UNIVERSITÀ DI PISA

Department of Computer Science
Bachelor's Degree in Computer Science

Analisi dei Bloom filter di Ethereum per aggregazione di eventi

-

Ethereum Bloom filters analysis for quick events aggregation

Candidate
Nicola Calise

Supervisors
Dott. Damiano Di Francesco Maesa
Matteo Loporchio

Academic Year 2021/2022

Contents

1 Introduction **3**

2 Background **7**
2.1 Cryptographic hash functions 7
2.2 Blockchain 8
2.2.1 Use cases 10
2.3 Ethereum protocol 11
2.3.1 State 12
2.3.2 Transactions 13
2.3.3 Blocks 14
2.3.4 Fees 15

3 Bloom filters and their application in Ethereum **17**
3.1 Bloom filters 17
3.2 Properties of Bloom filters 19
3.3 Bloom filters in Ethereum 21
3.3.1 Events 21
3.3.2 Receipts 23
3.3.3 Definition 24
3.3.4 Computing the false positive rate for a `logsBloom` . . 25
3.3.5 `logsBloom` saturation 25

4 Skip lists **27**
4.1 Generic Skip List 27
4.1.1 Skip list construction and search 28
4.2 Our version of skip list 29
4.2.1 Bloom filter based skip list 29
4.2.2 Construction time 31
4.2.3 Searching for events 31

5 Experimental results **35**
5.1 Data set 35
5.1.1 Ethereum header 35
5.2 Implementation 37
5.2.1 Skip list builder 38
5.2.2 Analysis with Python scripts 40
5.3 Analysis 41
5.3.1 Analysis of construction time of skip lists 42
5.3.2 Analysis of `logsBloom` 43
5.3.3 Analysis of the proposed Skip lists 47
5.3.4 Estimating the number of items inside a `logsBloom` . 49
5.3.5 `logsBloom` expansion 50

6 Conclusions and future developments **55**

Chapter 1

Introduction

During its diffusion, the Internet has radically changed our lives thanks to its main application: the Web. It makes information sharing simple, fast, and cheap, practically accessible to everyone. As all technologies, the Web has progressed continuously. This includes the terminology to describe its evolution state considering to the development techniques and technologies used. The first version of the Web was the Web1.0 defined by the development of web pages with a low interaction with users. The service offered by that version was only the supplying of information through web sites. An user could just land on a web page and read some information about the theme of the page [27]. Its evolution, named Web2.0 is the most diffused nowadays. It opened the way to the inception of new protocols and programming languages on the technical point of view, and an high "user to company" interaction as a service paradigm. Some relevant novelties introduced by this new paradigm were platforms called "social networks" which enabled users across the world to connect to each others and share information quickly and easily. For example Facebook allows users to share photo, videos and posts or Linkedin helps users to promote their professional images and look for an occupation. The main innovation that brings us from the Web1.0 to the Web2.0 is about a social nature, not technical one: the users are not just passive receivers of information, but rather partake actively in sharing and creating it on a third party platform.

In spite of many innovations, the Web2.0 presents some limitations. Some of these concern lack of guarantees on the security and integrity of data shared by the users. For example, if we use a digital payment service as PayPal we necessary refer and trust a centralized system managed by a private company, this could not provide us proof that our data (of-

ten sensitive information) are going to be handled as securely as we'd like. Furthermore, we could not avoid that most providers are able to perform censorship acts towards their users [28]. To solve these problems it's been proposed a new way to think of the Web, called Web3.0 or Web3. The main innovative concept behind the Web3 is decentralisation. No longer centralised entities should control the information diffusion and remote services. Systems and communities should be set up and managed by their own users in a fully decentralised fashion. The underlying idea is not recent but it has remained dormant for years waiting for the right technical tools to enable it. One of such "decentralisation enabling" tools is the blockchain.

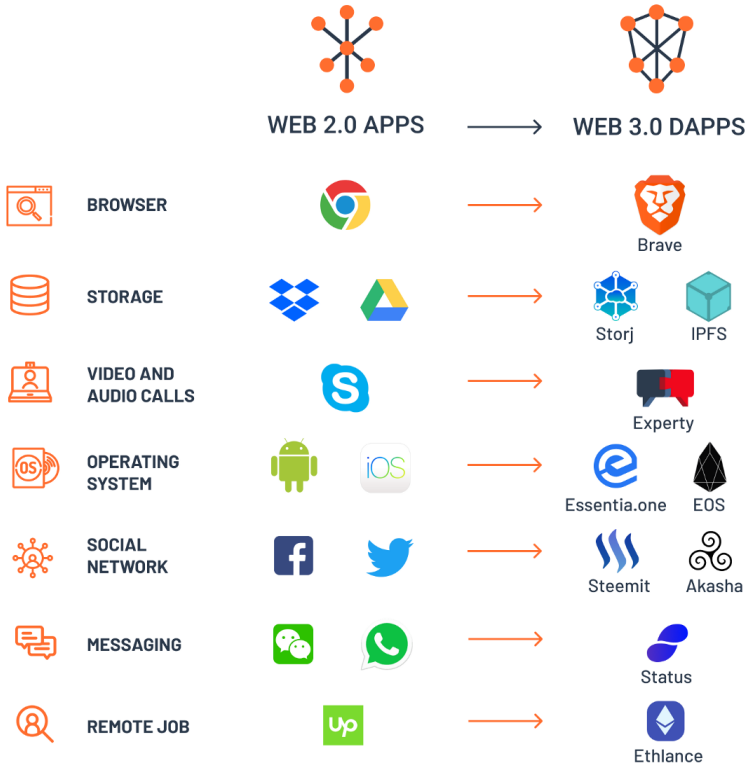

Figure 1.1: A picture representing the evolution from web 2.0 apps to web 3.0 dApps (source: [15]).

The blockchain paradigm started with several goals, one of them, maybe the most important one, was to ensure the secure transaction between customers who have no reason to trust the others. This is made possible by

employing a distributed consensus algorithm, making a single individual or organizations control over the system impossible. This system can also be described as a specialized kind of cryptographic state machine, producing a linked growing list of blocks. The chain could be compared to a Linked List data structure in which new blocks are securely linked to the last one incrementally, i.e. in an "append only" fashion. All of these record are replicated and stored across many computers over the internet, using a peer-to-peer networks, one of the key properties to make the system decentralized and distributed. One of the most important blockchain projects, and the one we focused on in this work, is Ethereum. It's main innovation has been the introduction of new tools aiming at allowing Turing complete code to be executed on chain. Such programs are named "smart contracts" and their execution follows the same decentralisation rules of standard writes to the chain. Another novelty of Ethereum was the introduction of smart contract events, a way to advertise messages to the off chain world by contracts.

The birth of Ethereum and other blockchain protocols was one of the enabler technologies to allow the birth of Web3. This is reflected on the key goals of Web3 [11]:

- Web3 is decentralized. This means services are managed by the users and there is no single owner, as blockchain protocols are.
- Web3 is permissionless. This means that every user has the same rights.
- Web3 has native payments. Which are blockchain backed cryptocurrencies.
- Web3 is trustless. This means it operates without trusting a third party entity, same as blockchain protocols.

Another result of the union of Ethereum and Web3 is the birth of dApps. dApps are a growing movement of applications that uses Ethereum to disrupt business models or invent new ones. They appear to users as standard remote applications but use Web3 principles. This allows this new kind of application to make novel decentralised protocols transparent to not-technical users. dApps possess their own backend code (often based on a smart contracts deployed on Ethereum or a similar blockchain protocol) running on a decentralized network and not in a standard server. They use a blockchain for data storage and smart contracts for their app logic. When a dApp is published in Ethereum it becomes public forever, everyone can use their features and no one can remove it from the web [10]. There are a lot

of examples of dApps for multiple types of services: trading, investments, crowdfunding, payments and more. Ethereum based dApps communicate on chain through smart contracts that produce events and transactions, so they need to access to the blockchain continuously.

At the moment in a blockchain protocol (specifically in Ethereum) if we want to retrieve information about transactions and events the solution is to read the corresponding chain block by block. Inside a block there are many information which are not relevant to the user, meaning they are going to download and read a lot of data they are not interested in. The complexity for searching a given information on chain is hence linear in the number of blocks to be scanned, and in Ethereum there are over fifteen millions blocks as of November 2022. As dApps rely on the information stored on chain to function, it is highly desirable to find a way to reduce the number of blocks to be read to retrieve any specific information.

To solve this problem, in this thesis we propose to use a skip list, a data structure built on top of each block and containing an index of previous block ranges. Adding such a data structure increase the blockchain size but improves the performance of items search. Aim of this thesis is to evaluate the implementation of skip lists over Ethereum, leveraging its native Bloom filters based events indexes. To do so, we define the novel data structure, we implement it on Ethereum and measure its performances. More precisely, we measure the growth of the number of ones inside the skip list Bloom filter indexes, because the higher this number is, the higher is the false positive rate (see Section 3.3.4), which limits the effectiveness of our proposal. As, even if searches are faster, high false positives may cause them to be useless. As a side effect, this work also provides an analysis of the current state of Bloom filters in Ethereum.

Chapter 2

Background

In this chapter we will introduce a number of different topics for a better understanding of the next chapters of the thesis. In particular, we will present the concept of cryptographic hash function, the fundamentals of blockchain, with particular attention to Ethereum.

2.1 Cryptographic hash functions

Cryptographic hash functions constitute a fundamental building block of blockchains. Given a sequence of bits of arbitrary length, a cryptographic hash function outputs a bit sequence of fixed length, called *digest*. Despite being computed deterministically, the digest "looks random" any user cannot infer the input from the output without a significant computational effort. This property makes this kind of hash functions useful for any application dealing with security. Below we present a formal definition of cryptographic hash function, taken from [18].

Definition 2.1.1 (Cryptographic hash function)**.** *A cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^n$ is a mathematical algorithm that maps data of arbitrary size to a bit sequence of fixed size (called* digest*). The function satisfies the following properties:*

1. *For all $x \in \{0,1\}^*$ it is computationally easy to calculate $y = H(x)$.*
2. *Preimage resistance. For all $y \in \{0,1\}^n$ it is computationally infeasible to find $x \in \{0,1\}^*$ such that $H(x) = y$.*
3. *Collision resistance. It is computationally infeasible to find two elements $x_1, x_2 \in \{0,1\}^*$ such that $H(x_1) = H(x_2)$.*

Cryptographic hash functions have many applications in security. For example they can be used in authentication systems to construct *message authentication codes* (MAC) [24], a family of hash functions parameterized by a secret key. Concerning blockchain systems, among the most popular algorithm adopted in this field we mention SHA-256 [16] and Keccak-256 [2], where 256 describes the number of bits of their output digests. The first one is widely used in Bitcoin [21] during the mining and address generation processes. The second one, on the other hand, is employed by Ethereum [32] and will also be discussed later during this thesis.

Diffusion An interesting property of cryptographic hash functions is called *diffusion*. In other words, by changing a single bit of the input string, then nearly half of the bits of the digest should change, too. The purpose of diffusion is to hide the statistical relationship between the digest and the input string. To illustrate this property, we now discuss with an example how a small change in the input generates a significant change in the output. Taking two words "Loro" and "Lora" and the SHA-256 algorithm, in Table 2.1 we can see the results of their hashing.

Input string	Digest (SHA-256)
Loro	`7ddb2822badeb5ebddca7b36d43370bc8bdbcbc16074e32b0a2118369a5c8321`
Lora	`3a5d51a2c047a94f43bf3f5e54d344f21c46f91ec18e76a531e00f6936079e8e`

Table 2.1: Example of diffusion in cryptographic hash functions.

