$ – such that $H(m) = H(m')$. A hash function that satisfy this property is called Collision Resistant Hash Function (CRHF).

Hash functions are one of the foundation layers of the concept of blockchain. Typically, each protocol decides a cryptographic hash function that is used every time hashing is needed. Bitcoin uses SHA-256, while Ethereum uses KECCAK-256, a more recent alternative [1, 2].

2.1.2 Hash chains

A hash chain is the sequential application of a cryptographic hash function to a message m. For example, $H(H(H(H(H(m)))))$ is a hash chain of length five ap-

plied to the string m using the cryptographic hash function H, it can be shortened as $H^5(m)$. Image 2.1.1 visualizes this idea.

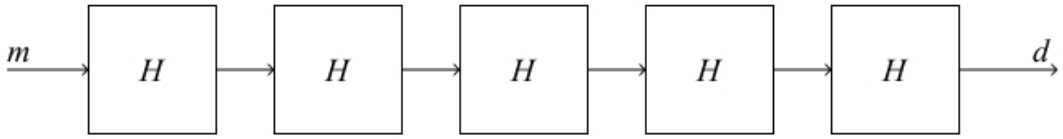

Figure 2.1.1: Example of a hash chain of length five.

This concept was proposed by Lamport as a way to securely store passwords on servers [8]. In his proposed protocol, the server just stores $H^n(p)$, where p is the password and n is relatively big number (e.g. 1000). When the user wants to authenticate, she sends $H^{n-1}(p)$ and the server computes $H(H^{n-1}(p))$ and checks if it corresponds to $H^n(p)$. If this check succeeds, the user is authenticated and the server replaces the stored value with $H^{n-1}(p)$. Next time she wants to authenticate again, she needs to send $H^{n-2}(p)$ and this value is checked against the previously sent digest $H^{n-1}(p)$. In this protocol, even if the transmission or the storage is not secure, the password is safe.

Blockchain technology uses this idea to form the immutable chain of blocks: each block contains the hash of the previous one. Modifying a block would result in changing its hash, this would break the chain since all the following blocks would have to be recomputed, changing each digest. As shown in Figure 2.1.2, it is a slightly different concept than the plain hash chain, since on each step, the hashing is done on the previous digest linked to raw data of the current block, so $d_n = H(b_n \mid\mid d_{n-1})$

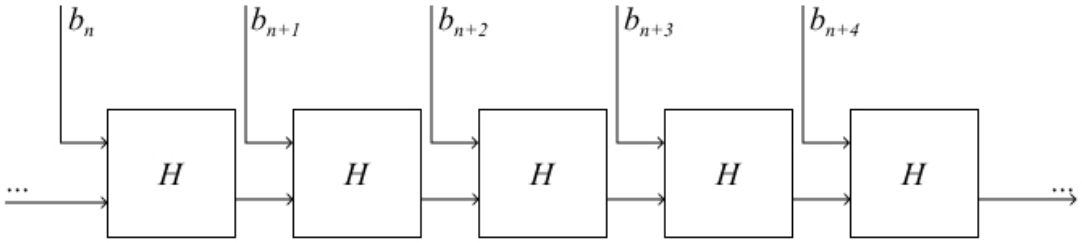

Figure 2.1.2: Example of how the hash chain concept is used in a blockchain.

2.1.3 Merkle trees

A Merkle tree is a data structure that generalizes the hash chain to efficiently prove membership of data. It is a binary tree in which each leaf node represents the hash of data, while intermediate nodes are computed as the hash of the two child nodes. Figure 2.1.3 is an example of a Merkle tree with 4 leaf nodes.

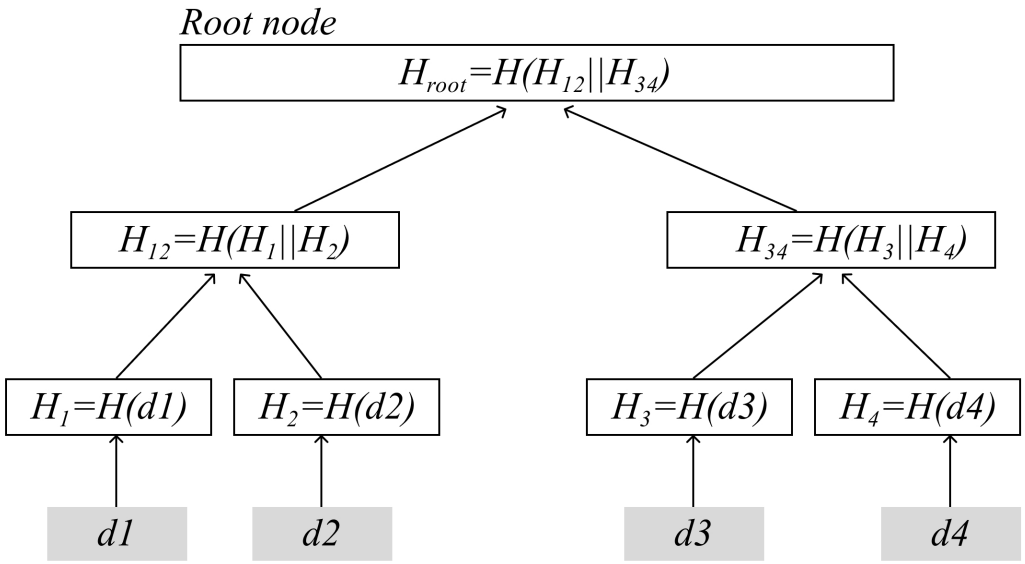

Figure 2.1.3: Example of a Merkle Tree with four leaf nodes.

Proving membership of data to a tree with 2^n leaves just requires n steps and n intermediate nodes. The algorithm simply reconstructs the tree starting from the node that must be checked, recalculating again the root node, also called *Merkle proof*. Data is proved to be part of the tree if the recalculated root node is equal to the given root node of the tree.

This exact concept is used in Bitcoin to prove membership of transactions to a block. Each block uses a Merkle tree where the leaf nodes are made by the hash of transactions. Inside each block it is stored the root of the Merkle tree. To prove that a particular transaction has been included in that block and not modified, it is necessary to rebuild the root node starting from the leaf of the transaction considered. This is particularly efficient for light clients. They do not store all the transactions, but just the block headers. If they need to prove that a certain transaction has been included in a block and not modified, they need to ask a full client just the intermediate hashes needed to rebuild the tree and not all the transactions of that block.

Ethereum uses a similar concept to store and verify membership of three different kind of information per block:

- State trie: it is a description of each account state used and modified at the current block.
- Transactions trie: similarly to Bitcoin, it describes each transaction included in the block.
- Receipts trie: it contains the receipts of the transactions included in the block.

All this data is stored in three different *Merkle Patricia Tries*. It is similar to a plain Merkle tree but it is faster to edit, does not depend on the order of data and has a limited depth.

2.1.4 Digital signatures

A digital signature is the digital counterpart of a handwritten signature in the physical world. It uses asymmetric cryptography to securely prove that a certain entity created a piece of digital data and did not modify it since signing.

This is achieved through the use of two keys:

- **Private key**: it is a random sample of bytes used by the signer to sign messages and, as the name suggests, is kept secret.
- **Public key**: it is obtained from the private key and shared to the verifier to check if the signature is valid.

The process of signing a message m works as follows:

1. The signer calculates $d = H(m)$.
2. The digest d is encrypted using the private key resulting in the generation of the signature σ, that is included in the message to prove authenticity.

To check if the signature is valid, the verifier needs the message m, the signature σ and the public key of the signer. Anyone with this data can independently check that the signature is authentic and thus that the entity who owns the private key related to the signature is the creator of the message.

There are multiple algorithms that make the public-key cryptography possible. In the sector of blockchain, the most used algorithm is **ECDSA** [9]. It is based on the discrete logarithm problem on elliptic curves over finite fields. On these curves, the problem of finding a k such that $P = kG$, where P is a known point on the curve and G is a generator point, has an exponential complexity.

Both in Bitcoin and in Ethereum, the addresses are obtained from ECDSA public keys in slightly different ways. Both networks use the same elliptic curve called *secp256k1*.

2.2 The blockchain

The concept of blockchain was first introduced by the pseudonym Satoshi Nakamoto when he/them presented Bitcoin, back in 2008. It makes use of a p2p network and the previously cited cryptographic technologies to create a distributed digital ledger without the presence of a centralized authority. Blockchain refers to an abstract concept that is implemented in a lot of different protocols, but Bitcoin and Ethereum remain by far the two most important implementations.

In all the blockchains, the distributed ledger is represented in form of blocks secured together as a hashchain. Data can be added to this ledger by anyone trough transactions. To include a transaction in a block, and so to write to the ledger, users must send it to the p2p network and pay a fee. The payment is done trough a *cryptocurrency* that has value just inside a specific network (e.g. Ether in Ethereum and Bitcoin in the Bitcoin network). The distributed ledger keeps track of the holdings of the cryptocurrency.

The consistency and the validity of what is written on the blockchain is based on the consensus. What is written in the ledger is correct if the majority of

the people participating in the network agrees on it. There are multiple ways of achieving so that are discussed in the next sections, but all these methods share the fact that they are computationally not efficient. All the computers participating in the network must perform the same calculations in parallel to check the data they receive and eventually agree on its correctness. This is expensive since millions of computers are performing the same calculations instead of working on different tasks in parallel.

This very inefficient computational pattern has become popular for the problem it solves: preventing double-spending without a central authority.

2.2.1 The double-spending problem

In the world of digital payments, the double-spending problem refers to the possibility, by malicious actors, to use the same unit of value more than once. Given the nature of digital money or assets, it is very easy to duplicate them and use them multiple times.

The traditional and easy way of solving this problem is by introducing a central authority that controls all the transactions. To make a payment, users need to go trough a central authority that guarantees the validity of the transaction. The central authority stores internally a list of holdings. This solution introduces the bottleneck of having a single point of failure. The central authority must be always online, must always behave honestly and must guarantee the security of the data it stores.

There were multiple historical attempts to remove, at least partially, this dependency from a central authority. A relevant attempt was ECash by David Chaum using RSA blind signatures in 1982 [10]. This protocol did not remove the concept of a central authority, but tried to reduce their power. Here the central authority, or bank, issues coins to users signing them with blind signatures. The peculiarity of this type of signature is that the signer does not know the message it is signing, so the coins are anonymous and untraceable. Users can send these digital coins between each other even if the bank is offline. The receiver then proceeds to send these coins to the bank to deposit the money into his bank account. In this protocol, double spending is prevented completely just if the bank is online but can also be detected offline if the same coin is sent multiple times to the same receiver.

In the concept of blockchain this problem is solved by letting all the participants of the network know and agree on who owns what and which transactions are taking place. If a user tries to submit a transaction that is spending money that she had already spent, the other participants of the network will not agree on including it into the blockchain and the money transfer will never occur.

For technical reasons it would be possible to encounter double spending also in blockchain transactions due to temporary forks. It is a situation in which there are two new blocks to include on the top of the chain at the exact same moment. This event will result in having two different chains, both valid. The blockchain protocols usually work in a way to always follow the longest available chain, so it is necessary to wait for the inclusion of new blocks and see which of the available chains outgrows the others. This is the reason why in blockchain transactions it is needed to wait for confirmations. A common heuristic in Bitcoin is to wait for

6 new blocks to be added on top of the desired one to be sure that it is final, this means waiting for around one hour. In Ethereum, with the recent migration to Proof of Stake, it is required a maximum of around 15 minutes to be sure that a block is finalized and can not be replaced.

2.2.2 Blockchain properties

The word blockchain is often accompanied with many adjectives to describe its properties. I here try to summarize and explain the most common ones:

- **Decentralized**: there is not a central authority that rule and control the network. No-one in the network has the power to control or change the rules. All the actors follow the protocol and have the same power.
- **Distributed**: the computation is performed by multiple distinct computers that interacts with each other through a p2p network. The failure of a machine does not interrupt the protocol.
- **Immutable**: it is not possible to alter the history of what has been written on the blockchain. Once a block is included and finalized it is not possible to modify it.
- **Permissionless**: everyone can actively participate in all the roles of the network without asking for permissions.
- **Permissioned**: in contrast to permissionless, permissioned blockchains requires that the actors receive explicit authorization to operate on the network. The most used example of this kind is Hyperledger Fabric[1].
- **Transparent**: everyone can independently download and read blockchain data. The only things needed are a computer and an internet connection. It is possible to get the data using specific software called clients, that are specific for each protocol.
- **Pseudoanonymous**: users that participate in the network does not need to prove their real identity. Every activity on the chain is linked to an address and not to a real person.
- **Account-based**: data is saved based on account. Each account has a balance that can be spent. In order to spend this balance, users need to prove they have a private key related to the account's address.
- **UTXO-based**: contrary to account-based blockchains, the model of UTXO just has the concept of transactions. Users need to prove they have the private key for unlocking the output of a transaction to spend its money. The balance of a user is simply the sum of the values of the outputs he can potentially unlock and spend.

[1]Hyperledger Fabric is a modular framework for enterprise blockchains: `https://www.hyperledger.org/projects/fabric`.

2.2.3 Consensus layer

As said in the previous sections, the fundamental concept in the blockchain technology is that the validity of the data stored is based on the consensus between the majority of the actors participating in the network. This section explains how this is achieved trough different types of *consensus layer*.

All public blockchains need to have a consensus layer to make the users agree on what is correct. There are many different types that have been invented in the years, but the two most important layers are *Proof of Work (PoW)* and *Proof of Stake (PoS)*.

2.2.3.1 Proof of Work

The Proof of Work consensus mechanism is the one introduced with Bitcoin back in 2008. It has been used also by Ethereum until the switch to Proof of Stake happened in September 2022.

The core concept in Proof of Work is that users need to solve complicated *puzzles* in order to write data on the blockchain. The solution to the puzzle shows that the user who found it has performed a lot of work and grants him the permission to write on the chain. All the other actors, that were not able to solve the puzzle, limit themselves to check the validity of the solution.

The PoW layer has a mechanism to algorithmically adjust the difficulty of the puzzles. Each blockchain protocol sets a fixed amount of time in between blocks, called inter-block time. In case of Bitcoin it is of 10 minutes. If the solutions to the puzzles are found faster than the expected inter-block time, the next puzzles will become more complicated. On the other hand, if the inter-block time is higher than the expected one, the following puzzles will be easier to solve.

The puzzle that must be solved in Bitcoin is the following: find a (double) hash of the data contained in the block's header such that is is lower than a threshold that depends on the current difficulty. It can be summarised as the following equation: $H(H(header)) < difficulty\ threshold$. The output of a hash function is a fixed amount of bits, so it can also be said that the puzzle is finding a digest that starts with a certain amount of zeros (and so that is a small number).

There is a field in block headers called *nonce* that is used for solving these puzzles. It is a 32 bits fields that can be freely set by the *miners* to cause the hash to change and meet the requirements of the puzzle.

The word *miner* comes from this concept. The users that try to find the solution to the challenge are called miners since they continuously try to find valid hashes that would provide them a precious reward.

Miners are economically incentivized to perform the hard work of mining since they receive a reward for each block mined and also all the transaction fees of the transactions included in the mined block.

In Bitcoin, these are the steps to follow to mine a block with PoW:

1. Choose the block's content. Include transactions, coinbase transaction and compute the Merkle root.

2. Choose a random 32 bit nonce and include it in the block's header.

3. Compute the double hash of the block's header. If the result is lower than the difficulty target the block is successfully mined. If not (and not all nonces has been tried), go to step 2.

4. If all nonces has been tried, go back to step 1.

PoW has proven to work well with Bitcoin, but it suffers of some limitations:

- High energy consumption: Bitcoin alone is estimated to use around 45.8 TWh per year for running the computers that perform mining. This is between the levels produced by the nations of Jordan and Sri Lanka [11].

- Risks of centralization: the hardware used for mining is becoming more and more expensive and it is a big entry barrier for small players. There is the risk that few big players together can take control of the blockchain. This happened in 2014, when the GHash.io mining pool had more than 51% of the hashing power of Bitcoin.

2.2.3.2 Proof of Stake

Proof of Stake is the most popular alternative introduced to reduce the drawbacks of Proof of Work. Here, the permission to append data to the blockchain is granted randomly between a set of *validators*.

To become a validator, a normal user has to *stake* a certain amount of coins. These coins are kept locked in a smart contract and can be used by the protocol as a warranty that the validator will behave correctly. If the validator fails to do his work, the coins staked can be *slashed*, like a fine to disincentive bad behaviours.

The idea is that miners, instead of spending money in computers and energy, spend their money by locking the coins. This has the advantage of not wasting energy and also gives more value to the coins themselves.

The most popular blockchain that uses this consensus layer is Ethereum. It switched from Proof of Work to Proof of Stake in September 2022, reducing emissions from 78 TWh/yr to just 0.0026 TWh/yr[2]. In Ethereum, users are required to stake 32 ETH, equivalent to 52,402 USD, to become validators.

2.2.4 51% attack

The 51% attack is the nature consequence of how the blockchain works. Everything is based on consensus, this attack simply gains the majority of it. It consists in the case in which a single entity owns more than half of the hashing power (in case of PoW) or more than half of the staked coins (in case of PoS).

In these cases, this single entity has the control of more than half of the network. In case of PoW, he can mine block faster that anyone else and so he can create a fork that will outgrow the current chain that in the meantime can be used to double spend money. In case of PoS, he has the control of the majority of attestations, so he can vote his own fork to be the preferred fork and again double spend money.

[2]More details about this can be found on the Ethereum website: `https://ethereum.org/en/energy-consumption/`.

The security of a blockchain against the 51% attack is related to how much it would cost to perform such an attack. It would cost billions of dollars to buy the infrastructure to perform a 51% attack against Bitcoin. The more a blockchain is used and the more it is secure against 51% attacks.

2.3 Ethereum

Ethereum is a decentralized, permissionless and account-based blockchain. It is the second blockchain in terms of market capitalization[3] and popularity. It was designed and developed by Vitalik Buterin and Gavin Wood since 2013. It went live in 2015. It was the first of many *Blockchain 2.0*, so a blockchain that allows users to create smart contracts using a Turing-complete programming language.

It was introduced to generalize the potentiality of the blockchain technology and make it available to other use cases than just money transfers. Thanks to the Turing-complete scripting language, it is possible to create applications that can implement any kind of logic that could benefit of a blockchain. Previously to Ethereum, it was common to create ad-hoc protocols for each use case. This created fragmentation and caused low adoption.

As stated by Vitalin Buterin in the announce post on the Bitcoin forum[4], the goal of Ethereum was to "to provide a platform for decentralized applications - an android of the cryptocurrency world, where all efforts can share a common set of APIs, trustless interactions and no compromises".

2.3.1 Ethereum as a state machine

The Ethereum blockchain can be seen as a state machine. There is a global state, called *World State*, that stores data related to each ever-used account. This state is modified trough transactions, after their inclusion in each block. Figure 2.3.1 visualizes this concept clearly.

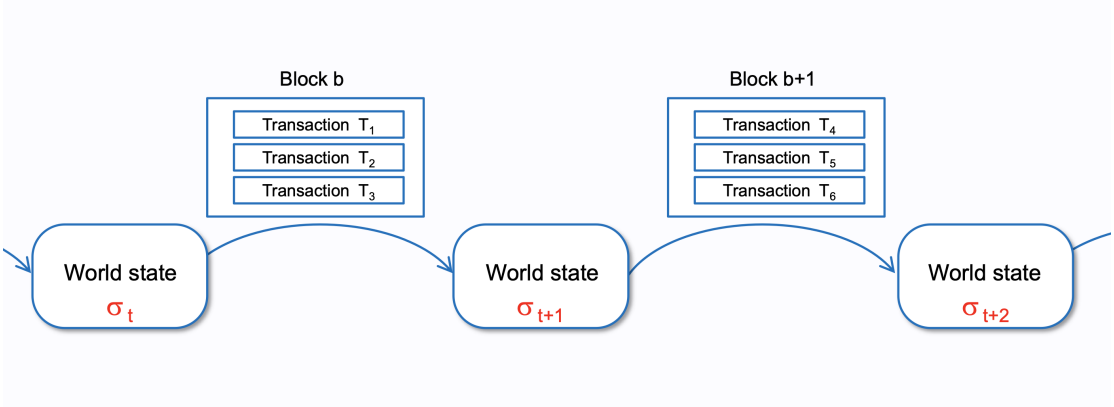

Figure 2.3.1: Ethereum visualized as a state machine [12].

The World State is a mapping between account address and account state. There are two types of account: Externally Owned Account (EOA) and Contract

[3]Market capitalization refers to the total value of all the cryptocurrency emitted.

[4]The thread can be found at the following link: `https://bitcointalk.org/index.php?topic=428589.0`

Account (CA). EOA is the traditional account based on a private key, used to generate transaction on the chain. It needs to store on the World State just two pieces of data: a nonce and its balance. The Contract Account refers to smart contracts, they can not generate transactions, they can just react to them. On top of nonce and balance, it needs to store on the World State also the storage hash and the code hash. These two hashes are used as pointers to retrieve the storage of the smart contract, so the place in which the software stores variable values, and the actual code to run when invoked from a transaction. Figure 2.3.2 shows the division of the World State.

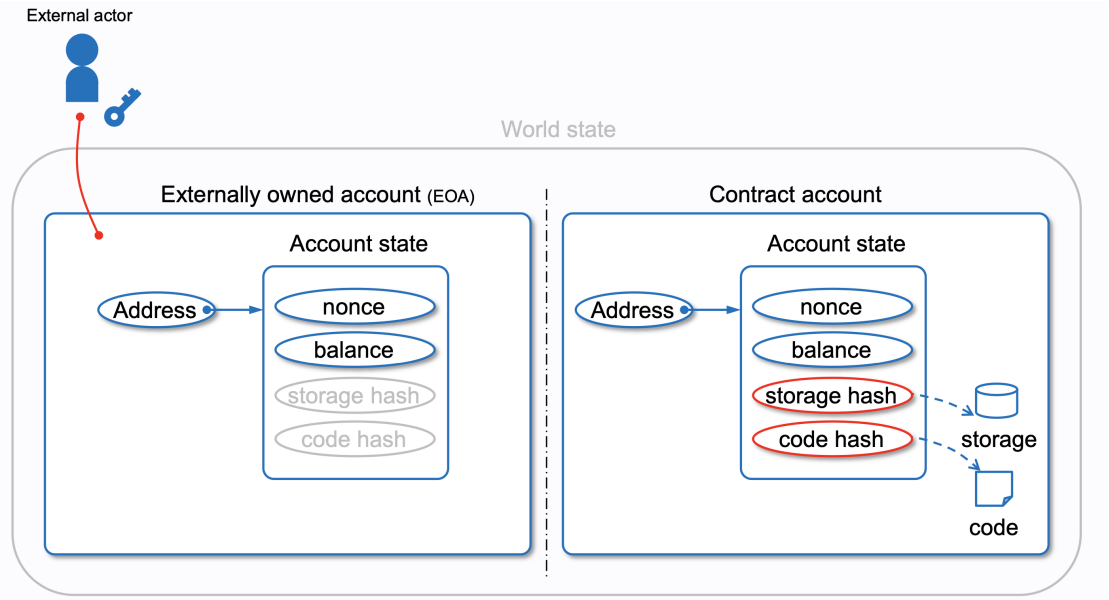

Figure 2.3.2: Ethereum World State visualized [12].

This World State needs to describe the state of millions of accounts and update it every 12 seconds, the average inter-block time in Ethereum. To do so, it uses a Merkle Patricia Trie that is optimized for this use case.

2.3.2 Ethereum Smart Contracts

Smart contracts are pieces of code that can be invoked trough transactions and that can modify the World State following a custom logic. They are written in high-level programming languages compiled to a stack-based bytecode called `Ethereum Virtual Machine code` or `EVM code`.

Smart contracts can do many things, such as:

- Store data on the blockchain: each smart contract has a storage in which it can store persistent data represented by *state variables*. Storing persistent data has a cost and is paid through *gas*.
- Interact with other smart contracts: during its execution, a smart contract can call other smart contracts and use their return values.
- Emit events that are indexed: it is possible to use the `LOG` opcode to store persistent data that is indexed by the Ethereum clients. This indexed data can be easily retrieved by traditional software through fast RPC calls.

- They can handle failures: it is possible for a smart contract to gracefully handle failures both totally or partially. Everything that is done in a failed branch of a transaction is not persistently stored on the blockchain.
- Access context data: it is is possible to access block and transaction data inside the execution of the code. It is very common to use the address of the transaction's sender to manage access to certain restricted functions.

But at the same time, smart contracts on Ethereum have some limitations that limit their applicability to real-world scenarios:

- No access to external data: from the code of a smart contract it is impossible to know everything that is outside of the blockchain. It is not possible to call a REST API to get information such as stock prices, weather or any other data. This is partly solved by *oracles*, that are specific smart contracts that can be queried to get these kind of information. These smart contracts are updated by transactions sent periodically by an external entity. It is an expensive operation and creates a potential single point of failure.
- Cannot connect to any other software: as said before, it is not possible for a smart contract to interact with other software that are not other smart contracts on the same chain. The link between traditional systems and smart contracts is always asynchronous, e.g. they can not cooperate during the execution of a transaction.
- No infinite loops: smart contracts can not execute endless loops. This is an obvious limitation, since the execution of a transaction must finish in order to include its outcomes in the World State.
- Hard to upgrade: it is not possible to change the code of a smart contract once it is deployed on the blockchain. There are some ways of doing it as explained in Chapter 6 but it is not trivial.
- Cannot initiate transactions: smart contracts are passive entities on the blockchain that can be invoked by external users. They can not create transactions by their own.
- Cannot accept randomness: the execution of a smart contract must be completely deterministic. It must be reproduced on each node participating in the network to attest its correct execution and so it can not accept any real randomness.
- Cannot store sensitive or private data: since Ethereum is a permissionless blockchain, everything stored there is public. Smart contracts can not hide anything.

2.3.2.1 Gas

The concept of gas has been introduced in Ethereum to have a unit of measure for pricing smart contract executions. Miners involved in the protocol must run the smart contracts' code on their machines to calculate the World State differences.

Malicious users could potentially send computationally intense transactions to congest the network. This is avoided in Ethereum using gas, which makes this kind of attack economically not convenient.

Each EVM opcode has an associated price that indicates how much gas it burns when it is executed. The amount of gas each transaction can burn is finite, so it is impossible to encounter infinite loops. If a user tries to code an infinite loop or even just an expensive code execution, the transactions will fail with the error *out of gas*.

Ethereum gas has a variable price that changes based on the congestion of the network. It is paid in Ethers by the sender of the transaction. Before EIP-1159[5], the price of gas was set by each user when he built the transaction. The fee that the transaction sender had to pay was simply $gasPrice * gasUsed$. Transactions with higher *gasPrice* were prioritized by miners since they rewarded more money for the same computational effort.

After EIP-1159, since August 2021, the gas fee has been split into two parts: *base fee* and *priority fee*. The *base fee* is a gas price calculated by the network based on congestion. The *priority fee* is an additional value given per unit of gas used to incentivize miners to include a specific transaction. For each transaction, $baseFee * gasUsed$ Wei[6] are burnt and $priorityFee * gasUsed$ Wei are given to the miner.

2.3.2.2 EVM

EVM stands for Ethereum Virtual Machine and is the virtual architecture of the computer that runs the smart contracts' bytecode. The EVM is a stack based machine composed of a stack, a volatile memory and a program counter. It can access the World State both to read and to write persistent data. Figure 2.3.3 shows the schema of the EVM.

The whole Ethereum network can be seen as a single instance of an EVM machine that runs the code of smart contracts invoked in transactions. In practice, this is done by every single node connected to the network, using a potentially different EVM implementation.

2.3.2.3 Solidity

Ethereum smart contracts can be programmed in many high-level programming language. Historically, two languages have been relevantly used: Viper and Solidity. Nowadays almost every smart contract is developed in Solidity, it is by far the most common language used for this purpose.

Solidity is a statically-typed programming language. The syntax is very similar to popular languages such as JavaScript. It works like an object oriented language: a contract is defined as a class that has a constructor and public or private attributes and methods.

It is compiled to EVM bytecode using *solc*, a compiler written in C++. After the compilation with the `--bin` flag, solc produces as an output the EVM bytecode that can be deployed to the Ethereum blockchain.

[5]EIP-1159 is an Ethereum Improvement Proposal that changed the fee market. It can be read here: `https://eips.ethereum.org/EIPS/eip-1559`.

[6]Wei is a fraction of a Ether. 1 Wei is 10^{-18} ETH.

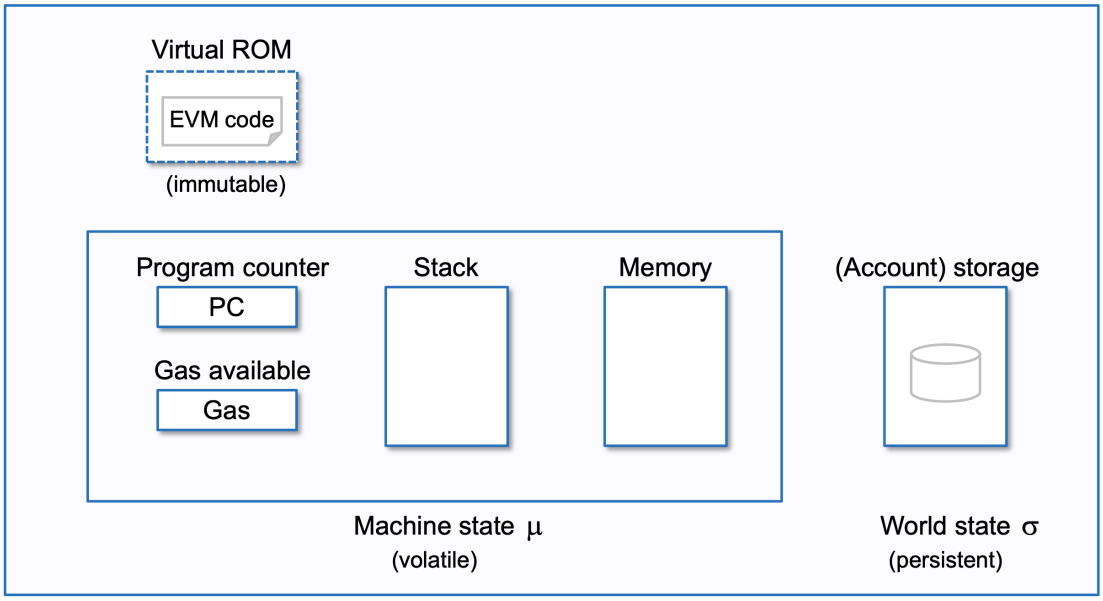

Figure 2.3.3: The Architecture of the Ethereum Virtual Machine (EVM) [12].

