Siva – The IPFS Search Engine

by

Nawras Nazar Khudhur

February 20, 2020

A dissertation submitted to the Faculty of the Information Engineering Course of the Graduate School of Engineering of Hiroshima University in partial fulfillment of the requirements for the degree of Master of Engineering.

Declaration of Authorship

I, Nawras Nazar KHUDHUR (M182220), declare that this thesis titled, ”Siva – The IPFS Search Engine” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Nawras Nazar KHUDHUR

Date: 2020

Acknowledgments

I would like to thank my supervisor Professor S. Fujita, and Associate Professor H. Okamura, Associate Professor S. Kamei for their support and contributions, not only during the preparation of this thesis, but also throughout my studies at Hiroshima University.

I would like to thank my colleagues at Information Engineering Laboratory of Hiroshima University and all of my friends for their warm-hearted support.

Finally, I must express my very profound gratitude to my family members, relatives and to my wife Hori for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thanks again.

Abstract

Recently, InterPlanetary File System (IPFS) has attracted considerable attention as a method to overcome the weaknesses of the current Web such as the single point of failure and an arbitrary control by the government. Each file stored in IPFS is associated with a unique cryptographic hash (CID) as a logical address to enable users to acquire the content of the file without being aware of the network address. In this paper, we design a *decentralized search engine* for IPFS which quickly returns a list of CIDs associated with a given collection of keywords. A key idea of the proposed method is to store the association between keywords and CIDs in the Distributed Hash Table (DHT) used in IPFS, and to utilize a *result cache* to accelerate the processing of partially duplicated queries. In addition, we use a variant of Bloom filter to quickly check the availability of the result cache. The performance of the proposed method is evaluated by simulation.

The simulation results indicate that it takes 1.7 s on average to respond to a query in a network of 1000 and 2000 nodes reduced by 43%, 31% respectively when the result cache is added to the search engine. Further, the search engine scales with the network size.

Contents

Declaration of Authorship i

Acknowledgments ii

Abstract iii

List of Tables vi

List of Figures vii

1 Introduction 1

2 Related Work 4

3 Preliminaries 6
- 3.1 Kademlia DHT 6
- 3.2 IPFS 7

4 Proposed Method 9
- 4.1 Overview 9
- 4.2 Keyword Extraction 11
- 4.3 Result Caching 11
  - 4.3.1 Handling of Single-Keyword Queries 12
  - 4.3.2 Handling of Conjunctive Queries 13
- 4.4 Hash Filter on Cache Results 14

4.5 Extra Step to Reduce Response Time 16

5 Evaluation **18**

5.1 Simulation Parameters 18

5.2 Query Response Time 19

5.2.1 Single-Keyword Queries Evaluation 20

5.2.2 Conjunctive Queries Evaluation 21

5.3 Scalability 22

5.3.1 Single-Keyword Queries 23

5.3.2 Conjunctive Queries 24

6 Concluding Remarks **28**

Bibliography **29**

Achievements **31**

List of Tables

5.1 Single-Keyword Query Experiment 1 21
5.2 Single-Keyword Query Experiment 2 22

List of Figures

3.1 The way of generating identifiers in IPFS. 8
4.1 Search-Index. 10
4.2 The search-index generation. 12
4.3 Proposed layers. 15
4.4 Search process & cache generation flow. 17
5.1 Yahoo! Webscope dataset. 20
5.2 Average query response time. 23
5.3 Cache hit-rate. 24
5.4 Average single-keyword query response time - Scalability. 25
5.5 Average query response time - Scalability. 26
5.6 Average query response time - Sub-query cache. 27

Chapter 1

Introduction

The concept of InterPlanetary File System (IPFS) was proposed by Juan Benet in 2014 [3] to overcome the weaknesses of the current World Wide Web such as the single point of failure and an arbitrary control by the government/corporation policies. In fact, IPFS beat the access block and the censorship to the Turkish version of Wikipedia conducted by the Turkish government in April 2017 [14, 13]. A key technique used in IPFS is to associate each file with a unique cryptographic hash (called Content Identifier CID) and to use the resulting hash as an address in the file system instead of the conventional network address such as `https://www.whitehouse.gov/` to realize a content-based networking without causing access bottlenecks (as will be described later, since any node associated with the hash stores a list of content holders for the corresponding file, the file content could be acquired from those nodes in parallel without any centralized authority)

The above mechanism of IPFS implies that we can easily acquire the content of a file *if we know the hash of the file*. Such a hash can be easily calculated by applying a predetermined hash function to the file content. However, it provides no means to acquire the hash of *un-owned* file nevertheless it is exactly the task of file search. To overcome this issue, some developers outside the IPFS community developed a centralized search engine based on the Elasticsearch, called ipfs-search[4]. Elasticsearch is a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents [5]. Elasticsearch supports a

horizontal scaling; meaning, it can be accessed in the same way whether it is running on one node or in a cluster of for example 300 nodes, and it automatically adjusts the distribution of indexes and queries across the cluster according to the number of events processed per second. However, the centralized solution with ipfs-search does not escape from *the curse of concentration* similar to the conventional web; namely, it still has a single point of failure and is affected by an arbitrary control by the government and corporation policies.

The objective of the current paper is to explore the design of a *decentralized* search engine for IPFS. The idea is simple but powerful. We store the association between keywords and hash values (CIDs) in IPFS in a distributed manner, where keywords are automatically extracted from given file. We then enhance the efficiency of the search of CIDs matching queries, with the following techniques:

1. The result for queries is cached in IPFS so that the duplicated calculation is avoided; and

2. To quickly check the existence of cached result for a combination of keywords, it uses a variant of Bloom filter called Cuckoo hash filter [6].

Similar idea has been proposed by the same authors in the context of standard distributed hash tables (DHTs) [7, 2]. In this paper, when a peer in the IPFS network issues a query to search for some content, we draw on the DHT to respond to that query. Meanwhile, the returned query results are cached either by the direct neighbors of the query-issuer node or by the nodes closest to it depending on whether the query was single-keyword query or conjunctive query.

The performance of the proposed method is evaluated by experiments using Simulation. The results of evaluations indicate that using the proposed caching techniques with the DHT reduces the query response time significantly. In addition the search engine scales with the network size as the average query response time increases only by 87% while the network size scaled by 900%. The remainder of this paper is organized as follows. Chapter 2 overviews related work. Chapter 3 explains the preliminaries. Chapter 4 describes the details of the proposed method. Chapter 5

summarizes the results of simulations. Finally, Chapter 6 concludes the paper with future work.

Chapter 2

Related Work

Since the only work on IPFS is the elasticSearch based ipfs-search search engine. We review some of the papers that take into consideration similar goals of processing conjunctive queries in a structured P2P environment as our work.

In Vahdat *et al* [11] first suggested hash function can be used to create a distributed inverted index. The inverted index is updated with each new document publishing. The selected keywords of a document are hashed using some hash function then appropriate peers are contacted to update the index accordingly. To reduce the bandwidth usage during conjunctive queries, they suggested using and caching a bloom hash filter. It reduces the bandwidth usage by drawing on bloom filter generation for the matched documents of a keyword, then send the bloom filter to the peer responsible for the next keyword instead of sending the whole matched document list thus minimizing the used bandwidth. Our current work is considering a different approach of using the hash filters and processing the conjunctive queries.

Abere *et al* [12] try to reduce the traffic consumed by conjunctive queries by constructing an index for the result cache corresponding to the query load. That is, only frequently issued queries are cached. The index is stored on a DHT called Distributed Cache Table. The selection of a query to be cached depends on the importance of the query like whether it can be used to answer more queries. A peer tries to find the results in DCT at first, if no cache was available, the query message is broadcasted to all the peers in the network.

Zeng and Wang [17] introduce a routing and lookup protocol based on CAN (content addressable Network). To reduce the routing hops with far nodes, it constructs a long distant link using network size estimation. With this approach, the new protocol provides high flexibility in the neighbor selection that a new node can choose from. Hence, it can be used to set physically close nodes to become neighbors in the logical network. Their results show a reduction in lookup latency compared to the traditional CAN. But they have not provided any results of conjunctive query lookup.