2.2 Blockchain

As already discussed in Chapter 1, a blockchain is a data structure made up of blocks. Each block includes a collection of transactions and each transaction describes a redistribution of funds between entities. Moreover, transactions in a block are organized in a specific data structure called *Merkle tree* [19], in order to guarantee their integrity. As its name suggests, a Merkle tree is a tree data structure in which every leaf node is labelled with the cryptographic hash of a data block, and every internal node is labelled with the cryptographic hash of the labels of its child nodes. With this kind of data structure it is not possible to modify the content of a leaf node without modifying every other node in the path from that leaf up to the root of the tree.

Blockchain structure The diagram of Figure 2.1 constitutes a plot representation of the structure of a blockchain, following the model of Bitcoin. Specifically, each block of the chain is composed by a *header* and a list of transactions. The header includes all cryptographic mechanisms that are needed to make the chain immutable, such as a hash pointer to the predecessor and the root of the Merkle tree built on the set of transactions. More in detail, in the Bitcoin blockchain the header of is made up of 7 different fields [22].

1. **Version**. This represents the protocol version number.
2. **Previous Block Hash**. Contains a hash pointer to the predecessor block. The hash pointer is a cryptographic hash of the header of the predecessor block. This mechanism ensures the immutability of the blockchain: if the content of one block changes, then it is necessary to recompute the hash pointers for all subsequent blocks.
3. **Merkle Root**. Contains the hash of the root of a Merkle tree built on the set of transactions.
4. **Timestamp**. This is the creation time of the block (represented using Unix Epoch).
5. **Difficulty Target**. The proof-of-work algorithm difficulty target for this block.
6. **Nonce**. This number represents the solution for the proof-of-work problem.

Proof of work Blocks of a chain are added according to a specific *consensus protocol*, run by all participants [34]. For instance, the consensus protocol adopted by Bitcoin is based on a mechanism called *proof of work*. Proof of work requires users of the blockchain to consume energy by solving a specific mathematical problem. The proof of work consists of a "race" where nodes with great computation capabilities and resources, called *miners*, have to compete to be the first to solve a mathematical problem. The first miner that solves this problem is rewarded with new cryptocurrency coins and is allowed to add the new block to the blockchain. This mechanism is necessary in order to guarantee the security and immutability of transactions between people and prevents frauds or mistakes. The reason why the proof of work suits this scenario is because finding the target hash is

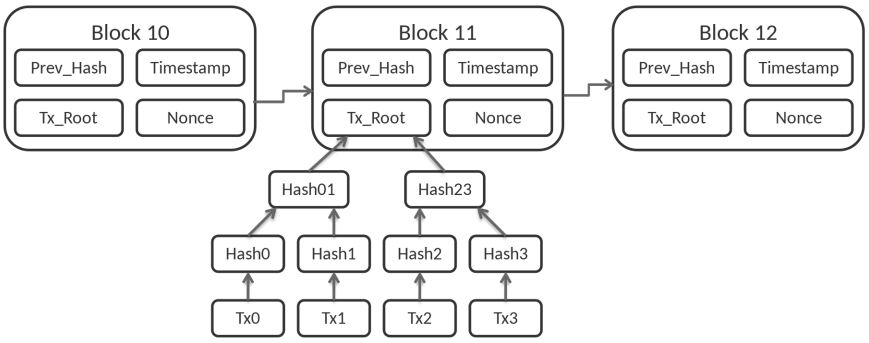

Figure 2.1: The structure of a blockchain, following the Bitcoin model.

difficult, while verifying the correctness of the solution is easy. The process of computing the proof of work is difficult enough to prevent the manipulation of transaction records. Another property is that when a target hash is found, it is easy for other miners to check its correctness.

2.2.1 Use cases

Blockchain technology has gained more and more popularity during the last few years. Despite its role in cryptocurrencies, its applications are not limited to this field of applications, as detailed in [17]. In this section, we discuss some of the most popular application of such technology.

- **Cryptocurrencies**. The most common use of blockchain technology is as a digital ledger for cryptocurrencies. Prominent examples in this regard are Bitcoin [21] and Ethereum [32]. A ledger can be seen as a book which contains a collection of financial transactions and accounts, so it can be used to record transfers of funds and assets between users. Cryptocurrency is an alternative form of payment created by using encryption algorithms to make safe transactions, wallets and information.

- **Financial Services**. Banks are interested in blockchain to improve the performance of the back office systems and reduce costs of work [29].

- **Domain Names**. Another use is the offering of domain name services using a private key with the purpose to make them uncensorable [1].

- **Games**. NFTs (non-fungible tokens) have settled in the video games monetization area. These games allow players to sell/buy the tradeable items for cryptocurrencies, which can eventually be exchanged for real money [12].
- **Food Safety**. Walmart is working with IBM to develop a blockchain technology for enhanced tracking and traceability of food products, to improve the safety of the food [31].

Difference between blockchains and databases A database is a centralized system, this means that if the main node fails the database will not work properly. Differently, blockchains are decentralized and distributed on multiple nodes across the network so the failure of one node has a limited impact on the entire system. Another difference is that a database is managed by an authority that can ensure user roles, while a blockchain system is backed by a peer to peer network without no central authority and where each node can connect with all other peers.

2.3 Ethereum protocol

Ethereum is a project which attempts to build a distibuted ledger protocol supporting full code execution [32]. The first appearance of it goes back in 2014, from a white paper published by Vitalik Buterin [5]. In this section, our focus is going to be reserved in some peculiarities of Ethereum, giving to the reader the necessary notions to understand the rest of the thesis properly. We start by introducing some differences between Ethereum and the general blockchain protocol described before.

Difference between Ethereum and other cryptocurrencies

We could reserve an entire chapter about the difference between Ethereum and other cryptocurrencies but now we are going to focus on two particular features that Ethereum implements as opposed to the other blockchains, namely *events* and *smart contracts*.

- **Smart Contract:** A "smart contract" is simply a software running on the Ethereum blockchain. It is a collection of code (its methods) and variables (its state) located at a specific address on the Ethereum blockchain [13]. This code is executed by the nodes of the Ethereum network in a decentralized fashion. It is useful to verify and facilitate

the right execution of a contract, allowing the partial or total exclusion of real contract terms.

- **Events:** Events are indexable signals which can be activated by a smart contract. An event matches a specific action done in the contract. For example, it may correspond to an ownership transfer of an NFT. They are important because their emission produces a log which is written in the blockchain to keep track and information about the operations done in the smart contract which emits the event. Events will be further examined and discussed in Section 3.3.

Now are going to talk about the structure of Ethereum. The blockchain is composed by three entities: states, transactions and blocks. We now describe each of them in the rest of this section.

2.3.1 State

Ethereum can be seen as a deterministic state machine, where balances and account data compose the state. New transactions and events cause a transition from the current state to a new one. The Ethereum world state is a mapping between addresses (i.e., 160-bit identifiers) and account states [32]. An account may correspond to a physical user (in that case it is called *externally-owned* account) or a smart contract. New accounts are being added and new smart contracts are being deployed continuously. It is important to notice that the world state is not stored explicitly inside the blockchain, but the implementation maintains the mapping in a data structure called *Merkle Patricia Trie* (a combination of a Merkle tree and a *Patricia trie* [20]). The root of the Merkle Patricia trie is embedded in the header of each block, thus making the data structure cryptographically dependent on. The account state comprises the following four fields, which we discuss below.

- **Nonce**. This value is a counter that indicates the number of transactions sent from the account.

- **Balance**. Number of *Wei* owned by this account. The term Wei refers to the smallest denomination of *ether* (ETH), which is the currency used on the Ethereum network [8]. In particular, we have that 1 ETH is equivalent to 10^{18} Wei.

- **CodeHash**. If the account corresponds to a contract, this hash value refers to the code of the contract itself. Indeed, contract accounts con-

tain executable code fragments to perform arbitrary operations (and modify the global state).

- **StorageRoot**. This field contains the 256-bit hash of the root node that encodes the storage contents of the account (implemented by means of a Merkle Patricia Tree).

Ethereum accounts can also be considered *dead* when the associated state is empty or non-existing.

2.3.2 Transactions

A transaction is a cryptographically signed operation which causes a transition of the world state in the Ethereum network, as highlighted in Figure 2.2. Transactions can only be initiated by externally-owned accounts. For instance, the simplest transaction is the one that transfers ETH from one account to another.

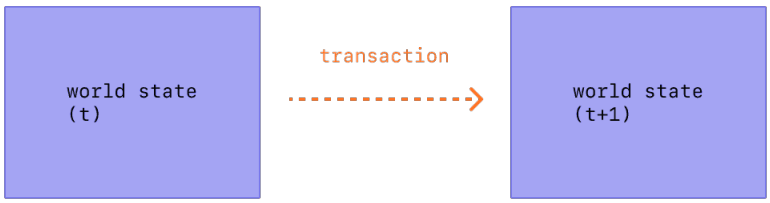

Figure 2.2: A transaction modifies the world state of the Ethereum state machine [14]

All transactions include several attributes. We now briefly discuss the most important ones.

- **From**. The sending address.
- **To**. The receiving address. If the receiver is an externally-owned account, then the transaction will transfer value. Conversely, if it corresponds to a smart contract, the transaction will use contract code.
- **Value**. The amount of Wei to be transferred to the receiver.
- **Nonce**. A counter indicating the number of transactions sent from the account.

- **GasLimit**. The maximum amount of *gas* units that can be consumed by the transaction. Units of gas represent computational steps. We will detail the role of gas in Section 2.3.4.

2.3.3 Blocks

In the Ethereum network, transactions are grouped into blocks. Similarly to what we have seen for Bitcoin, each block of the Ethereum blockchain is made up of a header and a list of transactions. As illustrated in Figure 2.3, the header contains several fields. Among the most important of them, we mention the following.

- **Number**. This value represents the height of the current block, i.e. the number of its ancestors. The genesis block, which is the first one added to the chain, has a number of zero.
- **ParentHash**. This value contains the Keccak 256-bit cryptographic hash of the header of the predecessor block. This mechanism ensures the immutability of the chain.
- **Beneficiary**. Defines the address of the miner that successfully mined this block. The mining rewards for this block are sent to this address.
- **StateRoot**. Contains the cryptographic hash of the root of the Merkle Patricia trie representing the world state up to this block.
- **TransactionsRoot**. Contains the root of a Merkle Patricia trie that groups the transactions of the block together.
- **LogsBloom**. This field contains a *Bloom filter* [4], a compact data structure that summarizes the event triggered by transactions inside the current block. This field is particularly important for the aim of the thesis and will be further discussed in the next chapters.
- **Difficulty:** A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp.
- **GasLimit:** A scalar value equal to the current limit of gas expenditure per block.
- **GasUsed:** A scalar value equal to the total gas used in transactions in this block.