This bytecode includes both the runtime code and the deployment code. The first one is the actual code that is stored on the chain, while the second part is the code needed to initialise the contract. Listing 2.1 shows an example of a basic smart contract taken from the Ethereum documentation. Listing 2.2 shows the result of the compilation of this contract. These are the 541 bytes that implement the logic of the Solidity code and that are stored on the blockchain.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  contract SimpleStorage {
5      // State variable to store a number
6      uint public num;
7
8      constructor(uint _num) {
9          num = _num;
10     }
11
12     // You need to send a transaction to write to a state
       variable.
13     function set(uint _num) public {
14         num = _num;
15     }
16
17     // You can read from a state variable without sending a
       transaction.
18     function get() public view returns (uint) {
19         return num;
20     }
```

```
}
```

Listing 2.1: Example of a simple Solidity smart contract that stores a variable on the blockchain and allows edits to it.

```
0x6080604052348015610000f575f80fd5b5060405161021d38038061021d83
3981810160405281019061003191906100745565b805f81905550610009f56
5b5f80fd5b5f81905091905056 5b6100538161004156 5b811461005d575f80
fd5b50565b5f8151905061006e8161004a565b92915050565b5f6020828403
1215610089576100886100 3d565b5b5f61009684828501610060565b915050
92915050565b610171806100ac5f395ff3fe__6080604052348015610000f57
5f80fd5b506004361061003f575f3560e01c80634e70b1dc14610043578063
60fe47b1146100615780636d4ce63c1461007d575b5f80fd5b61004b61009b
565b6040516100589190610c9565b60405180910390f35b61007b60048036
038101906100769190610110565b6100a0565b005b6100856100a9565b6040
516100929190610c9565b60405180910390f35b5f5481565b805f81905550
50565b5f8054905090565b5f81905091905056 5b6100c3816100b1565b8252
5050565b5f6020820190506100dc5f8301846100ba565b92915050565b5f80
fd5b6100ef816100b1565b81146100f9575f80fd5b50565b5f8135905061010
0a816100e6565b92915050565b5f602082840312156101255761012461 00e2
565b5b5f61013284828501610 0fc565b9150509291505056__fea264697066
7358221220ed271d25e577fb9fa9b0b77e93485c035d8fc87f28d3c0247cff
80d69910e46064736f6c63430008150033
```

Listing 2.2: EVM bytecode derived from compiling the contract shown in Listing 2.1. There are three parts divided by two underscores (__). The first part is the deployment code, the second part is the runtime code and the last part is the CBOR-endoded metadata. This is what is stored on the blockchain and what can be retrieved.

2.3.2.4 Token contracts, ERC20 and ERC721 standards

The most frequent use case of smart contracts are tokens. Tokens can be of two types: fungible or non-fungible. Fungible tokens are divisible, like money, while non-fungible tokens can not be split, like a piece of art.

At low level, fungible tokens are simply a mapping from account to the balance of the token that the account can spend. Non-fungible tokens, on the opposite, are simply a mapping from the token id to its owner. Those mappings are stored on the blockchain, and specific functions allow the owners of the tokens to transfer them.

Since the usage of tokens was so popular, the Ethereum community standardized them in two standards: ERC20[7], for fungible tokens, and ERC-721[8] for non-fungible tokens.

ERC20 specifies nine functions and two events that allows users to transfer their tokens or allow other users to spend them. Famous ERC20 smart contracts are stable coins (e.g. USDT, USDC) or DAO tokens (e.g. Uniswap, Lido DAO, Aave).

[7]Link to the EIP-20 official page: `https://eips.ethereum.org/EIPS/eip-20`.
[8]Link to the ERC-721 official page: `https://eips.ethereum.org/EIPS/eip-721`.

The same applies to ERC721, which specifies nine functions and three events that allows users to safely manage their non-fungible tokens. Popular ERC721 tokens includes CryptoPunks, CryptoKitties and the Ethereum Name Service.

The usage of standards allows developers to build applications that can interact with any token. For example, it is possible to develop decentralized exchanges and wallets that work with any smart contract that correctly implements one of the two standards.

2.3.3 Ethereum clients

Ethereum is just a definition of a protocol. It defines in details how participants of the network must behave. All the logic is implemented by complex software called *clients*. A computer that runs a client and is connected to the network is called an *Ethereum node*.

There are multiple different practical implementations of the Ethereum protocol, all the different kind of software interact with each other thanks to the protocol definition. It is actually very important for the network to have a diversity of the type of clients running simultaneously. This ensures that the network will still work in case a specific client receives a wrong update. Imagine the scenario in which there is only one client version in all the network and it receives a wrong update. The blockchain would stop to work correctly and it would be very hard to bring it back up with the correct state.

After the switch to PoS, clients are divided into two: execution and consensus clients. The execution client is responsible for managing transactions, executing smart contracts and updating the world state. The consensus client is responsible for running the proof-of-stake logic. These two clients interacts with each other with the *Engine API*[9].

Clients can be used in three different configurations:

- Light node: these nodes do not download blocks' data, they just download the headers. They can ask full or archive nodes other data if needed (transactions or world state). The purpose of light nodes is to give users the possibility to access blockchain data without the need of powerful machines. Light nodes do not participate in blocks validation.

- Full node: when run as a full node, the client download and validates all the history of the blockchain block-by-block. They can both start from the genesis block (block zero) or from a more recent block. After the verification, full nodes do not store all the data they download, they just keep in memory the most recent blocks that can be useful for maintaining the network. Full nodes can participate in blocks validation.

- Archive node: this is the most complete type of node. It behaves similarly to a full node but it does not delete data after verification. To run an archive node, very powerful machines are needed. To sync and store the history of Ethereum using Geth, it is estimated to take around 12TB of disk space

[9]The specification of the Engine API can be found here: `https://github.com/ethereum/execution-apis/blob/main/src/engine/common.md`.

(July 2023) and more than a week of time. There are clients optimized for storing all the history of the chain, such as Erigon, that takes just 3TB.

It is possible to interact with Ethereum clients through Inter-Process Communication (IPC) using JSON-RPC[10]. This is a protocol that defines how data must be sent to the clients and how clients should reply. All the communication is made with stateless network calls in which data is serialized as JSON. There are many libraries that implement this protocol in different programming languages, easing the process of communication with clients.

According to Ethernodes.org[11], 54.9% of the execution clients are running with Geth, followed by Nethermind (24.21%) and Erigon (10.48%). For the consensus clients, the most used is Prysm (45.79%) followed by Lighthouse (32.94%) and Teku (14.82%)[12].

2.4 Graph databases

Graph databases (GDBs) are databases optimized for storing and retrieving data structured as a graph. A graph is represented as nodes connected trough edges. Typically, graph data is highly connected, so moving between nodes by following edges is what graph databases optimize. In contrast, relational databases (RDBMS) needs to perform expensive joins to achieve so. It is possible to store graph data also with other kind of databases, but in graph databases, the storage level is optimized to perform graph traversal.

2.4.1 Dgraph

Dgraph [6] is an open-source distributed graph database written in Go. It features horizontal scalability while offering the benefits of ACID transactions.

Internally, a Dgraph cluster is composed of multiple processes that can be run on different machines. There are two types of nodes in a Dgraph cluster: Alphas and Zeros. A Zero instance is responsible for coordinating the cluster, managing distributed transactions and re-balancing data across the servers. An Alpha instance is responsible for storing data and indices. A cluster must have at least one Zero instance and one Alpha instance. To get the data, users can query directly the Alpha instances. Figure 2.4.1 shows an example of the composition of a cluster. Inside the cluster, consensus between the nodes is ensured using the RAFT Consensus Algorithm [13]. This protocol uses periodical *elections* to ensure that at any given time just one entity of the cluster is the *leader* and all the others are *followers*[13].

Data in Dgraph is stored using the concept of *posting list*. In Dgraph, the unit of data is stored as a composition of three values: `<subject> <predicate> <object>`.

[10]The specification of the execution API can be found here: `https://ethereum.github.io/execution-apis/api-documentation/` and the one for the consensus API can be found here: `https://ethereum.github.io/beacon-APIs/`.

[11]Data has been obtained from this dashboard: `https://ethernodes.org/`.

[12]Data retrieved from: `https://github.com/sigp/blockprint/blob/main/docs/api.md`.

[13]An interactive visualization of how the RAFT consensus algorithm works can be found here: `https://raft.github.io/`

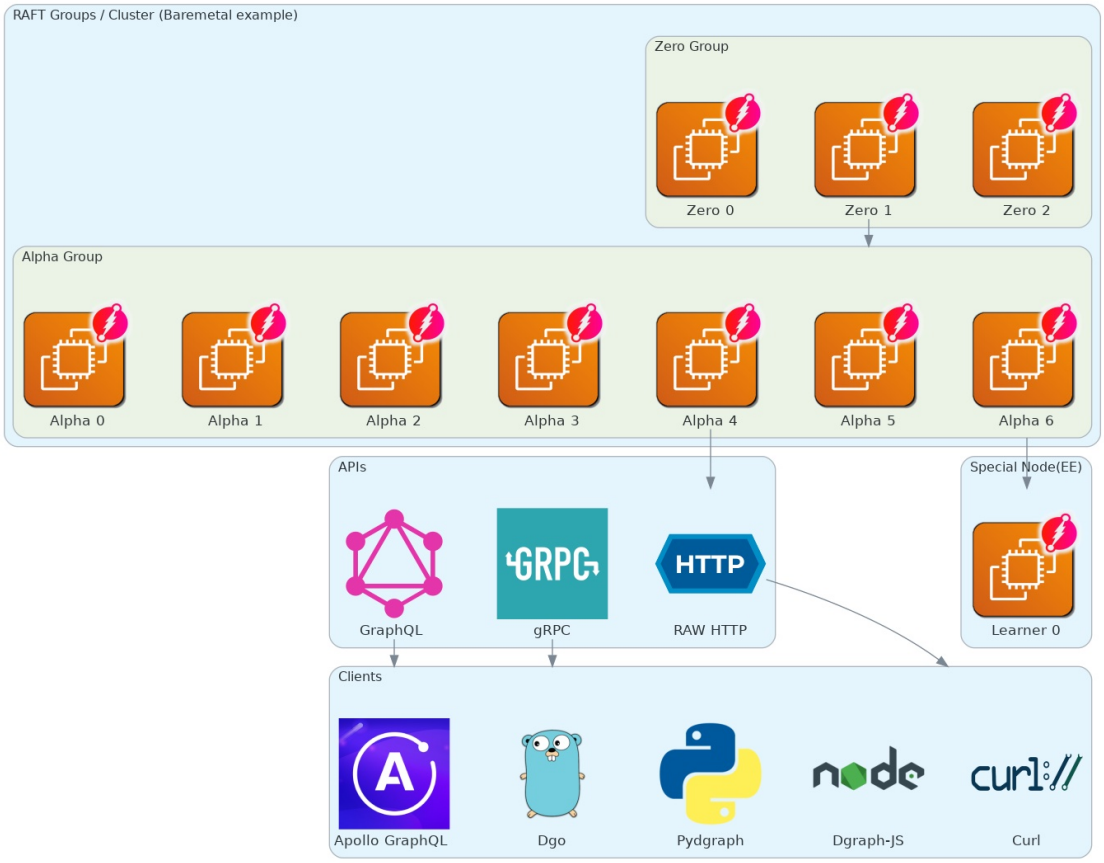

Figure 2.4.1: The architecture of a Dgraph cluster with three Zeros and seven Alphas. Taken from Dgraph official documentation.

Every node receives an unique integer id used as the subject. So, for example, a node representing a person with *uid* 1 and name "John" is stored as: `<1> <name> <John>`. The actual storage is done by *Badger*, a key-value database. Each unique `<subject> <predicate>` is stored as a key with the list of corresponding `<object>` as values. So, to complete the example, the person would be stored in the underlying key-value DB as `key: <name, 0x1>` and `value: <"John">`. Each value is a posting and is stored together with other values sharing the same subject and predicate, effectively forming a posting list.

Sharding in Dgraph is based on *tablets*: a tablet can not be split into multiple servers. A tablet groups data based on predicates. So, reusing the previous example, all the values of the predicate `name` form a tablet and so are stored in the same server. Having all the same predicates stored in a single machine allows to perform faster queries since the data to filter is all in one place and there is no need to perform network calls.

Indexes are implemented using the same concept of posting lists. The only difference is that indices are stored using `<predicate, token>` as keys instead of `<predicate, subject>`. Token is the piece of indexed data and depends on the type. For example, supposing a string `"Davide Aimar"` must be indexed using the *term* index, two posting lists will be defined: one with `key: <name,"Davide">` and one with `key: <name,"Aimar">` containing as values the references to all the uids that have that word in the name.

CHAPTER

THREE

PREVIOUS WORK

Data extraction and indexing from various blockchains is an essential topic for making Web3 and dApps working, there are different projects that have tried to address this problem. A lot of work has been done by companies, whose source code and methodology are not publicly available. There are also some open source or academic attempts that I use as a comparison with my work.

These projects target three main categories:

- Web3 and dApps developers, that need data to feed their applications.
- Data analysts, that need to analyse historical blockchain data.
- Blockchain users, that need to see the results of transactions.

In the next sections I list the state of the art tools available.

3.1 Etherscan

In the world of blockchain, *explorers* play an important role. They are web-based tools that allow users to see every kind of information with a user-friendly and interactive interface.

Etherscan[1] is the reference point for accessing data about the Ethereum blockchain. It is the most used Ethereum explorer that lets people browse historical data through a web interface. Here users can easily explore transactions, internal calls, token transfers and everything else related to the Ethereum protocol. It is useful for inspecting singular operations, but it can not be used for large-scale analysis.

One of the most important services they offer is the verification of smart contracts, they host 461,261[2] source codes (as of 17 May 2023) that have been verified to match exactly the deployed bytecode on the Ethereum chain.

The process of verification consists in providing Etherscan the exact source code of a Smart Contract, the version of the compiler used, the license selected and

[1] Online blockchain explorer available at `https://etherscan.io/`

[2] This data was calculated using the CSV file exported from `https://etherscan.io/chart/verified-contracts`

the contract address to verify. With this information, Etherscan tries to compile the given data and check if the resulting bytecode equals the one deployed on the blockchain. If the check succeeds, then the source code correctly describes the bytecode of the smart contract. There are plugins for the most used development tools like Remix[3] or Hardhat[4] that ease this process of verification.

This data is helpful for understanding the semantics of the code deployed, since it is hard to get valuable insights by just looking at the raw bytecode. Many studies are based on these verified contracts.

Etherscan has evolved from being just an Ethereum explorer. Etherscan developers use the Ethereum blockchain data and created API endpoints and offer these data to users for a fee. These API endpoints include the standard Ethereum JSON-RPC interface. There are also other more advanced indexes not supported by the standard RPC, e.g. transactions by address, ABI of contracts, token transfers.

On top of that, they also provide live and interactive charts[5] about historical Ethereum data.

The same company that is behind the Ethereum Etherscan explorer applied the same logic and technology to other EVM-compatible chains, like Polygon[6] or BNB Smart Chain[7].

It is important to note that, although they provide almost all the possible available Ethereum data, they have not shared technical details about how this data is extracted or how it is indexed. Users need to trust the company. Another problem is that the data they offer is not easy to get and manipulate for large-scale analysis. Their API service is meant to be used by developers for building dApps. It would be too expensive to use it for data analysis on all the history of the chain.

3.2 The Graph

The Graph [14] is a decentralized indexing protocol for blockchain data. It allows users to get structured on-chain data from other users via a GraphQL[8] interface.

All the data is organized in so-called **subgraphs**, which are independent data collections that index a small subset of a blockchain network. A common pattern is that a subgraph indexes data from one or a set of few smart contracts all part of a common protocol, like Uniswap[9]. All the available subgraphs can be found

[3]Remix is an online IDE to develop Ethereum smart contracts. It is possible to verify contracts using the *Etherscan Contract Verification* plugin. A detailed guide can be found here: `https://medium.com/@ezeamaka2/how-to-verify-smart-contracts-on-etherscan-in-remix-ide-92f6354933b4`.

[4]Hardhat is a popular Ethereum development environment. It has the *hardhat-verify* plugin that allows to verify contracts easily: `https://hardhat.org/hardhat-runner/plugins/nomicfoundation-hardhat-verify`.

[5]Interactive charts are available at `https://etherscan.io/charts`

[6]Explorer for the Polygon network `https://polygonscan.com/`

[7]Explorer for the BNB Smart Chain `https://bscscan.com/`

[8]GraphQL is an open source query language managed by Linux Foundation, originally created by Facebook.

[9]Uniswap is a decentralized cryptocurrency exchange `https://uniswap.org/`

on the *Graph Explorer*[10].

The underlying protocol is composed of multiple actors:

- Developers: people with technical knowledge that develop the needed code for creating and maintaining the indexes. As of now, the most important pieces of code needed are the mappings from Ethereum events to the stored data, written in AssemblyScript[11], and the subgraph manifest, a structured description of all the parts needed by the subgraph in YAML format.
- Indexers: they are responsible for operating a node. This implies indexing the data following a subgraph specification and serving queries to users.
- Curators: they are in charge of finding the best subgraphs to be indexed.
- Delegators: they secure the network by locking economical value to certain indexers they choose, giving them the possibility to serve more queries.

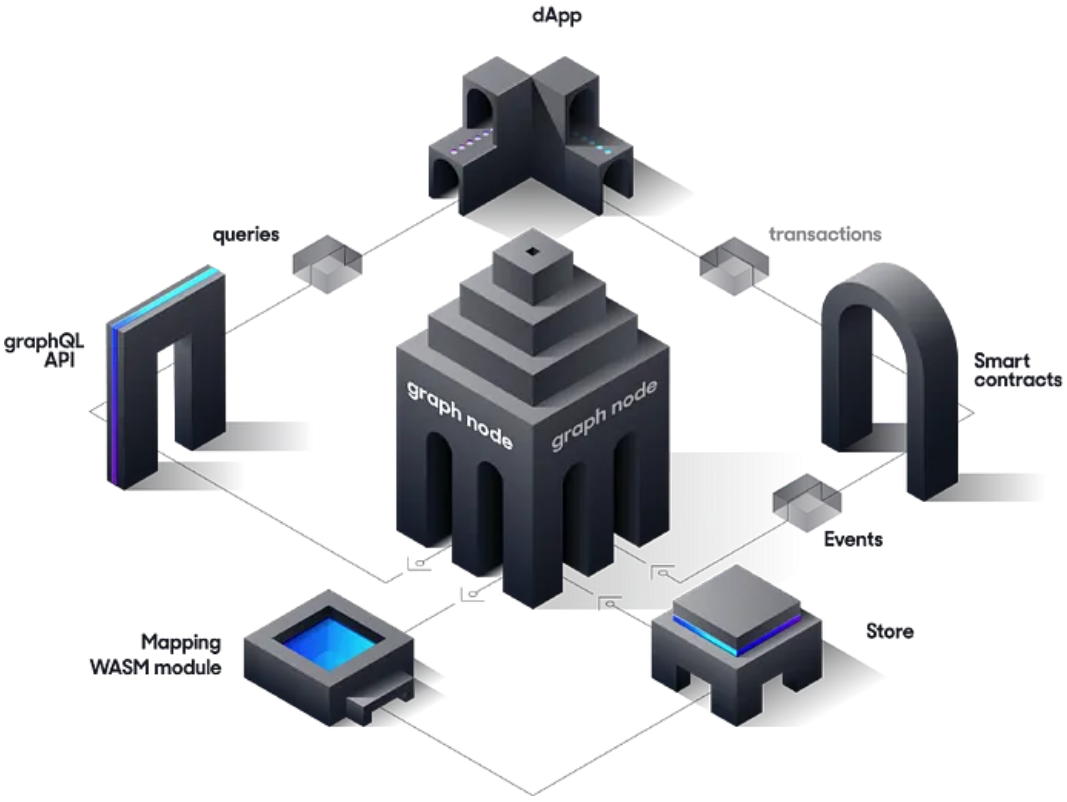

Figure 3.2.1: Data flow of The Graph indexing protocol[12].

All these actors are economically motivated to perform well. This is achieved via a token economy where the GRT is the currency. It is implemented on the Ethereum chain with an ERC-20 smart contract[13].

[10]The Graph Explorer collects all the subgraphs of the protocol, it is available at: `https://thegraph.com/explorer`

[11]AssemblyScript is a Typescript-like language that is compiled to WebAssembly.

[12]Source: `https://thegraph.com/docs/en/about/`

[13]`https://etherscan.io/token/0xc944e90c64b2c07662a292be6244bdf05cda44a7`

In order to index and serve queries, indexers have to stake at least 100,000 GRT tokens (roughly equal to 12K USD with the current change). These tokens can be slashed in case the indexer behaves maliciously. The more tokens the indexer stakes and the more queries it can serve. At the same time, indexers are rewarded with GRT tokens in two ways: query fees and annual rewards based on amount of queries served.

According to the specification of the subgraph file[14], the only allowed source of data are Ethereum contracts and mappings are restricted to logs. Figure 3.2.1 shows the flow of data in the protocol. In most cases, this is enough for dApps, since typically all the smart contracts are written in such a way that they emit meaningful events when things happen.

On the other hand, it is not possible to index all the other kind of information for performing other analysis, such as block data, contract deployments, transactions, contracts destruction, etc. It is also not possible to extract data that was not meant to be extracted, since the emitted events are pieces of information that the developers of the smart contracts explicitly wanted to expose and index.

As of today, the protocol continues to rely on a centrally hosted service that uses the subgraphs' logic and code to index data, but queries are served from this centralized server for free. This should change later in 2023, the network should slowly migrate from this centralized service to the decentralized protocol once the quality of the service will be comparable[15].

The Graph is the first attempt to decentralize indexing of blockchain data, it is still a project with a lot of work behind the scene. It is the most promising mechanism to make Web3 and dApps not dependent on centralized data ingestion services.

3.3 Ethereum-ETL

Ethereum-ETL [15] is an open-source tool for extracting data from the Ethereum blockchain following the **Extract-Transform-Load** pattern. It is written in Python and can be used through a CLI.
Raw data can be extracted to CSV or JSON files using these commands:

- `export_blocks_and_transactions`: it calls `eth_getBlockByNumber` RPC and maps the response to two files containing blocks and transactions.
- `export_token_transfers`: it calls `eth_getFilterlogs` applying a filter with the first topic set to the Keccak-256 hash of the Transfer event signature. Transfers are stored in a single file without distinction between ERC20 or ERC721.
- `export_traces`: it stores the internal transactions calling the `trace_block` method.

[14]Specification available at: `https://github.com/graphprotocol/graph-node/blob/master/docs/subgraph-manifest.md#15-data-source`

[15]This transition is explained in details here: `https://thegraph.academy/developers/sunsetting-the-hosted-service/`

All other kind of data can be extracted starting from the files that these previous commands store. It is possible to further extract:

- Transaction receipts and logs starting from transaction hashes exported from `export_blocks_and_transactions`.
- Contracts data, starting from the traces stored with `export_traces`.
- Token contracts with metadata, starting from contracts (this requires two previous steps of extraction).

Ethereum-ETL also provides the command `export_all` to perform extraction of all the common data in a single step (blocks, transactions, receipts, logs, contracts, token transfers and token metadata). However, this command is not present in the documentation and it can not extract contracts deployed by other contracts, just the ones deployed by Externally Owned Accounts.

3.3.1 Google BigQuery public dataset

Ethereum-ETL also allows for streaming data from an Ethereum node to the console. This functionality is used to ingest data into the popular Ethereum Google BigQuery public dataset[16]. This dataset is organized in six tables that I report here in Tables 3.3.1 to 3.3.6 since it is a good representation of the raw data that can be extracted from an Ethereum node.

[16]The Google's BigQuery dtaset can be explored here: `https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics`

Attribute	Type	Required
timestamp	Timestamp	Yes
number	Integer	Yes
hash	String	Yes
parent_hash	String	No
nonce	String	Yes
sha3_uncles	String	No
logs_bloom	String	No
transaction_root	String	No
state_root	String	No
receipt_root	String	No
miner	String	No
difficulty	Numeric	No
total_difficulty	Numeric	No
size	Integer	No
extra_data	String	No
gas_limit	Integer	No
gas_used	Integer	No
transaction_count	Integer	No
base_fee_per_gas	Integer	No
withdrawals_root	String	No
withdrawals	Record	Repeated
- index	- Integer	- No
- validator_index	- Integer	- No
- address	- Integer	- No
- amount	- Integer	- No

Table 3.3.1: Schema of table *Blocks* on the Google BigQuery public dataset. Taken from the official docs.

Attribute	Type	Required
log_index	Integer	Yes
transaction_hash	String	Yes
transaction_index	Integer	Yes
address	String	No
data	String	No
topics	String	Repeated
block_timestamp	Timestamp	Yes
block_number	Integer	Yes
block_hash	String	Yes

Table 3.3.2: Schema of table *Logs* on the Google BigQuery public dataset. Taken from the official docs.

Attribute	Type	Required
address	String	Yes
bytecode	String	No
function_signature	String	repeated
is_erc20	Boolean	No
is_erc721	Boolean	No
block_timestamp	Timestamp	Yes
block_number	Integer	Yes
block_hash	String	Yes

Table 3.3.3: Schema of table *Contracts* on the Google BigQuery public dataset. Taken from the official docs.

Attribute	Type	Required
transaction_hash	String	No
transaction_index	Integer	No
from_address	String	No
to_address	String	No
value	Numeric	No
input	String	No
output	String	No
trace_type	String	Yes
call_type	String	No
reward_type	String	No
gas	Integer	No
gas_used	Integer	No
subtraces	Integer	No
trace_address	String	No
error	String	No
status	Integer	No
block_timestamp	Timestamp	Yes
block_number	Integer	Yes
block_hash	String	Yes
trace_id	String	No

Table 3.3.4: Schema of the *Traces* table on the Google BigQuery public dataset. Taken from the official docs.

Attribute	Type	Required
token_address	String	Yes
from_address	String	No
to_address	String	No
value	String	No

Table 3.3.5: Schema of table *Token_transfers* on the Google BigQuery public dataset. Taken from the official docs.

Attribute	Type	Required
hash	String	Yes
nonce	Integer	Yes
transaction_index	Integer	Yes
from_address	String	Yes
to_address	String	No
value	Numeric	No
gas	Integer	No
gas_price	Integer	No
input	String	No
receipt_cumulative_gas_used	Integer	No
receipt_gas_used	Integer	No
receipt_contract_address	String	No
receipt_root	String	No
receipt_status	Integer	No
block_timestamp	Timestamp	Yes
block_number	Integer	Yes
block_hash	String	Yes
max_fee_per_gas	Integer	No
max_priority_fee_per_gas	Integer	No
transaction_type	Integer	No
receipt_effective_gas_price	Integer	No

Table 3.3.6: Schema of table *Transactions* on the Google BigQuery public dataset. Taken from the official docs.

It is possible to query these tables using SQL syntax, they can be joined on equal fields. There is the possibility to query this database for free up to a certain monthly limit of processing storage and amount of data extracted.

3.4 Dune Analytics

Dune Analytics[17] is a company that provides tools to query and visualize data from multiple blockchains. They support Bitcoin, Solana, Ethereum and other 8

[17]Blockchain analytics platform `https://dune.com/home`

EVM-compatible chains.

Their application is web-based. With their web-app it is possible to create queries using SQL syntax and visualize results in interactive charts. Multiple queries can be collected together to create dashboards[18].

3.4.1 Data architecture

Blockchain data was initially managed using PostgreSQL and SparkSQL until 2022, year in which they released their own engine called DuneSQL [19]. The migration to this technology is currently ongoing.

They started storing and indexing data with PostgreSQL. In this DBMS, entries are stored in pages following a row-oriented strategy, which means that all the attributes of a row are stored adjacently. This strategy is beneficial when it is necessary to retrieve all the attributes of a row while querying. However, it leads to poor performances when filtering for specific attributes, since the database has to load many bytes containing irrelevant data (i.e., all the other attributes of the row that are not used). To avoid this, it is possible to create indexes on columns. Indexes avoid the need to read and filter data. They provide direct access to the location of the requested data. Indexes provide very good query performances, but can be hard and slow to maintain when the amount of data grows. At Dune Analytics they tried this approach with traditional relational database (PostgreSQL) combined with many indexes, but they had to drop it stating that "the size of the datasets were so huge that the database was struggling to fully support them"[20].

They came up with DuneSQL, a query engine built specifically for managing blockchain data. It is a fork of Trino, an open source distributed SQL query engine designed to query data from heterogeneous sources. The DuneSQL fork is not open-source, so technical information can only be deducted from their blog articles or statements on the support community. The main features they added are *varbinary* data type for storing addresses and hashes, as well as *uint256* support for EVM data.

Data is physically stored on AWS S3 buckets using the Apache Parquet storage format[21]. This storage system is a mix between column and row oriented. Data is split in files based on rows, inside these files, there are further row groups and inside these groups data is divided by columns. Figure 3.4.1 visualizes this concept.

