Article

Keyword Search in Decentralized Storage Systems

Liyan Zhu, Chuqiao Xiao and Xueqing Gong *

School of Software Engineering, East China Normal University, Shanghai 200062, China; 51184501099@stu.ecnu.edu.cn (L.Z.); 52184501011@stu.ecnu.edu.cn (C.X.) * Correspondence: xqgong@sei.ecnu.edu.cn

Received: 20 October 2020; Accepted: 30 November 2020; Published: 2 December 2020

Abstract: The emerging decentralized storage systems (DSSs), such as InterPlanetary File System (IPFS), Storj, and Sia, provide people with a new storage model. Instead of being centrally managed, the data are sliced up and distributed across the nodes of the network. Furthermore, each data object is uniquely identified by a cryptographic hash (ObjectId) and can only be retrieved by ObjectId. Compared with the search functions provided by the existing centralized storage systems, the application scenarios of the DSSs are subject to certain restrictions. In this paper, we first apply decentralized B+Tree and HashMap to the DSSs to provide keyword search. Both indexes are kept in blocks. Since these blocks may be scattered on multiple nodes, we ensure that all operations involve as few blocks as possible to reduce network cost and response time. In addition, the version control and version merging algorithms are designed to effectively organize the indexes and facilitate data integration. The experimental results prove that our indexes have excellent availability and scalability.

Keywords: decentralized storage systems; keyword search; decentralized B+Tree; decentralized HashMap

1. Introduction

With the rapid development of internet technology, centralized storage has become an important business model in our daily life. Centralized storage not only provides a variety of storage services for both individuals and businesses but also supports different kinds of queries, thus meeting the needs of users. However, centralized storage systems depend on a trusted third party, which inevitably inherits the single point of failure drawback. Even if centralized storage systems are backed up for data availability, they still suffer from certain factors of force Majeure (such as political censorship), which can cause users to be unable to access their own data.

From the above point of view, data storage requires a more secure and free environment. The emerging DSSs, such as InterPlanetary File System (IPFS) [1], Storj [2], and Sia [3], can provide people with a new storage model. They are built on a peer-to-peer (p2p) network, so there is no need to rely on third-party platforms. In these systems, data are not managed by a central node but divided into blocks and distributed through the network. A data object can be accessed as long as it exists on any node. Each node in the network can share its free disk space, thus reducing the cost of decentralized storage. Users do not have to worry that they will not be able to access their own data because DSSs can be combined with blockchain to ensure data availability [4,5].

One of the key reasons why the traditional centralized storage systems can be applied to various fields is that they provide rich query services, which is exactly the defect of DSSs. In the DSSs, each node or data object is assigned a unique identifier (NodeId or ObjectId) by a cryptographic hash function. Analogously, each block has its BlockId, which is obtained by encrypting its stored content. Therefore, all data can only be accessed by hash values, which greatly limits the application of DSSs. In addition, centralized storage systems are not subject to encryption and they use different data...
structures to adapt to different scenarios. Inspired by this point, this paper aims to design a global index based on block storage to provide keyword search. However, there are several challenges. One is that there are no central nodes in DSSs, and each node manages its own data independently. The index should be directly accessed and updated by all nodes. Another is that the DSSs adopt the hash-based addressing and any modification to a block will result in a new block. If the index occupies multiple blocks, it must be effectively organized so that all operations can involve as few blocks as possible.

Driven by these demands, this paper chooses B+Tree and HashMap as the main index structures. These indexes are chosen for a few reasons, which are as follows. From the index type perspective, B+Tree and HashMap denote representatives of the tree and hash structures, respectively. In addition, these indexes have lower height and a reasonable number of branches, which is conducive to query and update. Finally, each of their nodes can store partial information, which is suitable for block storage.

The main contributions of our paper can be summarized as follows:

1. A query layer for DSSs is designed to support rich queries.
2. Well-known B+Tree and HashMap are used in combination with inverted index respectively as the global structures of the decentralized index, and add, update, and query operations are supported.
3. Version control is designed to manage multiple versions of the index, and index merging algorithms are proposed for different index structures to promote data integration.
4. The experiment is performed in a real IPFS cluster and the results show that the two indexes provide excellent availability and scalability.