Chapter 3

Preliminaries

3.1 Kademlia DHT

Kademlia is a distributed hash table that organizes the peers into a structured overlay network. Kademlia uses binary codes as the underlying ID space; namely, any entity including peers and files is assigned a binary string as the unique ID, where the distance between two strings is the difference of the corresponding bitwise *XOR* value (e.g., although 011 is at distance one from 010, 100 is at distance six from 010). In this regard, the distance is shortened between two keys when matching more prefix bits. The routing table of a Kademlia node consists of buckets where each bucket holds a maximum of k nodes called contacts. These buckets are known as k-buckets. The parameter k is a system-wide replication parameter to ensure data availability. The contacts information stored in the form of triples $\langle$IP address, UDP port, Node ID$\rangle$ with respect to the distance from the self node. More specifically, for any bucket i, $0 \leq i < B$, where B is the size in bits of the used ID, it is guaranteed that the distance to contacts is between 2^i and $2^{(i+1)}$

To find the k closest node to a key in Kademlia protocol, the initiator node first chooses α nodes it knows about from its closest k-bucket to create a shortlist for the search. It then sends a parallel, asynchronous find request to the α contacts in the shortlist. Each contacted node will then return k closest triples to the requested key. Upon receiving the intermediate lists, the shortlist is updated. It then contacts

another α nodes that are not queried yet. This operation is repeated iteratively until no more uncontacted nodes remain. Thus, closest k nodes obtained.

3.2 IPFS

IPFS consists of a number of autonomous peers connected by a Kademlia network [9]. In IPFS, each peer has its own public/secret keys and is assigned a unique NodeId which is calculated by applying SHA256 hash function to its public key and encoding it with Base58 encoding algorithm (see Figure 3.1 for illustration). Similarly, each file shared in IPFS is given a unique CID which is calculated by applying the same functions to the *content* of the file (note that SHA256 can be applied to any file including binary file). For example, the turkish wikipedia webpage from `https://tr.wikipedia.org` translated to /ipfs/QmT5NvUtoM5nWFfrQdVrFtvGfKFmG7AHE8P34isapyhCxX/ where the long string of characters is the CID of the index file of wikipedia.

Since IPFS implements Kademlia, the *index* of a file, which is a key-value pair to have CID of the file as the *key* and a list of content holders and their IP addresses as the *value*, is stored in a peer to have the closest NodeId with the CID. For example, for a key-value pair $\langle$10001, {{IP address, UDP port, 11001}, {IP address, UDP port, 11101}, ...}$\rangle$ and a list of candidate nodes 10110, 11110, 01111 the distance(CID, NodeId) for each NodeId is calculated by XORing each NodeId with the CID resulting in 7, 15, 30 respectively. Thus, the first node is chosen to be the holder of the key-value pair. With the above notions, the retrieval of a file with a designated CID proceeds as follows:

1. Identify a peer to have the index of the file through Kedemlia;
2. Acquire a list of content holders from the identified peer;
3. Acquire the file content from peers contained in the list; and
4. After receiving the requested file, cache the replica of file for a certain time and add itself to the list of content holders (the list is updated when the replica is

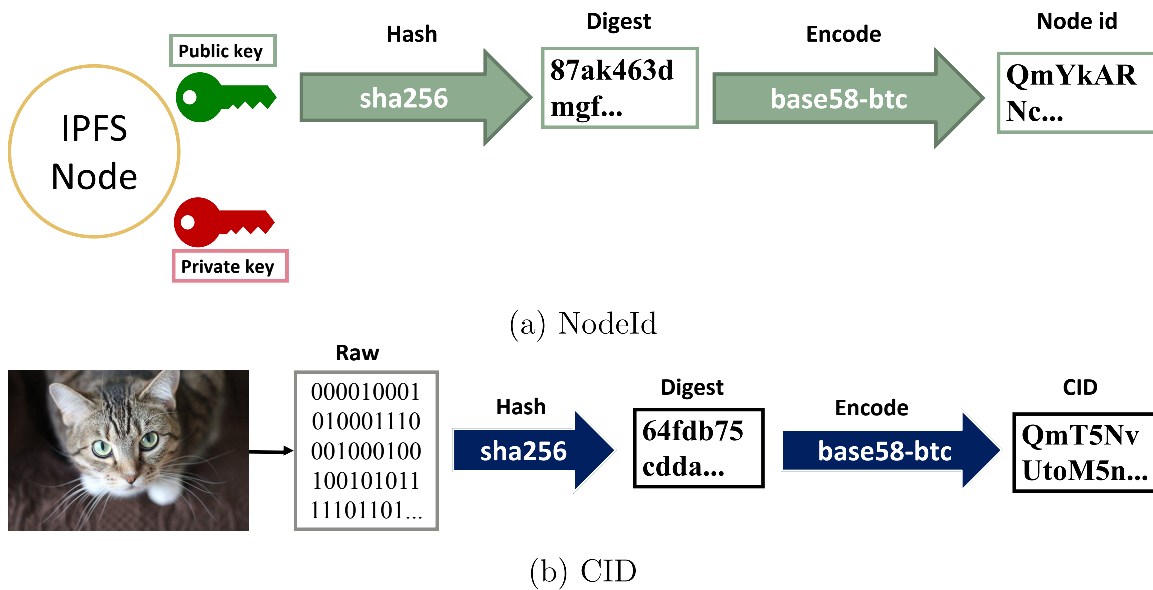

Figure 3.1: The way of generating identifiers in IPFS.

removed).

In terms of the system hierarchy, IPFS consists of three layers called Moving data, Defining data and Using data layers [8]. The first layer is called libp2p, and is divided into three sub-layers called network, routing and exchange sub-layers. The network sub-layer provides point-to-point connections between any two nodes within the IPFS network. The routing layer provides peer routing and content routing to find other nodes and requested data on the network respectively. The IPFS Block Exchange layer manages the transfer of the blocks of data among nodes. The second layer contains definition of IPNS, that is used as naming in IPFS network. The using data layer is where applications of IPFS are defined which depends on the mentioned layers. Our proposal falls on the top layer of IPFS that is using data layer.

Chapter 4

Proposed Method

4.1 Overview

The association between keywords and CIDs can be represented as *key-value pairs* in which the cryptographic hash of keyword is used as the *key* and the list of CIDs associated with the keyword is used as the *value*. See Figure 4.1 for illustration. In the following, we call this simple data structure the **search-index**. Note that this index can be naturally stored in the DHT of IPFS in a distributed manner. In this section, we propose a method to realize an efficient processing of single-keyword/conjunctive queries issued by the users in IPFS.

A single-keyword query is a query containing only one word which requests a set of CIDs associated with it, such query is processed easily by finding the closest nodes to the hash of the keyword then request the responsible node for the list of associated CIDs.

A *conjunctive* query is designated by a set of keywords which requests a set of CIDs associated with *all* keywords contained in it. In the following, we represent a conjunctive query as $\{key_1, key_2, \ldots, key_w\}$, where $w(\geq 2)$ is the number of keywords in the query. With the notion of search-index, such a conjunctive query can be naturally processed by taking an intersection of the sets of CIDs for each key_i after acquiring them from the corresponding peers through DHT. However, such a naive scheme takes a long computation time since it is dominated by the longest response