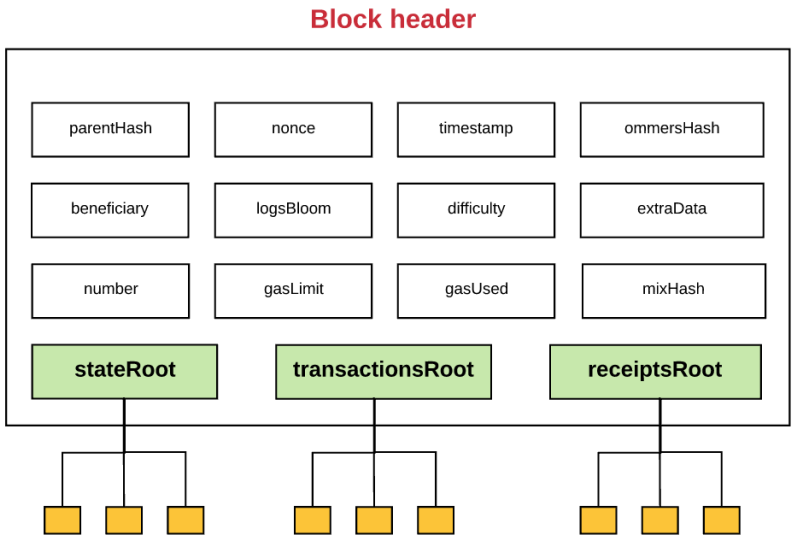

Figure 2.3: Illustration of the fields inside the header of an Ethereum block.

- **Timestamp**. The Unix timestamp indicating when the block has been added to the chain.
- **ExtraData:** An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer.
- **MixHash:** A 256-bit hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block.
- **Nonce:** A 64-bit value which,combined with the mixhash, proves that a sufficient amount of computation has been carried out on this block.

2.3.4 Fees

Similarly to what happens with cars, Ethereum contracts need a "fuel" to perform their computations. The fuel is called *Gas* [9] and we introduced it in Section 2.3.2. Now we show the definition of *gas* as stated in [3].

Definition 2.3.1 (Ethereum Gas)**.** *The Ethereum Gas is a unit of measure used to estimate the work done by Ethereum to make transactions or whatever interaction inside the network.*

An usage of gas occurs when an operation is completed, because Ethereum generates an amount of gas to be paid, the name of this quantity is *fee*. Now are going to introduce its formal definition, taken from the Ethereum yellow paper [32].

Definition 2.3.2 (Ethereum Fee)**.** *The fee schedule G is a tuple of scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.*

Chapter 3

Bloom filters and their application in Ethereum

3.1 Bloom filters

A Bloom filter [4] is a randomized data structure that can be used for representing a set of elements $X \subseteq \mathcal{U}$, where $\mathcal{U}$ is a *universe set*, in a compact way. This data structure can also be used for testing if a specific element x belongs to X.

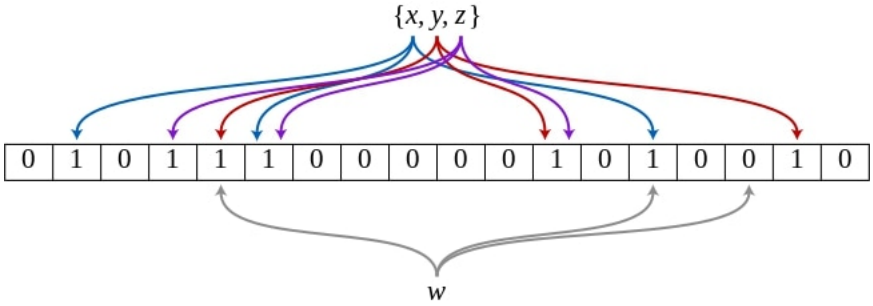

Figure 3.1: A picture representing an example of Bloom filter.

As highlighted in Figure 3.1, a Bloom filter is a bit array of m bits, all initially set to 0. The filter also employs k independent hash functions $h_1, \ldots, h_k$ that map elements of $\mathcal{U}$ to a position in the array. More formally, for each $i \in \{1, \ldots, k\}$ we have that $h_i : \mathcal{U} \to \{0, \ldots, m-1\}$.

Inserting elements To insert an element $x \in X$ we use the k hash functions and compute k different positions in the array, namely $p_1 = h_1(x), \ldots, p_k = h_k(x)$. Then we set $B[p_i] = 1$ for all $i \in \{1, \ldots, k\}$. To initialize the filter B with the content of the set X, we repeat this process for all elements $x \in X$. The whole process can be summarized by the pseudocode listed in Figure 3.2.

```
1 for each x in X do
2     for each i in {1,...,k} do
3         B[h_i(x)]=1;
4     end
5 end
```

Figure 3.2: Pseudocode for building a Bloom filter B for a set of elements X.

Testing the membership of an element After inserting all elements of X in B, we can use the filter to test whether a given element $x \in \mathcal{U}$ belongs to X. To this aim, we can proceed as follows. Once again, we use the k hash functions to get k positions in the array, namely $p_1 = h_1(x), \ldots, p_k = h_k(x)$. Then we check if the following property holds:

$$\forall i \in \{1, \ldots, k\} \; B[p_i] = 1.$$

At this point, we have two different possibilities.

1. If the property holds, we can conclude that x *might belong* to the set X. Notice that we cannot be sure about this fact, since the positions in the bit array might have been set during the insertion of another element $x' \neq x$.

2. If the property does not hold (i.e., if there exists $i \in \{1, \ldots, k\}$ such that $B[p_i] = 0$) we can conclude that definitely $x \notin X$.

The procedure we have just described can be summarized by the pseudocode of Figure 3.3. In this regard, we can observe that a particular feature of Bloom filters is the possibility of obtaining *false positives* when answering a membership query for an element x. As stated previously, if all bits for x are set to 1, we cannot be sure about the fact that $x \in X$. Indeed, given the possibility of having *collisions* on the hash functions, these positions might have been set during the insertion of another element. This might

lead us to conclude that $x \in X$ when in fact it is not (and this is why we call it a "false positive"). Conversely, if there is at least one bit that is still set to zero, we can be sure that $x \notin X$, because if it were inserted in the filter, all positions would necessarily be set to 1 (in accordance with what we described in Figure 3.2). For this reason, we can say that a Bloom filter has no *false negatives*.

```
1 for each i in {1,...,k} do
2     if B[h_i(x)]=0 then
3         return FALSE;
4     end
5 end
6 return TRUE;
```

Figure 3.3: Pseudocode for testing the membership of an element x using a Bloom filter B.

3.2 Properties of Bloom filters

As previously discussed, due to their randomized nature, Bloom filters could return erroneous answers to membership queries. Such answers are usually called false positives because we are declaring an element x a member of the set X when in fact it is not. As detailed in [4], it turns out, however, that the probability of getting a false positive (also called *false positive rate*) can be determined in advance if we know the following parameters.

- The number of elements to be inserted in the filter, denoted by n. This corresponds to the size of the set X.
- The number of bits used by the filter, denoted by m.
- The number of hash functions employed by the filter, which we denote by k.

Let us now consider an element $\overline{x}$. We would like to compute the probability that a membership query for $\overline{x}$ results in a false positive after the insertion of n elements in a filter with m bits and k hash functions. If we assume that each of the k hash functions selects each array position with the same probability, then the probability that a certain bit is not set to 1

by a certain hash function after the insertion of an element is equal to:

$$\left(1 - \frac{1}{m}\right)^k.$$

After the insertion of all n elements of the set X, the probability that the bit is still equal to zero can be computed as

$$\left(1 - \frac{1}{m}\right)^{kn}$$

and consequently the probability that the bit is set to 1 is

$$1 - \left(1 - \frac{1}{m}\right)^{kn}.$$

Now, in order for $\overline{x}$ to result in a false positive, each one of the k hash values $h_1(\overline{x}), \ldots, h_k(\overline{x})$ must correspond to a bit that is set to 1. The probability that this happens is given by

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

and with the approximation

$$\left(1 - \frac{1}{m}\right)^{kn} \approx \exp\left(-\frac{kn}{m}\right)$$

we get that the probability p of getting a false positive for $\overline{x}$ is

$$p = \left(1 - \exp\left(-\frac{kn}{m}\right)\right)^k. \tag{3.1}$$

We have to remember that the number of k hash functions must be positive. Thus, given m and n, the value of k that minimizes the false positive probability p is $k = \frac{m}{n} \cdot \ln 2$.

Other properties

In the following list, we briefly describe some other important properties of Bloom filters.

- Inserting a new element and testing the membership costs $O(k)$, where k is the number of hash functions used by the filter. Of course this property is true if we assume that the cost of computing a single hash function is constant.
- Unlike hash tables with open addressing for collision resolution, a Bloom filter has no limit on the number of elements. Thus, adding an item never fails.
- Union and intersection of Bloom filter with the same size are allowed and they can be implemented using the bitwise OR and AND operations.
  Particularly in Chapter 4 we will describe a structure which implements the union between Bloom filter.

3.3 Bloom filters in Ethereum

Bloom filters also find an application in the Ethereum blockchain [32]. Specifically, the header of each block of the Ethereum blockchain contains a field, called `logsBloom`, which corresponds to a Bloom filter for logging *events* triggered by transactions inside that block. In this section we introduce and discuss the main properties of Ethereum's `logsBloom` filters. Before doing that, however, we present the notion of event.

3.3.1 Events

Ethereum events constitute an important mechanism of communication between smart contracts [6]. Events also allow programmers to monitor the state of a contract and any possible change from the outside. In this section we describe how Ethereum events are generated by means of an example.

In Solidity, an event is declared by means of the `event` keyword and it typically takes a number of parameters. To fire an event, during the contract execution, the keyword `emit` is used. To clarify how events are declared and triggered, let us consider the `Transfer` event, which is one of the most common on the Ethereum blockchain and whose signature is described in Figure 3.4. The `Transfer` event is typically emitted when someone transfers tokens. Indeed, its signature is made up of three different variables:

1. the address from which the tokens are sent (the address `from`);
2. the recipient of the tokens (the address `to`);
3. the number of tokens transferred from the `from` address to the `to` address (the unsigned integer `value`).

```
event Transfer(address indexed from,
               address indexed to,
               uint256 value);
```

Figure 3.4: Signature of the `Transfer` event in Solidity.

Now let us consider the example contract called `MyContract` and whose code is listed in Figure 3.5. The contract defines a function named `doTransfer()` which triggers the `Transfer` event using the `emit` keyword. More precisely, when `doTransfer()` is called, the `Transfer` event is fired and its parameters are instantiated with the values of the local variables `addr1`, `addr2`, and `numTokens`.

```
contract MyContract {

    ...
    function doTransfer() public {
        ...
        address addr1 = 0xb794f5ea0ba39494ce839613fffba74279579268;
        address addr2 = 0xba52c75764d6F594735dc735Be7F1830CDf58dDf;
        uint256 numTokens = 100;
        ...
        emit Transfer(addr1, addr2, numTokens);
        ...
    }
    ...
}
```

Figure 3.5: An example of contract firing the `Transfer` event.

3.3.2 Receipts

The outcomes of transactions, as well as the events that have been triggered during their execution, are recorded in the so-called *transaction receipts* [32].

To understand the basic mechanisms behind Ethereum's `logsBloom` filters, we will focus in particular on the *logs* contained in a receipt. In order to understand what a log is, let us consider once again the `Transfer` event which we have previously described in Figure 3.4. This event has three parameters: `from`, `to`, and `value`. The first two are declared as `indexed`, while the third one is not. Now, let us suppose that a function inside a certain contract emits a `Transfer` event where the parameters are instantiated as in Figure 3.6.

```
1 emit Transfer(0xfbb1b73c4f0bda4f67dca266ce6ef42f520fbb98,
2               0x9e0f70dec65e4a62b5c4df1317f47fd2ef707d6c,
3               10235417200000000)
```

Figure 3.6: Emission of the `Transfer` event.

This invocation of `Transfer` generates the log described in Figure 3.7. As the reader can notice, the log contains a number of fields describing the invocation of the event. We have a field for each parameter of the event itself (i.e., `from`, `to`, and `value`) and each field contains the value passed for the emission (possibly encoded as a hexadecimal string). Then, there is an `address` field containing the address of the smart contract that generated the event. On the other hand, the `topics` array contains three different hexadecimal strings, which we describe below.