```
Select * from ethereum.transactions
where hash = 0x9ef65fe51...ff74219e
```

Listing 3.1: Shortened simple query on DuneSQL that took 3 minutes to run

[18]Example of a dashboard about history of Ethereum: `https://dune.com/hildobby/ethereum`

[19]DuneSQL was announced in this blog post: `https://dune.com/blog/dune-engine-v2`

[20]Source: `https://dune.com/blog/introducing-dune-sql`

[21]Detailed description of Parquet storage format `https://github.com/apache/parquet-format`

Indexing is done using the Parquet file's metadata. At the end of each file there is a part in which are stored min/max values of all the column values inside a row group, as shown in image 3.4.2. This information allows the database to skip reading the whole file if values are not in the desired range. While working great for certain types of data, it is not very useful with strings, especially if they are not sorted. That is the reason why even very simple queries can take several minutes to run. Listing 3.1 serves as an example of such a query.

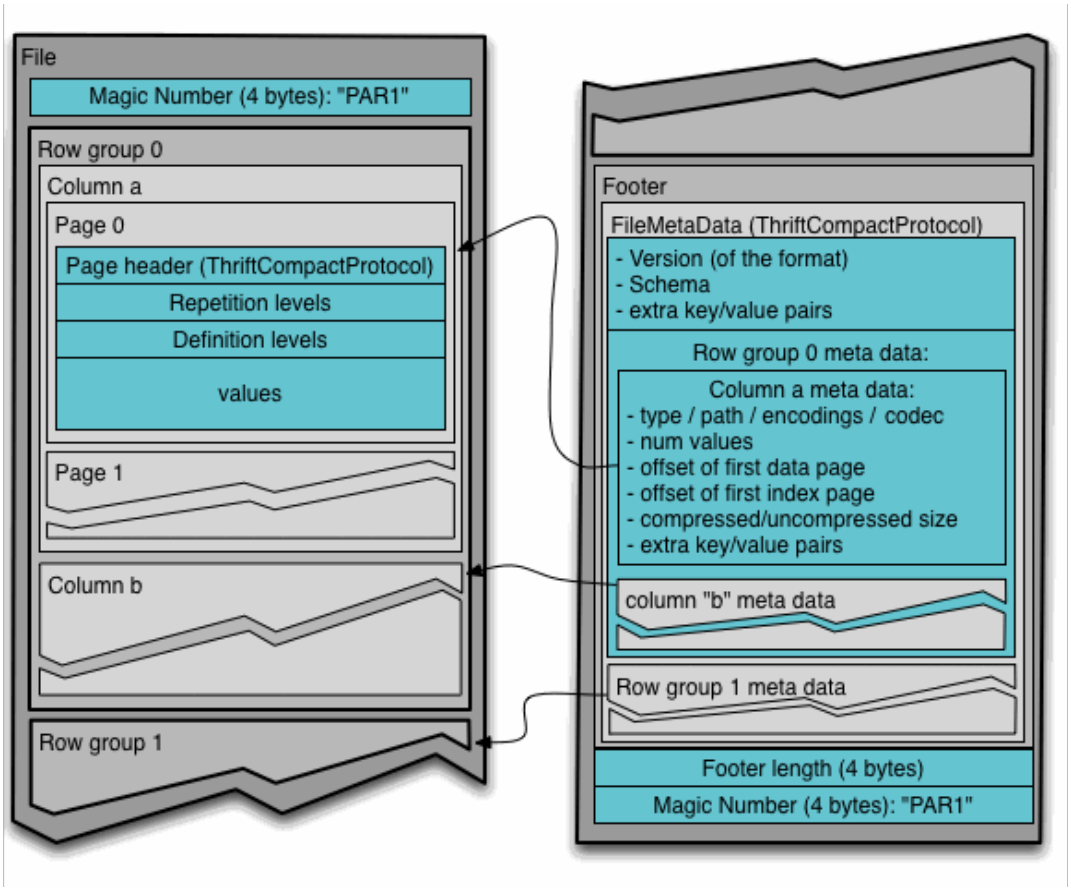

Figure 3.4.1: File structure in the Parquet storage format[22].

block_time		block_number		value		gas_limit		gas_price		gas_used	
2019-05-28 12:41		7848097		3.00		80000		66.00		98618	
2019-05-29 12:41		7848097		0.00		90000		50.00		21000	
2019-05-30 12:41		7848099		0.00		90000		41.00		36785	
2019-05-31 12:41		7848099		6.92		90000		41.00		37065	
2019-06-01 12:41		7848099		0.52		90000		40.00		53465	
2019-06-02 12:41		7848099		5.02		90000		40.00		52845	
2019-06-03 12:41		7848102		0.00		26000		40.00		21000	
2019-06-04 12:41		7848102		0.00		26000		40.00		21000	
2019-06-05 12:41		7848102		61.35		60000		35.00		98618	
2019-06-06 12:41		7848102		0.00		155554		35.00		98618	
MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX	MIN	MAX
2019-05-28 12:41	2019-06-06 12:41	7848097	7848102	0.00	61.35	26000	155554	35	66	21000	98618

Figure 3.4.2: Parquet min/max column index structure [23].

[22]Source: `https://github.com/apache/parquet-format`
[23]Source: `https://dune.com/docs/query/storage/`

3.4.2 Available data

On Dune Analytics, blockchain data is available in multiple layers:

- *Raw tables*: these tables contain raw data as it is stored on the various blockchains, without modifications. For EVM chains this means Blocks, Logs, Traces, Transactions and Withdrawals.
- *Decoded tables*: smart contracts with verified source code receive their own tables on which data is stored in a more readable way. Each event and function that is present in the contract's ABI corresponds to a table on Dune with the following name pattern: `project_chain.contract_[evt/call]_evtOrCallName`. Inside this table there will be columns for each of the event or function attributes, with the relative names. For example, all the *cryptokitties* `Transfer` events are stored in a table called `cryptokitties_ethereum.KittyCore_evt_Transfer` that will have, among other contextual information, three columns: *from*/*to* addresses and *tokenId*.
- *Spellbooks*: these tables are abstractions over raw and decoded data that give users an easy way to retrieve information without diving into the technicalities of complex decentralized protocols. They are open source and the community can contribute creating new spellbooks on the Github Repository[24]. The main power of spellbooks is to put together data from heterogeneous sources to gather useful insights. One of the clearest example is the `dex.trades`[25] spellbook, that puts together all the trades made on all decentralized exchanges of all the blockchains present on Dune Analytics.

3.5 XBlock-ETH

Zheng et al. [4] released a dataset containing raw Ethereum data and described the framework used for getting it, called XBlock-ETH. It was released in 2019. Data is periodically updated in chunks of 500,000 blocks, it currently covers blocks from 0 to 16,499,999. It is divided in different smaller datasets: *Block*, *Block Transaction*, *Internal Transaction*, *Contract Info*, *ERC20 Transaction*, *ERC721 Transaction*, *Token Info*. Data can be downloaded from their website[26] in CSV format.

This is a useful resource for getting Ethereum data without having the possibility to run a node; however it lacks important information such as logs and receipts.

It is not easy to use, since the massive amount of data in the CSV files must be parsed and cannot be easily queried or indexed, further transformation steps are needed. There are some Python scripts available on Github[27] for downloading

[24] All the spellbooks can be found on the Github repository: `https://github.com/duneanalytics/spellbook`

[25] Source code of the `dex.trades` spellbook: `https://github.com/duneanalytics/spellbook/blob/main/models/dex/dex_trades.sql`

[26] Xbloxk website where it is possible to get the data: `https://xblock.pro/xblock-eth.html`

[27] Repository of XBlock-ETH: `https://github.com/tczpl/XBlock-ETH`

and analysing the data, but the code used for the extraction is not open-source, so it is not possible to replicate the extraction.

3.6 Data-ether

DateEther [3] is a framework presented by Chen et al. for extracting and indexing Ethereum data. They tried a different approach for executing this task: they modified a Geth node to record and store data during the initial synchronization phase. They used ElasticSearch[28] for indexing and exploring data, but by the time of their research the size of data was relatively small compared to now.

While extracting data by modifying a node source code was efficient back in 2019, now Ethereum nodes have evolved and there are different and faster RPCs for getting internal transactions. Their way of extracting data was compared against `debug_traceTransaction` RPC of Geth, claiming to be 18.6x faster, but it was not compared against `debug_traceBlock` or `trace_block` of Erigon. From my work, using Erigon's `trace_block` RPC, I managed to get and loop trough all the internal transactions in around 7 hours. Doing that while synchronizing a node would have required at least 3-4 days.

Extracting data by modifying source code has another drawback: maintainability. Just Geth itself received 173 releases[29], modifying its source code would mean having to merge the code and resolving eventual conflicts every time a new release is published. This is not a problem using Ethereum RPC APIs since they do not change after upgrades of the nodes.

3.7 Web3 providers

The term *Web3 provider* is commonly used to refer to companies that offer access to Ethereum RPC API, avoiding developers of Web3 dApps the costs of running a node. They are included in this chapter since the majority of them also provide access to more sophisticated indexed and interpreted data. Some popular web3 providers include Alchemy[30], Infura[31], Quicknode[32] and Chainstack[33].

All of them have endpoints for getting NFTs data. It is possible to get all NFTs owned by an address by just calling an API instead of having to analyze all the chain. Chainstack also hosts on their servers all the supgraphs of *The Graph* protocol, offering users the possibility to use a more stable service instead of the decentralized alternative.

It is important to note that the web3 providers are profit-oriented companies. While they provide an important service in the world of dApps, it is essential to consider their cost implications. Using their services for analyzing historical data can be very expensive, since billing is done based on number of API requests to their server.

[28] ElasticSearch is a search engine based on the Lucene library `https://www.elastic.co/`

[29] Data obtained from the Github releases page: `https://github.com/ethereum/go-ethereum/releases`

[30] `https://www.alchemy.com/`

[31] `https://www.infura.io/`

[32] `https://www.quicknode.com/`

[33] `https://chainstack.com/`

Another important aspect to consider is the transparency of these services. As profit-oriented entities, these web3 providers do not necessarily make their source code and methodology open-source. This means that developers relying on their services need to trust the data they receive.

Depending on centralized companies poses also a risk to the actual decentralization of the web3. Access to the blockchain infrastructure is concentrated on a few companies, as noted by Wang et al. [16].

3.8 Comparison

Table 3.8.1 summarises the main differences between the analyzed tools in terms of *primary target*, *transparency* and *price*.

Tool	**Target**	**Open source**	**Price**
Etherscan	Blockchain users	No	Free explorer, paid apis
The Graph	Web3 developers	Yes	Billing based on usage
Ethereum-ETL	Data analysts	Yes	Free
Dune Analytics	Data analysts	No	Based on query credits, it has free plan
XBlock-ETH	Data analysts	No*	Free
DataEther	Data analysts	No*	Free
Web3 providers	Web3 developers	No	Billing on usage, they have free plans

* Source code not available, but technical papers describing the software were released.

Table 3.8.1: Comparison of state-of-the art tools for management of blockchain data.

My work aims to provide an open-source alternative for accessing Ethereum data. The main target are data analysts, but thanks to the performance of the database used, it can also be used by web3 developers. It is mostly inspired by Ethereum-ETL, but with particular focus on performance and semantics of smart contracts. Data is indexed using a particular graph database, Dgraph, that was not used in the other works.

CHAPTER
FOUR

METHODS

In this chapter, I introduce `eth2dgraph`, the new open-source software designed to facilitate the extraction and indexing of Ethereum data in Dgraph.

Eth2dgraph is developed using Rust, a system programming language that prioritizes both speed and safety. Rust was chosen mainly for two reasons. Firstly, for its emphasis on performance and parallelism, crucial for scaling the data extraction process and achieving good performance. Secondly, Rust benefits from a rich ecosystem of libraries specifically designed for handling Ethereum data, easing the process of development.

One of the key features of eth2dgraph is its integration with a decompiler, enabling the extraction at the scale of the Application Binary Interface (ABI) from the bytecode stored on the Ethereum blockchain.

The following sections provide details about the architecture of eth2dgraph, give a comprehensive overview of how the software operates, and how it has been constructed.

4.1 Data flow

Extracting and indexing Ethereum Smart Contract data is a process of moving and transforming bytes. Raw data stored in the Ethereum node must be taken, transformed and ingested into a database, Dgraph in this case, in order to be indexed.

The first attempt was to do this step through transactions. I ran a Dgraph cluster and instructed eth2dgraph to add data via DQL mutations. DQL stands for **Dgraph Query Language** and it is the language used to write Dgraph queries. This data flow was working but it was too slow for thinking about applying it to all the history of the Ethereum chain. I also had problems to parallelize data insertion, too many concurrent transactions were failing due to read-write conflicts. Figure 4.1.1 visualizes this data flow.

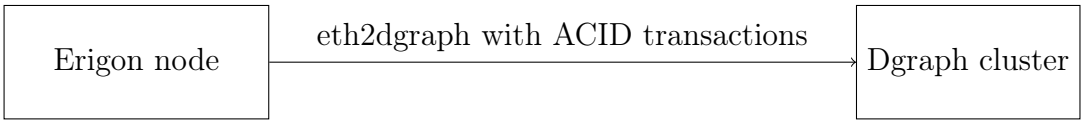

Figure 4.1.1: First attempt of data ingestion into Dgraph

To solve these problems, I changed the approach and followed an ETL (Extract, Transform, Load) process. Extraction and transformation are done in the same step by eth2dgraph. The output is stored in JSON files that can be later loaded into Dgraph using the Bulk Loader.

The Bulk Loader[1] is a tool provided by Dgraph. It is designed for performing the initial load of data into the database. It takes JSON or RDF N-Quads[2] data and stores it directly in Badger, the underlying key-value database used by Dgraph. This is the fastest way to ingest data into Dgraph. It maximizes concurrency and avoids problems related to ACID constraints since it is not operating on a live database. Figure 4.1.2 visualizes the final data flow chosen for eth2dgraph.

Figure 4.1.2: Second and final data flow

4.2 Data model

After seeing how data is indexed in other works, I decided to design the schema in a slightly different way. In eth2dgraph, raw data is interpreted to create a schema around the semantics that can be extracted from the blockchain. This semantics is indexed alongside the raw Ethereum data. Figure 4.2.1 shows the whole schema that was created. All the edges have the `@reverse` index, which means they can be traversed in both directions. Some of the attributes are indexed at load time, but it is easy to add indexes once the database is running. The complete description of both DQL and GraphQL schemas are described in Appendix 8.

[1]Detailed description of the Bulk Loader: `https://dgraph.io/blog/post/bulkloader/`

[2]N-Quads is a serialization format for RDF graphs data `https://www.w3.org/TR/n-quads/`.

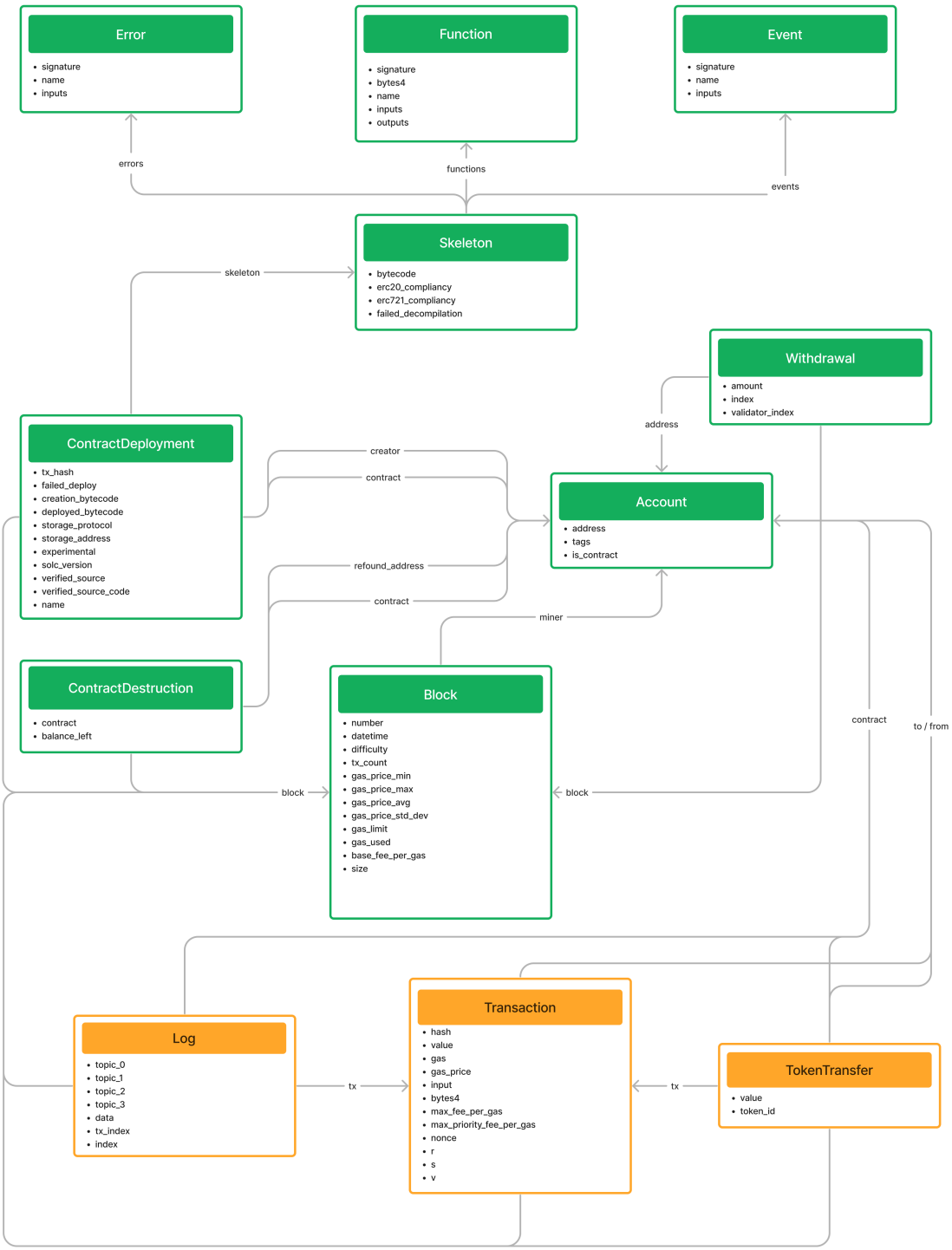

Figure 4.2.1: Schema of Ethereum indexed data in Dgraph

Data is modelled as a graph with many edges. This was done since Dgraph is designed and optimized to perform joins and traversals.

Entities in yellow (*Transaction*, *TokenTransfer* and *Log*), also called *dynamic*, are optional and can be skipped during data extraction. Excluding data about smart contracts usage allows the creation of a much smaller dataset that provides a static description of these contracts. The reason of this choice is that the volume of dynamic data is around `30-40x` bigger than static data. Having the option to extract a smaller dataset enables analysis on less powerful computers, improving

the accessibility of the research.

Here is a brief description of the schema:

- *Transaction* and *Log* are stored as they are retrieved from the Ethereum node. References to other entities are stored as edges for allowing graph traversal in queries. A *Transaction* represents a transfer of value between two EOAs or an invocation of a Smart Contract from an EOA. A *Log* is the result of an invocation of an Event in the code of a Smart Contract.

- *TokenTransfer* represents both an ERC20 or an ERC721 token transfer between two accounts. The value is stored as a string since Dgraph does not handle 256-bit integers.

- *Block* contains all the information related to a single Ethereum block header. This entity is connected to many others: *ContractDeployment*, *ContractDestruction*, *Transaction*, *TokenTransfer* and *Log*.

  The indexed attribute *datetime* is the only reference to time in the database, so it is possible to query it for specific dates and times and then get all the connected data from there.

  Here, it added a summary of the gas prices of the transactions included.

- *Withdrawal* represents a withdrawal from a validator. Withdrawals are used to remove the locked funds and stop being a validator in the proof of stake.

- *Account* represents an Externally Owned Account or a Contract Account. Its fundamental attribute is the *Address* of the account. There is also a boolean field *is_contract* indicating whether the account is a contract or not. This entity is extremely useful for querying data since with the standard Ethereum RPC interface it is not possible to query data based on account addresses. So, for example, it is impossible to get all the transactions from/to an address without having to download and filter all the transactions in the history of Ethereum.

- *ContractDeployment* and *ContractDestruction* represent respectively the birth and the death of the code of a Smart Contract. I decided to store them in this way since I found, analyzing the data, that a single Ethereum Address can receive more than one code deployment, contrary to what it may appear because of the theoretical immutability of Smart Contracts. An address can receive both a deployment with the same old code or with a new different code, in which case it is called *morphic* or *metamorphic* [17].

- I introduced the concept of *Skeleton* directly in the schema since it is a useful way of aggregating similar contracts together. A *Skeleton* is the bytecode of a contract without all the arguments of the PUSH opcodes. Having it directly in the schema allows us to easily find contracts sharing the same skeleton.

- *Event*, *Function* and *Error* are the parts that, together, form the ABI of the contracts. There is just one entity for each signature, so all the contracts implementing the same function/event/error point to the same entity.

Having them indexed in this way allows to search contracts based on the functionalities they implement.

4.3 Data extraction

Data is extracted using the Ethereum *Remote Procedure Call* (RPC) interface[3]. Eth2dgraph extracts data block by block. It needs three API calls per block to get all the data: `eth_getBlockByNumber`, `eth_getLogs` and `trace_block`.

To call these RPCs, I used the Rust library `ethers-rs`[4], which provides, among other functionalities, a full client implementation that wraps the standard Ethereum RPC interface and provides easy methods to interact with it in an asynchronous runtime.

At a high level, data returned from these RPC endpoints is parsed into Rust structs and later serialized into JSON files using the *serde*[5] crate. To implement this, all the Rust structs that must be serialized to Dgraph format implement a trait called *SerializeDgraph*. In Rust, a trait defines a collection of methods. A struct that implements a trait is guaranteed to have those methods implemented. It is a concept similar to the Interfaces in object-oriented programming (OOP) languages.

The trait *SerializeDgraph* requires one method to be implemented: `serialize_dgraph`. As the name suggests, it is a function that serializes a generic struct to the JSON format accepted by the Dgraph Bulk Loader.

The entities produced by eth2dgraph are linked together using the Dgraph's *blank nodes*. It is a way of referencing to an entity that has yet to be created. Dgraph will resolve all the references to a blank node generating and using the same `uid`.

The next sections explain how each piece of data is extracted.

4.3.1 Blocks and transactions

Blocks and transactions are both extracted from the data returned by the RPC `eth_getBlockByNumber`. This method accepts two parameters:

- *Block number* specifies the target block that is extracted
- *Hydrated transactions* is a boolean indicating whether to return or not all the details of the transactions in that block.

Eth2dgraph calls this RPC sequentially for each block with *Hydrated transactions* set to `true`. All raw transaction data is stored without modifications, same for the withdrawals. For the blocks I added a summary of the gas price, this is not returned by default from the RPC. These fields have been added:

[3]Official specification of the Ethereum RPC interface: `https://ethereum.github.io/execution-apis/api-documentation/`

[4]`https://docs.rs/ethers/latest/ethers/`

[5]Serde is a Rust framework for easily serializing and de-serializing data structures `https://github.com/serde-rs/serde`

- *gas_price_min*: the cheapest gas price of all the transactions included in the block, in Gwei.
- *gas_price_max*: the maximum amount paid for gas in the transactions included in the block, in Gwei.
- *gas_price_avg*: the average price of gas in Gwei of that block.
- *gas_price_std_dev*: the standard deviation of gas price in that block, in Gwei.

Gas price varies between each transaction. Sticking to the official RPC docs, it should be possible to obtain data about it just from the transaction receipts. This would imply getting more data and slowing down the process of data extraction.

Looking at the data returned from the Erigon node and analyzing its source code[6], I noticed that gas price was present even if not required by the protocol. I compared it to the data returned from the receipts and it matched, so I decided to use it and store this information in the database.

4.3.2 Logs

Logs are retrieved using the `eth_getLogs` RPC. This remote procedure call accepts an object as a parameter, it can be used as a filter to refine the call and get just the logs needed. It is possible to filter by topics, contract address and block range.

All the logs are already indexed in the Ethereum nodes by the fields on which it is possible to filter. It is the standard way to extract semantics from the chain. When specific conditions happen inside a call to a smart contract, it can emit a log with up to four indexed 256-bit words that will be stored by all the Ethereum nodes. These logs can represent any kind of information, e.g. token transfers, and token swaps.

Eth2dgraph is getting logs and downloading them block by block. The RPC *eth_getLog* is called for each block with a filter on the blocks range, with matching *fromBlock* and *toBlock* parameters.

4.3.3 Smart contracts

There are two ways to deploy a smart contract on the Ethereum blockchain:

- From EOAs, with a transaction to the address `0x0` containing as input data the deployment code of the smart contract.
- From other smart contracts, calling the EVM opcode `CREATE` or `CREATE2`, after having pushed on the stack the deployment code of the smart contract to deploy.

There is no RPC to directly get the list of contracts. Smart contracts are not indexed by the Ethereum nodes.

[6] `https://github.com/ledgerwatch/erigon/blob/35422986645832d1c9ce1107a59dbaf4e12f55dd/turbo/adapter/ethapi/api.go#L450`

It is relatively easy to extract smart contracts deployed in the first way, it is enough to loop through transactions and see the ones sent to the address `0x0`. The resulting transactions are potential contract deployments. To confirm this, it is possible to download their receipts, which contain the address of the newly created contract in case the deployment is successful. After finding the address, it is possible to get the deployed bytecode calling the RPC `eth_getCode`, which returns the actual code stored on the blockchain.

This way of extracting contracts has two drawbacks: it requires two extra calls to RPCs for each deployment and it misses all the contracts deployed by other contracts. Contracts are more likely to be deployed by other contracts rather than by users [18], so it is clear that this way is not ideal.

To extract all the contract deployments, it is necessary to inspect each individual interaction done on the blockchain, both between users and contracts (via transactions) and between contracts and other contracts (via *internal transactions*).

Internal transactions (also known as *traces*) are the result of a call to a smart contract, they describe each single operation performed in that call. They are not described in the Ethereum Yellow Paper [19] and they do not need to be stored by the nodes. They are just a detailed description of a transaction execution. They can be calculated having the transaction data, the bytecode to be executed and the state of the blockchain at the time of the transaction execution.

Erigon provides a RPC to collect all the traces of all the transactions in a block, it is called `trace_block`. Traces returned can be of four types: *Call*, *Create*, *Suicide* and *Reward*. They are structured as a directed graph: an internal transaction can generate many other internal transactions. To get deployments and destructions, it is sufficient to go through the traces graph and filter the Create and Suicide traces, they contain all the needed information.

Eth2dgraph is using the `trace_block` RPC to collect all the deployments and destructions of smart contracts.

4.3.4 Error propagation in traces

An Ethereum transaction can be successfully executed even if some of its parts have failed. An error in one internal transaction implies that all its child traces have no effect on the blockchain.

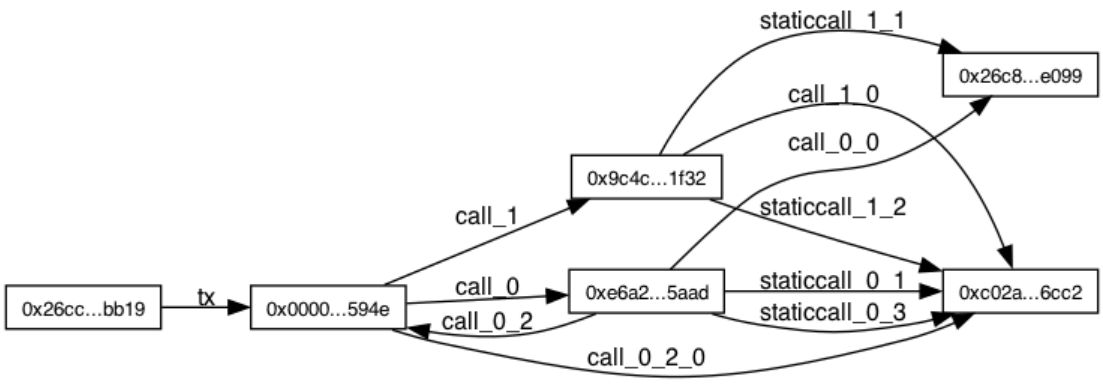

Figure 4.3.1: Example of the structure of traces in a transaction.

Figure 4.3.1 gives an example of the structure of traces in the transaction `0x7b4968c606e...4d941d66977`[7]. Supposing that `call_1` failed, all its child calls (`call_1_0`, `staticcall_1_1` and `staticcall_1_2`) will not change the state of the blockchain, despite being included in the traces returned by `trace_block`. Contracts deployed or destructed in an internal transaction that is part of a failed branch will not be effectively deployed or destroyed.

The only reference to the error is present in the single trace that failed, with an `error` field. To understand if a deployment or a destruction successfully completed, it is necessary to propagate the error from the failed traces to all its children. Eth2dgraph does so with the algorithm reported in Listing 4.1. It first groups traces of a block based on transaction hash. Then, for each transaction, it gets its failed traces. Then it loops again on all the traces of the transaction to see if any of them is a child of a failed one. This is done by searching a failed trace that has a matching starting address, e.g. a trace with address `1, 0, 0` has a matching starting address of the trace `1, 0, 0, 2`.

Contract deployments and destructions found in failed traces are stored in the output of eth2dgraph with a boolean field indicating it.