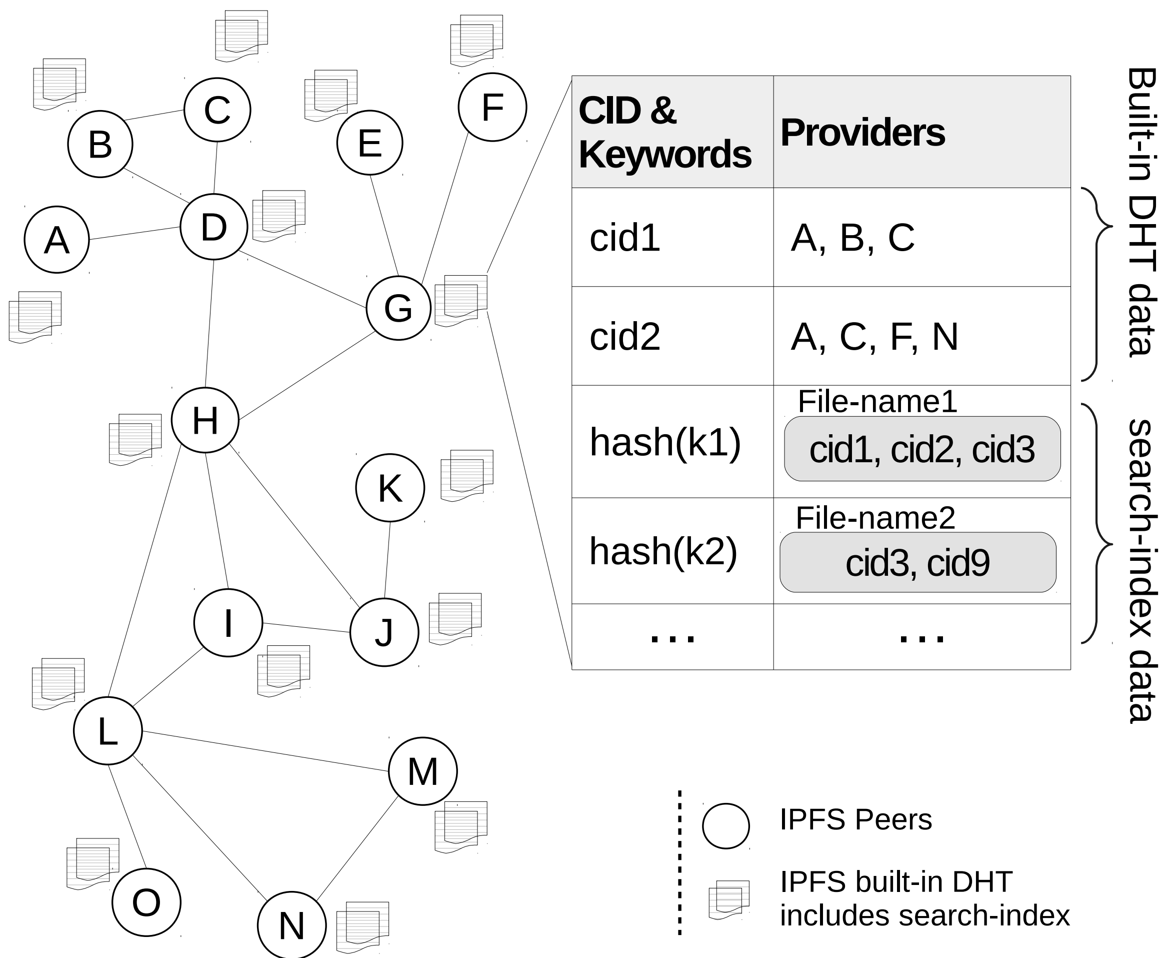

Figure 4.1: Search-Index.

time from corresponding peers which depends on the number of CIDs associated with the keywords and the network environment.

The proposed scheme improves the efficiency of the naive scheme with the notion of result caching and a variant of Bloom filter. More concretely, the proposed scheme consists of the following four components:

- IPFS Layer
  Stores the association between CIDs and lists of content holders.

- Search-Index Layer
  Stores the association between keywords and lists of CIDs.

- Result Cache Layer
  Stores the association between keywords sets and lists of calculated result for the keyword set. Used to avoid the recalculation of the repeated conjunctive queries.

- A Variant of Bloom Filter
  Quickly judges whether the full or partial result for a given keyword set is stored in the Result Cache layer.

4.2 Keyword Extraction

In the proposed method, the search-index is maintained in such a way that CID of a file is stored in a peer to have the closest NodeId to the hash of keyword for each keyword extracted from the file. Keyword extraction from a given file proceeds as follows (refer to Figure 4.2 for illustration):

1. At first, it extracts *metadata* such as author, created date, content of a page, and so on by using a metadata extractor such as Apache-Tika [1].

2. In case of text file, it calculates the TF-IDF weight of all words contained in the file and selects words scored higher than the mean value as the representative keywords of the file.

Finally, the CID of the processed file with its list of representative keywords and metadata are stored in the search-index.

Although the extracted words are certainly relevant with the given file, the *appropriateness* of them should be refined through interaction with the users. Concrete design of such an interaction is left as a future work.

4.3 Result Caching

Result caching is the process of storing the calculated result of a query and reuse it to reply future queries in order to avoid the re-calculation and reply faster. These

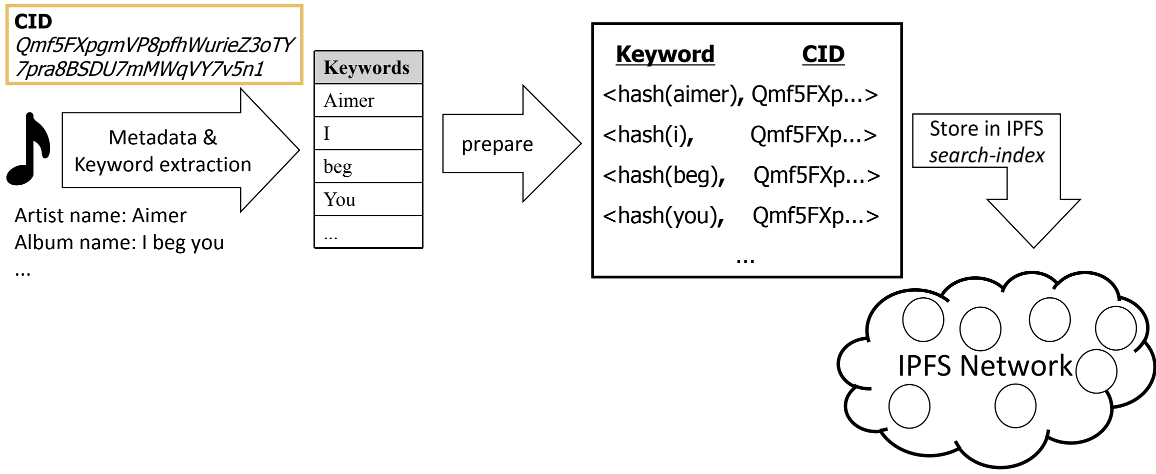

Figure 4.2: The search-index generation.

queries can be either partially similar or exactly the same as the stored query. Since search engines have a strong locality [16], that is, many users repeat the same or similar queries, result cache can be used to take advantage of it.

We take different a approach for processing the single-keyword queries, and conjunctive queries as discussed below.

4.3.1 Handling of Single-Keyword Queries

For a query consisting of a single-keyword, the set of CIDs matching the query has already been stored by the peers closest to the hash of the keyword in search-index (as discussed before). To accelerate frequent query reply, the query result, which is a set of CIDs matching the query, is cached by k direct neighbor nodes of the issuer node to have the closest NodeId to the hash of the query where k is a Kademlias system-wide replication parameter.

When a query is issued, the issuer node does not have any knowledge about holders of the cached results, thus, it will not look for any cache. Instead, it forwards the query to other nodes to find the responsible node holding the actual requested value. In the meantime, the intermediate nodes can reply the query directly and interrupt the search if they happened to have the cached result.

4.3.2 Handling of Conjunctive Queries

Result caching can reduce the number of sub-queries needed by any conjunctive query. For a conjunctive query, the result for the query, which is a set of CIDs matching the query, is cached by k nodes to have the closest NodeId to the hash of the conjunctive query. The query issuer node caches the results for the past queries locally so that the same query answered right away by itself.

Since the cache storage is limited and the query result is always getting cached once it is issued, the *Least Recently Used (LRU)* cache replacement algorithm is applied to avoid memory overflow. With LRU, the new cached results are pushed into the result cache storage by removing the least recently used cache record.