1. The first one is the cryptographic hash, computed using the Keccak-256 hash function, of the event *signature*. The event signature for `Transfer` is simply the string `Transfer(address,address,unit256)`.
2. The second one is the value associated with the `from` parameter, padded with zeros to 32 bytes on the left.
3. The third string corresponds to the value associated with the `to` parameter, padded once again with zeros to 32 bytes on the left.

The third parameter of the `Transfer` event, namely `value`, is not inserted in the `topics` array, since it is not declared as `indexed`. As a consequence, its value is inserted separatedly in the `data` field of the log and will not be recorded by the `logsBloom` filter, as will be detailed in the next section.

```
{
  "_from": "0xfbb1b73c4f0bda4f67dca266ce6ef42f520fbb98",
  "_to": "0x9e0f70dec65e4a62b5c4df1317f47fd2ef707d6c",
  "_value": 10235417200000000,
  "address": "0xbb9bc244d798123fde783fcc1c72d3bb8c189413",
  "data":
  ↪  "0x0000000000000000000000000000000000000000000000000245d0ec6557c00",
  "logIndex": 0,
  "topics": [
    "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
    "0x000000000000000000000000fbb1b73c4f0bda4f67dca266ce6ef42f520fbb98",
    "0x000000000000000000000000e0f70dec65e4a62b5c4df1317f47fd2ef707d6c"
  ]
}
```

Figure 3.7: A log for the emission of the `Transfer` event, represented using the JSON format.

3.3.3 Definition

As its name suggests, an Ethereum `logsBloom` filter records event logs, discussed in Section 3.3.2. The `logsBloom` for a given block is a Bloom filter made up of $m = 2048$ bits. As detailed in [32], the filter is initialized as follows.

1. We take all the log entries for all transactions included in the block. Each log entry O is structured as described in Figure 3.7.

2. For each log entry O, we consider the content of the `address` field and all the byte sequences included in the `topics` array. In other words, we consider the set $S = \{\texttt{address}\} \cup \texttt{topics}$.

   (a) For each sequence $s \in S$ we compute $H(s)$, namely the Keccak-256 digest of s. The result is a sequence of 256 bits (i.e., 32 bytes).

   (b) Then, let h_1, h_2, h_3 be the first three pairs of bytes of $H(s)$. Each one of these pairs is interpreted as an integer and is therefore reduced modulo $m = 2048$. The reduced values x_1, x_2 and x_3 are then used to compute a position in the bit array. Specifically, for each $i \in \{1, 2, 3\}$, we set to 1 the bit in position

$$pos_i = m - 1 - x_i.$$

Notice that for each element that is inserted in the `logsBloom` filter three different bits are set. As a result, we can say that the filter simulates the behavior of $k = 3$ independent hash functions.

3.3.4 Computing the false positive rate for a `logsBloom`

We can apply the formula we derived in Section 3.2 to estimate the false positive rate for a `logsBloom`. Indeed, in our case we have that the number of bits used by the filter is $m = 2048$, and the number of hash functions is $k = 3$. Therefore, by plugging these values in Equation 3.1, we get that

$$p = \left(1 - \exp\left(-\frac{3n}{2048}\right)\right)^3. \tag{3.2}$$

3.3.5 `logsBloom` saturation

Due to the fixed size of Bloom filters, they inevitably tend to fill up the more elements we add. The closer the filter gets to saturation (i.e., with all bits set to one) the higher the chance of false positives get when testing for an element presence. This degrades the accuracy, and so usefulness, of highly congested filters.

Clearly, the saturation problem needs to be taken into account when considering Ethereum's `logsBloom` as well. In such a scenario, a saturated filter would be useless as the high number of false positives would cause nodes useless work most of the time by wrongfully requiring them to download blocks that do not actually contain the events they searched for.
This particular situation is really hard to happen without human intervention because, it's necessary at least a number of events equal to the size of the filter divided by the number of hash functions. In our case that is 683 events, producing 2048 different positions to flip in a Bloom filter. However, we know that with a cryptographic hash function, every position has the same probability to be chosen. So a way to fill up a Bloom filter is trying to force the saturation. We know that `logsBloom` is 2048 bits and it sets 3 bits based on the hash of the item to point, so with 683 events we can fill up the entire `logsBloom` because we know a smart contract can make an undefined number of events. Next step is realize what are the data to insert in the logs to fill the Bloom filter. This basically just required taking 3-byte values, hashing them, checking what bits they would flip in the filter, and drop off any preimages that cross over with others. Finding for the right input data become exponentially more difficult going near to the full filter.

Not only this attempt requires high computational costs (the higher the bigger a Bloom filter is), but it also requires transaction fees to be paid for the events to be written on the filter on chain. So, let us estimate the gas cost of intentionally saturating a filter. Using the Ethereum yellow paper [32] we see that Log instructions cost 375 gas, plus 8 gas for each byte of topic data, 68 for each calldata (needed to call the function), and 21,000 for a single transaction fee. Summing all up, we would need $683 * (24 + 375 + 68) + 21,000 = 339,961$ gas to fill an entire block. Considering the current exchange rate of 1,279.43 USD for 1 Ether and the current average cost of 0.000000016 Ether per gas unit we obtain a cost of 6.96 USD to saturate a block [7]. This value may appear extremely low, however we have to take into account that it corresponds to the optimal lower bound of the cost to be paid. Moreover, new filter saturating events need to be computed anew for each block, requiring the computation to be done before a new block is created, i.e. every 14 seconds on average. Still the low amount leaves Ethereum `logsBloom` potentially vulnerable to such exploit.

Chapter 4

Skip lists

As we already said in Chapter 1, the goal of the thesis is the evaluation of the skip list for the Ethereum `logsBloom`. We are going to introduce them in this chapter, but now we need to recap why we need skip list. Ethereum is made block by block and if we want to find some information probably we need to scan many blocks. This operation could be wasteful if we don't find a way to skip some element for a search. The skip list is a solution for this problem. The entries of these skip lists that we are going to implement are Ethereum `logsBloom`, which we introduced in Section 3.3

4.1 Generic Skip List

Before proposing the formal definition of our skip list we present the original skip list concept, as described in [25]. Despite the many differences, they share a common factor, i.e. the goal of improving the time performance of traversing a list while skipping elements through targeted jumps.

Now we are going to give to the reader some relevant properties about a generic skip list:

- Skip list is a probabilistic data structure which means that a part of login of the implementation contains a degree of randomness.
- Skip list stores sorted elements.
- Skip list has $O(\log n)$ average complexity for search.
- Skip list has $O(\log n)$ average complexity for insertion.

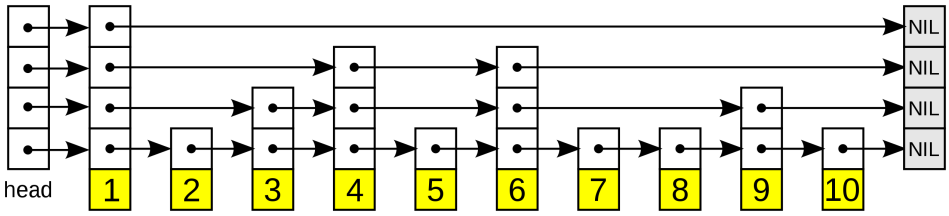

Figure 4.1: A picture representing an example of generic Skip List

4.1.1 Skip list construction and search

A standard skip list is composed by two layers, the first one is the bottom layer and it correspond to a linked list (the standard data collection), the second one is the top layer, a set of lines with jumps which links the elements of the linked list skipping some of them, using these jumps we reduce the number of nodes to lookup. The basic operation allowed are insertion, to add a new node inside the skip list, deletion, to delete a node, search, to search a specific element inside the set.

Searching

An effective example to understand the logic of skip list is the search.
We assume that the skip list stores a collection of sorted numbers:
$\{3, 7, 9...21, 25, 30..90\}$ and we want to verify if the number "25" belong to the set. We start from the first line and verify if "25" is inside the number in that line, if the answer is false we compare "25" with the number we found and based on the result we go for the next number in the line or for the line below.

Given:

- *key*: item to find
- S: the skip list

```
1 Search(key)
2     p = top-left node in S
3     while (p.below != null) do
4         p = p.below
5         while (key >= p.next) do
6             p = p.next
7     return p
```

Figure 4.2: Pseudo code of an algorithm for searching elements in skip list

The algorithm return p as the position of the element.

4.2 Our version of skip list

We want to use a skip list to help us in the intent of improving the search time on Ethereum `logsBloom`. We label our data structure as skip list, as it presents targeted jumps as the one described in Section 4.1. Our skip list uses its jump to point to the position 2^i with $i \in \{0, \ldots, n-1\}$ (where n is the number of jumps we want) of a list. For each jump, we have a descriptor, aggregating all indexes of the information inside the nodes included in that jump. This kind of skip list can work with any data structure as descriptor, as long as it allows the union of its underlying indexes, while keeping data integrity.

In our work we focus on Ethereum blocks, in which elements (i.e. events) are indexed inside a Bloom filter (the `logsBloom`). So Bloom filters will be used as single lists elements indexes and their union will be the descriptor of jumps. The obtained skip lists would then meet our goal of speeding up the search for event occurrences along the entire blockchain.

4.2.1 Bloom filter based skip list

We based our structure in Skip list because it fits perfectly with out intent of join the `logsBloom` preserving every data. In the following, we will assume that:

1. Each block b_i contains a set of events E_i;
2. The set of events E_i is summarized by a Bloom filter B_i which can be accessed from the header of the block b_i.

Based on the ideas proposed in [25] and [33], we call this data structure *skip list* and present its formal definition below.

Definition 4.2.1 (Skip list)**.** *Given a block b_i, the skip list for b_i is a sequence*

$$\langle B_{i,0}, B_{i,1}, \ldots, B_{i,m-1} \rangle$$

of m Bloom filters. For each $j \in \{0, \ldots, m-1\}$, the filter $B_{i,j}$ is defined as

$$B_{i,j} = \bigvee_{k=i-2^{j+1}+1}^{i-2^j} B_k$$

where $\vee$ denotes the bitwise logical OR operation.

With the help of Figure 4.3, we can illustrate the idea behind Definition 4.2.1.

Every entry the skip list contains a `logsBloom` which encloses information about the joint `logsBloom`s for that jump.
So now we have a list data structure with the properties to search and insert elements with a better average complexity, but this is not enough to conclude that this method will not have issues. `logsBloom` for Ethereum is a 2048 bit collection, the construction of skip list could fill up the entire collection with 'ones' and this could generated false positive. The aim of the next chapter is to do some experiments to verify the behavior of skip list's `logsBloom`.

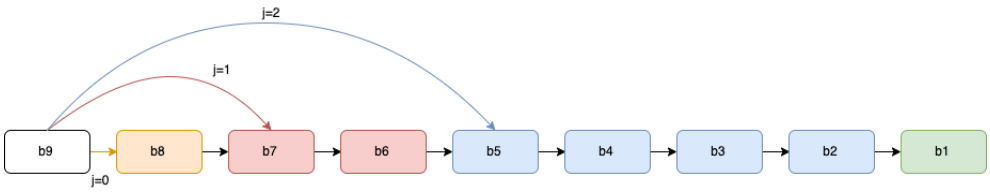

Figure 4.3: Picture representing an example of Skip List implemented by us.

Referring to Figure 4.3 we can now describe how the skip list is built. b9 is the starter block and it is the oldest one.
$j \in \{1, 2, 3\}$ are the jumps. Each block with the same color are the blocks which we joined their `logsBloom` to a unique `logsBloom` for a jump.