```
fn propagate_errors(traces: &mut Vec<Trace>) {
    // group traces by transaction hash
    let mut txs: HashMap<TxHash, Vec<&mut Trace>> = HashMap::new
    ();
    traces.iter_mut().for_each(|t| {
        if t.transaction_hash.is_some() {
            let group = txs
                .entry(t.transaction_hash.unwrap().clone())
                .or_insert(vec![]);
            group.push(t);
        }
    });
    // inside each transaction, mark trace as failed if a parent
    trace has failed
    txs.iter_mut().for_each(|(_, grouped_traces)| {
        // collect trace addresses of failed traces
        let failed = grouped_traces
            .iter()
            .filter(|t| t.error.is_some())
            .map(|t| t.trace_address.clone())
            .collect::<Vec<Vec<usize>>>();
        // loop again traces to flag ones whose parent failed
        grouped_traces.iter_mut().for_each(|t| {
            let address = t.trace_address.as_slice();
            let parent_failed = failed.iter().any(|f| address.
    starts_with(f));
            if parent_failed {
                t.error = Some("Parent␣failed".to_string());
            }
```

[7]Full tx hash: 0x7b4968c606e100d05158456d66d620ff6e96f00d68e3b6a426b774d941d66977

```
27          });
28      });
29  }
```

Listing 4.1: Algorithm for errors propagation in traces.

4.3.5 Accounts

As for the smart contracts, there is no RPC to get the list of accounts used on the Ethereum blockchain. They must be extracted as they are used.

Being a permissionless blockchain implies that there is no initial phase of registration or an official opening of an account. Users simply generate an address and start using it.

Eth2dgraph stores accounts every time they are used in the following cases:

- Senders or receivers of transactions.
- Receivers of validator withdrawals.
- Authors of blocks (miners).
- Receivers of SELFDESTRUCT reward.
- Deployers of smart contracts.
- Senders or receivers of token transfers.
- Addresses of contract deployments or destructions, marked as contracts.
- Addresses of contracts emitting logs, marked as contracts.
- Addresses of contracts emitting a token transfer, marked as a contract.

4.4 Semantics extraction

The previous section described how raw Ethereum data is extracted. In this section, I describe how eth2dgraph extracts semantics from this data.

The semantics extracted are meant to give a more comprehensive description of the smart contracts stored on the blockchain. The only information that can be taken from the Ethereum protocol about smart contracts is their EVM bytecode. The EVM bytecode is the byte representation of the contracts' compiled code that can be run by the Ethereum Virtual Machine implemented in all the nodes. While it is fundamental to the functioning of the protocol, it does not give any meaningful description of the smart contract for human analysis.

Eth2dgraph tries to put together pieces of information to allow for an easier analysis of such smart contracts.

4.4.1 ABI extraction

Smart contracts are described by an **Application Binary Interface** (ABI) that lists all the functions and events that are implemented.

Smart contracts can be seen as REST APIs. The application status is stored in the contract's state variables, similar to a database. The public functions exposed by the contract are like the API endpoints that can be used by users to interact with the application. This is done via stateless transactions (in the blockchain) and via HTTP calls (in the traditional REST API pattern). The ABI is the description of the exposed functions and events of a smart contract, similar to the specification of an API.

It is clear that having the ABI of a smart contract gives many useful insights about what it does. It can also be used to decode transactions and logs. In the ABI, there are the types of functions and events arguments. These types can be used to decode the bytes present in the transactions and logs data. It is possible to understand if they represent addresses, numbers, strings, raw bytes, etc..

To extract the ABI, eth2dgraph integrates heimdall-rs[8], an open-source EVM decompiler. Heimdall-rs uses symbolic execution to create the control flow graph (CFG) of the EVM bytecode. This is done via a custom implementation of the EVM. From the CFG, and looking at the function dispatcher part of the code, it is possible to locate functions and extract their inputs and outputs parameter types.

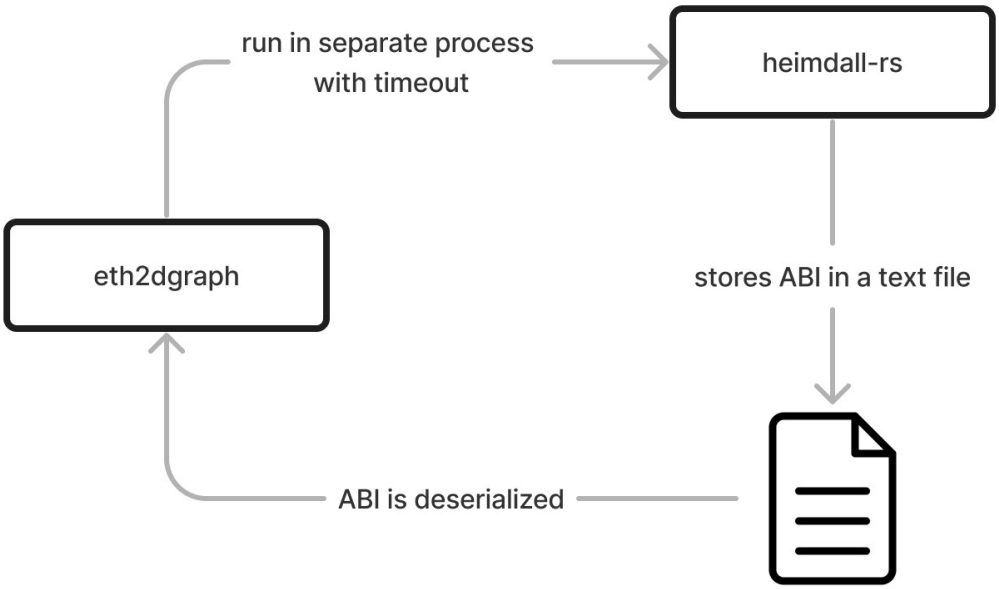

Figure 4.4.1: Heimdall-rs integration into eth2dgraph

It is possible to extract function names using a database of reverse hashes. Function selectors are stored in the bytecode as the first four bytes of the hash

[8]EVM decompiler written in Rust, source code available at: `https://github.com/Jon-Becker/heimdall-rs`

of the function's signature. It is possible to read these four bytes and see if there are matching signatures that were previously reverse-hashed. Heimdall-rs does this using the public etherface[9] database. Unfortunately, it is not possible to extract parameter names from the bytecode stored on the blockchain. This piece of information is removed during the compilation of the code.

Heimdall-rs is integrated in eth2graph using separate processes. Each time there is the need to run the decompiler, eth2dgraph creates a new process. This is done for isolation. For the nature of how the decompiler works, it can run in infinite loops during the symbolic execution. To avoid that, eth2dgraph has a parameter to set the maximum time spent waiting for de-compilation. The default is five seconds.

The output of the `heimdall-rs` process is stored in text files. When decompilation is done, these files are de-serialized by eth2dgraph into Rust data structures. Figure 4.4.1 shows how decompilation in handled in eth2dgraph.

4.4.2 Contracts skeleton and metadata

The *skeleton* of an EVM bytecode is the bytecode itself with all the arguments of the PUSH opcode set to zero. It allows to find bytecodes that are functionally identical between them. The concept of skeleton is used in many studies to reduce the total number of contracts to analyze [20][21].

Removing the PUSH arguments to the bytecode deployed on the blockchain is not enough to extract the skeletons. The Solidity compiler can add metadata at the end of the bytecode that must be removed before getting the skeleton.

Eth2dgraph identifies this part using a regular expression. Metadata is stored as CBOR[10] encoded data. After being split, the *runtime* part of the bytecode is processed for the skeleton extraction, while the metadata part is decoded. Metadata are also stored in the indexed data, these are the field included:

- *storage_protocol*: distributed file system protocol where the developer can eventually store the source code of the contract.
- *storage_hash*: location on the contract's data in the distributed file system.
- *compiler*: the version of the Solidity compiler used.
- *experimental*: whether the compilation was performed with experimental features activated.

The skeleton extraction is performed by looping through the bytes of the EVM bytecode. A byte between `0x60` and `0x7f` represents a PUSH instruction. If a byte is found in that range, the next bytes are replaced with zeros depending on the kind of PUSH found.

[9]Etherface is a database of Ethereum functions and event signatures, with related hashes, available at: `https://www.etherface.io/`

[10]CBOR is a binary data serialization format standardized in RFC8949 `https://www.rfc-editor.org/rfc/rfc8949.html`

4.4.3 Verified source code

It is possible to link contracts to their verified source code using the repository of *Smart Contract Sanctuary* [22]. After cloning the repository locally, eth2dgraph can be run with an option to include the source code of the discovered contracts in the indexed data.

This allows querying the contracts based on text inside the source code. Dgraph supports text matching with stemming stop word removal. Data can be queried with two query functions: `anyoftext` and `alloftext`. The first function matches for any of the term searched for, while the second function matches for all the search terms.

A query example with `anyoftext` is searching for all smart contracts that have in the source code the terms *token*, *erc20* or *erc721*. To be more precise, it would be possible to search with `alloftext` all the smart contracts that have in the source code the exact signature of the transfer function of either the ERC20 or ERC721 standards. All of this is doable in a single fast query.

4.4.4 Token transfers

The ERC20 and ERC721 token standards state that when a token ownership is transferred an event MUST be emitted. Listing 4.2 shows the event emitted for fungible token transfers, Listing 4.3 shows the one emitted for NFT transfers. Emitting an event means generating a log that is indexed by Ethereum nodes.

```
event Transfer(address indexed _from, address indexed _to, uint256
    _value)
```

Listing 4.2: Event emitted for ERC20 token transfer

```
event Transfer(address indexed _from, address indexed _to, uint256
    indexed _tokenId)
```

Listing 4.3: Event emitted for ERC721 token transfer

Both events share the same name and type. The only difference is in the last parameter. It is indexed in the case of ERC721 and not indexed in the case of ERC20. The signature of the event is not influenced by this difference, so all the logs that describe token transfers share the same signature.

Logs with the first topic matching the `keccak256` hash of the Transfer event signature are treated as token transfers by eth2dgraph.

After finding a matching log, the bytes composing the data field are split into 256-bit words and merged into the 256 words of the indexed topics.

The 256-bit words are then treated and decoded as follows:

- First word is the *from* address.
- Second word is the *to* address.
- Third word is the *value*. It represents the amount in the case of ERC20 or the token ID in the case of ERC721.

If the decoding succeeds, the log is parsed and later stored as a token transfer.

The distinction between ERC20 and ERC721 transfers is done based on the length of the topics array. Since ERC20 has the last parameter not indexed, the topic length is three words. For the ERC721 the topics length is four. This information is used to discriminate between storing the third word as *value* or as *token_id*.

The same way of semantics extraction can be used for other kind of logs, e.g. token swaps. Eth2dgraph implements the code for parsing token transfer since it is the most common log emitted on the Ethereum chain.

4.5 Software architecture

The Ethereum network stores data in the order of billions of entries. The process of extraction must be as efficient as possible to be able to get this data in a reasonable time. At the time of writing, August 2023, Ethereum has 18M blocks. Performing every single action described in the previous sections sequentially for each block results in executions that take many days or even weeks.

A lot of time is spent waiting for network data (the calls to the RPCs) or for the decompiler process to complete the decompilation. While it is not possible to make these steps faster, it is possible to parallelize them. Eth2dgraph has been developed to maximize concurrency of the machine where it is ran to minimize the time needed for data extraction.

This was done using async Rust and the *Tokio asyncronous runtime*[11]. Figure 4.5.1 gives a general overview of how tasks are used and how they communicate.

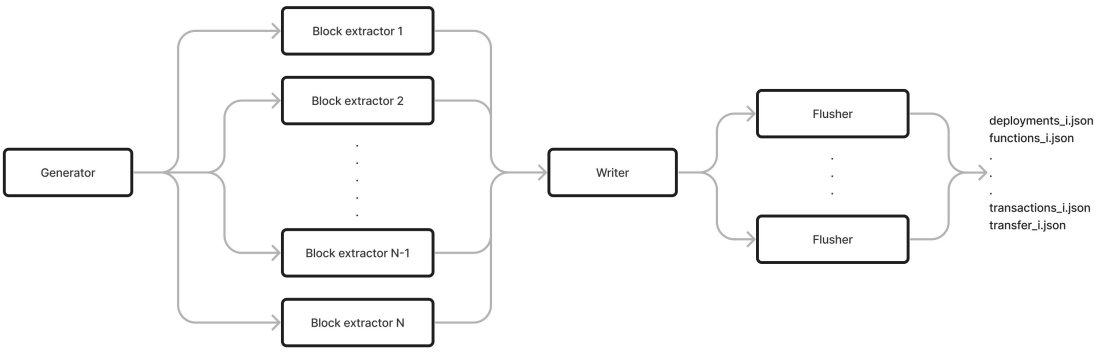

Figure 4.5.1: Software architecture of eth2dgraph

Here is a more detailed description of each task:

- The *generator* task loops trough the block range to extract and spawns, using `tokio::spawn`, an *extractor* task for each block that must be processed. It is also responsible for initializing all the data structures needed, the output data folders and the *writer* task.

  It is possible to set the limit of parallel extractor tasks to run usign the `-num-jobs` option. This is implemented using a semaphore. Each time an extractor task is created, it acquires a permit from a semaphore with a fixed

[11] Tokio provides a multi-threaded runtime for executing asynchronous Rust code `https://tokio.rs/`.

capacity. When the task ends, the permit is dropped and, as a consequence, a new permit is freed in the semaphore. In case the maximum number of tasks is reached, the generator waits until the semaphore has a free permit available. This was done to avoid overloading the system with millions of concurrent tasks.

- The extractor task is the one responsible for the actual extraction. It receives as an argument the block to process and it collects all the data related to it. This includes handling the decompilation. All the steps related to data extraction were described in the previous sections.

  When data is ready to be stored, it is sent to the writer task using a bounded *multiple-producer single-consumer* (mpsc) channel.

- The writer task is responsible to collect all the data sent by all the extractors and merge it. It stores data in buffers, one for each data type. When buffers reach a certain size, that can be set with the `-size-output` option, they are sent to a *flusher* task to be stored to disk. The flusher tasks are spawned on demand by the writer task using `tokio::spawn_blocking`. All their join handles are stored in a vector to wait for their termination at the end of the extraction.

- The flusher task is responsible for storing and compressing the output of the extraction. It receives a vector of a generic type `T` that implements the trait `SerializeDgraph`. It compresses this vector using gzip with a compression level that can be set as an option. Finally, it stores it in an output file. Data is stored divided by type and with incremental file names.

All these tasks are ran on the multi-threaded Tokio runtime. They are managed by the Tokio scheduler that implements a *non-preemptive*, also called *cooperative*, scheduling strategy. This means that tasks are switched when they explicitly ask for it, using `.await`, giving back the control to the scheduler.

In this way it is possible to maximize the parallelism of the extraction process. For example, when a task is downloading block's data, it calls `await` on the library function responsible for networking. This call tells the Tokio scheduler that the task is paused and cannot continue, so it is replaced with another task that has work to do. The task will be resumed when the network data is ready to be used.

4.5.1 Decompilation cache

One of the biggest bottlenecks of the extraction process was the decompilation step. Spawning a dedicated process for handling decompilation of all the 60M contracts takes a lot of time. The main problem is that the decompiler, for how it is designed, can encounter infinite loops during the building of the control flow graph. To avoid that, it is ran with a timeout of a few seconds, but even with that, it was slowing down the process by a lot.

The solution implemented in eth2dgraph is caching the decompilation based on bytecode skeletons. Two contract deployments sharing the same EVM skeleton are decompiled only once. The decompiled ABI of a skeleton comes from the decompilation of the first contract found with that skeleton.

The reason of this choice is that contracts sharing the same skeleton also share the same code logic. In theory, there could be a difference in the function names, since they are stored as arguments of PUSH instructions, but, from the data analyzed, this almost never happens.

To test the reliability of the caching logic, I compared ABIs extracted from decompiling each single contract to the ones got using the cache. To make the test more reliable, I ran it on various block ranges spanning trough the history of the Ethereum blockchain. Table 4.5.1 reports the results of the tests. Full match means that the ABI got from the cache is exactly the same as the one got from a new decompilation run. Partial match indicates that the ABI got from the cache has the same exact function and event names, but there is at least one difference between types based on assumptions made by the decompiler (e.g. bytes instead of address). Mismatch indicates that the ABI got from the cache has at least one different function or event name.

Blocks range	**Cache hits**	**Full matches**	**Partial matches**	**Mismatches**
6000000-6001000	1373	1373	0	0
10000000-10001000	596	592	4	0
12008000-12009000	1296	1295	1	0
15505000-15506000	101	101	0	0
16001000-16002000	120	100	19	1
17000000-17001000	39	39	0	0

Table 4.5.1: Precision of the decompilation caching logic

In total, 99.29% of the cache hits resulted in full matches, 0.68% in partial matches and just 0.03% in mismatches.

This level of accuracy showed that caching decompilation based on skeletons is an effective way of reducing the time needed for semantics extraction. The slight loss of precision is justified by the boost in performance, that made it possible to scale the extraction of ABIs to all the history of the chain in a single machine. It allowed to reduce the number of decompilation runs from 60M to 470k.

The cache is implemented in the code with a shared `HashMap`. The implementation of the concurrent hashmap used is the one provided by the `DashMap`[12] crate. The *key* of the hashmap is the hash of the skeleton's bytecode and the *value* is an *atomic unsigned integer*. The role of the number in the cache is of indicating how many failures that specific skeleton has encountered during decompilation. It is used for trying multiple times to decompile a skeleton that fails to decompile

[12]DashMap provides a concurrent hashmap that is faster and easier to use that the combination of `RwLock` and `HashMap`. It is available at:`https://github.com/xacrimon/dashmap`

even with different contracts' bytecodes. These is an hard limit of ten attempts, after which the skeleton is stored without the ABI and no more decompilation processes will be spawned.

This way of extracting semantics has a direct implication on the schema of data. The ABI extracted by the decompiler is linked to the skeleton and not to the contract deployment itself. Figure 4.5.2 shows the result of this design choice in the schema.

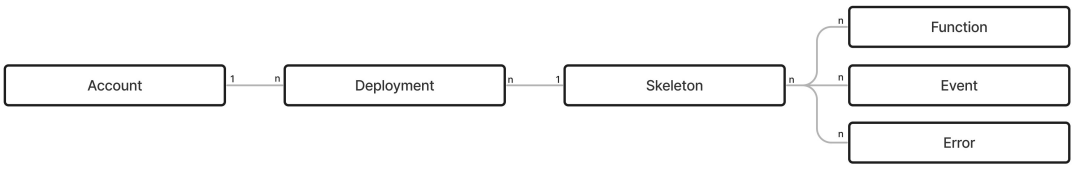

Figure 4.5.2: Storage of contracts' information

4.6 Similarity calculation

Smart contracts' deployments are clustered based on bytecode skeletons: many smart contracts share the same skeleton and thus they are linked to the same skeleton entity in the database. To link these skeleton clusters together, some similarity metrics are needed. Linking skeleton clusters helps the process of data analysis; it ease the recognition of patterns and connections.

Eth2dgraph implements two similarity metrics:

- *Interface similarity*: inspired by Di Angelo and Salzer [23], it calculates similarity between skeletons as the Jaccard index of the sets of function and event names. Two skeletons are similar if they have similar ABIs. It is calculated as

$$Sim = \frac{|A \cap B|}{|A \cup B|} \tag{4.1}$$

  where A and B are the sets containing function and event names. It gives a number between 1 (identical ABIs) and 0 (non-overlapping ABIs).

- *Bytecode similarity*: The interface similarity logic works greatly if the smart contracts implement many functions or events. It does not work very well if most of the code lies in internal functions and the exposed ones are just a few, like `start()` or `run()`. To face this problem I added a second similarity metric that just considers the contracts' bytecodes. Inspired by Kiffer et al. [18], the metric used is the cosine similarity between hypervectors containing the frequencies of opcodes' 5-grams. It is computed as follows:

  1. The two bytecodes to analyze are decoded to extract the opcodes, without their arguments.
  2. From the opcodes, the 5-grams and their frequencies are calculated. For example these instructions:

```
PUSH1
PUSH1
MSTORE
CALLVALUE
DUP1
ISZERO
PUSH2
JUMPI
PUSH1
```

give these 5-grams with related frequencies as values:

```
[
    ("PUSH1 PUSH1 MSTORE CALLVALUE DUP1", 1),
    ("PUSH1 MSTORE CALLVALUE DUP1 ISZERO", 1),
    ("MSTORE CALLVALUE DUP1 ISZERO PUSH2", 1),
    ("CALLVALUE DUP1 ISZERO PUSH2 JUMPI", 1),
    ("DUP1 ISZERO PUSH2 JUMPI PUSH1", 1),
]
```

3. This results in having two hypervectors, here called A and B, in the dimension of the 5-grams. A and B are structured as shown in the previous listing. The cosine similarity is the cosine of the angle θ between these two vectors. It is possible to compute it as

$$cos(\theta) = \frac{A \cdot B}{||A||\,||B||} = \frac{\sum_{i \in \Omega} A_i B_i}{\sqrt{\sum_{i \in \Omega} A_i^2 \cdot \sum_{i \in \Omega} B_i^2}}$$

In the previous formula, Ω is the set that results from the intersection of the dimensions of A and B. This gives a number between 0 (completely dissimilar bytecodes) and 1 (identical bytecodes). The suggested threshold for considering similarity suggested by Kiffer et al. is of 0.90.

The process of similarity calculation is done on demand after data is loaded into Dgraph. The same binary that is responsible for the data extraction part also integrates commands to perform data analysis.

It is possible to run eth2dgraph with the `analyse similarity` command to calculate both the previous metrics. Comparing all skeletons with each other is a heavy calculation, with a quadratic complexity with respect to the number of skeletons. It is possible to restrict the calculation to a single skeleton, giving as an argument the address of a contract.

To optimize the performances, calculation of similarity is done in parallel using the *parallel iterators* of the *Rayon*[13] crate.

The output of the calculation is a text file containing the RDF triples that describe the similarities. It is possible to import it into the live database by running a mutation or using the Dgraph's *live loader*.

Similarity values are stored as edge attributes, called *facets* in Dgraph. It is possible to query the data filtering by similarity value. Listing 4.4 shows an example of how to retrieve similar skeletons filtering by similarity value.

[13]Rayon is a data-parallelism library that easily introduces parallelism into existing sequential code `https://docs.rs/rayon/latest/rayon/`

```
{
    q(func: uid(0x180c753f6)) {
        Skeleton.similar_code @facets(gt(similarity, 0.95))
   @facets(similarity) {
            uid
        }
        Skeleton.similar_interface @facets(gt(similarity, 0.95))
   @facets(similarity) {
            uid
        }
    }
}
```

Listing 4.4: Example DQL query for getting similar skeletons.

CHAPTER

FIVE

RESULTS

5.1 Infrastructure used

Eth2dgraph was developed and used on a server provided by the Decentralised Systems Engineering Lab of NTNU University[1]. The server's specifications are reported in Table 5.1.1.

Parameter	Value
CPU	64 cores, 256 threads
RAM	1.5 TB
OS	Ubuntu 22.04
Disk	16 TB SSD array

Table 5.1.1: Specification of the server used for the work.

On the same machine, there was an archive Ethereum node that was used to get the data. The RPC calls did not have to go through the network, all the process was done in a single machine. The client used was Erigon[2]. It was run with the command reported in Listing 5.1 and the RPC daemon with the command reported in Listing 5.2.

```
erigon \
    --datadir="our data location" \
    --chain=mainnet \
    --authrpc.jwtsecret="JWT secret location"\
    --private.api.addr=0.0.0.0:9090 \
    --http.api=eth,debug,net,trace,web3,erigon
```

Listing 5.1: Erigon command

```
rpcdaemon \
    --datadir="our data location" \
    --http.addr=0.0.0.0 \
```

[1]Decentralised Systems Engineering Lab `https://www.ntnu.edu/idi/dse`

[2]Erigon is an Ethereum client written in Go `https://github.com/ledgerwatch/erigon`