Under the above caching strategy, the requester of a new conjunctive query $\{key_1, key_2, \ldots, key_k\}$ conducts the query processing in the following manner: It first searches the network for peers holding the fully cached results. For example, for a query $Q = \{key_1, key_2, key_3\}$, k closest nodes for $hash(\{key_1, key_2, key_3\})$ are determined and queried. If no result is found, all of the possible combinations ($>$1 keyword and $<$whole query) of the keywords in query Q such as $\{key_1, key_2\}, \{key_1, key_3\}, \ldots$ are calculated. Then, the sub-queries $subQ_1, subQ_2, \ldots$ are issued for each combination to query the result cache layer for partially cached results. The final result is calculated by intersecting results from sub-queries such that $subQ_1 \cap subQ_2 \cap \ldots = Q$. When sub-queries are not enough to produce query Q, the search-index is used to find results for the remaining keywords by issuing the necessary single-keyword queries.

The above steps are highly time consuming when the results are not cached yet because it is going to retry cache check for all possible combinations of the conjunctive query. Finally, as the last resolve, it starts creating sub-queries for each keyword separately to query the search-index. We approach this issue by introducing another layer as discussed in the following section.

4.4 Hash Filter on Cache Results

To solve the issue mentioned in the previous section, a hash filter layer is added. Figure 4.3 illustrates the proposed extra layers to the search engine. Hash filters are used as membership testing against a set of existing data. In this case, the hash filter layer can quickly judge about the existence of the requested result cache or its combinations in the network; therefore, bypassing the unsuccessful cache searches. That means, instead of going through the distributed cache directly. Hash filters can respond by ensuring that the query is not in the cached result set.

Because the cache result is not static, which means it will be updated or deleted after some period of time, deletion feature of the hash filter is needed. Thus, we propose using a variant of bloom filter called Cuckoo filter which is a new data structure, represented in a paper by Fan *et al* in 2014 [6]. Cuckoo filters improve upon the scheme of the bloom filter by giving deletion, restricted counting, and bounded false positive probability, whereas still keeping up a similar space complexity.

In this proposal, every peer holds the same version of the hash filter and uses it to decide whether to conduct the search keyword by keyword then combine the results or retrieve results from the cache layer for the conjunctive query directly. It will also give the ability to find out about partially cached results that can serve the requested query best (intersecting fewer subsets to get the query result). For the same simple example $Q = \{key_1, key_2, key_3\}$, at first, the hash filter is questioned about the existence of cache results for the whole query Q. If the answer is negative, the hash filter is asked about the existence of cache results for the generated combinations/sub-queries starting from largest combinations. Finally, the new query is formalized with reference to the hash filter results. For example, assuming only $\{key_1, key_2\}$ is cached according to the hash filter layer. Then, two sub-queries $\{key_1, key_2\}$ and $\{key_3\}$ are created and issued simultaneously.

To sum up, with the combination of the above mentioned techniques, the proposed search engine works as follows (for the illustration refer to Figure 4.4):

1. User x makes a query search for some query Q.

2. If the query is single-keyword query, the search processeds normally by finding

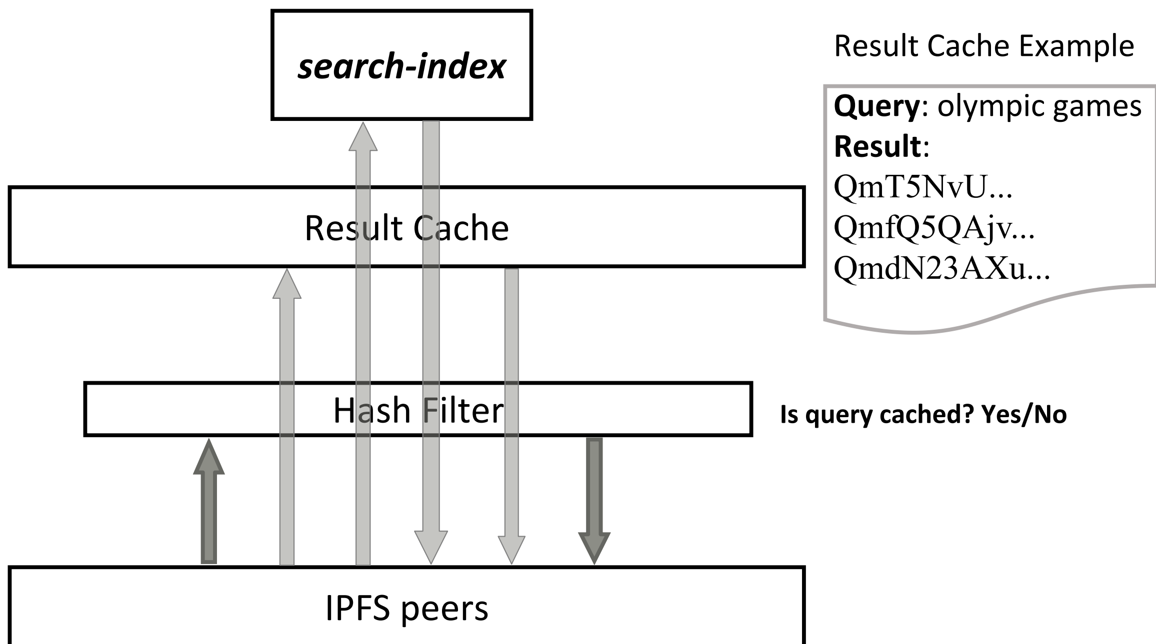

Figure 4.3: Proposed layers.

the closest nodes to the hash of the query. In this case, the issuer node depends on the intermediate nodes to get a cache hit and return results faster.

3. Otherwise, the cuckoo hash filter is used to find out about the existence of fully or partially cached results for the conjunctive query.

4. If the query result was fully cached, the result cache layer is searched by finding the responsible peers holding the cache.

5. Otherwise, hash filter layer is asked about partially cached results, and the final result is calculated by issuing the extra needed queries if partially cached results were not enough.

6. If the query Q was not cached fully nor partially, it proceeds to the keyword by keyword search on the search-index, then the calculated result is cached on the closest nodes.

4.5 Extra Step to Reduce Response Time

Since we expect the cache effectiveness reduction when the network grows very large, which means the query response time reduces gradually. We suggest mixing the approach of handling single-keyword queries with conjunctive queries. We propose caching the intermediate results of the conjunctive queries by considering them as the seperate single-keyword queries. More specifically, The search engine caches the results of single-keyword sub-queries. The responsible nodes to hold this cache, are k direct closest (to the key) neighbors found in the issuers routing table. In this case, the hash filter does not update and the search engine depends on intermediate nodes to get a cache hit. Since Kademlia tends to keep the most probably available nodes in the routing table, this strategy considers the availability of the nodes automatically without extra calculations.

With this approach, when a conjunctive query divided into sub-queries of single-keyword, the intermediate node which get the sub-query request, at first searches its local cache, if the cache is found, it interrupts the search by returning the result directly. Otherwise, it proceeds normally by returning the k closest nodes it knows about.