- The jump $j = 0$ includes the `logsBloom` of the block b8.

- The jump $j = 1$ includes the `logsBloom` built with the bitwise OR of the blocks b7 and b6.
- The jump $j = 2$ includes the `logsBloom` built with the bitwise OR of the blocks b5, b4, b3 and b2.

4.2.2 Construction time

The construction of our skip list follow a naive method. To build a skip list we scroll every block of the dataset, so the the computational cost of the operation is steady on $O(n)$. This approach follows Definition 4.2.1 introduced in the last section. An alternative method to build a skip list using a more efficient way is to join the `logsBloom` of the first and the last block of a jump instead of joining every blocks of a jump. This method allow us to build a skip list in logarithmic time, because we do one operation for each jumps of the skip list.

4.2.3 Searching for events

The problem we would like to solve is the following. Starting from a block b_u we want find every occurrence of a specific event E from the predecessors of b_u in an efficient way (by visiting the lowest possible number of blocks). Particularly, we are interested in finding every occurrence of E in the blocks between b_l and b_u, with $l \leq u$. The skip list allows us to speed up the search process by means of Bloom filters. An high-level description of the search process is presented as follows: each block containing at least an occurrence of E is going to be returned. We can observe that the result is not necessarily exact because we are using Bloom filters, which provide *approximate* information because of their nature.

1. We start from b_u and we check the Bloom filter B_u of the block, which resumes all events inside the block itself. We test if B_u contains the event E, if the answer is affirmative we add b_u to the result.
2. At this point, after analysed b_u, we examine its skip list in ascending order to find out which are the previous blocks that own E.
3. Before, it's necessary determinate the *maximum possible jump*, or the maximum $0 \leq j \leq m-1$ such that the block $u - 2^j$ is inside the range $[l, u]$. We call that maximum value j_{max}.

4. Now, for each $j \in [0, j_{max}]$ we analyse the number j entry of the skip list of b_u. Example: referring to Figure 4.3, we assume that b_9 is our b_u. We analyse the jump $j = 0$ (the yellow one), then the jump $j = 1$ (the red one) and then the jump $j = 2$ (the blue one). If one these filters return me an affirmative response, then we explore recursively the blocks relative to him. For example if the filter of the jump $j = 2$ answer me that it includes the event, then I'm going to explore the range of blocks between b_2 and b_5 (the blue blocks).

5. When we terminate to explore the skip list of b_u, we can jump on the first jump outside our skip list, to continue the research. Referring again to Figure 4.3 we mean the block b_1, the green one.

We can resume this procedure in the pseudocode reported in Figure 4.4. In particular we are going to find the occurrences of the event e in the range of block $[l, u]$.

```
1  Search(l, u, e):
2      R = {}
3      while (u >= l) do
4          curr = b[u]
5          if (curr.B contains e)
6              R = R + {curr}
7          jmax = max {j | 0 <= j <= m-1 && l <= u-2^j}
8          for (j = 0; j <= jmax; j++) do
9              l' = u - 2^(j+1) + 1
10             u' = u - 2^j
11             if (curr.S[j] contains e)
12                 R = R + Search(l', u', e)
13         u = u - 2^(j+1)
14     return R
```

Figure 4.4: Pseudo code of the searching algorithm used to finding event occurrences along the blockchain.

Computational cost

A property in common with the generic skip list presented in Figure 4.1, is the possibility to look for an occurrence of an item inside a set in a logarithmic time. This is the time computational complexity of the algorithm

(showed in Figure 4.4) if we stop the algorithm when we find an occurrence of the element we were searching for.

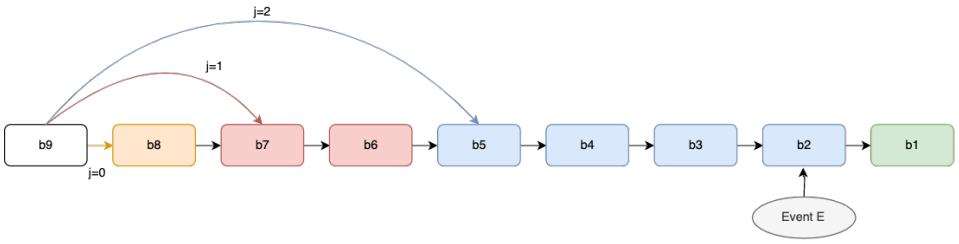

Figure 4.5: Example of Skip List that we implemented including an event inside a block.

Referring to Figure 4.5 we are going to simulate the process of searching for an event, illustrating the final search route in Figure 4.6. We suppose to have an event E and an occurrence of it located inside the block b_2. We build the skip list basing it on the blocks of the figure Figure 4.5, and we want to find that occurrence of E. Following the algorithm, we check if E belongs to the jump $j = 0$ and $j = 1$ obtaining a negative answer for both of them. So we go to check the jump $j = 2$, but this time we got an affirmative answer because the occurrence belongs to that jump. We remark that the occurrence could be present when the three bits retrieved from the three Bloom filter's hash functions (with E as input) are set to 1.

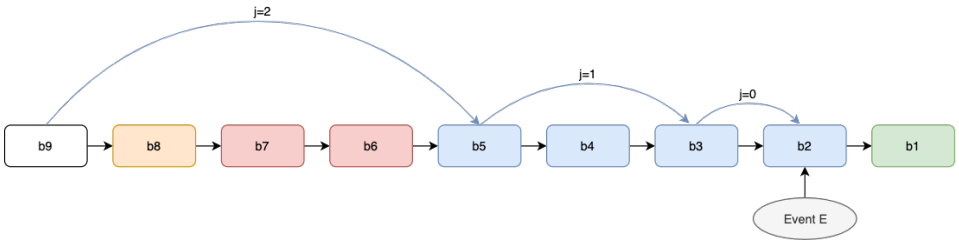

Figure 4.6: Example of route to find the occurrence E inside the skip list.

At this point we have to explore recursively the blocks contained in the jump $j = 2$, in this case the blocks b_5, b_4, b_3 and b_2 and we know that one of these contains the occurrence of E (ignoring false positives). We start the verify from the block b_5 (the first of the jump $j = 2$) and after two further jumps (see Figure 4.6), we are going to find the occurrence in the block b_2,

so the program returns as result the position b_2, because we want to find only the first occurrence. The procedure takes a logarithmic time because we skip every jump which does not contain the element we are looking for, and the jumps we skip become bigger and bigger exponentially. When we skip a jump, the space of blocks is halved, because the number of blocks inside the discarded jumps is equal to the number of the blocks inside the jump we are checking to find the occurrence.

Chapter 5

Experimental results

In the previous chapters we introduced what blockchain, Bloom filters, and skip lists are. In this chapter we evaluate our proposed skip list's implementation in Ethereum. Skip lists allow us to scan a blockchain with improved time performance at the cost of a chance to find false positive. We know a skip list is constructed through bitwise OR operations between `logsBloom` for each jump. This means that, in general, with these operations the number of 'ones' for each Bloom filter might grow until filling them. When this happens, even if the skip lists improve the time of scanning blocks we are going to get more false positives.

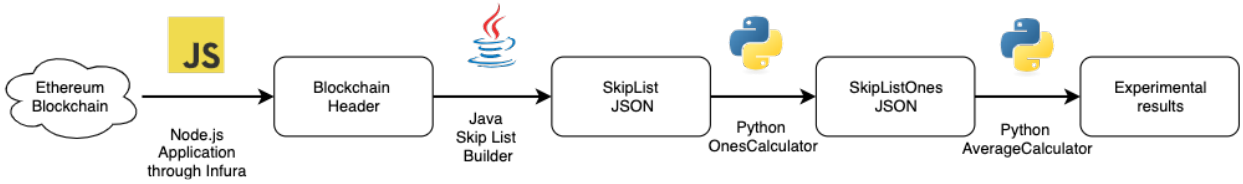

Figure 5.1: Data flow of our experiments.

5.1 Data set

5.1.1 Ethereum header

The first thing to do to follow our line is get a proper dataset and what we really need are 2 elements inside the Ethereum header: position and `logsBloom`. We used a javascript library called Web3 to download some blocks distributed in different dataset to evaluate many different possibility.

Of course every results we are looking in the analysis section will refer to the same one. The implementation of this was made in a node.js project. We used two methods, one to get the numbers of blocks in Ethereum and another to read the headers of N blocks starting from the last block created.

```
//Number of blocks in the blockchain
web3.eth.getBlockNumber()
    .then(console.log);

//Read the 3150-th block
web3.eth.getBlock(3150)
.then(console.log);
```

Figure 5.2: Two methods used from web3.js

With a loop we can compute the getBlock() function for N numbers from r where r is the result of getBlockNumber() to r-N [30].

The final result is a JSON file containing a list of N headers.

```
{
    "number": 3,
    "hash": "0xef95f2f1ed3ca6.....9a2c4e133e34b46",
    "parentHash": "0x2302e1c0b972d0093...d42504c05eaf9c88",
    "baseFeePerGas": 58713056622,
    "nonce": "0xfb6e1a62d119228b",
    "sha3Uncles": "0x1dcc4de8dec75d7a....142fd40d49347",
    "\texttt{logsBloom}": "0x0000000000000.....00000000000000...",
    "transactionsRoot": "0x3a1b038751.....397b8acf415bee",
    "stateRoot": "0xf1133199d44695d....a94cda515bcb",
    "miner": "0x8888f1f195afa192cfee860698584c030f4c9db1",
    "difficulty": '21345678965432',
    "totalDifficulty": '324567845321',
    "size": 616,
    "extraData": "0x",
    "gasLimit": 3141592,
    "gasUsed": 21662,
    "timestamp": 1429287689,
    "transactions": [
        "0x9fc76417374aa880d4449a1...d2d66f4c9c6c445836d8b"
    ],
    "uncles": []
}
```


Figure 5.3: Example of an Ethereum block header parsed as JSON

As we said in the last chapter our work will be done using the `logsBloom` field.

5.2 Implementation

We made a Java program (Called "Skip list builder") to read the JSON file containing the Ethereum headers, built the skip list and produce a JSON containing a representation of the skip list.
After building the skip lists it is necessary to elaborates and evaluates the result to find out the behavior of `logsBloom`.
To simplify the work of analysis we used Python as programming language, especially three libraries; Numpy to calculate in an easy way some mathematical functions, pandas to manipulate inputs from skip list's files and Pyplot to produce graphics.
We wrote two main Python scripts to import the skip lists result and elaborate all data to find out the number of ones inside a skip list's `logsBloom`. This because with the growth of jumps if the number of ones increase means

that we are going to catch false positive and this is what we do not want to. The last Python script was made to calculate the time needed to build a skip list with a specific number of jumps. We could make all calculations inside the same program but we wanted to produce different level of data to allow us to think about different uses in the future.

5.2.1 Skip list builder

Classes

Here it is a list of classes used in the program:

- **LogsBloomInfo:** class which contains the position and the relative `logsBloom` of a block and the getter and setter methods..
- **LogsBloomManager:** this class is an handler class which makes the final SkipListLogsBloomFinalInfo list using SkipListLogsBloom class methods.
  Some relevant methods are:
  - *initLogsBloomList*: which runs for each block of our dataset the method to build the skip list.
  - *printOrResult*: which help us to visualize the results of OR operations.
- **SkipListLogsBloom:** This class implements Collections and methods to build the skip list.
  Some of them are:
  - *populateListLogsDeiSalti*: Which is the method computes the OR operations between jumps using other utility methods.
  - *stringToBitSet*: This method convert a string to a BitSet, the data structure we used to perform the OR bitwise.
- **LogsLister:** class which contains a list of LogsBloomFinalInfo object to be written in a JSON file. The only methods contained are getter and setter.
- **SkipListLogsBloomFinalInfo:** class which represent the final skip list, it contains an integer who is the starter position of the skip list and an HashMap with the key as the jump and the value is the bitwise or of the `logsBloom` in the specific jump.