```
    --http.api=eth,debug,net,trace,web3,erigon
```

Listing 5.2: RPC daemon command

5.1.1 Benchmark of the Erigon's RPC interface

To give a clearer overview of the environment in which data extraction was done, I performed a load test against the Erigon node using a modified version of *flood*[3]. I tested the throughput and the success rate, varying the requests per second, of the three RPCs used by eth2dgraph: `eth_getBlockByNumber`, `eth_getLogs` and `trace_block`. Flood was modified to generate RPC calls with the exact same parameters used by eth2dgraph, spread over random block numbers. The actual network calls were performed by *vegeta*[4], that is specifically designed to measure HTTP services with a constant request rate. Each load test was conducted for 30 seconds. Figures 5.1.1 to 5.1.6 show the results of this test. The slowest RPC, and so the bottleneck of data extraction, resulted to be `trace_block`.

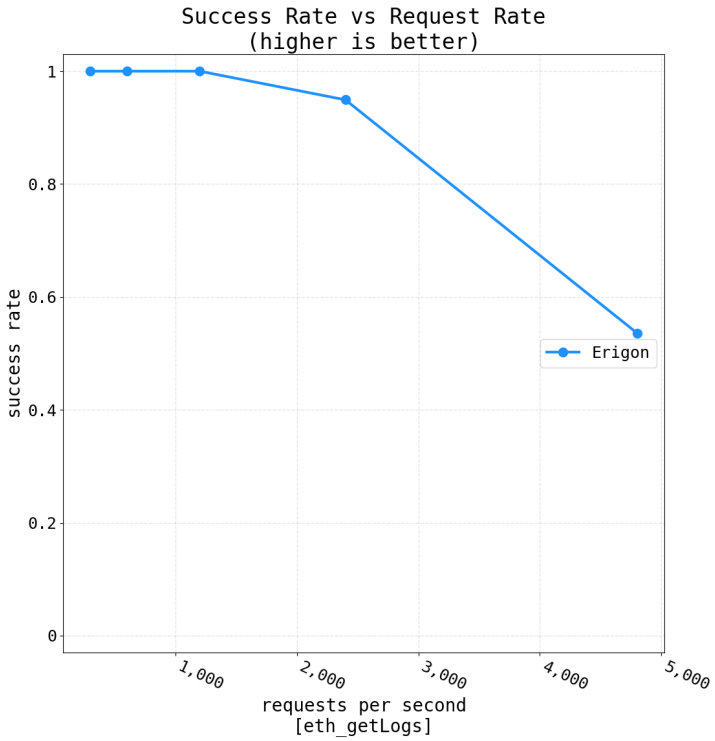

Figure 5.1.1: Success rate of `eth_getLogs`. After 1200 requests/s, Erigon starts to fail handling some requests. At 5k requests/s half of the requests fail.

[3]Flood is an open-source tool for load testing Ethereum nodes `https://github.com/paradigmxyz/flood`

[4]Vegeta is a tool written in Go for load testing HTTP services `https://github.com/tsenart/vegeta`

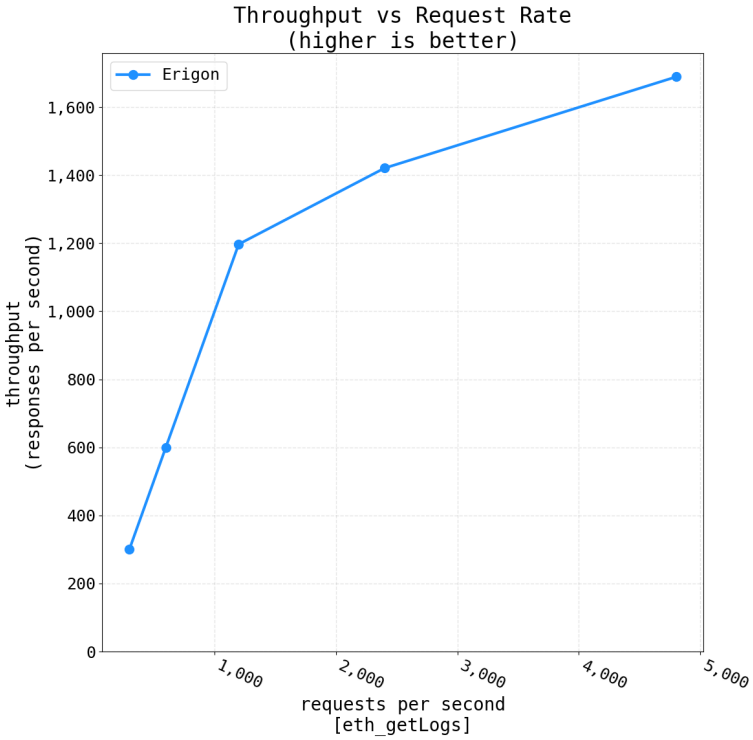

Figure 5.1.2: Throughput of `eth_getLogs`. After 1200 requests/s Erigon can't keep the requests rate.

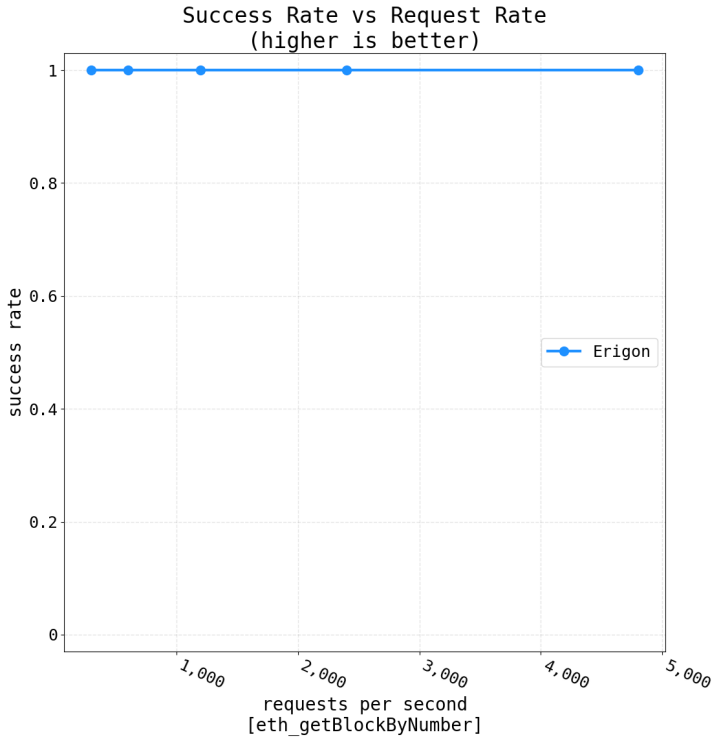

Figure 5.1.3: Success rate of `eth_getBlockByNumber`. Erigon shows perfect performance on this RPC. It can successfully reply to 5k requests/s.

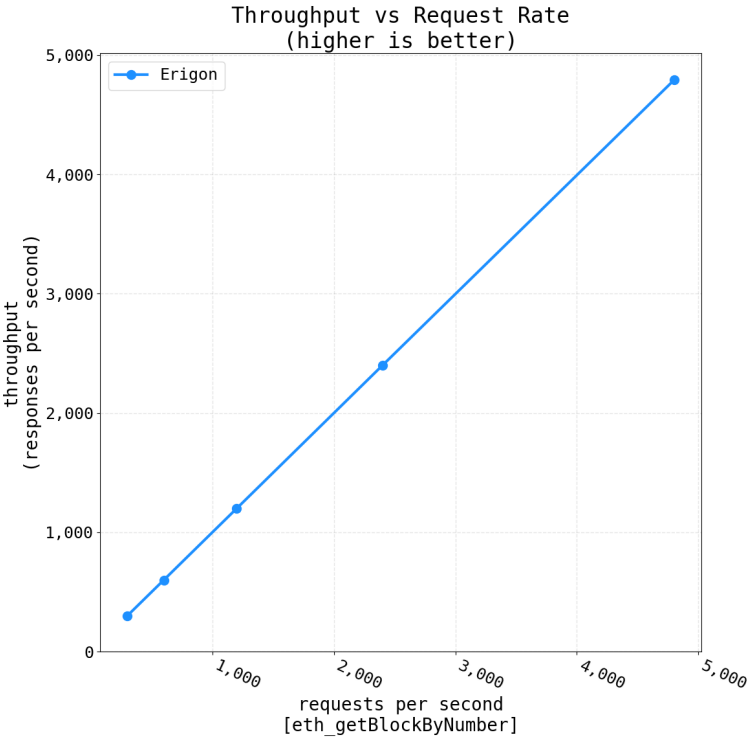

Figure 5.1.4: Throughput of `eth_getBlockByNumber`. Erigon can keep the throughput even at 5k requests/s.

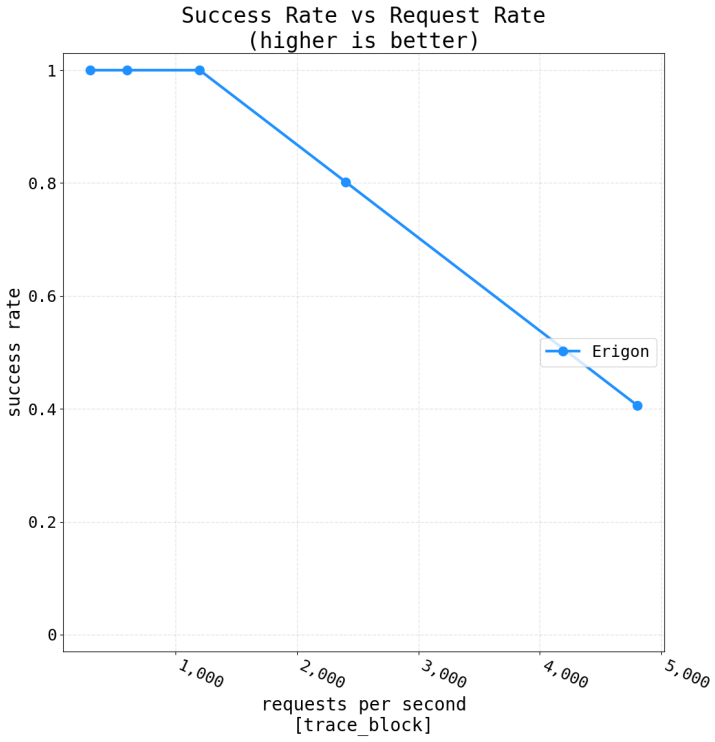

Figure 5.1.5: Success rate of `trace_block`. Erigon starts to degrade after 1200 requests/s. At 5k requests/s, just 40% of the requests are successfully handled.

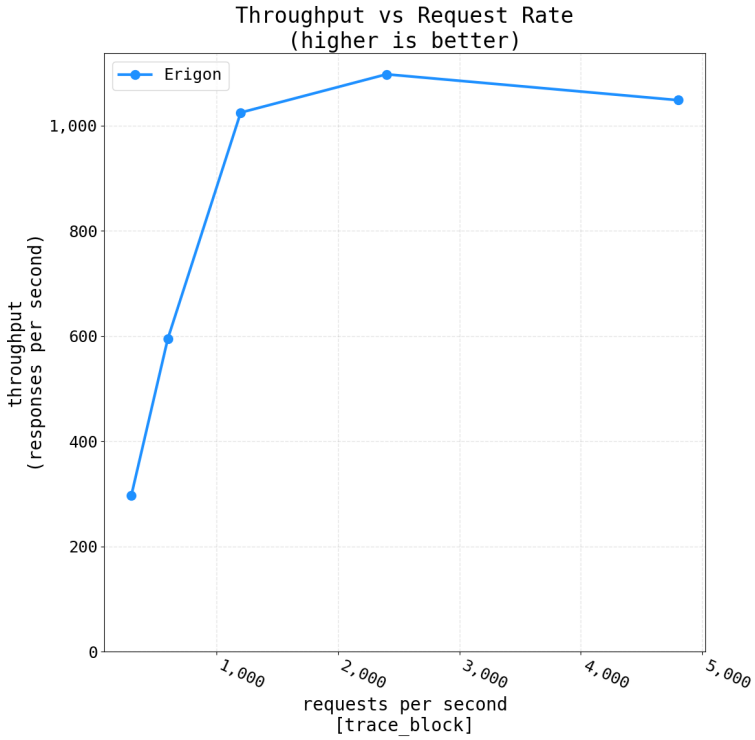

Figure 5.1.6: Throughput of `trace_block`. It reaches the maximum of around 1200 responses/s at 2400 requests/s.

5.2 Optimal number of concurrent tasks

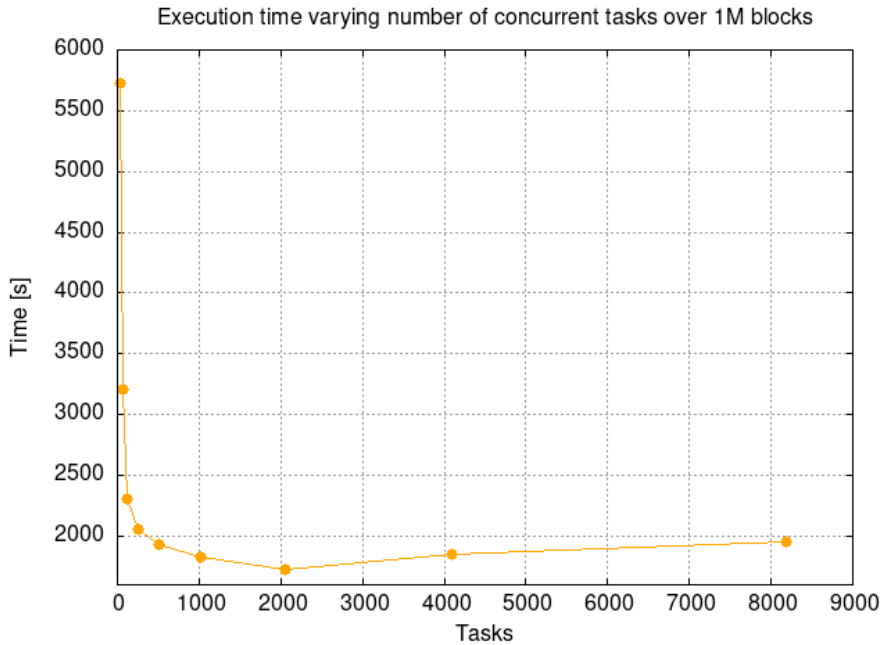

Figure 5.2.1: Extraction time varying the number of concurrent tasks.

Eth2dgraph can be run with a variable amount of concurrent tasks that perform

data extraction. This value can be set with the `-num-tasks` option. Having too many concurrent tasks can overload the system where eth2dgraph or Ethereum node run. On the other hand, having just a few tasks makes the extraction process very slow. The best is to find the best balance point. To find a good value for the number of tasks, I ran eth2dgraph with different amounts of tasks over one million blocks, from 10M to 11M. Figure 5.2.1 shows the results of this test. As observed from the data, the optimal number of tasks for the machine used is around 2048, which is eight times the amount of available threads.

5.3 Extraction of data

5.3.1 Extraction and Transformation

To perform the extraction and transformation of Ethereum data to Dgraph format, eth2dgraph was run with the command reported in Listing 5.3. The extraction was performed from block 0 to block 17,265,420, using 2048 concurrent tasks.

```
eth2dgraph extract \
    -o extraction_output \
    -f 0 \
    -t 17265420 \
    --num-tasks 2048 \
    --include-tx \
    --include-transfers \
    --include-logs \
    -s smart-contract-sanctuary-ethereum/ \
    --size-output 100000 > eth2dgraph_extraction.logs & disown
```

Listing 5.3: Eth2dgraph extraction command used.

Table 5.3.1 reports general statistics about the extraction process. The detailed sizes of the output folders are reported in Table 5.3.2.

Figure 5.3.2 and Figure 5.3.1 shows respectively the RAM and CPU usage of the server during the process of data extraction. Data was obtained using the command `top -bn1 | awk '/Cpu/ { print $2 }'` for the CPU and `free -m | awk '/Mem/{print $3}'` for the memory.

Parameter	Value
Total time	7h 15m 21s
Block/s	660.97
Contracts	60,016,663
Contract/s	2,297.6
Decompiler failures	508,990
Output size	957 GiB

Table 5.3.1: Statistics about extraction and transformation process.

Folder	Size
/	957GiB
/dynamic	934GiB
/static	24GiB
/static/blocks	1.2GiB
/static/events	548KiB
/static/destructions	3.4GiB
/static/deployments	18GiB
/static/skeletons	903MiB
/static/errors	36KiB
/static/functions	8.5MB
/dynamic/transfers	129GiB
/dynamic/logs	263GiB
/dynamic/transactions	543GiB

Table 5.3.2: Size of extracted data divided by folders.

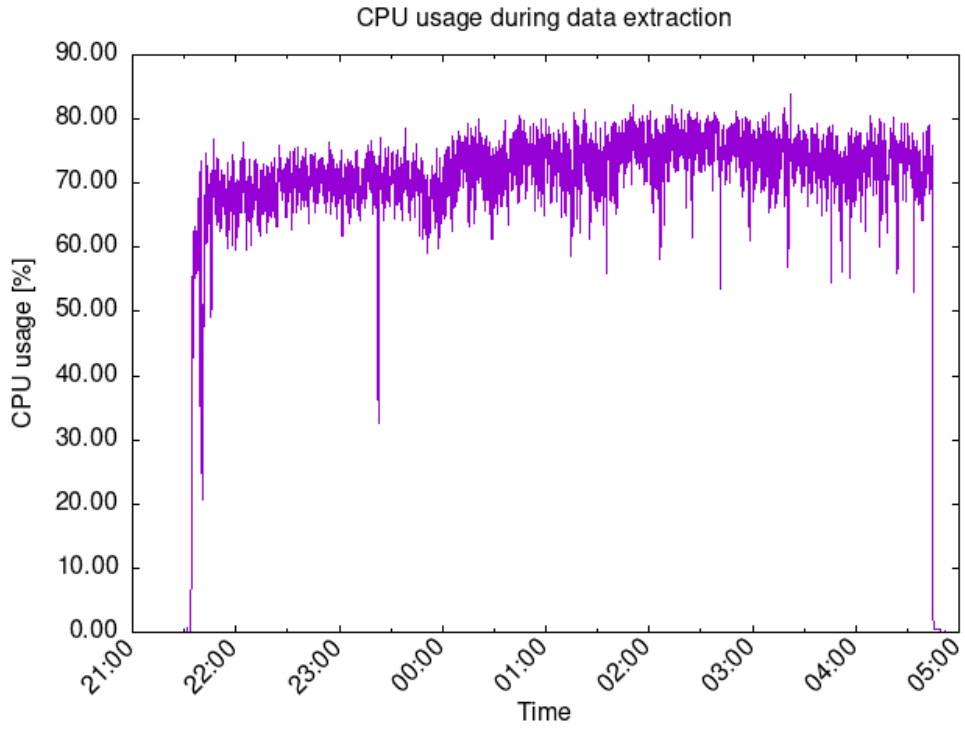

Figure 5.3.1: CPU usage of the server during data extraction.

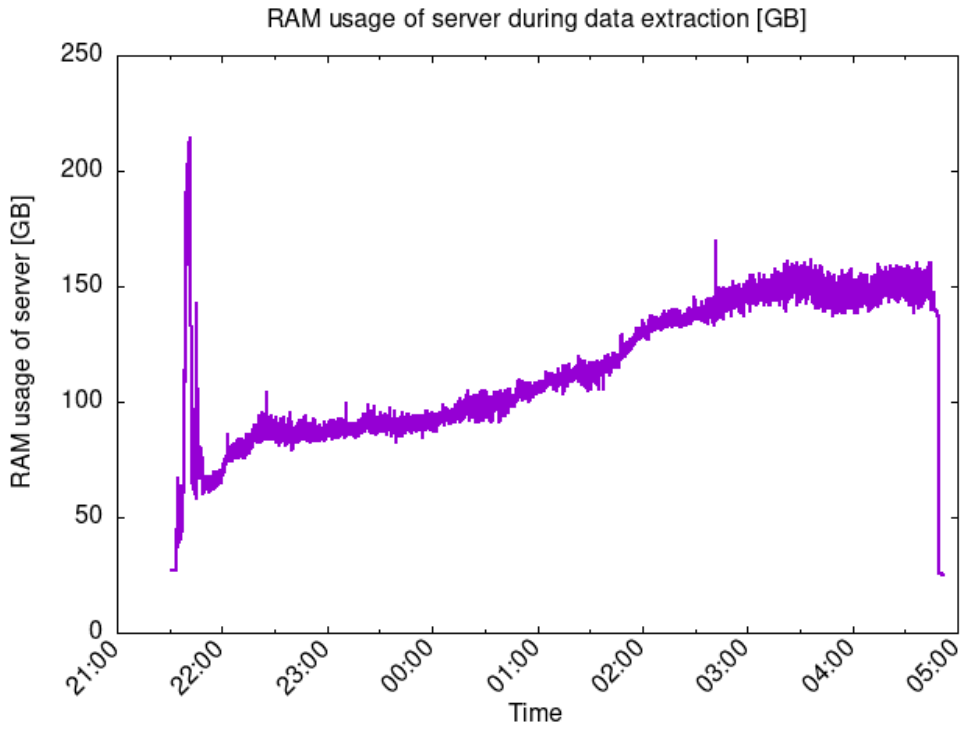

Figure 5.3.2: Memory used by the server during data extraction

5.3.2 Import in Dgraph

Data was imported in Dgraph using the Bulk Loader. I faced a problem during the import of the complete dataset, the loader kept crashing. Analyzing the logs, it seemed that the crash was due to a hard limit on the size of local buffers. Removing this hard-coded limit allowed me to complete the import of data. I also submitted this fix[5] in the open-source codebase of Dgraph. It was accepted, merged in the main branch and later included in the 23.0.1 release.

To perform the bulk import, I first ran with Listing 5.4 an instance of `zero`, the Dgraph's node responsible of coordinating the distributed cluster. With the `zero` running, the bulk import process was started with the command reported in Listing 5.5.

```
dgraph zero --my=localhost:5080
```

Listing 5.4: Command used for running `zero`.

```
dgraph bulk -f "<data-location>" \
  -s "<dql-schema-location>" \
  -g "<graphql-schema-location>" \
  --out "./out" \
  --map_shards=4 \
  --reduce_shards=1 \
  --zero=localhost:5080 \
  --mapoutput_mb=4096 \
  --num_go_routines=64 \
  --cleanup_tmp=false
```

Listing 5.5: Command used for running `bulk loader`.

[5]Pull Request can be seen here: `https://github.com/dgraph-io/dgraph/pull/8841`

The import of the complete dataset took 52 hours. Divided into 28 hours for the MAP phase and 24 hours for the REDUCE phase. It resulted in being the bottleneck of the process, it took around 7.5 times the amount of time needed for extracting the data.

For memory heavy operations, Dgraph does not rely on the Go garbage collector, it uses *jemalloc* to manually allocate memory. Figures 5.3.3 and 5.3.4 show the RAM allocated by Dgraph during the phases of MAP and REDUCE of the bulk import. At least 400GiB of RAM are needed. Both steps have shown a big spike in memory allocation at around half of the process. The reasons of these spikes are not clear.

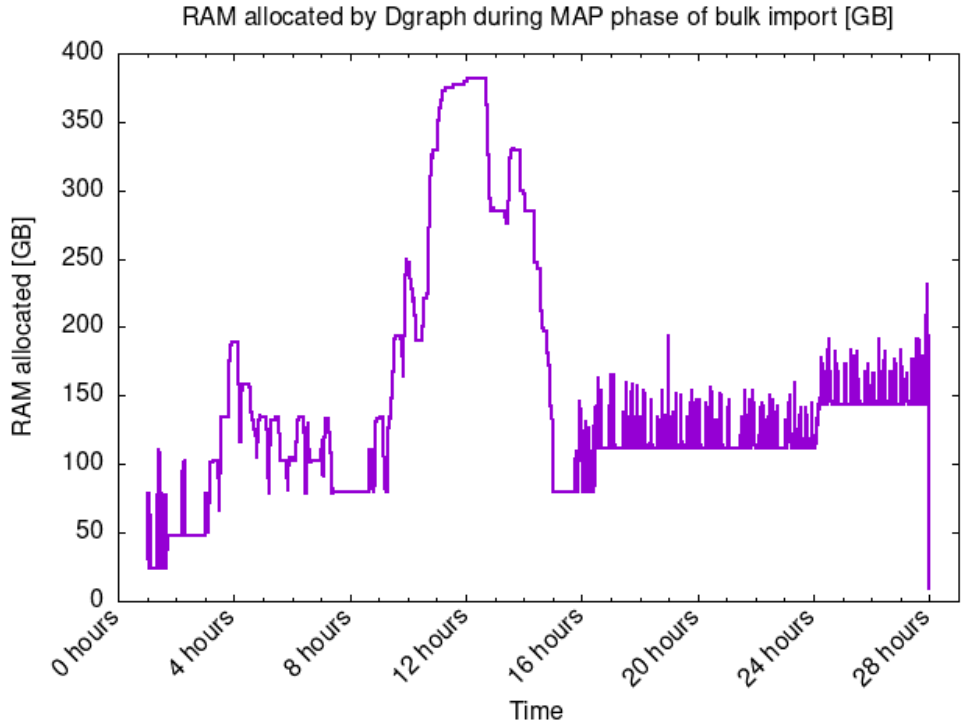

Figure 5.3.3: RAM allocated by Dgraph with jemalloc during the MAP phase of the bulk import.

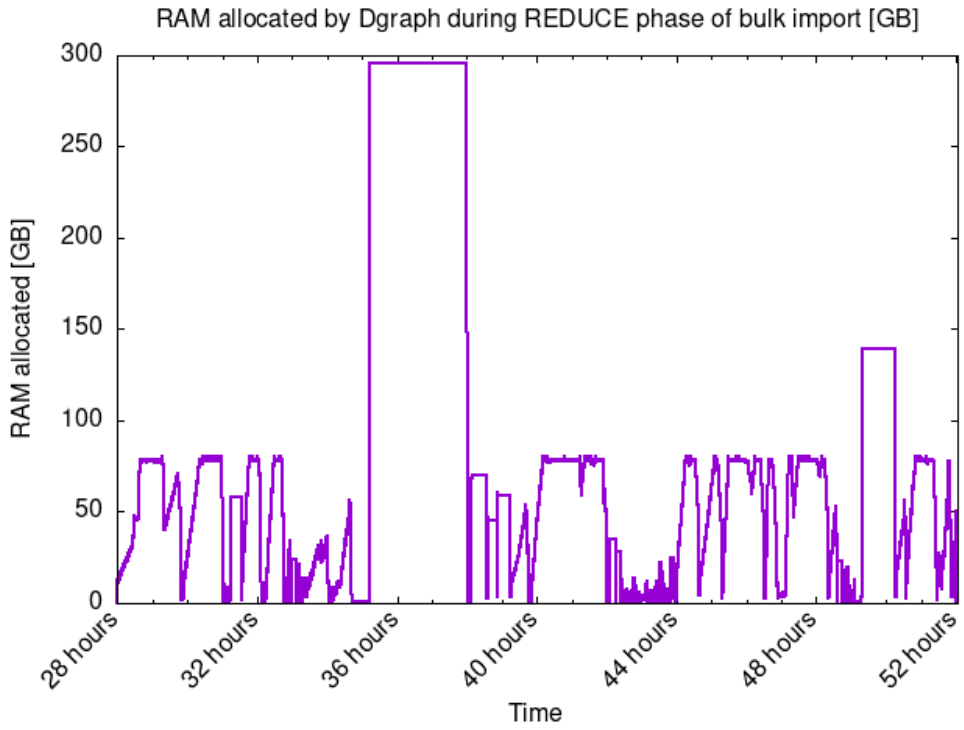

Figure 5.3.4: RAM allocated by Dgraph with jemalloc during the REDUCE phase of the bulk import.

The result of the bulk import is a folder called `p` that contains the actual data in Dgraph's binary format. The size of this folder was of 2.5 TiB.

From this folder, `alpha`, the node responsible for managing the actual data, was run with the command shown in Listing 5.6. All the process was done with a locally-compiled version of Dgraph, but it is possible to perform the same steps using Docker. The details of the imported data, divided by types, are reported in Table 5.3.3.

In contrast to importing the entire dataset, the import of the static data alone was completed in 1 hour, with an output of 112 GiB. This smaller dataset statically describes all the smart contracts of the Ethereum blockchain, without information on their usage.

```
dgraph alpha
    --my=localhost:7080 \
    --zero=localhost:5080 \
    --security whitelist=0.0.0.0/0 \
    --cache "size-mb=20000; percentage=50,30,20;" \
    --badger="compression=snappy; numgoroutines=64;"
```

Listing 5.6: Command used for running the `alpha` instance.

Type	Entries	Disk size	Uncompressed size
Transaction	1,967,716,025	1.3TiB	2.8TiB
Log	2,795,971,346	823.6GiB	2.2TiB
TokenTransfer	1,437,470,051	181.1GiB	414.0GiB
ContractDeployment	60,016,663	82.0GiB	161.3GiB
Account	286,391,265	29.7GiB	52.2GiB
ContractDestruction	55,152,100	9.7GiB	22.5GiB
Block	17,265,421	5.7GiB	16.7GiB
Skeleton	467,318	1.5GiB	7.0GiB
Withdrawal	3,688,662	285.7MiB	956.5MiB
Function	139,603	42.1MiB	84.9MiB
Event	9,690	2.1MiB	4.0MiB
Error	545	118.5KiB	228.6KiB

Table 5.3.3: Cardinalities and sizes of entries stored in Dgraph[6].

5.4 Querying data

Dgraph exposes two endpoints for querying the data: one for DQL at `/query` and one for GraphQL at `/graphql`.

DQL is the query language built by the Dgraph team to query and mutate data in this database. It has powerful features such as query variables, math on attributes, recursive queries and shortest path search. Under the hood, every GraphQL query is translated to DQL before being executed, so everything that can be done with GraphQL can be done with DQL, but not the opposite.

To easily perform DQL queries, Dgraph provides a web application called *Ratel*[7]. It gives a friendly user interface to get and visualize query results. Figure 5.4.1 shows an example of query in which it is visualized the ABI of a contract. Figure 5.4.2 shows accounts linked by transactions.

[6]Data about sizes was obtained querying the `/state` endpoint of the `alpha` instance.

[7]Ratel is a web application for querying, visualizing and managing Dgraph's data: `https://github.com/dgraph-io/ratel`

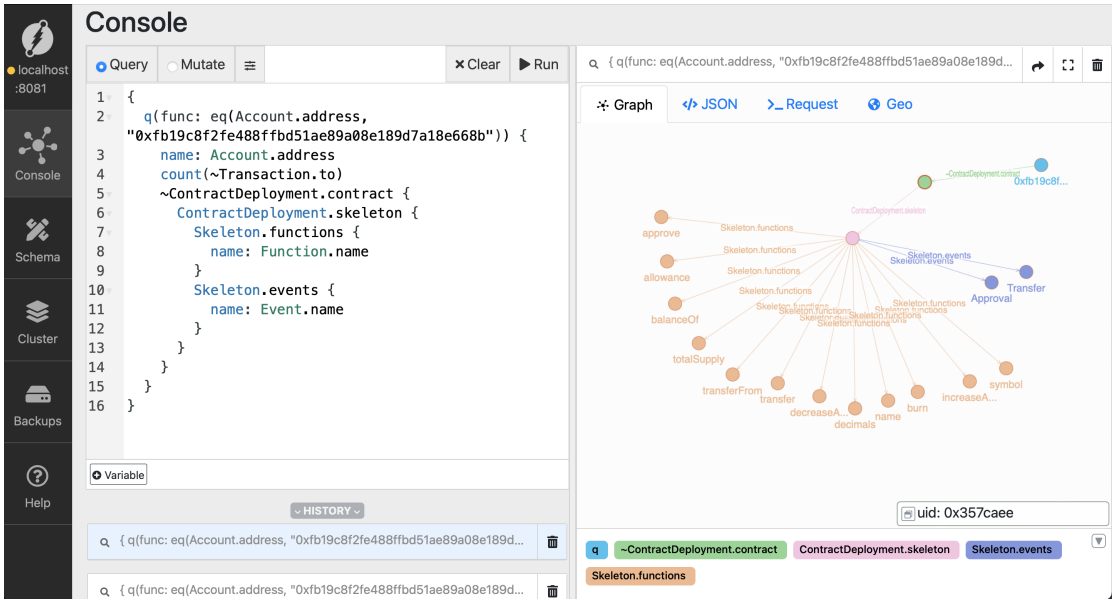

Figure 5.4.1: Visualization of Contract's ABI in Ratel.

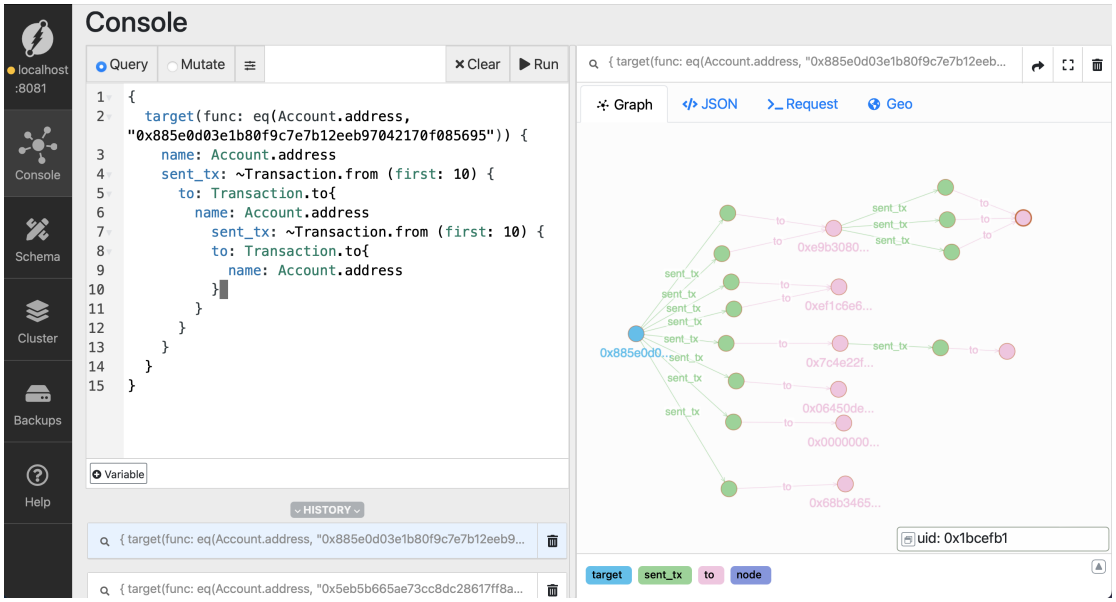

Figure 5.4.2: Visualization of Accounts linked by Transactions in Ratel.

5.4.1 Query performance

In this section, I provide examples of queries with their corresponding execution times. The processing time mentioned in table Table 5.4.1 is included by the database in the query responses. It does not include decoding, encoding or network transfers, just the time needed by Dgraph to get the data.

```
{
    q(func: eq(Account.address, "0
   xd8da6bf26964af9d7eed9e03e53415d37aa96045")){
        ~Transaction.from{
            expand(_all_)
        }
```

```
    }
}
```

Listing 5.7: Query to get all the transactions sent by a specific address. Response included 1071 transactions.