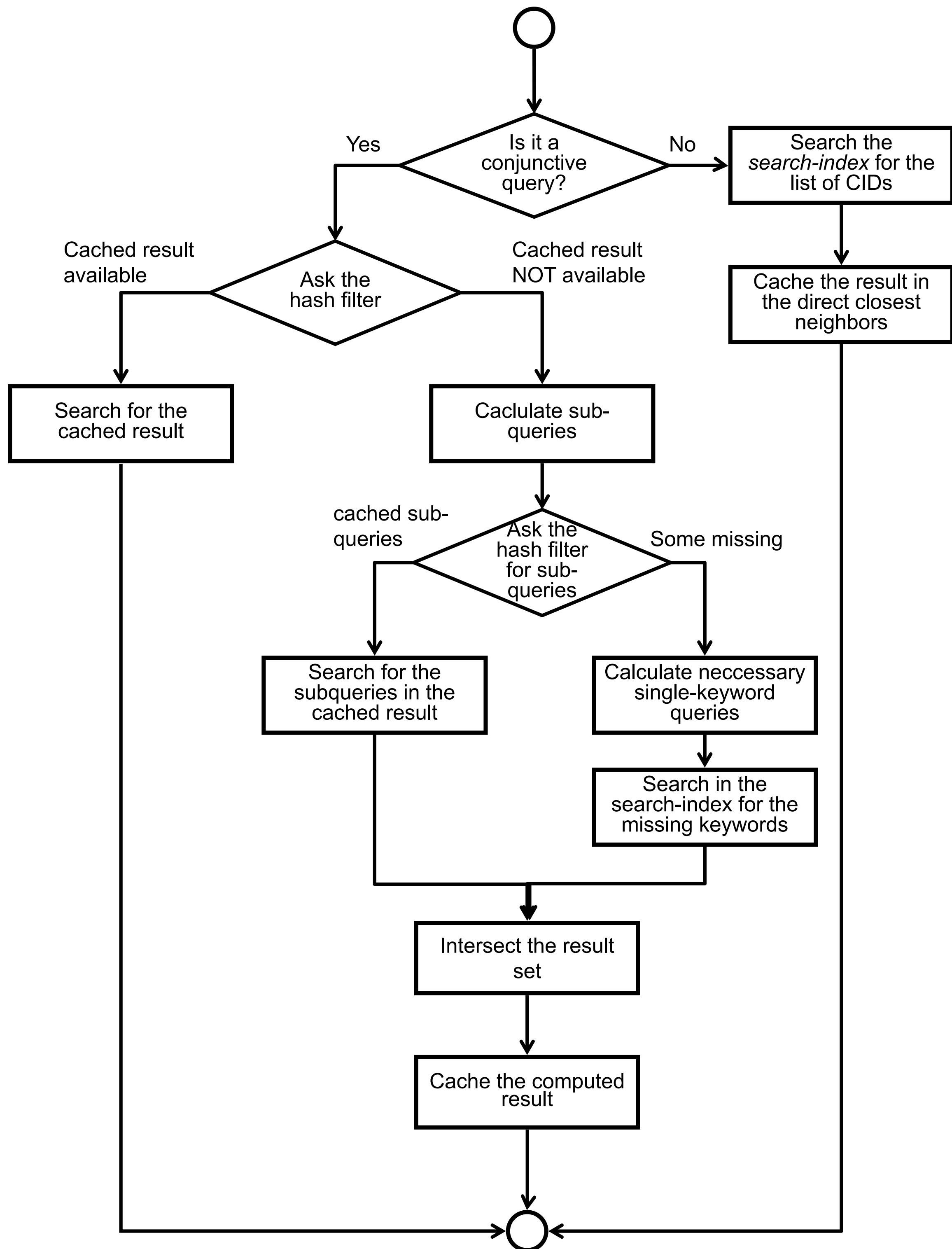

Figure 4.4: Search process & cache generation flow.

Chapter 5

Evaluation

To evaluate the performance of the proposed method, we simulate a P2P environment using Peersim simulator [10]. The objectives of these experiments are to show the result caching effectiveness in term of query response time and the scalability of search engine with respect to the network size.

5.1 Simulation Parameters

In the simulated environment the nodes are organized using Kademlia DHT similar to the actual IPFS network. The used hash function to generate identifiers is SHA256 same as used in IPFS. The Kademlia tuning parameters k is 20, α is 3 and B is 256 bits. The P2P environment is set to be static, which means the state of peers does not change during the simulation. The nodes are in a random topology of the degree of 5. The used dataset is from Yahoo! Search Engine provided by Yahoo! Webscope [15]. The dataset contains 1,500,000 queries in total with 1,198 unique queries. Figure 5.1 shows the distribution of the dataset that follows a power law distribution with some degree of α.

We have calculated the CID for each unique document in the dataset and built the $\langle$keyword, CID$\rangle$ association for the dataset by combining the CIDs to the hash of the corresponding queries. The produced collection is then stored in the search-index which is distributed according to the used Kademlia DHT. Hence, each peer in the

network is responsible to maintain a portion of the search-index that is closer to their NodeId.

Since we are only interested in showing the change in the *ratio* of query response time compared to the conventional approach, a uniform link latency (100 ms) is applied between peers in the network. We set the same cache size for all peers in the network and specified by a number of units, where each unit represents a storage for one query result. The random queries were drawn according to the distribution of the dataset. Any peer can issue the next query randomly with equal probability. Once a query is issued, the obtained results are cached by the k nodes having closest NodeId to the hash of the conjunctive query. As for single-keyword queries, The nodes responsible for caching query results are direct closest neighbors of the issuer node whose NodeId is closest to the hash of the query text. Finally, for the cache replacement, we have implemented the LRU cache replacement scheme in this simulation. The hash filter layer is implemented using cuckoo hash filter as it was proposed and we assumed that all peers have the same version of hash filter at any point of time.

5.2 Query Response Time

In this paper, we define the response time for a query to be the time which a query is issued to which the result is returned to the issuer node. With this definition $query\ response\ time = \sum_{i=1}^{n} 2 * (latency)$ where n is the number of iterations needed until the corresponding peer is reached. Since the conjunctive queries may divide into sub-queries, the response time for conjunctive queries is the maximum response time of the issued sub-queries.

To evaluate the effect of network size on the performance of the proposed solution, we have simulated two different networks of 1000 nodes and 2000 nodes with the same basic parameters. To show the sufficient cache size for the best query response time, we repeat the simulation comparing different cache size parameters.

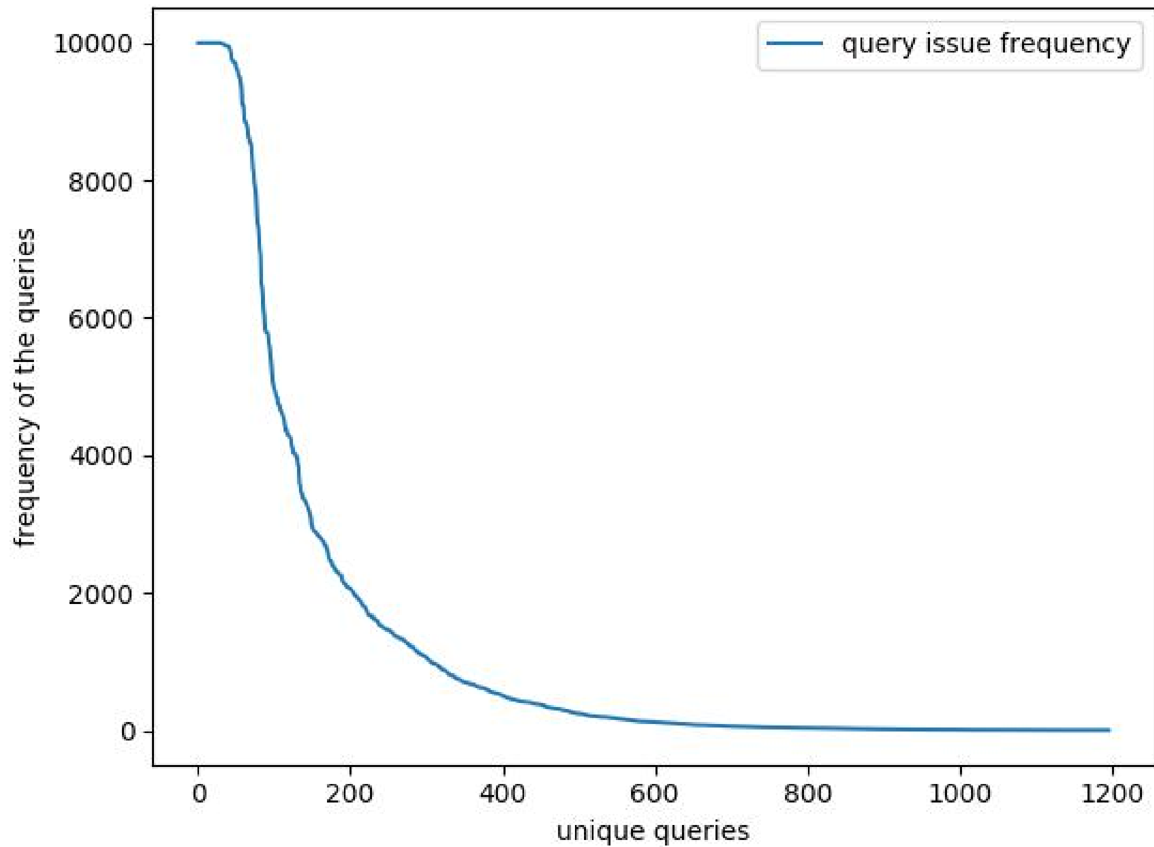

Figure 5.1: Yahoo! Webscope dataset.