- **SkipListTimer:** class useful to keep track of the building time of a skip list.
- **Utils:** Class which contains a list of utility methods, for example:
  - *booleanToChar*: convert a boolean to a char. (0 or 1).
  - *stringToBinary*: convert string to a binary string.

Execution

The program acts in this order: at the beginning, inside the main class, it opens the JSON file containing the eth's headers and makes a list of `logsBloom`Info with position and `logsBloom` of every blocks readed, After it instantiates a `logsBloom`Manager Object that takes as constructor parameter the `logsBloom`Info list, the number of blocks we read (N) and the number of jumps for the skip list (M) and call the public method init`logsBloom`List to start the computation.
We are moving inside the Manager, in the init method which scans all element inside the list and foreach of them populates a list of `logsBloom`FinalInfo, beacuse every blocks we used for the `logsBloom`Info list should be a starter index of a skip list which do M jumps.
The final result is a LogsLister Object which contains the list of blocks we read with the relative skip list made with the bitwise-or of the `logsBloom` of the relative jumps. Backing in the main class, this LogsLister will be converted in a JSON and written in a file, the structure of the logsLister is done specifically to be analyzed from other scripts in order to extrapolate statistics and produce plots.
The JSON result show a list of "skip list" with the position of the starter block and the list of all `logsBloom`s for each jump. Another result produced and generated by the program is a JSON file containing a list of long which represent the building time of any skip list, this is going to be useful to get a comparison between skip list with different number of jumps. Another relevant features implemented in the program is a class which calculates the building time of the skip lists using the system nanoTime() method. In the end of the program the class generate a JSON object with 2 entries, the number of the jumps chosen and a list with the building time for each skip lists.

5.2.2 Analysis with Python scripts

First Python script, calculating ones inside every skip list jump

The first one take as input file the JSON result of the Java program from Subsection 5.2.1

After having read the JSON file we looped the list and used a function to calculate the number of ones for each jump. Starting from it we can compute other statistics as the percentage of ones for each jump. After all we produce a new JSON files containing the same structure of the input with the difference of the `logsBloom` value that do not represent the `logsBloom` binary string but the number of ones in that string. Though the principal data we need is the number of ones, we can easy add in the result stats like $averageOfOnes$ referring to the same jump.
Beyond the main file we used a class which we defined, called *logsAnalysis* to contain every entry of the skip list. This class implements some utility methods to manage and elaborate the data from the skip list JSON. The following list is a list of utility functions used:

- $makeAvgOnes$: calculate the average of ones for a `logsBloom`
- $getNumberOfOnes$: counts the number of 'ones' inside a binary string.
- $makeListWithJump$: make an object with an entry $jump$, $avgOnes$ and $numberOfOnes$ which they are the results from the last two functions.

And this is a list of method from *logsAnalysis*:

- $insertObject$: which take as param an object from $makeListWithJump$: and build a new list with the numbers of 'ones' for all the jump for each the skip list.
- $makeFileJson$: convert the *logsAnalysis* skip list collection to a JSON to write the result in a file.

Second Python script, Average of ones for each jump

The second one took the final JSON produced in the first script and make the most important result for all the thesis, or rather the average of ones for every equal jump.

All operations are resumed in a single main file which has some functions explained in the following list:

- *buildStructResult* : this function build the structure for the JSON result.
- *makeCalcResult* : this function aggregates all counts of ones for each kind of jump in k lists, where k is the number of jumps for the skip lists.
- *populateResult* this function take every entry of the list generated in the last function and make the average of ones for every jump, this values are the main result of all the thesis.

For example if the first script produces a JSON with two entry with "position" 31 and 32 which inside their "listOfJumpAndNumber" there is one element with "jump" 1 having as values of "one" 300 and 200 the average for every jump "1" is 250.
This result is going to be the most important because if we look the average of 'ones' for each jump and if with the increment of jump the number of 'ones' increase it means that `logsBloom`s will fill up and we will have a higher false positive rate. We further examine these behaviors in Section 5.3.

Third Python script, time needed to build a skip list and other scripts

The third Python script takes as input a JSON file with two elements, the number of jumps for the skip list that we built n and a list with the time needed to build each one l. The program calculates the average value for the list, which refer to the time in nanoseconds need to build a skip list with n jumps. We also produced other Python scripts useful to generate tables and plots which the reader may see in Section 5.3.

5.3 Analysis

In this section we present the experimental results obtained on the previously described dataset of Ethereum `logsBloom` data.

5.3.1 Analysis of construction time of skip lists

Before discussing about the dataset and the implementation of skip lists, we stop to examine the time needed to built them, analyzing the result from the third Python script presented in Section 5.2.2.

The following figure is a table which has, in the first column, the number of jumps of a skip list, in the second one the average time measured in nanoseconds to build a skip list with that number of jumps. The average is made up using a collection of 1024 skip lists.

Jump	Average time (ns)	ratio with the previous jump
2	162133	-
3	445146	2.7455
4	1031354	2.3169
5	2291881	2.2222
6	4541186	1.9815
7	9454679	2.0820
8	17388066	1.8390

Table 5.1: Average building time in nanoseconds for skip lists

As we can see in Table 5.1 the average building time increases radically following an exponential growth as we can see from the third column and in Figure 5.4, which visualizes a graphic elaboration of the data from the table.

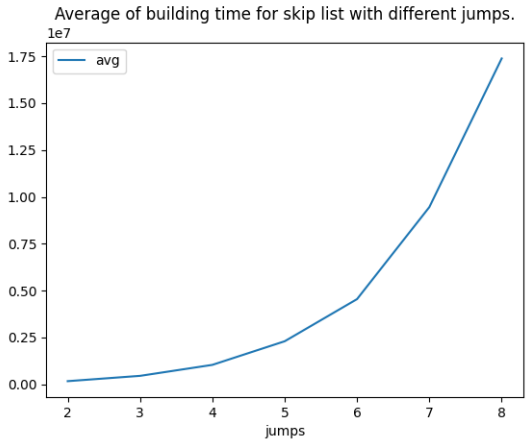

Figure 5.4: Trend of building time average for skip lists having different numbers of jumps

The X-axis represents the number of jump for a skip list and the Y-axis represents the nanoseconds average. In the example reported in the Figure 5.4 we can observe that the bigger is the number of jumps, the bigger is the time to build a skip list with that number of jumps. It follows an exponential curve, which is congruent with the growing of the number of operations that we do to build that list. This happens because the growth of the jumps number correspond to a growing of the width of the jumps, so our expectation could be that increasing the number of jumps the time is going to rise following the same shape. Obviously repeating the experiment the exacts times are going to change, because they depends on the computer, but the significant outcome is not the specific time but the growth of them in relation to the growth of the number of jumps for the skip lists.

5.3.2 Analysis of `logsBloom`

In this subsection we are going to talk about our dataset, we used a list starting from the block number 13 916 166 and with a size of 100 000 units, these blocks refer to the period from January 1st, 2022 to January 16th, 2022.
We conducted a preliminary analysis of this datasets, or rather we converts every `logsBloom` to binary strings to know the number of '1' foreach `logsBloom` and reminding us that the size of a `logsBloom` is 2048 bits, we compute three mathematical indexes to find some preliminary information

about the set.

- **Average:** the average of '1' inside a `logsBloom` is this dataset is 777, so the 37.94 percent of every bits are '1'.

- **Standard Deviation:** the standard deviation of this dataset is approximately 401. This value helps us to know if the average is made by similar values or opposite values. (numbers of '1' in a `logsBloom`) and this is good to know the reliability of the statistics. Interpreting this result we can figure out that our standard deviation is high so the there is a dispersion of data from the dataset and this is good for us because we can understand there aren't peaks of single values.

- **Coefficient of variation:** Another index we can figure out is the coefficient of variation that we can compute doing the following formula:

$$\frac{\text{Std. deviation} \times 100}{\text{Average}}$$

  This result is useful when we want compare two results from two different dataset and understand which one has the bigger standard deviation in relation to the average.
  Our coefficient of variation is approximately 51%.

Now we are going to compare them with some coefficients from different datasets which refer to the same period of ours but for the years 2019, 2020, 2021.

Year	Average of ones	Standard Deviation	Coefficient of Variation
2019	356	198	**55%**
2020	439	247	**56%**
2021	806	161	**20%**
2022	777	401	**51%**

Table 5.2: Analysis of dataset from last 4 years.

These results mean our dataset is more reliable than the 2021 dataset because the Variation Coefficient is higher so the distribution of ones is spreader. As we can see the 2019 and 2020 coefficient is quite higher then our so this means that they are reliable too, but obviously they are a bit old. A further information we can retrieve from the Table 5.2 is the big growing of the ones average. The 2019 was 356 and in just two years it has doubled

and more, this could means that at the moment we have a bigger rate to find false positive for a single `logsBloom` than some years ago.
Another important analysis is showed in the following Figure 5.5, the distribution of ones in the `logsBloom` for the entire dataset.

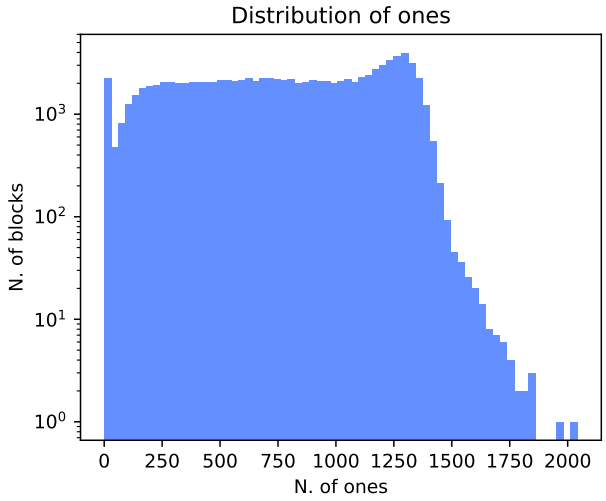

Figure 5.5: Distribution of ones in Ethereum's `logsBloom` filters (Dataset of year 2022).

- The X-axis indicates the numbers of ones (from 0 to 2048).
- The Y-axis indicates the number of blocks with x ones. The measuring unit is 10^y with y from 0 to 4.

As the reader can see there are only 2 blocks with a number of ones around 2.000 and little more than 10^3 blocks have 0 'ones'.
Beyond these two specific cases, the distribution is quite flat from the blocks with 250 'ones' to the blocks with around 1100 'ones', after grows up to 1250 and after decreases until it reaches about 1800.

Distribution of ones from last years

In this subsection we are going to analyze three others plots (as the Figure 5.5) referring to the year 2019, 2020 and 2021 and develop a comparison between these results. The first figure show the plots about 2019 and 2020 together because they have a similar coefficient of variation behaviour, as

we can see in the Table 5.2, which is better than value of the dataset we used for our experiments, this means that these two could be reliable for our purpose (differently with the 2020's one).