The rest of the paper is organized as follows: In Section 2, related work and preliminaries are discussed. The two indexes are described in detail in Section 3. The version control is presented in Section 4. The experimental results are provided in Section 5. Lastly, the conclusions are drawn in Section 6.

2. Preliminaries and Related Work

2.1. Decentralized Storage Systems

Figure 1 shows an abstract decentralized storage architecture, including the network layer, local storage, block layer, and object layer. The network layer uses various configurable underlying network protocols to manage connections between nodes. This layer represents the cornerstone of decentralized systems, and avoids the problem of a single point of failure. Nodes in this layer have equal rights and jointly build a fully autonomous environment. Moreover, distributed hash table (DHT) [6,7] is also implemented in this layer. The discovery and connection of nodes and the transmission and positioning of data are all implemented with DHT. In general, a tuple <ObjectId or BlockId, [NodeId1, NodeId2, ...]> is stored in the DHT, and the information of other nodes that have been connected is stored in a routing table. At regular intervals, nodes exchange information in the DHT with each other to obtain more data.

![The architecture of decentralized storage systems.](image-url)
The local storage layer provides the upper-layer service. For instance, in the IPFS, the information in blocks that make up a file is stored in the Flatfs structure, while the blockchain [8] mostly employs LevelDB to store data.

The Block layer is responsible for dividing, decoding, and encrypting [9] data. These are local operations without involving other platforms, which ensure data security and user privacy. A block is the smallest unit of network distribution and local storage. In IPFS, the default block size is 256 kB. For each BlockId, the local node selects several nodes with the closest logical distance from the routing table, and sends <BlockId, NodeId\textsubscript{local}> to them. The logical distance is calculated by XORing BlockId and NodeId. The more prefix “0” exists in the result, the closer the logical distance is. In this cycle, a BlockId will eventually be mapped to the node with the closest logical distance to it in the network. Searching a BlockId is the same process, but, during the mapping process, if a node has the BlockId, results will be returned to the requester, and the query process will be terminated. This actually relies on the idea of consistent hashing [10].

The object layer is commonly organized by the block layer. It uses merkleDAG [11] to maintain the connection between blocks. ObjectId can help users verify whether the data object has been modified. The ObjectId of a maliciously modified data object cannot be correctly parsed into multiple BlockIds, so it cannot be accessed normally. In addition, when multiple nodes in the network upload the same data object, they will get the same ObjectId, which is a measure of DSSs to prevent creating data duplicates. The mapping and query processes of ObjectId are the same as for BlockId, which ensures that data of each node can be accessed by other nodes in the network. Based on such a safe and reliable architecture, this paper adds an index layer to provide search service.

2.2. Related Work

Many solutions have been proposed to implement the keyword search in DSSs, and they can be roughly divided into three categories: DHT-based search, centralized search, and global index-based search.

2.2.1. DHT-Based Search

This method maps keywords to nodes in the network. Any node can use the DHT substrate to determine the current live nodes that are responsible for a given keyword. In [12], the author used an XML data file as a local index to convert all queries into XML paths. The relational databases were used in [13], where queries were converted into SQL statements, and the database can also be used as a storage platform for original data. Reference [14] did not use files but proposed a conversion method of logical expressions and keywords. In some of the related studies, common index structures were used. Siva [15] organized the queryable data in an inverted index, and wrote it directly into DHT in the key-value form. In addition, QueenBee [16] sorted the results based on the inverted index. Reference [17] proposed a novel structure named Summary Prefix Tree (SPT) to solve the superset search problem. Unfortunately, the exchange of information via DHT is the main reason why p2p networks generate massive network traffic [18,19]. In addition, each keyword needs to be mapped via DHT, which makes this problem even more serious.

2.2.2. Centralized Search

The IPFS-search [20] is the representative of centralized search. It used some interfaces of IPFS to download files from the network and saved them in Elasticsearch. EtherQL [21] used a synchronization manager to get blockchain data from the Ethereum [8] network, parsed out the fields of blockchain, and stored them into MongoDB. Although it is a very reasonable idea to use existing mature tools to provide high-quality search services, this violates the principle of data discretization.
2.2.3. Global Index-Based Search

In fact, in the DHT-based search method presented in Section 2.2.1, each node maintained a local index, and the DHT aggregated these index information. The distributed wiki search [22] can be regarded as a prototype of the global index search. It is the search engine of a distributed Wikipedia mirror [23], which has been archived in the official projects of IPFS. A two-layer HashMap and inverted index were combined as the index structure in it, called Wiki HashMap. The index was stored in blocks, and operations on the index were also based on blocks. However, it could be operated only locally and supported only creation and query functions. More importantly, these operations always took a long time. In this paper, we optimized this novel structure. The deletion is a tricky problem in DSSs [24], which may lead to malicious destruction of data, so it is not provided for the time being.