5.2.1 Single-Keyword Queries Evaluation

In the single-keyword query experiment, the nodes are unaware whether a particular query result is cached or not. To respond to a query using the cache, each intermediate routing node checks its cache upon receiving query messages along the way until the actual responsible node that holds the stored result is reached. If the cached result were found on any intermediate node, it will send back the result to the issuer node. On the other hand, the issuer node will stop listening to further responses about the issued query.

We issued 54000 single-keyword queries, and cache size starts from 10 unit to 100 units doubling each time for two different network size 1000 nodes and 2000 nodes. Each unit of the cache represents storage for one query result.

Table 5.1: Single-Keyword Query Experiment 1

Cache Size	Avg Query Response Time	Avg # of hops	Cache Hit %
0	1878	21.59	0
10	1282	14.82	36
20	948	11.34	55
40	619	7.67	74
80	374	5.25	87
100	321	4.89	89

Network Size 1000 nodes, issued queries 54,000

The proposed approach to handle single-keyword queries showed a significant reduction in the average query response time as summarized in tables 5.1, 5.2. the average query response time reduced by 83% compared to no usage of cache. We can also obaerve that the average number of hops for each query is reduced considerably from 21.59 hops when result is not cached to 4.89 hops when 100 unit of cache is added, which is about 77% reduction in the number of hops needed by a query Additionally, with larger network size, the average query response time reduced by 81% when result cache of 100 units is added to the peers. Again, the number of hops reduced noticeably when cache layer is added. From 22.30 hops when result is not cached to 6.16 hops when 100 unit of cache is added, which is about 72%.

In both cases, we can see the cache hit-rate also increases along with the increasing of the cache size indicating that the peers take advantage of the used result cache when processing search queries.

5.2.2 Conjunctive Queries Evaluation

In the conjunctive query experiment, The cache size starts from 100 units to 500 units adding 100 unit to each simulation. The simulation issued 216,000 conjunctive queries (of two or three keywords with the same probability). We compare the results with the conventional approach (0 unit cache size). Figure 5.2 shows the distribution of

Table 5.2: Single-Keyword Query Experiment 2

Cache Size	Avg Query Response Time	Avg # of hops	Cache Hit %
0	1963	22.30	0
10	1297	14.98	38
20	967	11.59	56
40	647	8.06	74
80	412	6.24	86
100	365	6.16	88

Network Size 2000 nodes, issued queries 54,000

the average query response time corresponding to the different cache size parameters for the two networks of 1000 and 2000 nodes. We can observe from the figure that adding the proposed cache layer and the hash filter layer to the search engine reduces the average query response time significantly. The average response time reduced by about 10% for both networks of 1000 and 2000 nodes after adding a 100 units of cache to the nodes in the network. The reduction rate increases when cache size increases. Using 500 units of cache, reduces response time by 43% in the network of 1000 nodes, and by 31% in the network of 2000 nodes. In Figure 5.3, the cache hit-rate is showed in the network of 1000 nodes. The increasing in the cache size gives more opportunity to have a cache hit for the succeeding queries. Adding 100 units of cache helped 12% of queries get cache hit. Increasing the cache size to 500 units, the search engine replied 46% of the queries from cache. We can also find that the most effective cache size for the simulated network size is between 200 units to 400 units as the highest reduction rate falls between these two cache sizes as 29% raising to 44% of the query replies are from cache.

5.3 Scalability

The only work toward IPFS search engine (the mentioned ipfs-search [4]) can be assumed easily scalable as it is based on Elasticsearch. Elasticsearch databases can scale

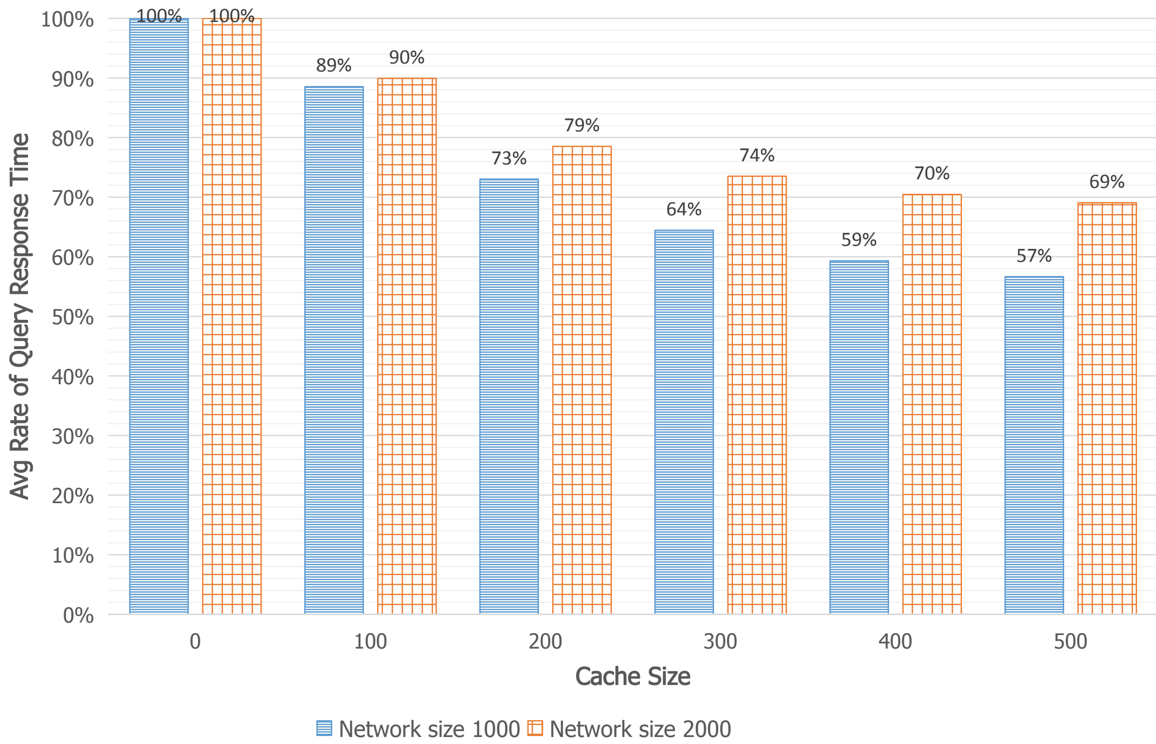

Figure 5.2: Average query response time.

to hundreds of nodes and respond to thousands of users. But due to the need of high performance servers to operate, the only major concern in scaling the Elasticsearch can be the financial cost.

Thus, we have evaluated our P2P solution for the scalability of the network size.

5.3.1 Single-Keyword Queries

The proposed scheme is tested for the scalability of the network size issuing 54000 queries and letting each node have 100 unit of cache. Figure 5.5 shows the average time a query takes to return the results while growing the network size starting from 500 to 5000 nodes adding 500 nodes each time. We notice a linear growth of the needed time for a query with increasing the network size, but not as fast as the growth of the network size. The average time starts from 365ms raising only by 65% while the network size increased by 900%.