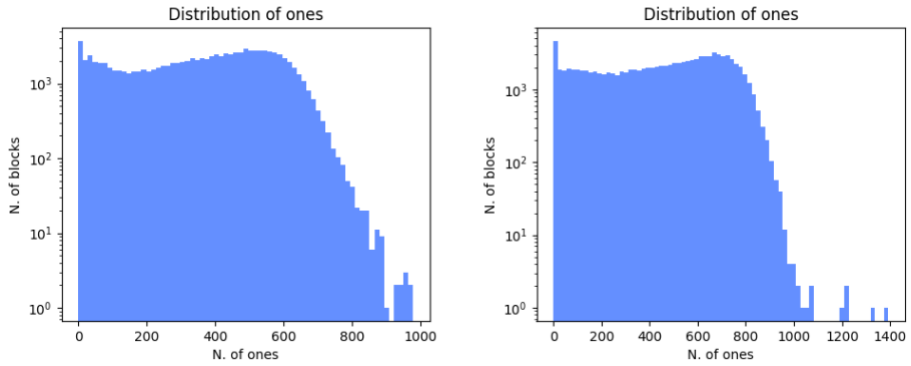

Figure 5.6: Distribution of ones in Ethereum's `logsBloom` filters (Dataset of years 2019 and 2020).

The difference between these two datasets is given mainly by the average of ones which is bigger in the year 2020. This fact confirms the filling of the `logsBloom` in the recent years. As the reader may notice another factor which influences the average is the appearance of a new maximum value for the number of 'ones' in the 2020 plot, which is near 1400, while in the 2019 the maximum was almost 1000. Now we are going to have a look at the 2021 plot separately because this dataset is going to be a bit weird if compared to the others.

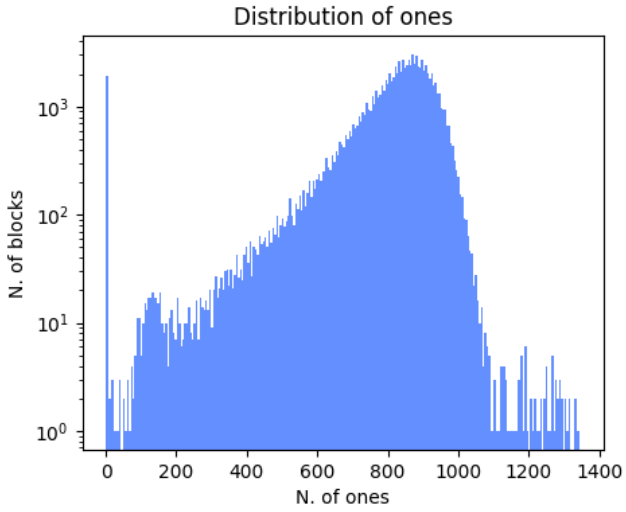

Figure 5.7: Distribution of ones in Ethereum's `logsBloom` filters (Dataset of year 2021).

As illustrated in the Figure 5.7 the trend of the distribution is completely different from the three plot we saw before. We can see a big quantity of `logsBloom` with 0 'ones' (in order of 10^3) and after a rising which terminate around 900 and the blocks from 800 to 1000 are the majority. Another confirm is given by the fact that even if the maximum number of ones is near 1300 and there are less blocks after 1100, the average of this here is bigger than the average of the year 2022, see Table 5.2. These data point out the low scattering of values which is projected in the coefficient of variation, which is the lowest of the group by far, as we can see in Table 5.2.

5.3.3 Analysis of the proposed Skip lists

In this subsection we are going to analyze the results and define a final evaluation.

To build the skip lists and to analyze their behaviour we used 1024 blocks from our dataset introduced in the Section 5.3.2. Though it is quite smaller the entire dataset, it is enough to produce reliable values.

The following Figure 5.3 shows a table where every row refers to a jump and the columns refer to a specific value about the jump.

1. First column refers to the entry of the jump

2. The second one refers to the average number of 'ones' of every skip list for that jump.

3. The third shows the relative percentage of the average where the 100% is 2048.

The reader may notice that the presence of ones inside `logsBloom` grows very fast up to around 90% at the fourth jump. This result is caused by the high average of ones inside a single Ethereum `logsBloom` (as we can see in the last Figure 5.5 and because their position distribution is uniformly random. We can infer that even if we implement the skip list to improve the query performance of an element, with increasing of width for a jump (and we need just few jumps) we will find an higher false positive rate and this behaviour is to the detriment of time efficiency.

Entry	Avg. number of ones	% of saturation
0	738.669274	0.360678
1	1186.850279	0.579517
2	1606.735376	0.784539
3	1867.811864	0.912018
4	2001.823529	0.977453
5	2043.381579	0.997745
6	2047.990602	0.999995
7	2048.000000	1.000000

Table 5.3: Skip list result of saturation of filters with $m = 8$ entries.

The next figure represents a plot about the trend of the average of 'ones' for each jump for the skip lists, of course this refers to the same data exposed in the last Table 5.3.

- The X-axis indicates the numbers of ones (from 0 to 2048).
- The Y-axis indicates the number of the jump, denoted by entryId.

As we can see the growing has a logarithmic shape which drops and becomes constant when reaches 2048.

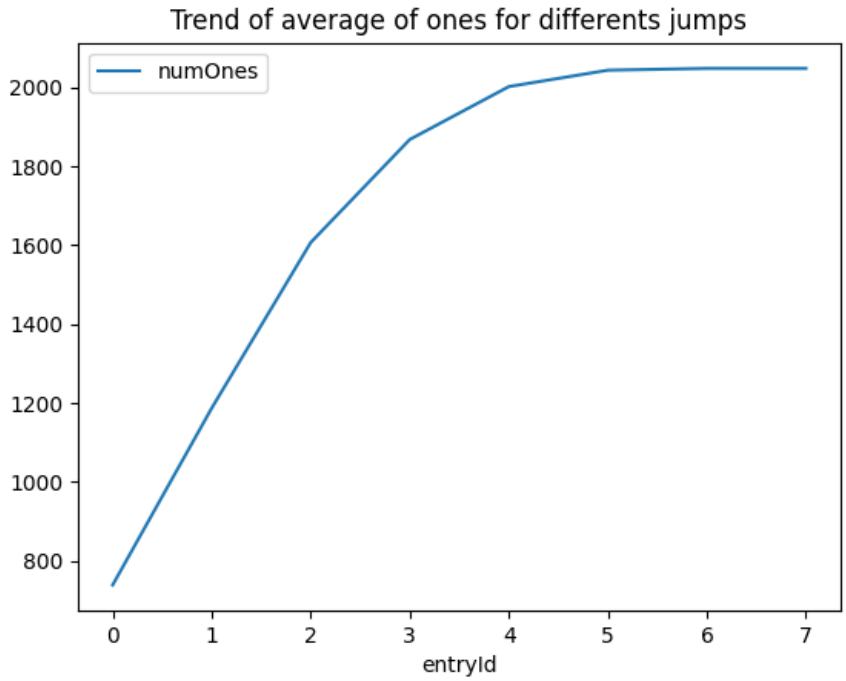

Figure 5.8: Plot about the growth of number of 'ones' from our skip lists.

Changing some parameters or configuration of skip list, our expectations could not be different from the result of these analysis.
If we increase the number of jumps obviously the trend of the plot will be constant after a number of jumps similar to the plot illustrated in Figure 5.8. With these configurations is difficult to expect a different outcome if not finding a different number of ones inside a `logsBloom`, but this is not related with us.
A similar situation was described in the Section 3.3.5, where we talked about a different type of saturation of a `logsBloom`. In the next chapter we are going to argue some idea to find out a solution.

5.3.4 Estimating the number of items inside a `logsBloom`

As described in [26], it is possible to estimate the number of items inserted in a `logsBloom` just by looking at the number of bits that are set to 1. More

precisely, with the help of the following equation

$$n^* = -\frac{m}{k} \ln \left(1 - \frac{X}{m}\right) \tag{5.1}$$

we can compute n^*, which is the estimated number of items inside the `logsBloom`. In the equation, we have that:

- m is number of bits for a `logsBloom`. In our case, it is $m = 2048$;
- k is the number of hash functions, which in our case is equal to 3;
- X is the number of bits set to 1 inside a `logsBloom`;

The variables took the same values from the example used in Section 3.3.4, except for X for which we take the value 777 as experimentally measured in the previous sections. Doing the computation we figure out that the expected value of items inside a `logsBloom` is 325.

5.3.5 `logsBloom` expansion

What we realized in the last section examining the Figure 5.8 is that this kind of skip list built using this kind of Bloom filter could not center our aim.

Increasing the size of `logsBloom`

With the analysis of the `logsBloom` we saw the high average of ones for a block (around 38%), so a possibility of improvements could be to increase their size. Referring to the Equation 3.1 we can change the m parameter, which was the number of bits in the filter. For example, if we double up it, it reaches 4096 bits, using this data as input of the equation we got 0.95% false positive for that Bloom filter, against the 5.4% of the 2048-bits Bloom filter we could obtain better results. The next table show the false positive rate difference between Bloom filters with different sizes.

Number of bits for a Bloom filter	% of false positive
2048	5.43%
3072	2.01%
4096	0.95%
5120	0.52%
6144	0.31%
7168	0.20%
8192	0.14%

Table 5.4: False positive rate for Bloom filters with different sizes.

The next plot shows the false positive rate trend. It falls with the increasing of the size of the Bloom filter.

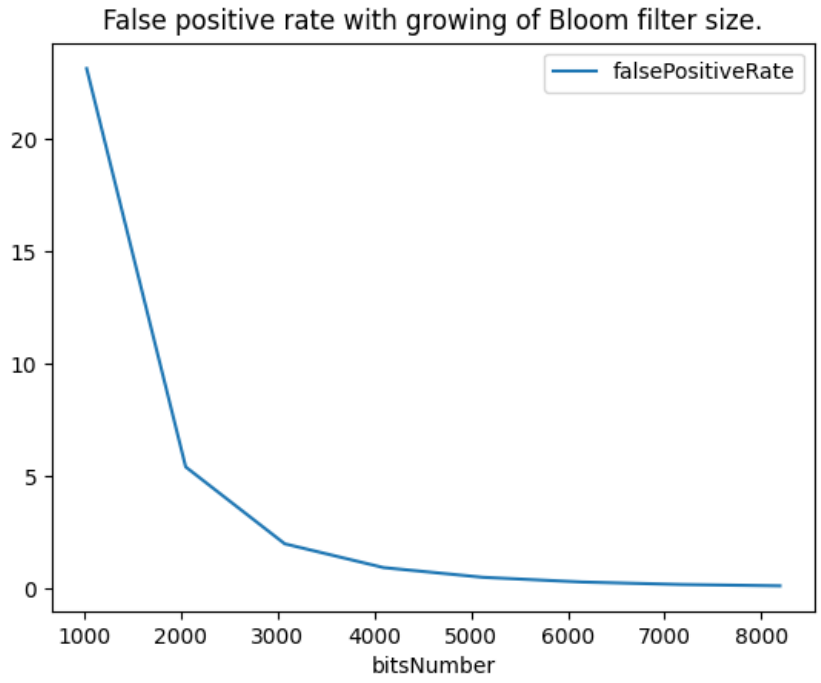

Figure 5.9: False positive rate for Bloom filters with different sizes.

We used the same parameters of the example in the Equation 3.1, we

only changed the size of the filter. The X-axis represent the bits size of Bloom filter and the Y-axis represent the false positive rate. We can observe how the false positive rate falls reaching a value near the 0%. This is a good a indication because beyond the lower false positive rate if the number of bits becomes bigger, the number of collisions when we do the bit-wise or operation is going to be smaller. What we can do now is simulate the saturation of skip list's Bloom filter increasing the size of `logsBloom`. We know that the number of items inside a `logsBloom` is 325 (using Equation 5.1), and we assume that the number of items double itself for each jumps of the skip list. The following table represent an hypothetical number of items inside a 2048 bits `logsBloom` for each jump.