3. Decentralized Index

In this section, the B+Tree and HashMap, which are two extensible, updatable, decentralized index structures, are introduced. First, the structure of B+Tree and its operation are described, and then the optimized HashMap and its advantages are presented.

3.1. Inverted Index

Inverted index consists of a dictionary and a posting list. The dictionary contains all the terms of text files, such as filename and content. For non-text files, in addition to filename, Apache Tika [25] is used to extract their metadata as basic information. The terms in the dictionary are arranged in lexicographic order and the posting list consists of multiple triples \((H_f, F_t, T_f)\), where \(H_f\) denotes the file’s hash, \(F_t\) represents the frequency of the terms in the file, and \(T_f\) represents the total number of terms in the file. Since the final results are sorted by Term Frequency-Inversed Document Frequency (TF-IDF) [26] algorithm, the posting list stores a collection of two-tuples \((H_f, TF_t)\). The frequency and inversed document frequency of a term in a file \((TF_t, IDFT_t)\) can be respectively calculated by:

\[
TF_t = \frac{F_t}{T_f},
\]

\[
IDFT_t = \log\frac{N_f}{1 + L_{tuples}},
\]

where \(N_f\) represents total number of files in the index, and \(L_{tuples}\) denotes the number of tuples contained in posting list of each term. \(TF_t \times IDFT_t\) indicates the relevance between the file and the term, and the higher the relevance is, the higher the ranking will be.

The data division of the inverted index is based on the characteristics of the superstructure. In the B+Tree, the inverted information belonging to the same leaf node is divided into one group. If the data size of this group exceeds the block size, the posting lists will be scattered into multiple blocks according to the lexicographic order of terms. Therefore, a group contains one or more blocks. However, in the HashMap, data distribution is uneven, so the bucket-merging strategy is used, which means that, under the premise of not exceeding a block size, two or more adjacent buckets are stored in one block. In this way, the inverted index is distributed in two kinds of blocks: single block and mixed block. A single block is marked with ‘1’, corresponding to a bucket, while a mixed block is marked with the size of the data it contains, which usually includes two or more buckets.

3.2. Decentralized B+Tree

The B+Tree was originally optimized for disk access, especially the nodes of the tree that are suitable for the size of disk pages for easy access. Analogously, it is also suitable for block access of decentralized storage.
The architecture of the B+Tree is illustrated in Figure 2, where it can be seen that it consists of two core elements, leaf nodes (labeled as 0) and index nodes (labeled as 1). The leaf node mainly contains the terms and their hashed posting lists, while the index node mainly contains the index keys and the hash values of its children. The root node is also classified as an index node. The decentralized B+Tree is similar to a regular B+Tree, except that the nodes store hash values instead of pointers. Furthermore, each node occupies a block.

The difference between decentralized B+Tree and traditional B+Tree in that there is no pointer between the leaf nodes in the former one. Namely, the previous leaf node does not save the BlockId of the next leaf node. As if a leaf node is updated, all predecessor nodes must be updated due to a new BlockId, which results in a high cost.

3.2.1. Create

The B+Tree is first created in memory, as shown on the left side of Figure 2. Suppose that the block size is $B$ bytes, BlockId occupies $S$ bytes, and each term occupies $T$ bytes. Then, the number of the branches for each index node is calculated by:

$$N_{\text{branches}} = \frac{B - S}{S + T},$$

while the number of terms stored in each leaf node is calculated by:

$$N_{\text{terms}} = \frac{B}{S + T}.$$  

The whole process does not involve node splitting. When one node is full, data are written directly into the next node. Ultimately, the tree is uploaded to the network starting from the leaf nodes. Each node occupies a block, and the BlockId obtained by calculation is recorded in the upper node until the $H_{\text{root}}$ is generated. Only with $H_{\text{root}}$ can we perform other operations on the index.