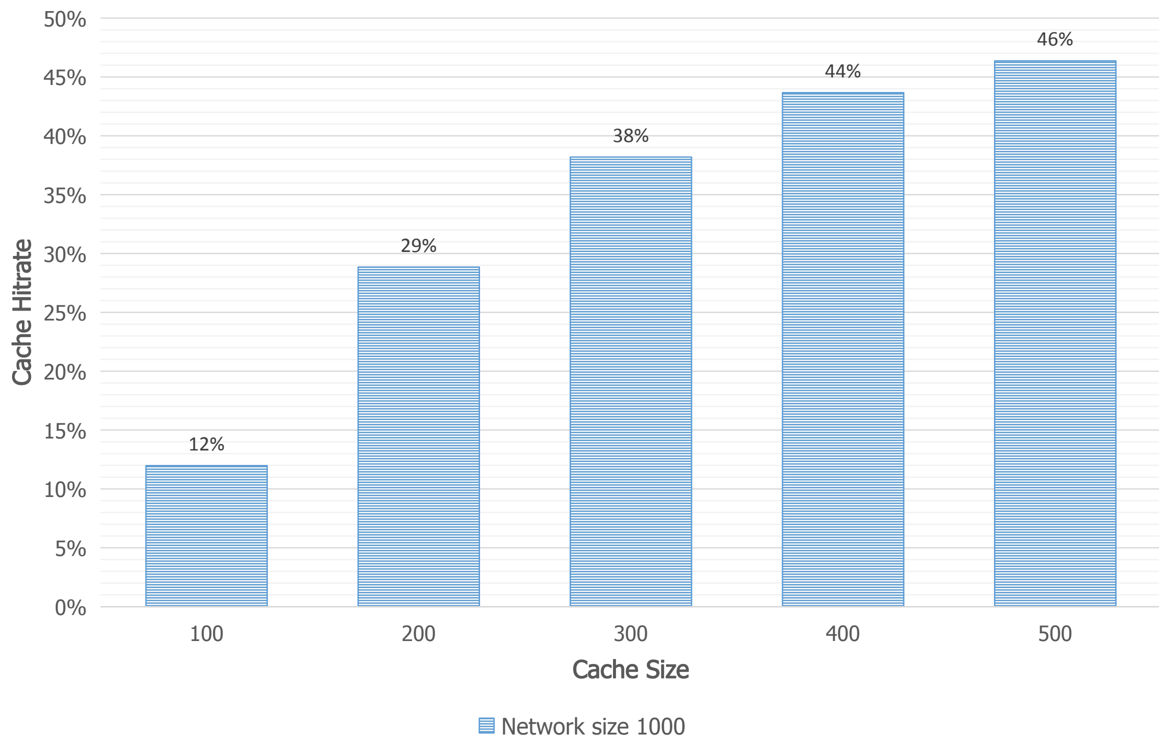

Figure 5.3: Cache hit-rate.

5.3.2 Conjunctive Queries

Next, We start the evaluation of this approach for the conjunctive queries in a scaled network up to 5000 nodes then compare the results with using only conjunctive query cache.

We set each node to have a cache size of 500 units and issued 216,000 queries. Figure 5.5 shows the average time a query takes to return the results. We increased the network size starting from 500 to 5000 nodes adding 500 nodes each time. The squared line shows the scalability result when the conjunctive query result is cached but not the intermediate results. We notice a linear growth of the needed time for a query with increasing the network size, but not as fast as the growth of the network. The average query response time starts from 847 ms raising only by 87% to 1581 ms while the network size is increased by 900%. When comparing to the second approach that caches the result of sub-queries in the direct closest neighbors of the issuer node.

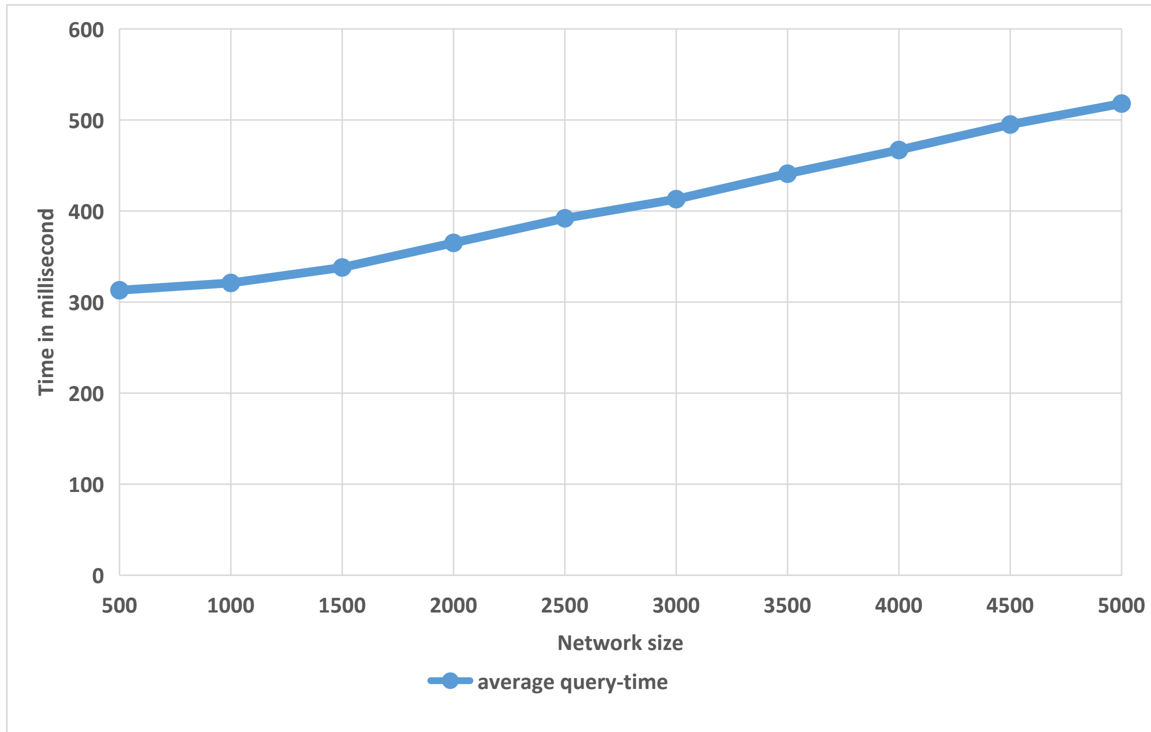

Figure 5.4: Average single-keyword query response time - Scalability.

We can read from its trend line, even for a growing network the query response time increases very slowly and steadily by 36% for the same 900% growth in the network.

We compared the effectiveness of our approach with random node selection in the network of 5000 nodes. Figure 5.6 summarizes the results of the comparison. In the large network of 5000 nodes, the direct closest neighbors cache performs 40% better than the random selection of nodes. We can also confirm that caching the single-keyword sub-queries in general performs 40% and 58% better than just caching the results of conjunctive queries.

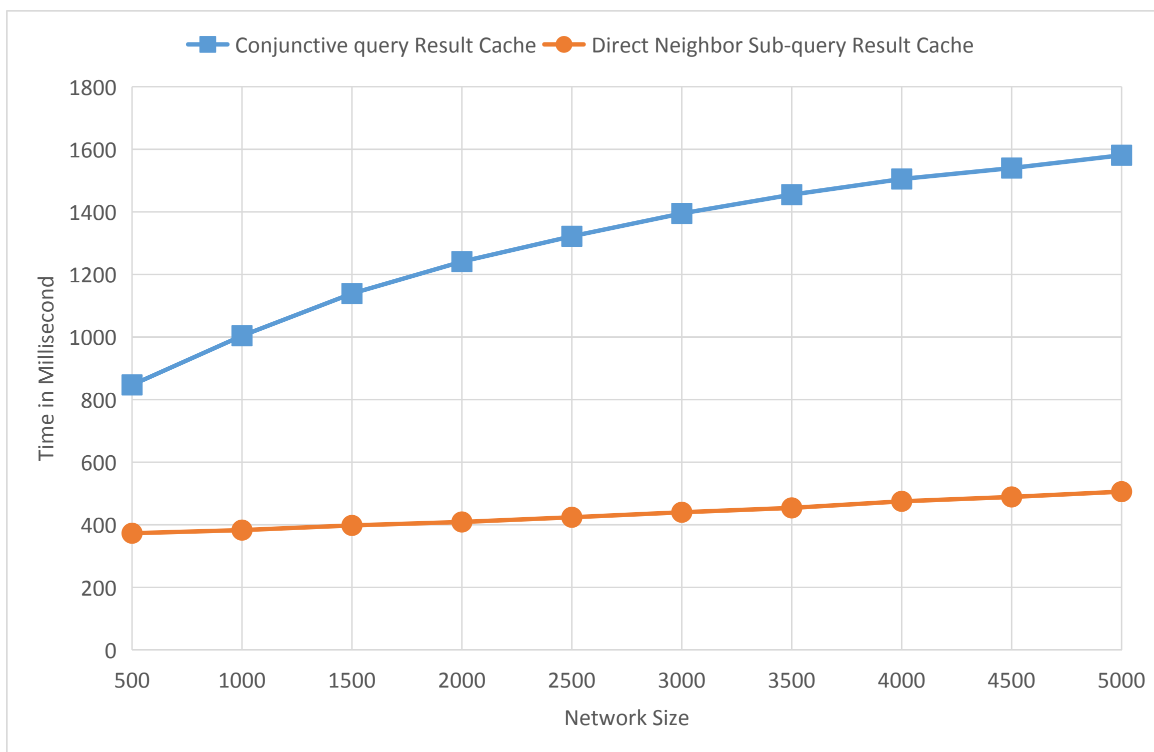

Figure 5.5: Average query response time - Scalability.