Jump 1	Jump 2	Jump 3	Jump 4	Jump 5	Jump 6	Jump 7	Jump 8
325	650	1300	2600	5200	10400	20800	41600

Table 5.5: Number of items in Bloom filters jumps assuming "worse" cases.

Now we know how many items could include a `logsBloom`, so we can invert the Equation 5.1 to obtain the number of ones X^*, given m which is the size of the `logsBloom`, k the number of hash functions which is 3, and n is the number of elements inside in according with Table 5.5.

$$X^* = m\left(1 - \exp\left(-\frac{kn}{m}\right)\right)$$

Applying the formula to every entry of the table using many size-different Bloom filter we obtain a list of expected ones for each jumps for different Bloom filter. The results are exposed in the following table.

m	1	2	4	8	16	32	64	128
2048	776	1257	1742	2002	2046	2047	2047	2048
4096	867	1551	2515	3485	4005	4093	4095	4095
8192	919	1735	3102	5030	6971	8010	8187	8191
16384	946	1838	3470	6205	10061	13943	16020	16375
32768	960	1893	3676	6940	12411	20122	27887	32041
65536	967	1921	3786	7353	13881	24823	40244	55775
131072	971	1935	3842	7572	14706	27763	49646	80488
262144	973	1942	3871	7684	15144	29413	55527	99293

Table 5.6: Projection of number of ones for different Bloom filter jumps with different sizes.

The next table represents the same data of the last one but displaying the percentages.

m	1	2	4	8	16	32	64	128
2048	37.87	61.40	85.10	97.78	99.95	99.99	99.99	100
4096	21.18	37.87	61.40	85.10	97.78	99.95	99.99	99.99
8192	11.11	21.18	37.87	61.40	85.10	97.78	99.95	99.99
16384	5.77	11.11	21.18	37.87	61.40	85.10	97.78	99.95
32768	2.93	5.77	11.11	21.18	37.87	61.40	85.10	97.78
65536	1.47	2.93	5.77	11.11	21.18	37.87	61.40	85.10
131072	0.74	1.47	2.93	5.77	11.11	21.18	37.87	61.40
262144	0.37	0.74	1.47	2.93	5.77	11.11	21.18	37.87

Table 5.7: Projection of percentages of ones for different Bloom filter jumps with different sizes.

The first comparison we can do is between the results retrieved from our skip list analysis showed in Table 5.3 and the first row in Table 5.6. We can see that the results we obtained and the expected results from the formula are similar, ours results are quite better because the number of ones for each jumps is a bit smaller then the expectation results. So referring to the same dataset we can expect that increasing the size of `logsBloom` we could find better results than the Table 5.6. Analyzing the other entries of the table we can notice an expectation of the saturation of Bloom filter. The results show us the lower saturation when we chose a bigger filter. This is a good behaviour because at this point we know that a good way to resolve the

problem of the saturation of the skip list's Bloom filter is to increase the number of the bits for a `logsBloom`.

The last statistic we want produce is made by applying the Equation 3.1 to every jumps of each size variant of skip lists, to calculate new false positive rates. We used as n the results from the Table 5.5 and as m growing sizes of Bloom filter. The results are summarized in the next table.

m	1	2	4	8	16	32	64	128
2048	5.43	23.15	61.64	93.49	99.85	99.99	99.99	100
4096	0.95	5.43	23.15	61.64	93.49	99.85	99.99	99.99
8192	0.14	0.95	5.43	23.15	61.64	93.49	99.85	99.99
16384	0.019	0.14	0.95	5.43	23.15	61.64	93.49	99.85
32768	0.002	0.019	0.14	0.95	5.43	23.15	61.64	93.49
65536	0.0003	0.002	0.019	0.14	0.95	5.43	23.15	61.64

Table 5.8: Percentage of false positive rate for different variants of Bloom filters with different numbers of items.

As we can see in the Table 5.8, the rate falls down increasing the size.

Another way to solve the problem is changing the data structure used for the skip list. We can use every structures which have a computational time complexity better than $O(n)$ that support the union operation to aggregate the elements. An example could be using a Cryptographic Accumulator, a space- and time-efficient data structure used for set-membership tests [23].

Chapter 6

Conclusions and future developments

The goal of the thesis was to evaluate a way to improve events search time performances in Ethereum, as, currently, the only method to retrieve information from the chain is to inspect every blocks sequentially. Ethereum chain contains millions of blocks, making this approach computationally unbearable for arbitrary old information. Our idea is to use a data structure that helps us to skip unnecessary blocks. To this aim we proposed and implemented a skip list data structure based on Ethereum's `logsBloom` filters. A Bloom filter is a probabilistic data structure that permits to know if an item belongs to a set in constant time, and our skip list has a logarithmic computational complexity for searching elements. Our main contribution is the combination of these two data structures, outlined in Chapter 4, thus allowing for efficient information retrieval on the Ethereum blockchain.

The experimental results, presented in Chapter 5, sheds light on this new data structure, for example measuring how fast the underlying filters fill up as the size of the jumps increases. With the real Ethereum data, after just three jumps the average of 'ones' inside a Bloom filter reaches 90%. This is not a good behaviour if what we want to do is speed up the search process, because a Bloom filter filled with 'ones' would most likely produce false positives. This means that, even if with a Skip list we can skip a lot of blocks, bigger is the size of the jump, bigger is the rate to find a false positive which voids the whole query process. This is why we have also measured how much the Ethereum filter sizes should be expanded to support desired false positives rates.

Future developments

We can highlight three different main ways to follow to improve on our proposal. The first one is to practically implement on Ethereum the increased Bloom filter sizes as proposed in Section 5.3.5. This would reduce the false positive rate at the expense of increased block sizes (see Section 5.3.5). The second one is to consider different data structures instead of Bloom filter. To be usable, they should have the same properties, such as allowing for efficient union operation and preserving key integrity of data guarantees. For example we could replace the default Ethereum Bloom filters of our implementation with block level cryptographic accumulators of Ethereum events. The third, and final direction could be to apply our proposal to different blockchain systems other than Ethereum, evaluating the performances in such contexts.

Bibliography

[1] Durga Prasad Acharya. *10 Blockchain Naming Service (BNS) Platforms to Get Your Blockchain Domain Name*. Accessed on 2022-11-21. URL: https://geekflare.com/finance/blockchain-naming-service-bns-platforms/.

[2] Guido Bertoni et al. "Keccak". In: *Advances in Cryptology – EUROCRYPT 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 313–314.

[3] bit2me. *Cos'è il gas in Ethereum?* Accessed on 2022-11-21. URL: https://academy.bit2me.com/it/que-es-gas-en-ethereum/.

[4] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[5] Vitalik Buterin. *Ethereum Whitepaper*. Accessed on 2022-11-21. URL: https://ethereum.org/en/whitepaper/.

[6] Jitendra Chittoda. *Mastering Blockchain Programming with Solidity: Write production-ready smart contracts for Ethereum blockchain with Solidity*. Packt Publishing Ltd, 2019.

[7] CoinGecko. *Cryptocurrency Prices by Market Cap*. Accessed on 2022-11-21. URL: https://www.coingecko.com.

[8] Ethereum. *Currency for our digital future*. Accessed on 2022-11-21. URL: https://ethereum.org/en/eth/.

[9] Ethereum. *Ethereum Gas and Fees*. Accessed on 2022-11-21. URL: https://ethereum.org/en/developers/docs/gas/.

[10] Ethereum. *Ethereum-powered tools and services*. Accessed on 2022-11-21. URL: https://ethereum.org/dapps/#what-are-dapps.

[11] Ethereum. *Introduction to Web3*. Accessed on 2022-11-24. URL: https://ethereum.org/it/web3/.

[12] Ethereum. *Non-fungible tokens (NFT)*. Accessed on 2022-11-21. URL: https://ethereum.org/en/nft/.

[13] Ethereum. *Smart Contract*. Accessed on 2022-11-24. URL: https://ethereum.org/en/developers/docs/smart-contracts/.

[14] Ethereum. *TRANSACTIONS*. Accessed on 2022-11-24. URL: https://ethereum.org/en/developers/docs/transactions/.

[15] FiatToCrypto. *Why Web 2.0 is failing us and how Web 3.0 will help*. Accessed on 2022-11-21. URL: https://blog.cryptostars.is/why-web-2-0-is-failing-us-and-how-web-3-0-will-help-82ce40a4d426.

[16] Helena Handschuh. "SHA-0, SHA-1, SHA-2 (Secure Hash Algorithm)". In: *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011, pp. 1190–1193.

[17] Damiano Di Francesco Maesa and Paolo Mori. "Blockchain 3.0 applications survey". In: *Journal of Parallel and Distributed Computing* 138 (2020), pp. 99–114.

[18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[19] Ralph C. Merkle. "Protocols for Public Key Cryptosystems". In: *Proceedings of the 1980 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 14-16, 1980*. IEEE Computer Society, 1980, pp. 122–134.

[20] Donald R Morrison. "PATRICIA—practical algorithm to retrieve information coded in alphanumeric". In: *Journal of the ACM (JACM)* 15.4 (1968), pp. 514–534.

[21] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. https://bitcoin.org/bitcoin.pdf. Accessed on 2022-11-21. 2008.

[22] naukri.com. *Structure of a Block in Blockchain*. Accessed on 2022-11-24. URL: https://www.naukri.com/learning/articles/structure-of-a-block-in-blockchain/.

[23] Ilker Özçelik et al. "An Overview of Cryptographic Accumulators". In: *CoRR* abs/2103.04330 (2021).

[24] Bart Preneel. "HMAC". In: *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011, pp. 559–560.

[25] William Pugh. "Skip lists: a probabilistic alternative to balanced trees". In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[26] S Joshua Swamidass and Pierre Baldi. "Mathematical correction for fingerprint similarity measures to improve chemical retrieval". In: *Journal of chemical information and modeling* 47.3 (2007), pp. 952–964.

[27] theastrologypage.com. *Che cos'è il web 1.0?* Accessed on 2022-11-24. URL: https://it.theastrologypage.com/web-1-0.

[28] VentureBeat. *The Web2 problem: How the power to create has gone astray.* Accessed on 2022-11-24. URL: https://venturebeat.com/security/the-web2-problem-how-the-power-to-create-has-gone-astray/.

[29] Venkatesha N. Vysya and Anjani Kumar. *Blockchain adoption for financial services.* Accessed on 2022-11-21. URL: https://www.infosys.com/industries/financial-services/white-papers/documents/blockchain-adoption-financial-services.pdf.

[30] web3js. *web3.eth Docs.* Accessed on 2022-11-24. URL: https://web3js.readthedocs.io/en/v1.2.11/web3-eth.html#getblock.

[31] Frode Langmoen Wiggs Civitillo and Baan Slavens. *Brewing a more traceable and sustainable beer industry with blockchain.* Accessed on 2022-11-21. URL: https://www.ibm.com/blogs/blockchain/2021/07/brewing-a-more-traceable-and-sustainable-beer-industry-with-blockchain/.

[32] Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger.* 2014.

[33] Cheng Xu, Ce Zhang, and Jianliang Xu. "vchain: Enabling verifiable boolean range queries over blockchain databases". In: *Proceedings of the 2019 international conference on management of data.* 2019, pp. 141–158.

[34] Shijie Zhang and Jong-Hyouk Lee. "Analysis of the main consensus protocols of blockchain". In: *ICT express* 6.2 (2020), pp. 93–97.