3.2.2. Update

The update includes local update and remote update. The blocks required for remote update do not exist locally. As mentioned in Section 2.1, these blocks need to be obtained from other nodes.
In the update process, a new term involves only a few blocks, as the hash value provides a shortcut. Starting from the root block, the new term is compared with the index keys of each layer to obtain the successor blocks until a leaf block is obtained. Then, the term and its posting list are inserted into the block, and the new BlockId generated is recorded to the upper node. Finally, a new index is created. In Figure 3, the distribution of index blocks is shown in the upper left part, where $N_1$ denotes the node that does the update operation, and $B_1$, $B_2$, and $B_3$ are stored on $N_2$, $N_3$, and $N_4$, respectively. In the update process, $N_1$ first gets $B_1$, $B_2$, $B_3$ locally, and then updates the data to $B_3$. Since the change in content will cause the recalculation of BlockId, three new blocks denoted as $B_1'$, $B_2'$, and $B_3'$ are generated. At the same time, for blocks that are not modified, such as $B_4$, their BlockIds will be written directly into the new blocks. In general, the update process generates not only a new version of the index but also a new copy of the blocks involved during this period, which improves index usability, and also promotes index discretization.

3.2.3. Query

Actually, query is the first step in the update process. If the term to be queried does not exist in the leaf node, this means that the data do not exist; otherwise, the results will be found and returned. In addition to keyword query, the decentralized B+Tree can also conduct range query. The query cannot start directly from leaf blocks but by extracting blocks within a given range from the root block. Finally, the results are taken from multiple data blocks, combined, and returned.

3.3. Decentralized HashMap

As a common data structure in memory, HashMap has a high level of performance in keyword search, but it usually takes up a large space. In particular, its uneven data distribution can cause some trouble to bucket persistence. Thus, certain optimization strategies should be used in the decentralized HashMap.

3.3.1. Create

As shown in Figure 4, the node types of decentralized HashMap include map and bucket. As is well-known, the structure of HashMap usually changes with the number of terms and its expansion is a tricky issue. In order to avoid affecting other blocks, the following strategy is adopted: If the terms stored in a bucket reach the upper limit $M$, all terms in this bucket will be hashed again and put into a
new map block in the next layer. Similarly, the HashMap is first built in memory, and then uploaded to the network from the bucket layer.

In order to facilitate searching, a map node takes up only one block. Suppose that the size of a block is $B$ bytes, the hash value of a term occupies $N$ bits, and the blockId occupies $L$ bits. Then, $N$ is the largest integer that satisfies the following equation:

$$2^N \leq \frac{8B}{N + L}.$$  \hspace{1cm} (5)

![Figure 4. The structure of HashMap.](image)

3.3.2. Update

If all terms are updated individually, the efficiency is definitely very poor. Therefore, the branch update method, which is an iterative process, is adopted. Starting from the root layer, at each layer, first, the hash value of the terms to be updated in this layer is calculated, and then the total number of terms corresponding to each hash key is calculated. It should be noted that, if a hash key already exists, the number of original terms must be added to the count. If the number exceeds $M$, the terms are rehashed, and the next level of maps is created to perform inserting; otherwise, updating is performed as described in Section 3.1. The problem of splitting the mixed bucket is considered, and a mixed bucket can be split into multiple buckets. The main objective is to ensure that terms with the same hash value are stored in the same block after splitting.

3.3.3. Query

The query process and time of HashMap are irregular, and they depend on the layer where the term is located. In each layer, if the term exists, the result will be returned directly; otherwise, the term is rehashed and found in the next layer. Generally, the HashMap does not exceed three layers. If there is no result, nothing will be returned.

4. Version Control

In the DSSs, if two nodes operate on one block at the same time, they will not interfere with each other. Because if a block is distributed to other nodes, it is no longer under the control of the original node. More importantly, in a decentralized system, modification means generating a new
block; that is, the original block still exists, and its content remains unchanged. Namely, the new block has both the original-block content and the new content. This is a natural version control mechanism, but it is adverse to managing the decentralized indexes because, as long as any node has $H_{root}$, it can operate this index and generate a new version. The indexed content and version numbers will increase exponentially. In terms of practicality, each node has its own requirements, and certain index information is useless to other nodes; therefore, version control \[27,28\] is necessary.