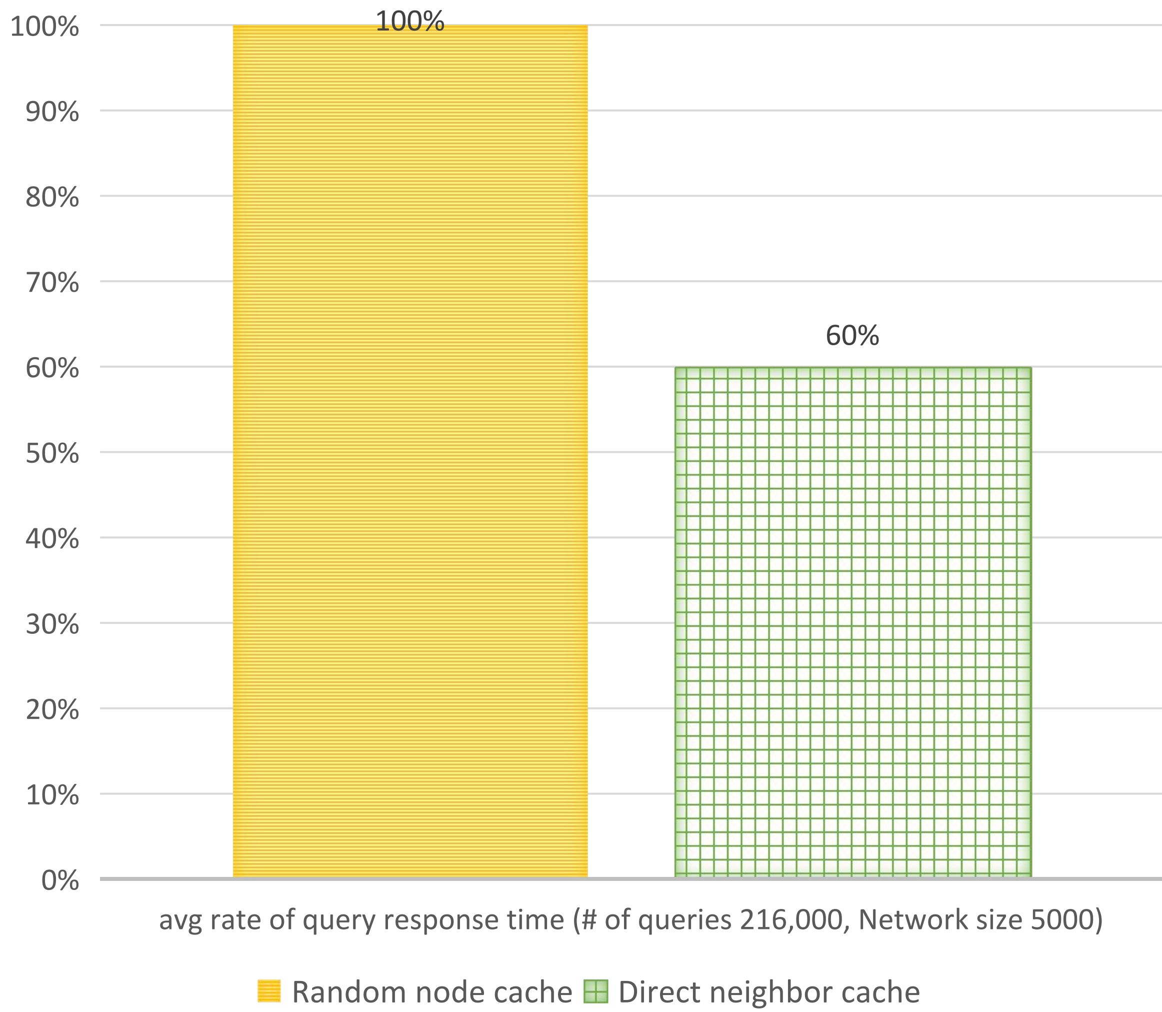

Figure 5.6: Average query response time - Sub-query cache.

Chapter 6

Concluding Remarks

This paper proposes a design of a scalable decentralized search engine for IPFS that can respond to the queries very efficiently. The DHT is used to hold the auto-generated indexed data and a combination of result cache layer with the hash filter layer used to process the conjunctive queries. The result of evaluations indicates that the proposed layer combo reduces the average query response time in networks of 1000 and 2000 nodes by 43%, 31% respectively. In addition, the average query response time increases by 87% when the network size is increased by 900%. By adding the sub-query result cache on direct closest neighbors, we achieved even better scalability. The sub-query result cache increased only by 36% for the 900% expansion of the network.

We leave the following issues as future work: 1) to evaluate the overall performance of the proposed method in the actual IPFS environment; 2) to develop a concrete design of ranking and correcting the indexed data; and 3) to propose a method to return the results depending on some similarity degree.

Bibliography

[1] Apache. Apache tika - a content analysis toolkit. https://tika.apache.org/, 2018. Accessed: October 12, 2018.

[2] T. Ariyoshi and S. Fujita. Efficient processing of conjunctive queries in p2p dhts using bloom filter. In *International Symposium on Parallel and Distributed Processing with Applications*, pages 458–464, Sept 2010.

[3] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[4] Mathijs de Bruin. Search engine for the interplanetary filesystem. https://github.com/ipfs-search/ipfs-search, 2018. Accessed: October 10, 2018.

[5] Elasticsearch. Elasticsearch. https://www.elastic.co/products/elasticsearch, 2018. Accessed: October 10, 2018.

[6] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, New York, NY, USA, 2014. ACM.

[7] Koji Kobatake, Shigeaki Tagashira, and Satoshi Fujita. A new caching technique to support conjunctive queries in p2p dht. *IEICE - Trans. Inf. Syst.*, E91-D(4):1023–1031, April 2008.

[8] Libp2p. Libp2p. https://github.com/libp2p/libp2p, 2018. Accessed: October 23, 2018.

[9] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[10] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[11] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 21–40, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[12] Gleb Skobeltsyn and Karl Aberer. Distributed cache table: Efficient query-driven processing of multi-term queries in p2p networks. In *Proceedings of the International Workshop on Information Retrieval in Peer-to-Peer Networks*, P2PIR '06, page 33–40, New York, NY, USA, 2006. Association for Computing Machinery.

[13] IPFS Team. Uncensorable wikipedia on ipfs. `https://blog.ipfs.io/24-uncensorable-wikipedia/`, 2017. Accessed: December 20, 2018.

[14] Turkeyblocks. Wikipedia blocked in turkey. `https://turkeyblocks.org/2017/04/29/wikipedia-blocked-turkey/`, 2018. Accessed: November 26, 2018.

[15] Yahoo. Yahoo! webscope dataset anonymized yahoo! search logs with relevance judgments version 1.0. `http://labs.yahoo.com/Academic_Relations`, 2018.

[16] Yinglian Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1238–1247 vol.3, June 2002.

[17] B. Zeng and R. Wang. A novel lookup and routing protocol based on can for structured p2p network. In *2016 First IEEE International Conference on Computer Communication and the Internet (ICCCI)*, pages 6–9, Oct 2016.

Achievements

1. Nawras Khudhur and Satoshi Fujita, "Siva the ipfs search engine." In *Proceedings of CANDAR'19* pages 150–156. **Selected as a CANDAR 2019 Outstanding Paper.**

2. Nawras Khudhur and Satoshi Fujita. "Siva – The IPFS Search Engine." In *Bulletin of Networking, Computing, Systems, and Software* 8.2 (2019) pages 98-103.