```
{
    BOREDAPE as var(func: eq(Account.address, "0
   xbc4ca0eda7647a8ab7c2061c2e118a18a936f13d"))
  q(func: uid(BOREDAPE)) {
        transfers: ~TokenTransfer.contract @filter(eq(
   TokenTransfer.token_id, "1020" )) {
            TokenTransfer.block {
                Block.number
                Block.datetime
            }
            TokenTransfer.from {
                Account.address
            }
            TokenTransfer.to {
                Account.address
            }
        }
  }
}
```

Listing 5.8: Query to get all the transfers of BoredApe NFT with id *1020* (686 transfers).

```
{
    var(func: type(Transaction)) {
        countVar as count(uid)
    }
    agg() {
        count: max(val(countVar))
    }
}
```

Listing 5.9: Query to count all the transactions in the database.

```
{
    q(func: eq(Function.name, "trade")) @normalize {
        ~Skeleton.functions {
            ~ContractDeployment.skeleton {
                ContractDeployment.contract {
                    address: Account.address
                }
            }
        }
    }
}
```

Listing 5.10: Query to get the addresses of contracts that implement a function with name *trade* (554 addresses).

```
{
    var(func: ge(Block.number, 12965000))  {
        price as Block.base_fee_per_gas
```

```
        used as Block.gas_used
        gasBurnt as math(price * used)
    }
    q() {
        totBurnt as sum(val(gasBurnt))
        ethBurnt as math(totBurnt / 1000000000)
        usdBurnt: math(ethBurnt * 1870)
    }
}
```

Listing 5.11: Query to compute the ETH burnt after London upgrade.

Query	Processing time [ms]
Listing 5.7	1671.46
Listing 5.8	416.70
Listing 5.9	73881.76
Listing 5.10	165.65
Listing 5.11	31868.36

Table 5.4.1: Processing time of DQL queries.

5.5 Comparison with Ethereum-ETL

Eth2dgraph was compared against Ethereum-ETL, one of the most popular open-source tools to export Ethereum data. The two tools serve the same purpose: extract and transform Ethereum data to be more usable. Ethereum-ETL does so by exporting data to CSV files, that can eventually be loaded into relational databases, while eth2dgraph does so by exporting data to compressed JSON files that can be imported into Dgraph.

At a high level, the two tools work in a similar way. They both fetch data using the Ethereum RPC interface. They both can be used through a CLI.

Ethereum-ETL does not integrate a decompiler to get the ABI of the smart contracts. It simply stores the raw four bytes of the function selectors that can be found as arguments of the `PUSH4` opcode at the beginning of the contracts' bytecodes.

Another design difference is in the steps needed to get the data. Ethereum-ETL needs multiple extraction steps to get certain kinds of information. For example, to get smart contracts data, it first needs to get and store the list of contracts' addresses, then, for each address, it fetches the contract's data. Eth2dgraph does everything in a single extraction step using the traces, to make the process faster.

The comparison was made by extracting data on various block ranges. There are multiple ways of using Ethereum-ETL. For this comparison, it was run with the command `export_all`. This command produces a similar output to the one of eth2dgraph, with the difference that it includes transaction receipts but it does not use the traces. The usage of receipts instead of traces means that all the contracts created by other contracts are not included in the output data.

Listings 5.12 and 5.13 shows the commands used to run the tools for the comparison. Both tools were run on the same machine and using the same Ethereum node. For the run on 400k blocks, Ethereum-ETL was run with batch size[8] reduced to 20. This was done since the default size of 100 caused the program to crash. The results are reported in Table 5.5.1.

```
ethereumetl export_all \
    -s <from> \
    -e <to> \
    -p http://localhost:8545 \
    -o <output-folder> \
    -w 2048
```

Listing 5.12: Command for running Ethereum-ETL in the comparison

```
eth2dgraph extract \
    -f <from> \
    -t <to> \
    --num-tasks 2048 \
    --include-tx \
    --include-logs \
    --include-transfers \
     -o <output-folder>
```

Listing 5.13: Command for running eth2dgraph in the comparison

Blocks range	Ethereum-ETL	eth2dgraph	Speedup
12000000-12001000	1m 57s	8s	14.6x
16000000-16010000	14m 18s	53.48s	16x
17000000-17100000	2h 56m 53s	6m 45s	26.2x
14000000-14400000	13h 30m 21s	31m 10s	48x

Table 5.5.1: Results of performance comparison between Ethereum-ETL and eth2dgraph.

Overall, eth2dgraph was at least one order of magnitude faster than Ethereum-ETL in all the extractions performed. This shows the effectiveness of the design choices taken while developing eth2dgraph. With growing block ranges, eth2dgraph becomes faster because the caching logic allows it to avoid decompiling contracts.

[8]The batch size indicates how many calls are sent together to the Ethereum node.

CHAPTER

SIX

ANALYSIS OF DATA

In this chapter, I analyse the data that have been extracted by `eth2dgraph`. Each section describes an independent analysis.

All the results refer to data from block 0 to block 17,265,420 of the Ethereum `mainnet`. It corresponds to the period of time between July 30 2015 to May 15 2023 (7 years, 9 months, and 15 days).

6.1 General data overview

This analysis is conducted to give a general overview of the data extracted. There are some interesting analysis results that help to understand the current state of the Ethereum chain. Here are some interesting points:

- 60,016,663 smart contract deployments, of which 213,406 failed, were found with 59,429,189 unique addresses. 88% of them (52,343,783) never emitted logs. 87% of them (51,974,197) never received a single transaction. Just 5.78% (3,435,332) of contracts' addresses both received at least one transaction and emitted at least one log. This shows that the vast majority of the deployed smart contracts are never actively used. Figure 6.1.1 shows the distribution of smart contracts based on their active usage.

- There are 2.8B logs emitted. 10 smart contracts alone, reported in Table 6.1.1, emitted the 24.25% of all logs. Top 100 contracts emitted 38.83% of the logs. Top 1,000 contracts emitted 57.53% of the logs. 33,287 smart contracts (0.056% of the total) emitted 90% of the logs. Most of the activity on the chain is restricted to a relatively small group of smart contracts.

- Transactions are more evenly distributed compared to logs. There are 165,684,328 distinct EOAs that have sent transactions. Top 10 senders sent 6.96% of all transactions. 90% of transactions were sent by top 25% addresses.

- There are 286,391,265 distinct addresses that have received transactions. The top 10 receivers received 18.55% of all the transactions, and there is just one receiver in the top 10 that is not a contract[1]. 90% of transactions were received by the top 57.85% receivers.

[1] It is an address of the Coinbase exchange `0xa090e606e30bd747d4e6245a1517ebe430f0057e`.

- 61.92% of transactions are sent to smart contracts. 0.25% are sent to the null address `0x0`. The remaining 37.83% are transactions from EOAs to other EOAs or unclaimed addresses.

- Figure 6.1.2 shows the history of smart contracts deployments. The growth in deployments was exponential until the beginning of 2018. After an initial period in which there were more deployments done by users than by contracts, now the majority of smart contracts are deployed using the CREATE or CREATE2 opcodes. This confirms the trend observed by Kiffer et al. [18]. They observed this phenomenon in data until 2018. The difference in deployments by users or by contracts has grown since then. Since 2019, the difference has been of at least one order of magnitude.

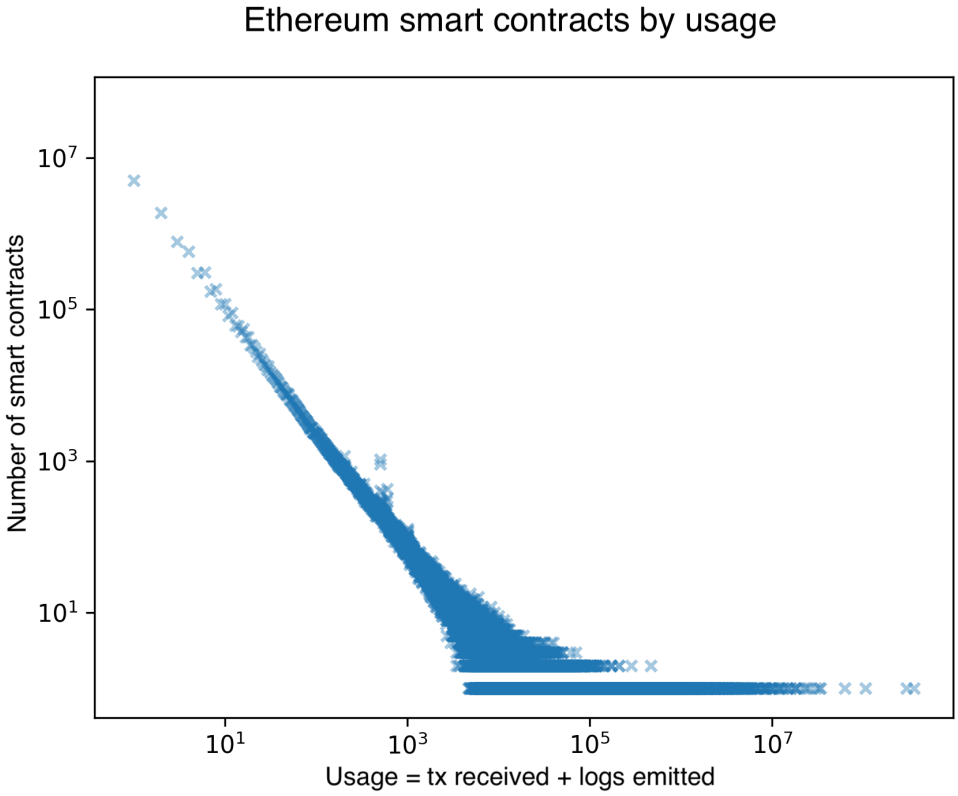

Figure 6.1.1: Ethereum smart contracts by usage, note the log scale on both axes.

Name of contract	Contract address	Logs emitted
Wrapped Ether	0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2	282,095,104
Tether USD	0xdac17f958d2ee523a2206206994597c13d831ec7	196,788,993
USD Coin	0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48	74,321,927
XEN	0x06450dee7fd2fb8e39061434babcfc05599a6fb8	30,438,737
DAI Stablecoin	0x6b175474e89094c44da98b954eedeac495271d0f	20,283,129
Seaport	0x00000000006c3852cbef3e08e8df289169ede581	16,764,010
ChainLink Token	0x514910771af9ca656af840dff83e8264ecf986ca	16,698,857
Wyvern Exchange	0x7be8076f4ea4a4ad08075c2508e481d6c946d12b	15,735,740
SHIBA INU	0x95ad61b0a150d79219dcf64e1e6cc01f0b64c4ce	12,607,046
Forsage	0x5acc84a3e955bdd76467d3348077d003f00ffb97	12,323,018

Table 6.1.1: Top 10 smart contracts per logs emitted.

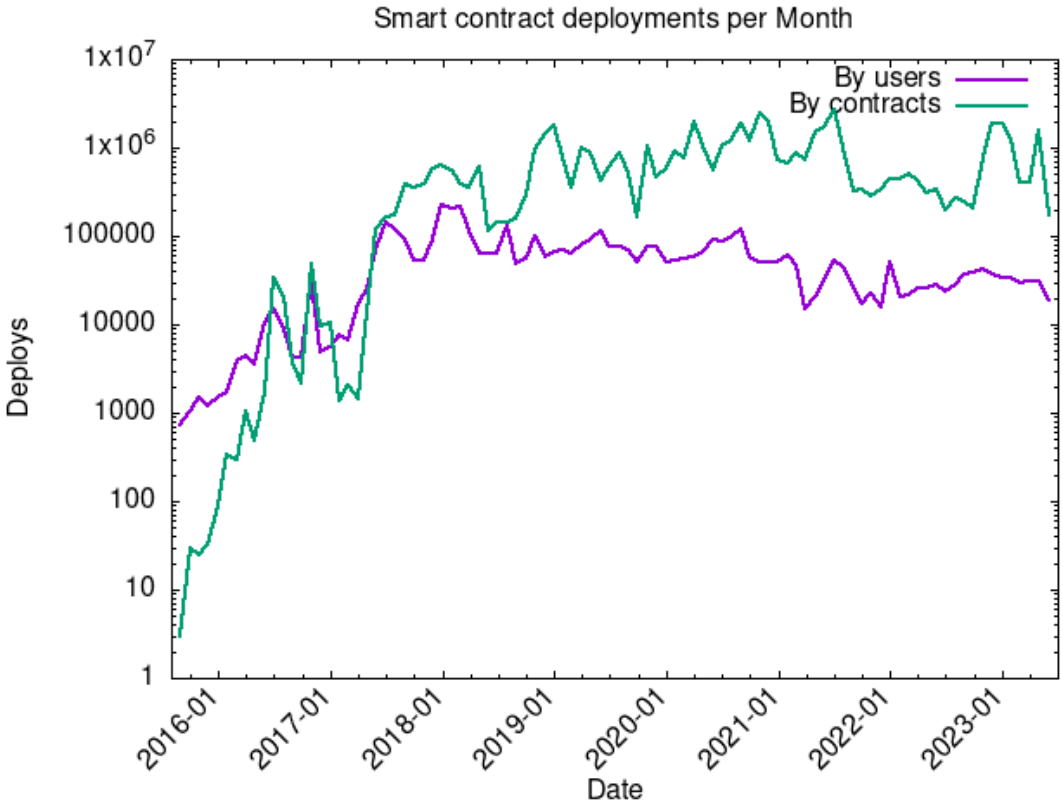

Figure 6.1.2: Smart contract deployments over time grouped by month.

6.2 Skeleton clusters

The skeleton of a smart contract is its deployed bytecode without the arguments of the PUSH opcodes and the eventual metadata appended at the end. Section 4.4.2 describes this concept and explains how eth2dgraph extracts the skeletons from

the Ethereum chain.

Out of the 60M smart contracts deployments in the history of Ethereum, just 467K distinct skeletons were found. This allows us to link smart contracts with each other based on skeleton equality. On average, each smart contract has 128 semantically identical siblings.

The distribution of smart contracts by skeleton is not uniform. There are 361,546 skeletons (77.4%) that correspond to a single deployed Ethereum smart contract. At the same time, there are skeletons that correspond to millions of smart contract deployments. The most frequently used skeleton matched 12.2M deployments and was related to gas tokens, as shown in Section 6.2.1.

6.2.1 Most deployed skeletons

I analyze the top 10 skeletons found on the Ethereum chain by a number of deployments. These 10 distinct skeletons correspond together to 44,389,576 smart contract deployments (73.96% of the total).

- The most used skeleton is related to gas reserves. It is the way gas tokens store gas when it is bought. The concept of the gas token is analyzed in Section 6.4. This skeleton has a very simple bytecode that just allows a particular address, with a length of 14 bytes, to destroy the contract. The size of this skeleton is 21 bytes. It has been deployed 12,240,689 times.

- The second skeleton is the implementation of the ERC-1167[2] logic. This bytecode consists of a minimal proxy that forwards all the calls it receives to a fixed hard-coded address. The size of this skeleton is of 45 bytes. It has been used 11,168,872 times.

- The third skeleton is again related to gas tokens. It is the same logic as the first described skeleton but with the allowed address of 15 bytes instead of 14. It has been used 6,829,142 times. Its size is 22 bytes.

- The fourth skeleton is simply the empty bytecode. It is valid in the Ethereum protocol to have empty smart contracts. 4,877,139 deployments with no bytecode were found.

- The fifth skeleton is again related to gas tokens. It is the same logic as the previous two but with the length of the allowed address of 20 bytes. 2,138,723 deployments matching this skeleton were found with a size of 27 bytes.

- The sixth skeleton represents 1,665,668 user wallets of the *Bittrex* exchange[3]. Each of these smart contracts is a controlled wallet. This means that each contract represents a user of the exchange, but the control over the Ethers and tokens remains under the company. The point of having these controlled wallets is to give users a unique address to send their tokens or Ethers. The size of this skeleton is of 502 bytes.

[2]Specification of the ERC-1167 Minimal Proxy Contract: `https://eips.ethereum.org/EIPS/eip-1167`

[3]Bittrex is a crypto exchange platform: `https://global.bittrex.com/`.

- The seventh skeleton has been used 1,549,146 times and represents an *OwnableDelegateProxy*. It is a proxy contract, so it simply forwards the received calls to another contract that implements the actual logic, using `delegatecall`. This specific type of proxy has two additional properties:
  - *Ownable*: it stores the address of the owner and allows it to modify the implementation address. The ownership can eventually be transferred.
  - *Upgradable*: it is possible to update the address of the implementation, changing where the proxy is forwarding the calls.

  All the previous logic is implemented in a bytecode with a size of 1073 bytes.

- The eighth skeleton has been used 1,542,310 times and represents a *forwarder contract*. This skeleton has two main public functions: *flush()* and *flushTokens(address)*. They are used to transfer ETH and tokens to a fixed parent address. The point of this kind of contract is to have multiple receive addresses for the same wallet. ETH transfers are automatically sent to the parent address, while tokens can be flushed with a transaction. Bitgo[4] uses this contract in their implementation of the multi-signature wallet[5]. The size of this skeleton is of 785 bytes.

- The ninth skeleton is a proxy used for the Ambi Multisig wallet as found by di Angelo and Salzer [21]. It has been deployed 1,202,291, with a size of 88 bytes.

- The tenth skeleton has been used 1,175,596 times and it is the exact same forwarding logic of the eighth skeleton. The few small differences are probably due to the compiler version or optimization level. Its size is 789 bytes.

I calculated cosine and interface similarity of the top 10 skeletons to find similarities between them and all the other skeletons. This formed seven clusters shown in Table 6.2.1.

These 7 clusters describe 75.29% of all the deployments that happened in the Ethereum blockchain. They can be grouped in just 4 distinct categories: *gas token*, *proxy*, *wallet* and *empty contract*.

[4]Bitgo is a digital asset trust company: `https://www.bitgo.com/`

[5]Source code of the multi-signature wallet: `https://github.com/BitGo/eth-multisig-v2/tree/master`

# Group	Distinct skeletons	Deployments	Category
1	5	21,787,384	Gas token
2	6	11,169,089	Proxy
3	1	4,877,139	Empty contract
4	31	2,732,644	Wallet
5	5	1,863,898	Wallet
6	20	1,55,2654	Proxy
7	3	1,202,787	Wallet

Table 6.2.1: Clusters formed by grouping top 10 skeletons with their similars.

6.2.2 New skeletons over time

An interesting metric to observe is when the skeletons were first seen on the blockchain. This is a different indicator to the one shown in Figure 6.1.2, since it just shows when semantically new smart contracts are deployed, avoiding all the replicas.

Figure 6.2.1 shows, for each month, the number of new skeletons found on the Ethereum chain. While the number of monthly deployments has not increased since 2021, the number of new monthly skeletons has kept increasing. This is also visible in Figure 6.2.2, especially in the year 2022 in which the ratio between deployments and new skeletons was low compared to the past. From this data, it appears that code reuse is dropping.

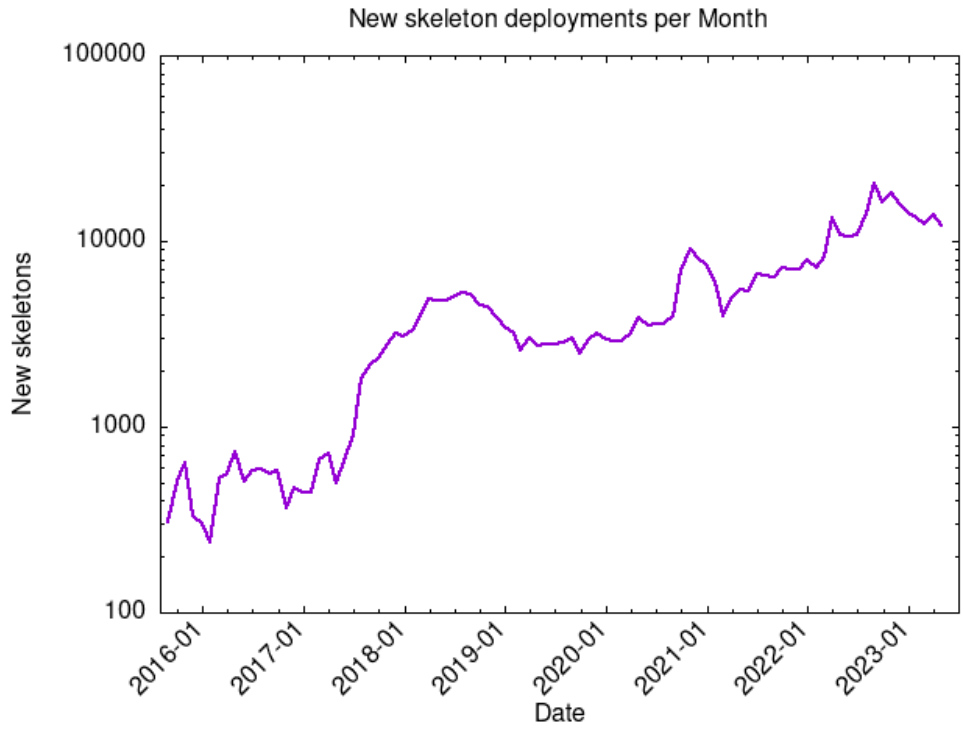

Figure 6.2.1: Deployments of new skeletons over time, grouped by month

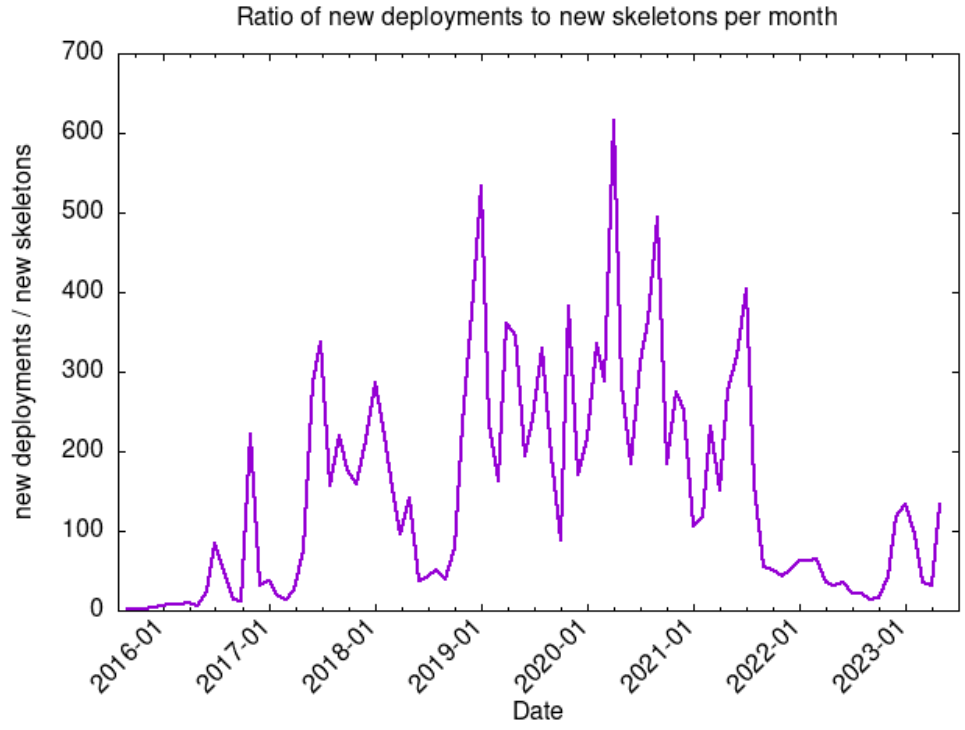

Figure 6.2.2: Ratio of deployments to new skeletons over time, grouped by month. High values imply more duplicates deployed.

6.3 Metamorphic contracts

Smart Contracts are commonly thought to be immutable: once deployed on the Ethereum blockchain their code can not be changed. This was true until the introduction of the `CREATE2` opcode in EIP-1014, included in the Constantinople Upgrade. This new opcode gives developers more control over the deployment address of the contracts created.

With the `CREATE` opcode, the new address is calculated as

$$a = keccak256(RLP(d, n_d))[12:]$$

in which d denotes the *deployer* address and n_d the deployer *nonce*. The nonce is updated by the protocol after every deployment. This prevents the possibility of deploying twice at the same address.

With `CREATE2`, the newly created address is computed as

$$a = keccak256(\texttt{0xff} \,||\, d \,||\, s \,||\, keccak256(c))[12:]$$

where `0xff` is a constant byte, s is a salt picked by the deployer and c is the initialization code of the contract. The developer invoking `CREATE2` has full control over all the variables, so it is easy to predict and manipulate the new address. The address to which the contract is deployed must be empty, this means that no contracts were ever deployed there or they were all previously destroyed.

With `CREATE2` it is possible to deploy a smart contract to a certain address a, then destroy it and re-deploy it again with the same bytecode at the same address a. This event is called *resurrection* by Fröwis and Böhme [17].

Since the initialisation code of a contract can read the blockchain state, it is possible to use it in a way such that the same initialisation code, run multiple times, results in different deployed bytecodes. An easy way of doing so is by instructing it to ask for a third smart contract for the code to deploy. This third contract can change the code it gives back from time to time. This is one way of deploying different bytecodes at the same address, creating a *metamorphic* smart contract. Listings 6.1 and 6.2 give an example of this pattern with pseudo code.

```
third_contract = address("0xab12...5134")
return third_contract.get_code()
```

Listing 6.1: Pseudo initialization code that gets the code to deploy from another contract.