4.1. Version Note

Since there are no management nodes to help manage index versions, a version note is designed for each version. The version note mainly includes the hash value $H_v$ of the previous version note, the root hash $H_{root}$ of current version’s index, the number of indexed files $N_f$, the number of terms $N_t$, and the creation time $V_t$.

4.2. Version Merge

The pseudo-code of the B+Tree merging process is given in Algorithm 1. The idea is to merge small trees into large trees, where small trees denote trees with a small number of stored words ($H_{root2}$).

The specific steps are as follows. First, traverse $H_{root2}$ and find its leaf block $n$. Then, take the first word (firstKey) in $n$ and find the leaf block $n_1$ in $H_{root1}$, where the firstKey is located. If the hash values of the two blocks are the same, then the data stored in them are also the same, and there is no need to modify them; otherwise, insert the words in $n$ and their inverted index into $H_{root1}$. In this way, the newly generated $H_{root1}$ becomes the new root of the large tree.

**Algorithm 1: B+Tree Merge**

\[
\text{Input: } H_{root1}, H_{root2} \\
\text{Output: } H_{root1} \\
\text{1 Traverse } root_2 \text{ and } n \text{ is the node in } root_2; \\
\text{2 for } n \neq \text{nil and } n.Type == 0 \text{ do} \\
\text{3 firstKey = n.Keys[0];} \\
\text{4 } n_1 = \text{find}(H_{root1}, \text{firstKey}); \\
\text{5 if } n.Hash == n_1.Hash \text{ then} \\
\text{6 Skip;} \\
\text{7 else} \\
\text{8 } H_{root1} = \text{update}(H_{root1}, n); \\
\]

The pseudo-code of the HashMap merging is given in Algorithm 2. Similarly, this process is based on the idea of merging a small HashMap into a large HashMap. A HashMap is merged by layers. The specific steps are as follows. The HashMap is merged by layers. For each layer, merge the same hash keys in the two indexes, and write different hash keys and the inverted index into the large HashMap ($H_{root1}$). One of the more challenging problems is that each time a mixed block is merged, it is necessary to calculate whether the number of keys with the same hash values exceeds $M$ or the updated data exceed the block size. If so, the data in the bucket need to be split to generate new blocks.
Algorithm 2: HashMap Merge

**Input:** $H_{\text{root}1}, H_{\text{root}2}$

**Output:** $H_{\text{root}1}$

1. For each layer of $H_{\text{root}2}$;
2. for $H_{\text{root}2}.\text{HashList.length} > 0$ do
3. if hash values in $H_{\text{root}2}.\text{HashList}$ also exists in $H_{\text{root}1}.\text{HashList}$ in the same layer then
   4. write inverted index into $H_{\text{root}1}.\text{layer.Bucket}$;
5. else
6. write hash into $H_{\text{root}1}.\text{layer.HashList}$;
7. write inverted index into $H_{\text{root}1}.\text{layer.Bucket}$;
8. remove($H_{\text{root}2}.\text{HashList}$, hash values);
9. generate new $H_{\text{root}1}$

4.3. Garbage Collection

In the DSSs, each node can operate only its own data and has no right to interfere with the data of other nodes, so garbage collection is local. In the process of version merging, the data of the old blocks will be contained in the new blocks, so they can be deleted to save storage overhead. However, in reality, old blocks are not deleted immediately, but marked as non-permanent. When garbage collection is performed regularly, whether a block should be deleted is determined based on its status.

5. Experiment Evaluation

5.1. Experiment Setup

5.1.1. Datasets

In order to evaluate the decentralized B+Tree and HashMap, two sets of experiments were conducted. The first experiment was performed to compare the performance of Decentralized HashMap and the Wiki HashMap. Three datasets were taken from an online Wikipedia library [29]: “wiki-chemistry”, “wiki-physics” and “wiki-history”, as shown in Table 1; these sets contained 7706, 11,774 and 19,087 words, respectively. In another experiment, three well-known datasets from the information retrieval were used to compare the performances in create, update, and query operations between the decentralized B+Tree and HashMap. The datasets were American National Corpus(ANC), Brown Corpus, and the CMU Pronouncing Dictionary, and they contained 63,000, 42,069, and 115,694 words, respectively. In the ANC, 8000 words were randomly selected for the creation—then 5000 and 50,000 words for the update operation.