```
bytes code_to_deploy;

function setCode(bytes calldata _data) public {
    code_to_deploy = _data;
}
function getCode() public returns (bytes memory) {
    return code_to_deploy;
}
```

Listing 6.2: Pseudo code of a contract that gives back the code to be deployed.

Another more complicated way to replace the deployed code of a contract is by combining `CREATE` and `CREATE2` together. `CREATE2` is used to reset the nonce used in `CREATE`. This can be done with the following steps:

1. A deployer contract D creates a *factory* contract, here called F, at address A_f using `CREATE2`. A_f is computed as $keccak256(\texttt{0xff} \,||\, A_D \,||\, s \,||\, keccak256(c))[12:]$, in which s is any random salt and c is the initialization code of F.

2. Through F, a new contract C_1 is created using the `CREATE` opcode at address A_{c1}. A_{c1} will be calculated as $keccak256(RLP(A_f, n_d))[12:]$, in which n_d, the nonce, is zero.

3. The factory F is destroyed with `SELFDESTRUCT` and redeployed again by D at the same address A_f and with the same code. This is achieved using `CREATE2` with the same parameters. This step is needed to reset the nonce of F.

4. The contract C_1 is destroyed with `SELFDESTRUCT`.

5. Now, F can deploy a new contract with arbitrary code using `CREATE`. The newly created contract will have the same address A_{c1}, since it is calculated as $keccak256(RLP(A_f, n_d))[12:]$ with n_d equals to zero, as in step 2.

6.3.1 Overview of metamorphic contracts usage

Fröwis and Böhme [17] analyzed metamorphic contracts until July 2021. They found 41 accounts that received deployments with different bytecodes. From a manual analysis, they concluded that this phenomena was used by just a few experienced users. They did not find any malicious use of this pattern. Most of the cases were about smart contracts related to the front-running infrastructure.

I here analyze the usage of this pattern in the data until May 15 2023, almost two years of data after the analysis conducted by Fröwis and Böhme.

A total of 267,461 accounts received multiple deployments of the **same** bytecode, these are the resurrected accounts. For the metamorphic pattern, there are 524 distinct accounts that have mutated bytecode over time. Out of these 524, 295 have probably used the pattern with just `CREATE2`, since all the initialization codes were identical. While the remaining 229 accounts probably used the pattern combining `CREATE` and `CREATE2` since the deployments used different initialization codes. In total, these 524 accounts recorded 1,774 deployments and received 8,687,083 transactions. A CSV dump with many information related to metamorphic contracts has been published online at this address: `https://gist.github.com/davideaimar/e115098af481b16d6755b2e5acc04309`.

Figures 6.3.1 and 6.3.2 show the first appearance of metamorphic contracts over time. Figure 6.3.1 shows all the data, while Figure 6.3.2 filter out three outliers. These three outliers are three days in which there were more than 100 first appearances of metamorphic smart contracts. These contracts are clearly correlated with each other.

The cause these outliers were just two addresses[6] that performed 852 deployments to metamorphic smart contracts in just three days. One of the addresses

[6] `0x3c3e8ab1e3327f24c917cd28789c9464adcf8198` and `0x6b25909c6141daf60ddf7c0700cedce07a9493d7` with respectively 596 and 256 metamorphic deployments.

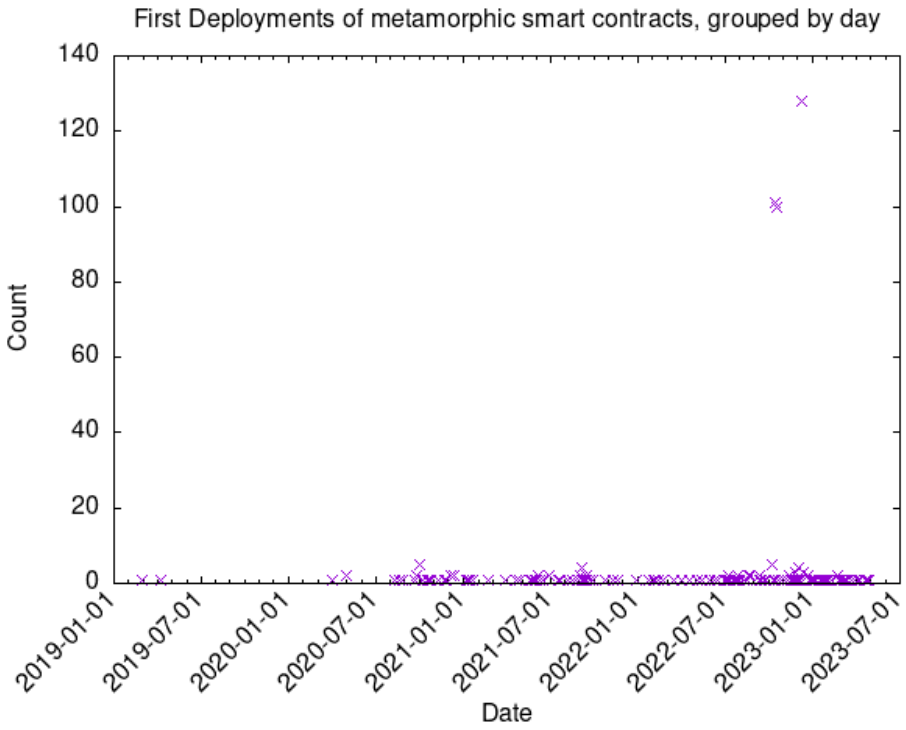

Figure 6.3.1: First deployments of the metamorphic smart contracts.

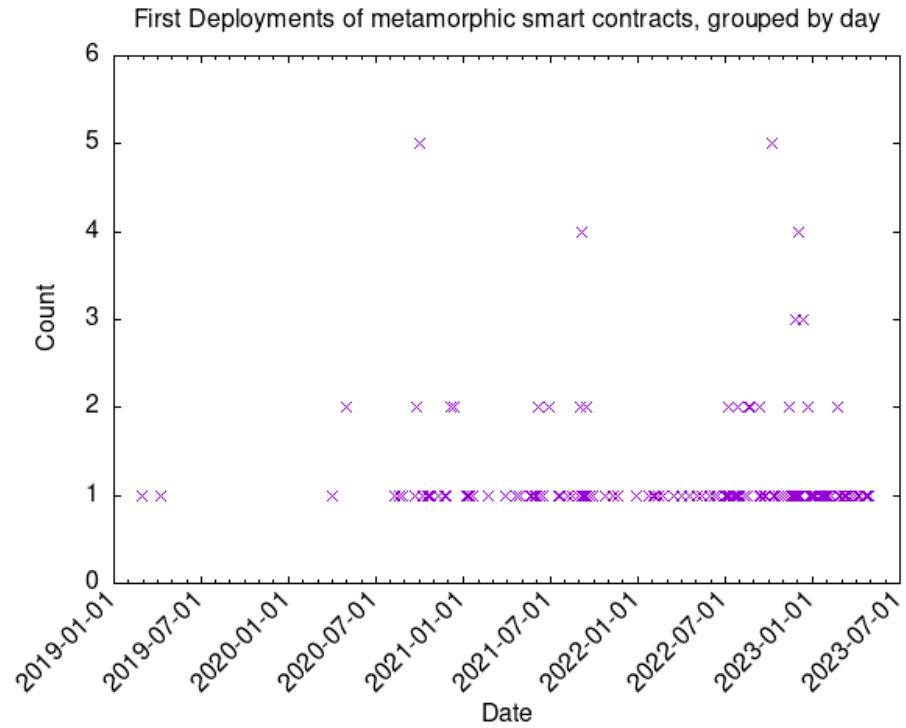

Figure 6.3.2: First deployments of the metamorphic smart contracts without the three outliers.

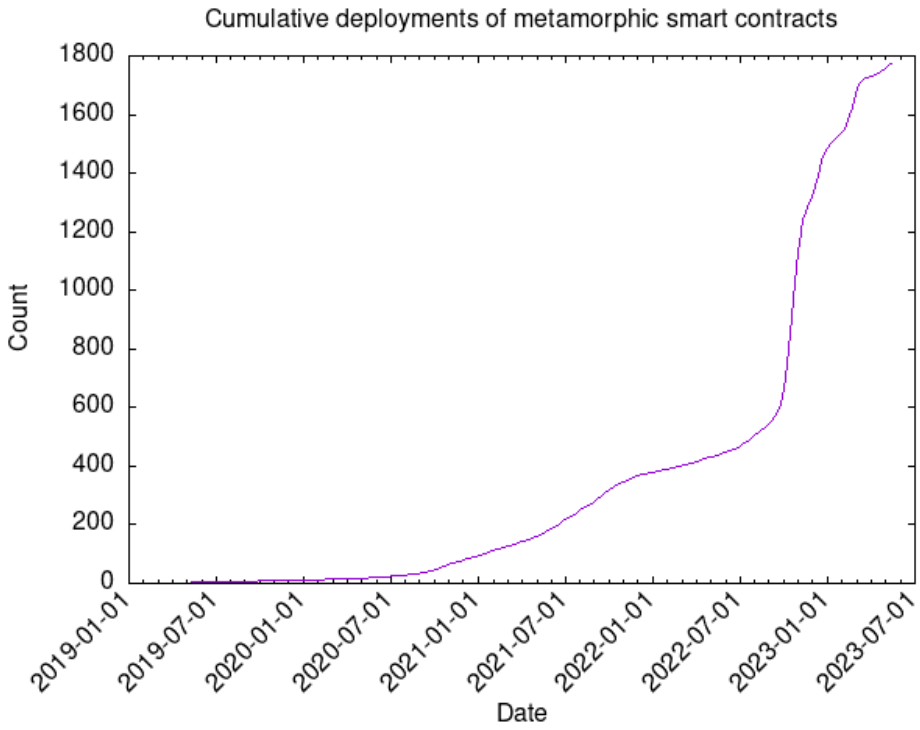

Figure 6.3.3: Cumulative sum of all the metamorphic deployments.

is used for MEV[7] activity, while the other deployed metamorphic contracts that were all used to mint the XEN[8] token.

Figure 6.3.3 reports the cumulative sum of deployments to the 524 metamorphic smart contracts. In general, the usage of the metamorphic pattern has increased in the last period, especially since 2022.

As found by Fröwis and Böhme, many metamorphic contracts use *vanity addresses*. These addresses have the peculiarity of having at least 7 leading zeros, meaning that they are smaller than the typical 20 bytes addresses. They are used to save on gas fees since it is less data that has to be stored on the blockchain. Out of the 524 metamorphic smart contracts found, 74 have a vanity address.

Understanding the purpose of metamorphic smart contracts is not easy. All the deployments do not have verified source code. Most of them have low-level bytecode with no decompiled functions (1660 out of 1781 deployments).

I manually analyzed the top 110 metamorphic contracts by number of transactions received. I tried to understand their usage and I was able to flag the type of 98 contracts. 12 contracts were unclear what they were used for. Out of the 98 flagged contracts, 93 (94.89%) contracts were found to be associated with MEV activity. This means that they were flagged as MEV by Etherscan or by Eigenphi[9]. The remaining 5 contracts were all used to mint the XEN token.

This confirms the trend observed by Fröwis and Böhme, the metamorphic

[7]MEV refers to multiple practices to maximize the extractable value from the block. It includes frontrunning, arbitrage and liquidations.

[8]XEN is an ERC-20 token available at `https://etherscan.io/token/0x06450dEe7FD2Fb8E39061434BAbCFC05599a6Fb8`

[9]Eigenphi is a platform that collects and analyse blockchain data related to MEV activity. It is available at `https://eigenphi.io/`.

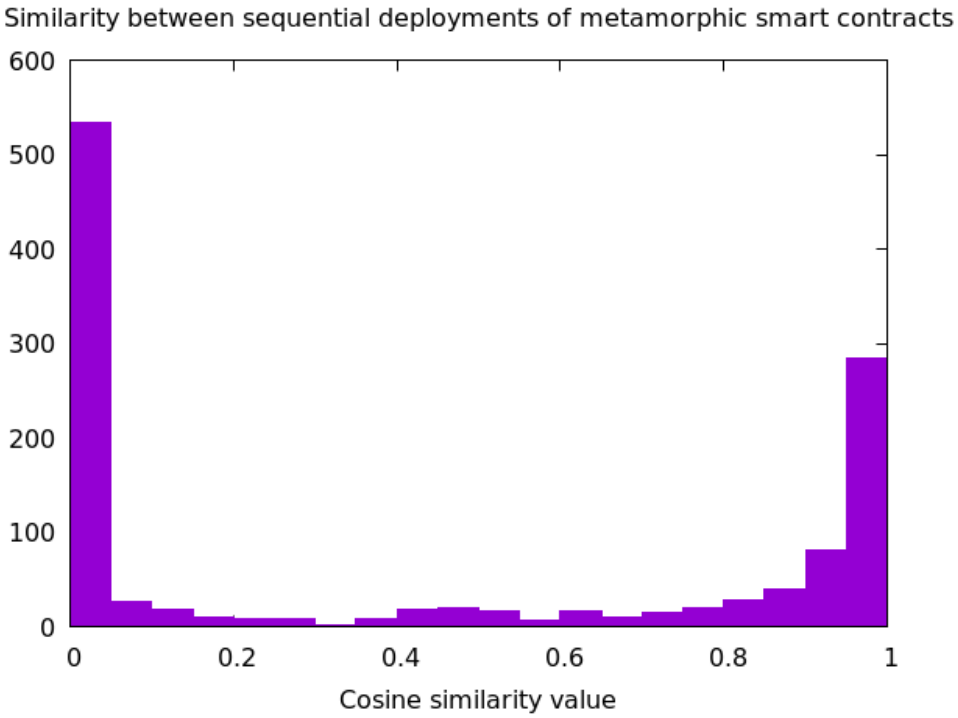

Figure 6.3.4: Similarity values of all metamorphic deployments.

smart contracts are still restricted and used by a small number of users. Most of the metamorphic contracts that are effectively used for, or related to MEV activity. In many cases, the usage of this pattern is probably done for reusing vanity addresses, since they are hard to find and can have an impact on the MEV revenues.

6.3.2 Similarity between metamorphic deployments

To understand how much the code changes in between deployments of metamorphic contracts, I computed the cosine similarity between sequential deployments. For example, if a contract has received 3 deployments, I computed the similarity between the first and the second bytecodes and then between the second and the third bytecodes. The calculation of the similarity has been done as described in Section 4.6.

The resulting values are plotted in Figure 6.3.4. A considerable amount of deployments completely changed the bytecode, with a similarity value of zero. Inspecting these bytecodes, I observed that most of these deployments were identical and followed a single pattern:

- The first and the second deployments were empty bytecodes.
- The third deployment was an implementation of the ERC-1167 minimal proxy contract.

All of the deployments that followed this pattern were done by the same address and were used to mint XEN tokens.

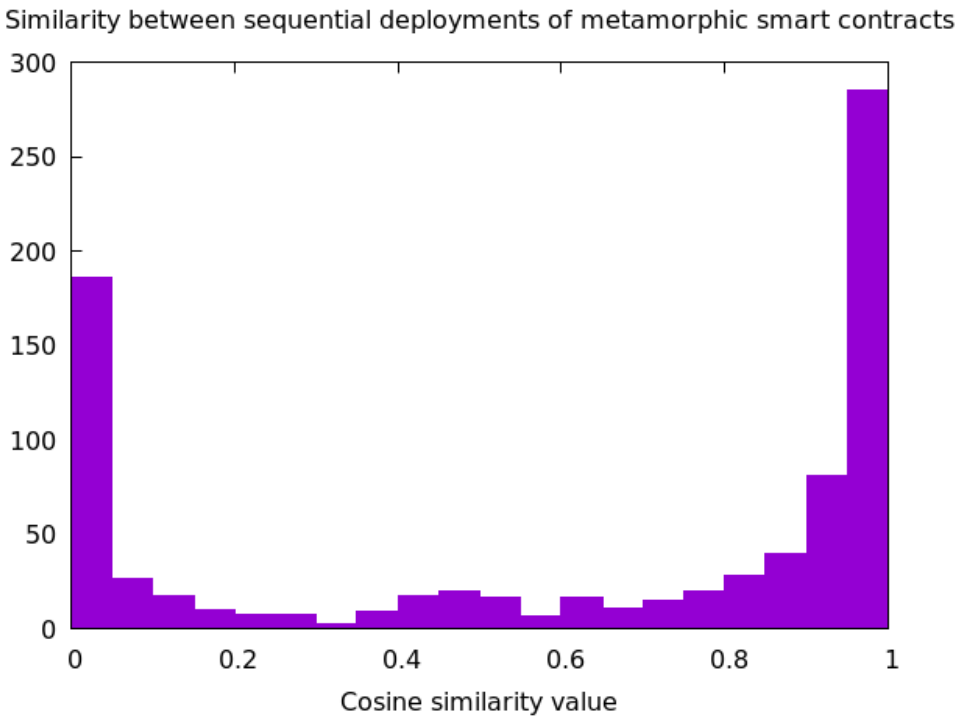

Figure 6.3.5: Similarity values of metamorphic deployments excluding a pattern that occurred identically multiple times.

Excluding these deployments from the visualization showed that in the majority of cases the new bytecode is very similar to the one that is replaced. It is visible in Figure 6.3.5.

6.4 Gas tokens

The aim of this analysis is to study the impact of the *GasToken* pattern on Ethereum.

GasToken is a pattern that was heavily used on the Ethereum blockchain to save on gas fees. It exploited the concept of refund provided by the op-codes `SELFDESTRUCT` and `SSTORE`. I analyze and focus on the pattern that uses `SELFDESTRUCT`. It works by creating and destroying basic smart contracts, used as gas reserves.

This pattern caused the creation of many fuzzy contracts and state slots that increased the size of the Ethereum state. It was the main reason for the adoption of EIP-3529 [24] on August 5th 2021. This EIP removed refunds for `SELFDESTRUCT` and reduced `SSTORE` refunds, effectively killing gas tokens.

The following information explains how this pattern worked before EIP-3529. These two concepts are the fundamentals of this pattern:

- When a contract is deployed, the creator needs to pay 32000 gas + 200 gas for each non-zero byte stored.
- When a contract is destroyed, a refund of 24000 gas is provided to the destroyer, after paying 700 + 5000 gas for calling `CALL` + `SELFDESTRUCT`.

So the idea is that users deploy fuzzy contracts when gas is cheap and destroy them when gas is expensive. Gas from the refunds can cover up to 50% of the gas used by the calling transaction that triggered the destructions, this is a limit introduced by the Ethereum protocol.

To make this concept accessible, there are a few smart contracts that abstract the logic into simple tokens. For each token minted, there are many underlying smart contracts deployed. This token can easily be transferred between users. When the token is freed, the underlying smart contracts are destroyed and the owner of the token gets a discount on the gas of the transaction. The two most used tokens are CHI[10] and GST2[11].

To be profitable, the price of the gas when tokens are bought must be at least half of the price of the gas when they are sold.

Gas price has been historically very volatile, so the existence of this pattern makes sense. Figure 6.4.1 shows the historical fluctuation of gas prices.

6.4.1 Identification of gas reserves

I identified all the smart contracts ever deployed that were used as gas reserves. The logic of a gas reserve contract is basic. It simply allows one hard-coded address to destruct the contract. Listing 6.3 shows the code that implements this logic, it is translated to the EVM bytecode reported in Listing 6.4

```
if (msg.sender == GAS_TOKEN_ADDRESS) {
    SELFDESTRUCT(msg.sender);
```

[10] Chi Gastoken on Etherscan: `https://etherscan.io/token/0x0000000000004946c0e9F43F4Dee607b0eF1fA1c`

[11] GST2 token on Etherscan: `https://etherscan.io/token/0x0000000000b3F879cb30FE243b4Dfee438691c04`

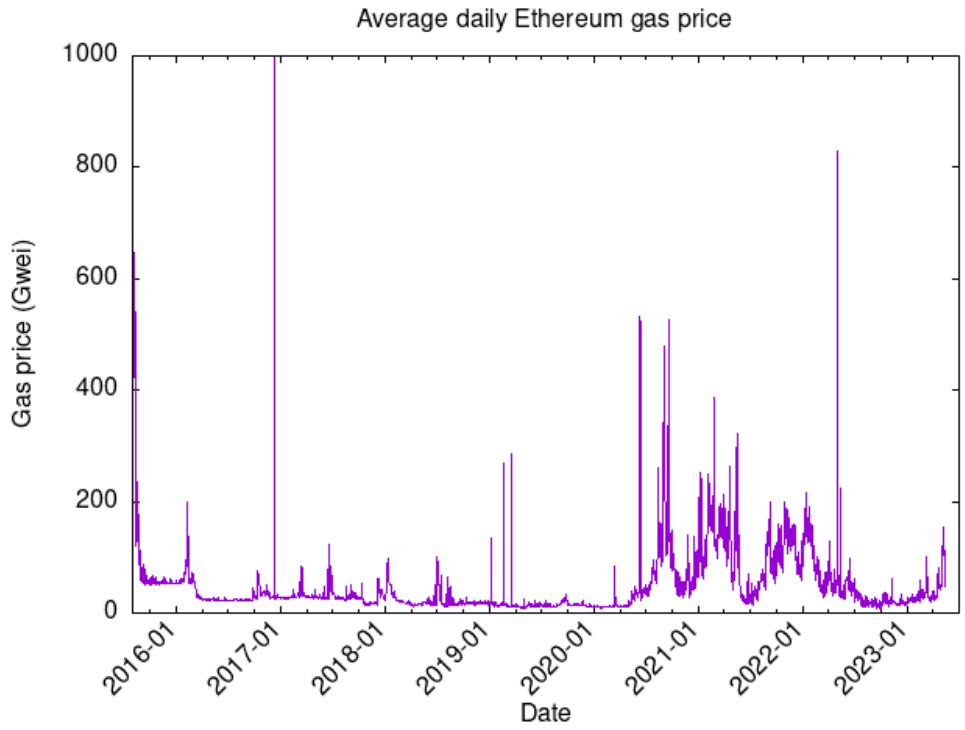

Figure 6.4.1: Average daily Ethereum gas price over time.

```
}
```

Listing 6.3: Pseudo code of the gas reserves.

```
PUSH* <address of token contract>
CALLER
XOR
PC
JUMPI
CALLER
SELFDESTRUCT
```

Listing 6.4: EVM bytecode of the gas reserves.

The `PUSH` opcode is represented as `"*"` because there are different implementations of the gas reserve contract. Here it is possible to perform optimisations. The shortest the allowed address is and the fewer bytes are needed to be stored in the contract bytecode. This results in cheaper deployments and more efficiency of the pattern.

For example, the GST2 gastoken has this address: `0xb3f879cb30fe243b4dfee438691c04` that has just 15 bytes instead of the standard 20. The CHI gas token, a more recent and optimized alternative, uses `0x4946c0e9f43f4dee607b0ef1fa1c` that is one less byte. Finding these short addresses is a very resource-intensive computation and requires trillions of iterations and hashes.

I identified all the gas reserves using the skeletons. There are five distinct skeletons that were used as gas reserves, with the only difference in the type of `PUSH`. Some of the gas reserves were deployed multiple times at the same address. The data found is reported in Table 6.4.1.

Skeleton	PUSH	Deployments	Distinct addresses
6d00...003318585733ff	14	12,216,500	12,188,707
6e00...003318585733ff	15	6,809,029	6,765,219
6f00...003318585733ff	16	568,116	525,202
7000...003318585733ff	17	9,577	9,577
7300...003318585733ff	20	2,138,608	2,138,608

Table 6.4.1: Gas reserves found on Ethereum.

A total of 21,741,830 successful gas reserve deployments were found, more than one third of all the deployments of Ethereum. 90.1% of them have been destroyed, the remaining contracts are still alive. Users should pay gas to destroy them but without a reward is very unlikely that this will ever happen. There are 2,158,422 contracts that can potentially be stuck there forever since there is no incentive to remove them from the blockchain. The timeline of deployments and destructions is shown in Figure 6.4.2. It is clearly visible how the London upgrade successfully killed this pattern.

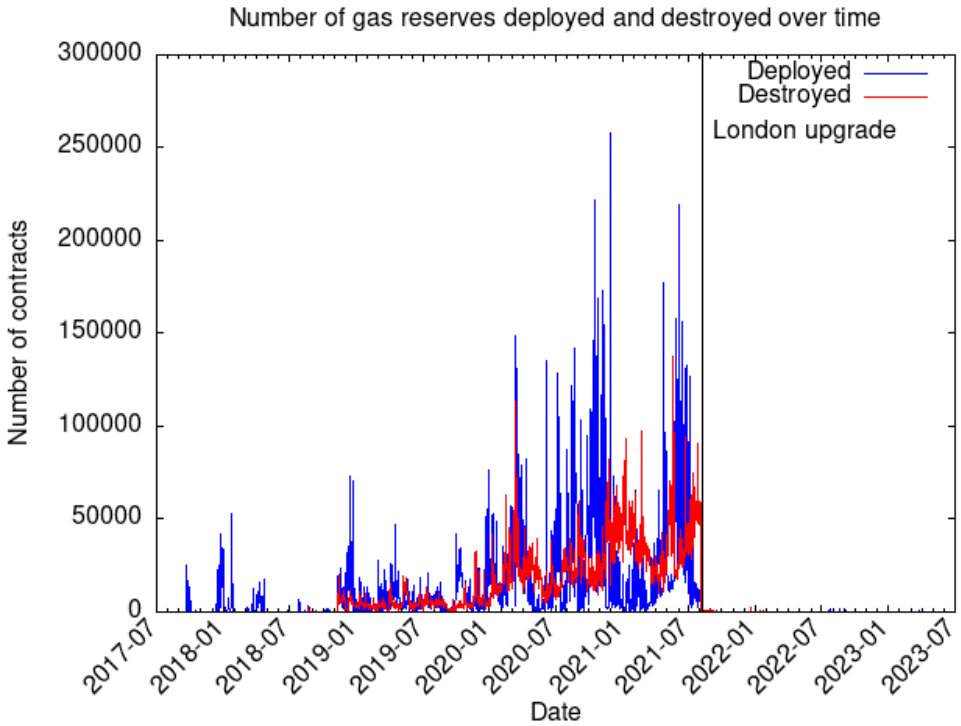

Figure 6.4.2: Deployments and destructions of gas reserves over time.

6.4.2 Quantification of eth saved

I estimated the amount of Eth saved using the GasToken pattern. The calculation is performed on each gas reserve as:

$$eth_{saved} = eth_{refund} - eth_{mint}$$

where eth_{refund} and eth_{mint} are calculated as

$$eth_{refund} = gas_{refunded} * gas\_price_{refund\_time}$$

$$eth_{mint} = gas_{mint} * gas\_price_{mint\_tine}$$

This is calculated on each gas reserve that was successfully deployed and then destroyed. This calculation is meant to give an order of magnitude of the amount saved.

These are the limitations and assumptions made for this calculation:

- The gas spent for the small logic introduced by the token contracts that manage most of the gas reserves is not considered.
- Gas spent for the transactions that deployed the reserves is not considered. It highly depends on how many deployments were done in a single transaction. For example, doing 100 deployments in a single transaction (equivalent to minting 1 GST2 token) makes the gas of the transaction count for just 0.58% of the deployment cost. I assume that all deployments were done in batches greater than 100 so the transaction cost is negligible.
- I assumed that all the destructions were done in such a way to not cover more than 50% of the transaction gas with the refunds. Not doing so would mean wasting the refunded gas.
- The gas prices used are the ones related to the blocks, obtained as the averages of the prices of gas in all the transactions. It is possible that many reserves were deployed by the miner themselves without paying for the gas.

The obtained value estimated a total saving of around 14K Eth, corresponding to 22,930,740 USD. The code used for this estimation is reported in Listing 6.5.

```
# df contains all the gas reserves deployments

# Type specifies wich PUSH was used (14, 15, ...)
# 24_000 is the refund
# 5_700 is the gas price for triggering SELFDESTRUCT + CALL
gas_refunded = 24_000 - 5_700

# 32_000 is the cost of CREATE
# 200 is the cost for each byte stored
df['deploy_gas_used'] = 32_000 + 200 * ( 7 + df["type"] )
df['deploy_cost'] = df['deploy_price'] * df['deploy_gas_used']
df['destroy_reward'] = gas_refunded * df["destroy_price"]
df['profit'] = df['destroy_reward'] - df['deploy_cost']
saved = df['profit'].sum()
```

Listing 6.5: Code for computing the total Eth saved with the GasToken pattern.

6.5 Most deployed functions and events

Understanding what are the most commonly used functions and events gives a hint about what most smart contracts are doing.

I present in Tables 6.5.1 and 6.5.2 the most frequently deployed functions and events extracted using the Heimdall EVM decompiler. As explained in Sections 4.4.1 and 4.5.1 the decompilation is performed on the deployed bytecode of the smart contracts, with a caching logic based on EVM skeletons. This means that two contracts sharing the same EVM skeleton received just one decompilation and the linked functions and events are the same. The numbers of this analysis depend directly on the accuracy of the decompiler.

Function	Skeletons	Deployments
`sweep(address,uint256) -> bool`	96	2,794,735
`flush()`	182	2,775,935
`flushTokens(address)`	106	2,762,395
`owner() -> address`	57,643	2,226,240
`tokenFallback(address,uint256,bytes)`	68	1,897,675
`implementation() -> address`	1,472	1,688,820
`proxyType() -> uint256`	96	1,581,439
`upgradeTo(address)`	1,102	1,577,083
`transferProxyOwnership(address)`	177	1,556,545
`proxyOwner() -> address`	185	1,556,297
`upgradeabilityOwner() -> address`	127	1,554,012
`upgradeToAndCall(address,bytes)`	8	1,550,005
`transferOwnership(address)`	46,597	581,149
`balanceOf(address) -> uint256`	57,144	563,664
`name() -> bytes memory`	55,842	535,428
`symbol() -> bytes memory`	54,714	530,246
`totalSupply() -> uint256`	54,511	529,351
`transfer(address,uint256)`	46,024	526,022
`approve(address,uint256) -> uint256`	51,984	515,168
`decimals() -> bool`	44,682	503,589

Table 6.5.1: Top 20 functions by number of deployments.

Event	Skeletons	Deployments
`TokensFlushed(address,uint256)`	14	2,720,369
`Upgraded(address)`	1,115	1,577,868
`ProxyOwnershipTransferred(address,address)`	165	1,556,455
`OwnershipTransferred(address,address)`	41,985	516,317
`Transfer(address,address,uint256)`	47,380	494,780
`Approval(address,address,uint256)`	48,499	482,358
`OwnerChanged(address)`	133	430,676
`DeedClosed()`	4	430,173
`SafeModeActivated(address)`	51	231,379
`TokenTransfer(address,address,uint256)`	11	84,781
`Transfer(address,uint256)`	64	63,185
`TokenReleased(address,uint256)`	14	49,736
`Burn(address,uint256)`	3,136	37,404
`OwnershipRenounced(address)`	2,408	24,993
`ApprovalForAll(address,address,bool)`	10,063	22,451
`AdminChanged(address,address)`	520	20,353
`LogSetOwner(address)`	196	16,921
`DelegateUpgraded(address,address,uint256)`	2	14,076
`DelegateRolledBack(address,address,uint256)`	2	14,076

Table 6.5.2: Top 20 events by number of deployments.

All the functions are related to proxy, wallet, or token contracts. For the events, it is the same situation. There is the event `DeedClosed` that is related to ENS[12] and `DelegateUpgraded` and `DelegateRolledBack` that are used for the Rocket Pool[13] protocol, but both of these protocols are token-based.

[12]ENS (Ethereum Name Service) is a decentralized name service `https://ens.domains/`.
[13]Rocket Pool is a decentralized staking pool `https://rocketpool.net/`.

6.6 Contracts metadata

Smart contracts deployed using the Solidity compiler have the default option to include CBOR-encoded data at the end of the deployed bytecode. This piece of information includes:

- The hash of the contract metadata. The metadata includes any kind of information related to the smart contract, such as the ABI, the documentation, the settings of the compiler, etc. It also includes the hash of the source code, so a change in the source code causes a change in the metadata that consequently changes its hash. This hash can be used as an address to store and retrieve the actual metadata and source code of the contract on a decentralized file system.
- The type of the hash (`bzzr0`, `bzzr1` or `IPFS`).
- A flag stating if the compilation was done with experimental features of the compiler enabled.
- The version of the Solidity compiler used.

All of this data has been extracted by eth2dgraph with a regex as explained in Section 4.4.2. Here is an overview of what has been extracted.

6.6.1 Hash of metadata

A total 17,491,909 deployments included the hash of the metadata in the bytecode stored on the blockchain. Out of these, 1,164,973 (6.66%) used `IPFS`, 15,636,747 (89.39%) used `bzzr0` and 690,189 (3.94%) used `bzzr1`.

Analyzing the values of the hashes, there were just 770,719 distinct hashes found. This means that, on average, each smart contract compiled with the Solidity compiler gets deployed 22.7 times, another indicator of the high code reuse in the Ethereum blockchain. There are five occurrences in which the same metadata hash has been deployed more than 1M times, all of these deployments were done by just a few distinct addresses. On the other hand, there are 690,157 occurrences in which the hash was only used once.

6.6.2 Experimental compilations

The Solidity compiler allows to activate experimental features that are not already included by default in the decompiler. This can be done at the beginning of the code writing `pragma experimental <feature-name>`. Inside the CBOR encoded data appended at the end of the generated bytecode there is a boolean stating whether any experimental feature was used or not.

Out of the 17,491,909 deployments that included the CBOR encoded data, 1,113,139 (6.36%) had experimental features activated. These contracts with experimental features received a total of 5.8M transactions.

6.6.3 Solc versions

2,090,487 smart contracts included the version of the Solidity compiler in the CBOR-encoded data. The version is included just from Solidity v0.5.9 onward. There are 1,299 smart contracts that included fake Solidity versions (e.g. 100.67.137, 116.153.33, etc.) and were removed in this analysis. 32 deployments were found to use pre-releases of the compiler, in these cases Solidity appends the exact commit used. The total numbers of major versions found are reported in Table 6.6.1.

Major Solidity version	Number of deployments
0.5	925,506
0.6	287,552
0.7	215,924
0.8	660,206

Table 6.6.1: Numbers of deployments found per major version of the Solidity compiler.

Figure 6.6.1 shows the distribution over time of the various major Solidity versions found. Generally, it is possible to observe how old compiler versions remain heavily used even after the release of multiple new versions. The versions 0.6 and 0.7 almost never managed to have a higher amount of daily deployments than the 0.5 version. The latest version, 0.8, managed to overtake the 0.5 after around one year since its release. Now it is the most used Solidity version.

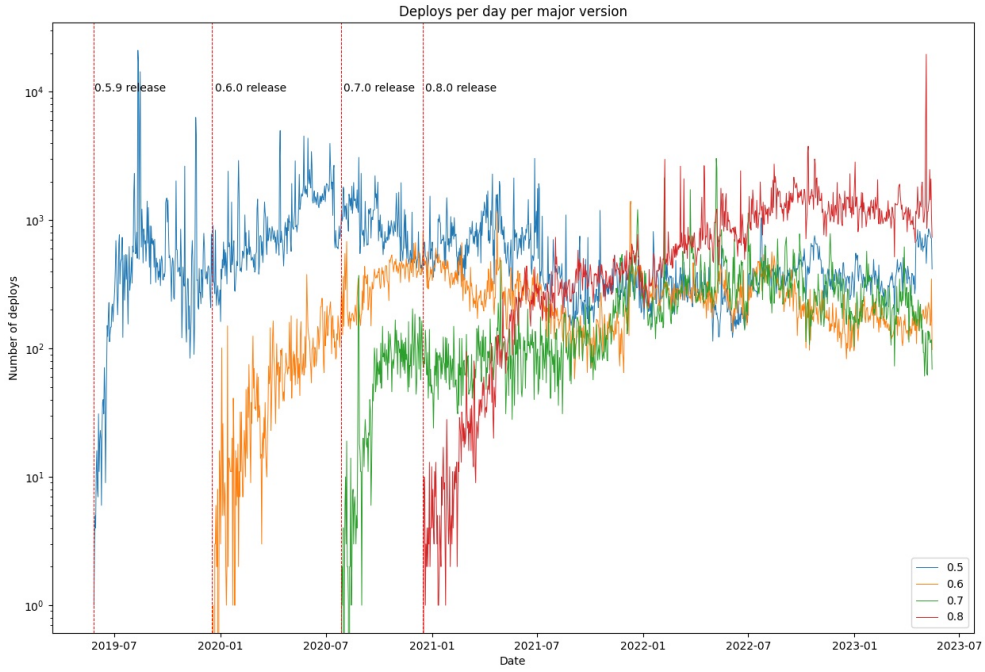

Figure 6.6.1: Deployments over time divided by major Solidity compiler versions. Each data entry represents the daily amount of deployments found per version. Keep in mind the log scale.

CHAPTER

SEVEN

DISCUSSION

7.0.1 Dgraph for Ethereum data

As blockchain technology gains popularity, the challenge of managing data from these networks becomes increasingly critical. Dgraph proved to be a viable solution for managing Ethereum data. This kind of data adapts well to graph databases.

However, there are some drawbacks that were encountered when using Dgraph:

- EVM data is often represented using 256-bit integers or raw bytes. Dgraph does not support these data types. They must be stored as text with all the related disadvantages, such as more disk space used and the impossibility of doing math operations in the queries.
- When used with a large dataset, Dgraph encountered some limitations with the usage of memory. It crashed both during the bulk import and during the execution of large queries. These problems were solved by modifying the database source code or tweaking the queries to use less memory. However, they show that Dgraph is still not well-tested against large datasets.
- Insertion of live Ethereum data into a cluster with all the history of the blockchain was slower that the production of data from the blockchain. The underlying Badger database gets stuck periodically logging `L0 was stalled`. This made it impossible to keep the dataset updated with the latest blocks.

Dgraph is a relatively new technology. It was born in 2016 and is currently under active development. The project maintainers showed interest in this use case and actively helped me to solve the problems I faced with the database.

7.0.2 Challenges of blockchain data management

The process of data extraction using the Ethereum RPC interface worked well and proved to be an optimal solution. The biggest problem of this process is the amount of time and computational resources it needs, as described in Chapter 5.

It is important to highlight that these data refer to the Ethereum blockchain. Layer 2 protocols and other blockchains are even more critical as they are already

producing more data than Ethereum. Polygon[1] is producing blocks 6 times faster than Ethereum, with an average of one new block every two seconds. The BNB Chain[2] is another popular blockchain that is growing at a pace of a new block every three seconds.

Another relevant concern is that blockchains will grow indefinitely. The size of an instance of a Geth archive node is growing at around 3.5TB per year[3]. Soon, it will not be possible to handle all this data on a single machine, since vertical scalability is not infinite. A distributed approach will be the only viable way in the future.

This expensive entry barrier makes it hard to perform such an operation. People interested in analyzing Ethereum data are more likely to use one of the few centralized services instead of running their infrastructure.

The lack of research and tools in this field could potentially create a gap between companies and the open-source community. This would reduce the decentralization of the blockchain ecosystem, making an important element such as data analysis dependent on private companies.

7.0.3 Domain-specific data analysis

If blockchains continue to grow in size indefinitely as they are designed to do, it will be unfeasible to get and index all the historical data in a single place. Chances are that it will be a similar challenge of indexing all the history of the World Wide Web on a single machine or cluster of machines right now.

The potential solution to this problem is reducing the domain of the managed data. As seen in the analysis reported in Chapter 6, most of the traffic on the chain is restricted to a relatively small set of smart contracts. These smart contracts implement different and independent protocols. Information about what happens in these protocols can be extracted, indexed and analyzed independently from each other.

The Graph [14] proposes a solution of this kind. As noted in Chapter 3, this decentralized indexing protocol gathers together data that is indexed independently from the various decentralized protocols implemented by smart contracts. This particular indexing protocol is even more strict in term of amount of data since it just considers the logs, ignoring transactions and blocks data. This technique allows for better scalability, but lacks giving a bigger picture of the traffic in a blockchain network.

7.0.4 Future work

There are some areas of this research that could be explored more in-depth in future works. In theory, eth2dgraph should work with any EVM-compatible chain, such as Polygon, since the RPCs used are the same. This has not been tested because of the lack of available nodes.

[1]Polygon is a layer 2 EVM blockchain based on Ethereum.

[2]BNB Chain is a layer 1 EVM compatible blockchain developed by the Binance exchange.

[3]This graph made by Etherscan shows the historical size of a Geth archive node: `https://etherscan.io/chartsync/chainarchive`.

Due to the infrastructure available, Dgraph was used in a single machine. It would be interesting to test its behaviour with a cluster distributed over multiple servers. This would test the horizontal scalability of Dgraph.

Another area of improvement is the streaming of live data. Currently, eth2dgraph supports live data insertion from the Ethereum network to an active Dgraph cluster using ACID transactions. It proved to work well when the database does not have a lot of data already indexed, but it was slower than the production of data from the blockchain when the database had all the Ethereum history indexed. The optimization of this part of the tool was not the priority of this work, so this could be an area of improvement.

Talking more broadly, the optimal solution to the problem that this research tries to address would be to have blockchain clients that directly give the option to freely index data. This would remove the redundancy of having to store data in two different places: one in the EVM client storage and one in the database storage. These *Blockchain Analytics Clients* could specifically target data analysts and would not need the ability to participate in the consensus layer of the protocol.

CHAPTER

EIGHT

CONCLUSIONS

This research has investigated a novel solution to the problem of Ethereum data management using a graph database. Thanks to the release of eth2dgraph, it is now possible to easily index Ethereum data using Dgraph and query it with DQL and GraphQL.

To answer research question 1, the schema used to manage data gives a clear image of what can be extracted from EVM blockchains without relying on centralized services. It is reported in Figure 4.2.1. The only kind of information that can not be extracted without relying on centralized services is the source code of smart contracts. It is possible to extract functions and events implemented by smart contracts from their bytecodes, even if the data is not perfectly accurate.

Generally, most of the semantics related to on-chain operations can be obtained from logs. In this work, logs referring to token transfers were parsed to allow faster and easier queries. The same can be done for other domains, e.g. token swaps, token approvals, or other protocol-specific use cases.

To answer the second research question, with the solution provided in this Master's Thesis, it is possible to estimate that at least 6TB of fast SSDs, 400GB of RAM, and a CPU of 64 cores are needed to perform independent extraction and indexing of all Ethereum data as of August 2023.

Although the entry barrier is high, it is still possible to run an archive node, extract all the historical data, and index it in a database all in the same machine. Apart from computational resources, it is an operation that also takes time. It takes at least a few days to sync a node, around seven hours to extract the data and more than two days to ingest it into the database.

The analysis on the extracted data conducted in Chapter 6 serve as an example of the potentiality of this data. They show possible ways of using and interpreting Ethereum data. From these analysis it appeared that most of the traffic on the network is restricted to a small number of smart contracts. This fact suggested that more efficient analysis can be done focusing on a specific decentralized protocol, instead of having to manage all the historical Ethereum data.

REFERENCES

[1] Satoshi Nakamoto. *Bitcoin: a Peer-to-Peer Electronic Cash System*. Tech. rep. Oct. 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 01/31/2023).

[2] Vitalik Buterin. *Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform*. Tech. rep. 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[3] Ting Chen et al. "DataEther: Data Exploration Framework For Ethereum". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 1369–1380. DOI: 10.1109/ICDCS.2019.00137.

[4] Peilin Zheng et al. "XBlock-ETH: Extracting and exploring blockchain data from Ethereum". In: *IEEE Open J. Comput. Soc.* 1 (May 2020), pp. 95–106. DOI: 10.1109/OJCS.2020.2990458.

[5] Santiago Bragagnolo et al. "Ethereum query language". en. In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. Gothenburg Sweden: ACM, May 2018, pp. 1–8. ISBN: 978-1-4503-5726-5. DOI: 10.1145/3194113.3194114. URL: https://dl.acm.org/doi/10.1145/3194113.3194114 (visited on 02/15/2023).

[6] Jain Manish. *Dgraph: Synchronously Replicated, Transactional and Distributed Graph Database*. Tech. rep.

[7] Phillip Rogaway and Thomas Shrimpton. "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance". In: *FSE*. Vol. 3017. 1979, pp. 371–388.

[8] Leslie Lamport. "Password authentication with insecure communication". In: *Communications of the ACM* 24.11 (1981), pp. 770–772.

[9] Don Johnson, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1 (2001), pp. 36–63.

[10] David Chaum. "Blind signatures for untraceable payments". In: *Advances in Cryptology: Proceedings of Crypto 82*. Springer. 1982, pp. 199–203.

[11] Christian Stoll, Lena Klaaßen, and Ulrich Gallersdörfer. "The carbon footprint of bitcoin". In: *Joule* 3.7 (2019), pp. 1647–1661.

[12] Takenobu Tani. *Ethereum EVM illustrated*. Access: 08-09-2023. URL: https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf.

[13] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm (extended version)". In: *Proceeding of USENIX annual technical conference, USENIX ATC*. 2014, pp. 19–20.

[14] Jannis Pohlmann Yaniv Tal Brandon Ramirez. *The Graph: A Decentralized Query Protocol for Blockchains*. Tech. rep. Mar. 2018, p. 12. URL: https://github.com/graphprotocol/research/blob/master/papers/whitepaper/the-graph-whitepaper.pdf.

[15] Evgeny Medvedev and the D5 team. *Ethereum ETL*. 2018. URL: https://github.com/blockchain-etl/ethereum-etl.

[16] Qin Wang et al. *Exploring Web3 From the View of Blockchain*. 2022. arXiv: 2206.08821 [cs.CR].

[17] Michael Fröwis and Rainer Böhme. "Not all code are create2 equal". In: *6th Workshop on Trusted Smart Contracts (WTSC'22)*. 2022.

[18] Lucianna Kiffer, Dave Levin, and Alan Mislove. "Analyzing Ethereum's Contract Topology". In: Oct. 2018, pp. 494–499. ISBN: 978-1-4503-5619-0. DOI: 10.1145/3278532.3278575.

[19] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[20] Monika Di Angelo and Gernot Salzer. "Identification of token contracts on Ethereum: standard compliance and beyond". In: *International Journal of Data Science and Analytics* (Sept. 2021), pp. 1–20. DOI: 10.1007/s41060-021-00281-1.

[21] Monika di Angelo and Gernot Salzer. *Wallet Contracts on Ethereum – Identification, Types, Usage, and Profiles*. 2021. arXiv: 2001.06909 [cs.CR].

[22] Martin Ortner and Shayan Eskandari. *Smart Contract Sanctuary*. URL: https://github.com/tintinweb/smart-contract-sanctuary.

[23] Monika Di Angelo and Gernot Salzer. "Assessing the Similarity of Smart Contracts by Clustering their Interfaces". In: Dec. 2020. DOI: 10.1109/TrustCom50675.2020.00261.

[24] Vitalik Buterin and Martin Swende. *EIP-3529: Reduction in refunds*. Ethereum Improvement Proposals, no. 3529, April. 2021. URL: https://eips.ethereum.org/EIPS/eip-3529.

APPENDICES

A - COMPLETE SCHEMA OF INDEXED DATA

Here are both the DQL and GraphQL schema of the data extracted. Listing 1 reports the DQL schema and Listing 2 the GraphQL one.

```
<Account.address>: string @index(hash) @upsert .
<Account.tags>: [string] @index(hash) .
<Account.is_contract>: bool @index(bool) .
<Block.base_fee_per_gas>: float .
<Block.datetime>: datetime @index(hour) .
<Block.difficulty>: string @index(hash) .
<Block.gas_limit>: int .
<Block.gas_used>: int @index(int) .
<Block.gas_price_avg>: float @index(float) .
<Block.gas_price_max>: float .
<Block.gas_price_min>: float .
<Block.gas_price_std_dev>: float .
<Block.number>: int @index(int) @upsert .
<Block.size>: int .
<Block.tx_count>: int .
<Block.miner>: uid @reverse .
<Block.withdrawals>: [uid] @reverse .
<ContractDeployment.block>: uid @reverse .
<ContractDeployment.contract>: uid @reverse .
<ContractDeployment.creation_bytecode>: string .
<ContractDeployment.creator>: uid @reverse .
<ContractDeployment.deployed_bytecode>: string .
<ContractDeployment.experimental>: bool .
<ContractDeployment.failed_deploy>: bool .
<ContractDeployment.skeleton>: uid @reverse .
<ContractDeployment.solc_version>: string .
<ContractDeployment.storage_address>: string .
<ContractDeployment.storage_protocol>: string .
<ContractDeployment.tx_hash>: string @index(hash) .
<ContractDeployment.name>: string @index(trigram) .
<ContractDeployment.verified_source>: bool @index(bool) .
<ContractDeployment.verified_source_code>: string @index(term) .
<ContractDestruction.balance_left>: string .
<ContractDestruction.block>: uid @reverse .
<ContractDestruction.contract>: uid @reverse .
<ContractDestruction.refound_address>: uid .
<ContractDestruction.failed>: bool @index(bool) .
<ContractDestruction.tx_hash>: string @index(hash) .
<Error.inputs>: string @index(trigram) .
<Error.name>: string @index(exact) .
<Error.signature>: string @index(hash) @upsert .
```

```
<Event.inputs>: string @index(trigram) .
<Event.name>: string @index(exact) .
<Event.signature>: string @index(hash) @upsert .
<Function.inputs>: string @index(trigram) .
<Function.name>: string @index(exact) .
<Function.bytes4>: string @index(hash) .
<Function.outputs>: string @index(trigram) .
<Function.signature>: string @index(hash) @upsert .
<Skeleton.bytecode>: string @index(hash) .
<Skeleton.erc20_compliancy>: int @index(int) .
<Skeleton.erc721_compliancy>: int @index(int) .
<Skeleton.errors>: [uid] @reverse .
<Skeleton.events>: [uid] @reverse .
<Skeleton.failed_decompilation>: bool .
<Skeleton.functions>: [uid] @reverse .
<Skeleton.similar_code>: [uid] .
<Skeleton.similar_interface>: [uid] .
<TokenTransfer.block>: uid @reverse .
<TokenTransfer.contract>: uid @reverse .
<TokenTransfer.from>: uid @reverse .
<TokenTransfer.to>: uid @reverse .
<TokenTransfer.tx>: uid .
<TokenTransfer.value>: string .
<TokenTransfer.token_id>: string @index(hash) .
<Transaction.block>: uid @reverse .
<Transaction.from>: uid @reverse .
<Transaction.gas>: int .
<Transaction.gas_price>: int .
<Transaction.hash>: string @index(hash) @upsert .
<Transaction.input>: string .
<Transaction.bytes4>: string @index(hash) .
<Transaction.max_fee_per_gas>: int .
<Transaction.max_priority_fee_per_gas>: int .
<Transaction.nonce>: int .
<Transaction.r>: string .
<Transaction.s>: string .
<Transaction.to>: uid @reverse .
<Transaction.v>: string .
<Transaction.value>: string .
<Log.contract>: uid @reverse .
<Log.block>: uid @reverse .
<Log.tx>: uid @reverse .
<Log.topic_0>: string @index(hash) .
<Log.topic_1>: string @index(hash) .
<Log.topic_2>: string @index(hash) .
<Log.topic_3>: string @index(hash) .
<Log.data>: string .
<Log.tx_index>: int .
<Log.index>: int .
<Withdrawal.address>: uid @reverse .
<Withdrawal.string>: int .
<Withdrawal.index>: int .
<Withdrawal.validator_index>: int .
<Withdrawal.amount>: int .
type <Account> {
  Account.address
  Account.tags
  Account.is_contract
```

```
}
type <Block> {
  Block.number
  Block.datetime
  Block.difficulty
  Block.tx_count
  Block.gas_price_min
  Block.gas_price_max
  Block.gas_price_avg
  Block.gas_price_std_dev
  Block.gas_limit
  Block.gas_used
  Block.base_fee_per_gas
  Block.size
  Block.miner
  Block.withdrawals
}
type <ContractDeployment> {
  ContractDeployment.contract
  ContractDeployment.block
  ContractDeployment.creator
  ContractDeployment.tx_hash
  ContractDeployment.failed_deploy
  ContractDeployment.creation_bytecode
  ContractDeployment.deployed_bytecode
  ContractDeployment.skeleton
  ContractDeployment.storage_protocol
  ContractDeployment.storage_address
  ContractDeployment.experimental
  ContractDeployment.solc_version
  ContractDeployment.verified_source
  ContractDeployment.verified_source_code
  ContractDeployment.name
}
type <ContractDestruction> {
  ContractDestruction.contract
  ContractDestruction.block
  ContractDestruction.tx_hash
  ContractDestruction.balance_left
  ContractDestruction.refound_address
  ContractDestruction.failed
}
type <Error> {
  Error.signature
  Error.name
  Error.inputs
}
type <Event> {
  Event.signature
  Event.name
  Event.inputs
}
type <Function> {
  Function.signature
  Function.name
  Function.inputs
  Function.outputs
}
```

```
type <Skeleton> {
  Skeleton.bytecode
  Skeleton.functions
  Skeleton.events
  Skeleton.errors
  Skeleton.failed_decompilation
  Skeleton.erc20_compliancy
  Skeleton.erc721_compliancy
  Skeleton.similar_code
  Skeleton.similar_interface
}
type <TokenTransfer> {
  TokenTransfer.contract
  TokenTransfer.from
  TokenTransfer.to
  TokenTransfer.value
  TokenTransfer.block
  TokenTransfer.tx
  TokenTransfer.token_id
}
type <Transaction> {
  Transaction.hash
  Transaction.from
  Transaction.to
  Transaction.block
  Transaction.value
  Transaction.gas
  Transaction.gas_price
  Transaction.input
  Transaction.bytes4
  Transaction.max_fee_per_gas
  Transaction.max_priority_fee_per_gas
  Transaction.nonce
  Transaction.r
  Transaction.s
  Transaction.v
}
type <Log> {
  Log.contract
  Log.block
  Log.tx
  Log.topic_0
  Log.topic_1
  Log.topic_2
  Log.topic_3
  Log.data
  Log.tx_index
  Log.index
}
type <Withdrawal> {
  Withdrawal.address
  Withdrawal.amount
  Withdrawal.index
  Withdrawal.validator_index
}
```

Listing 1: DQL schema of indexed data

```
type Account {
  address: String! @id @search(by: [hash])
  tags: [String] @search(by: [hash])
  is_contract: Boolean @search
  token_sent: [TokenTransfer] @dgraph(pred: "~TokenTransfer.from")
  token_received: [TokenTransfer] @dgraph(pred: "~TokenTransfer.to
   ")
  transactions_sent: [Transaction] @dgraph(pred: "~Transaction.
   from")
  transactions_received: [Transaction] @dgraph(pred: "~Transaction
   .to")
  created_contracts: [ContractDeployment] @dgraph(pred: "~
   ContractDeployment.creator")
  logs: [Log] @dgraph(pred: "~Log.contract")
  deployments: [ContractDeployment] @dgraph(pred:"~
   ContractDeployment.contract")
  destructions: [ContractDestruction] @dgraph(pred:"~
   ContractDestruction.contract")
  transfers: [TokenTransfer] @dgraph(pred: "~TokenTransfer.
   contract")
  mined_blocks: [Block] @dgraph(pred:"~Block.miner")
  withdrawals: [Withdrawal] @dgraph(pred:"~Withdrawal.address")
}

type Withdrawal {
  amount: String! @search
  index: Int
  validator_index: Int
  address: Account! @dgraph(pred:"Withdrawal.address")
  block: [Block] @dgraph(pred:"~Block.withdrawals")
}

type Block {
  number: Int! @id @search
  miner: Account @dgraph(pred:"Block.miner")
  datetime: DateTime @search(by: [hour])
  difficulty: String @search
  tx_count: Int @search
  gas_price_min: Float
  gas_price_max: Float
  gas_price_avg: Float @search
  gas_price_std_dev: Float
  gas_limit: Int
  gas_used: Int
  base_fee_per_gas: Float
  size: Int
  deployments: [ContractDeployment] @dgraph(pred: "~
   ContractDeployment.block")
  destructions: [ContractDestruction] @dgraph(pred: "~
   ContractDestruction.block")
  transfers: [TokenTransfer] @dgraph(pred: "~TokenTransfer.block")
  transactions: [Transaction] @dgraph(pred: "~Transaction.block")
  withdrawals: [Withdrawal] @dgraph(pred:"Block.withdrawals")
  logs: [Log] @dgraph(pred: "~Log.block")
}

type Transaction {
  hash: String! @id @search(by: [hash])
```

```
  value: String!
  gas: Int @search
  gas_price: Int @search
  input: String
  bytes4: String @search(by: [hash])
  max_fee_per_gas: Int
  max_priority_fee_per_gas: Int
  nonce: Int
  r: String
  s: String
  v: String
  from: Account! @dgraph(pred:"Transaction.from")
  to: Account! @dgraph(pred:"Transaction.to")
  block: Block! @dgraph(pred:"Transaction.block")
  logs: [Log] @dgraph(pred: "~Log.tx")
}

type Function {
  signature: String! @id @search(by: [hash])
  name: String @search(by: [exact])
  inputs: String @search(by: [trigram])
  outputs: String @search(by: [trigram])
  skeletons: [Skeleton] @dgraph(pred:"~Skeleton.functions")
  bytes4: String @search(by: [hash])
}

type Event {
  signature: String! @id @search(by: [hash])
  name: String @search(by: [exact])
  inputs: String @search(by: [trigram])
  skeletons: [Skeleton] @dgraph(pred:"~Skeleton.events")
}

type Error {
  signature: String! @id @search(by: [hash])
  name: String @search(by: [exact])
  inputs: String @search(by: [trigram])
  skeletons: [Skeleton] @dgraph(pred:"~Skeleton.errors")
}

type ContractDeployment {
  tx_hash: String @search(by: [hash])
  failed_deploy: Boolean @search
  creation_bytecode: String
  deployed_bytecode: String
  storage_protocol: String
  storage_address: String
  experimental: Boolean @search
  solc_version: String @search(by: [hash])
  verified_source: Boolean @search
  verified_source_code: String @search(by: [term])
  name: String @search(by: [trigram])
  contract: Account! @dgraph(pred:"ContractDeployment.contract")
  block: Block! @dgraph(pred:"ContractDeployment.block")
  creator: Account! @dgraph(pred:"ContractDeployment.creator")
  skeleton: Skeleton @dgraph(pred:"ContractDeployment.skeleton")
}
```

```
type ContractDestruction {
  tx_hash: String @search(by: [hash])
  balance_left: String
  refound_address: Account! @dgraph(pred:"ContractDestruction.
   refound_address")
  contract: Account! @dgraph(pred:"ContractDestruction.contract")
  block: Block! @dgraph(pred:"ContractDestruction.block")
}

type Skeleton {
  bytecode: String! @search(by: [hash])
  erc20_compliancy: Int @search
  erc721_compliancy: Int @search
  failed_decompilation: Boolean @search
  deployments: [ContractDeployment] @dgraph(pred:"~
   ContractDeployment.skeleton")
  functions: [Function] @dgraph(pred:"Skeleton.functions")
  events: [Event] @dgraph(pred:"Skeleton.events")
  errors: [Error] @dgraph(pred:"Skeleton.errors")
  similar_code: [Skeleton]
  similar_interface: [Skeleton]
}

type TokenTransfer {
  value: String!
  token_id: String
  tx: Transaction
  block: Block! @dgraph(pred:"TokenTransfer.block")
  contract: Account! @dgraph(pred:"TokenTransfer.contract")
  from: Account! @dgraph(pred:"TokenTransfer.from")
  to: Account! @dgraph(pred:"TokenTransfer.to")
}

type Log {
  topic_0: String @search(by: [hash])
  topic_1: String @search(by: [hash])
  topic_2: String @search(by: [hash])
  topic_3: String @search(by: [hash])
  data: String
  tx_index: Int
  index: Int
  contract: Account! @dgraph(pred:"Log.contract")
  block: Block @dgraph(pred:"Log.block")
  tx: Transaction @dgraph(pred:"Log.tx")
}
```

Listing 2: GraphQL schema of indexed data

B - DATA RETURNED FROM ETHEREUM RPCS

To give more context, Listings 3 to 5 include three examples of data returned from the RPCs that were used for extracting Ethereum data.

eth_getBlockByNumber


```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "baseFeePerGas": "0x1ced5e1e0c",
        "difficulty": "0x0",
        "extraData": "0x7273796e632d6275696c6465722e78797a",
        "gasLimit": "0x1c9c380",
        "gasUsed": "0xd0b9ef",
        "hash": "0x...",
        "logsBloom": "0x...",
        "miner": "0x1f9090aae28b8a3dceadf281b0f12828e676c326",
        "mixHash": "0x...",
        "nonce": "0x0000000000000000",
        "number": "0x1067381",
        "parentHash": "0x...",
        "receiptsRoot": "0x...",
        "sha3Uncles": "0x...",
        "size": "0x135f5",
        "stateRoot": "0x...",
        "timestamp": "0x6455fe73",
        "totalDifficulty": "0xc70d815d562d3cfa955",
        "transactions": [
            ...,
            {
                "blockHash": "0x...",
                "blockNumber": "0x1067381",
                "from": "0x...",
                "gas": "0x1725d",
                "gasPrice": "0x1cf069692b",
                "maxPriorityFeePerGas": "0x30b4b1f",
                "maxFeePerGas": "0x1df5b2c880",
                "hash": "0x...",
                "input": "0x...",
                "nonce": "0x8",
                "to": "0x...",
                "transactionIndex": "0x9b",
                "value": "0x0",
```

```
                "type": "0x2",
                "accessList": [],
                "chainId": "0x1",
                "v": "0x1",
                "r": "0x...",
                "s": "0x..."
                },
                ...
        ],
        "transactionsRoot": "0x...",
        "uncles": [],
        "withdrawals": [
            ...,
            {
                "index": "0x285032",
                "validatorIndex": "0x7125d",
                "address": "0
   x562ab7dca86f8947f5b066a663d83452954a71a2",
                "amount": "0xbcd8a1"
            },
            ...
        ],
        "withdrawalsRoot": "0x..."
    }
}
```

Listing 3: Data returned from eth_getBlockByNumber RPC with hydrated transactions.

eth_getLogs

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    ...,
    {
      "address": "0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48",
      "topics": [
        "0x...",
        "0x...",
        "0x..."
      ],
      "data": "0x...",
      "blockNumber": "0xe524e1",
      "transactionHash": "0x...",
      "transactionIndex": "0xb",
      "blockHash": "0x...",
      "logIndex": "0x0",
      "removed": false
    },
    ...
  ]
}
```

Listing 4: Data returned from eth_getLogs RPC.

trace_block

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
  ...,
  {
    "action": {
      "from": "0xaeec6f5aca72f3a005af1b3420ab8c8c7009bac8",
      "callType": "call",
      "gas": "0x11170",
      "input": "0x",
      "to": "0xef86d3164a50df780e24e9226bcddf3c8606e424",
      "value": "0x2b5e3af16b1880000"
    },
    "blockHash": "0x...",
    "blockNumber": 5469798,
    "result": {
      "gasUsed": "0x0",
      "output": "0x"
    },
    "subtraces": 0,
    "traceAddress": [],
    "transactionHash": "0x...",
    "transactionPosition": 0,
    "type": "call"
  },
  ...
  ]
}
```

Listing 5: Data returned from trace_block RPC.