Table 1. The comparison of Wiki HashMap and Decentralized HashMap.

| Datasets          | Numer of Blocks | Create Time   | Query Time   |
|-------------------|-----------------|---------------|--------------|
|                   | Wiki HashMap    | Decentralized | Wiki HashMap  | Decentralized | Wiki HashMap  | Decentralized |
| Wikipedia-en-chemistry | 4376           | 4             | 17.77 min    | 2.47 s        | 650.23 ms    | 700.49 ms     |
| Wikipedia-en-physics  | 6394           | 6             | 25.41 min    | 4.79 s        |               |              |
| Wikipedia-en-history     | 7092           | 8             | 42.08 min    | 5.23 s        |               |              |
5.1.2. Environment

The experimental environment was based on the version 0.7.0 of IPFS. The two proposed indexes worked along with the relevant APIs supported by go-ipfs-api [30]. All experiments were conducted on a cluster with 15 commodity machines in a local area network. Each machine was equipped with a Windows 10 platform with Lenovo Intel(R) Core(TM) i5-3470 CPU @3.2 GHz, 8 GB RAM, 931 GB disk capacity and launched an IPFS node. Nodes on all machines constituted a private network.

5.2. Wiki HashMap

The comparison results of Wiki HashMap and Decentralized HashMap are shown in Table 1, where it can be observed that two proposed measures significantly improve the performance of Wiki HashMap. The hash value in the first layer of Wiki HashMap takes 12 bits, and the second layer takes 8 bits, so $2^{20}$ terms could be stored, which far exceeds the number of words that need to be stored, so the conflict rate is very low, almost one bucket per word. For this reason, it needs to occupy and upload a large number of blocks, while the Decentralized HashMap adopts a 12-bit hash value when the block size is 256 kB, which is used in Wiki HashMap, and a multi-layer structure. On the basis of increasing the collision rate, it adopts the bucket merge strategy to achieve better results.

Wiki HashMap does not support update operation, so only the results of the create and query processes are analyzed. Based on a large number of experimental results, the creation time is directly proportional to the number of blocks; for this reason, the creation time of the Decentralized HashMap is greatly reduced. However, Wiki HashMap performs better in query operation because the Decentralized HashMap needs to separate the query results from the mixed block. The Wiki HashMap only supports local query, while the Decentralized HashMap supports local query and remote query. Overall, the Decentralized HashMap has better performance.

5.3. Decentralized B+Tree and HashMap

5.3.1. Create

The creation process was completed at one node without including the other nodes. The response time included the construction of memory index and the upload of index blocks. The experimental results are shown in Figure 5, where it can be seen that, for the same dataset, as the data block increases, the create time decreases rapidly. This is because uploading blocks takes up most of the time, as presented in Figure 6. When a data block increases, this means that it stores more data, which reduces the number of blocks that need to be uploaded. Analogously, a dataset with fewer words has a shorter response time. However, based on the results of different structures, the HashMap occupies fewer blocks but takes longer time than B+Tree. It can be seen that the bucket merging strategy has played a very important role, but, according to the principle of time for space, numerous calculations are required to place the data. The B+Tree performs well because, in the process of memory creation, the data stored in each block have been determined. Therefore, in the creation of HashMap, the amount of calculation affects the final result.
5.3.2. Local Update

Local update was to update the index that already exists locally. During this period, the node would not send requests to other nodes. In the experiment, 5000 and 50,000 words were inserted into the indexes of Section 5.3.1, respectively.

The results of update time are shown in Figures 7 and 8. Overall, HashMap performs better. Based on the results presented in Figures 9 and 10, the HashMap gets (oldBlocks) and uploads (newBlocks) fewer blocks during the update process. However, as shown in Figure 8, when the block size is larger than 128 kB, B+tree gradually shows its advantage. In fact, during the update process, a large amount of node splitting occurs in B+Tree and HashMap. The difference is that the leaf nodes of B+tree do not contain the data of other leaf nodes, and the buckets may be mixed. Therefore, in the update process of HashMap, different buckets are separated and merged, resulting in excess calculations. These calculations determine the total time consumed in the case when the amount of data are large or the block size is large. In addition, it can be seen that the two structures only involve a small part of the blocks, which is a tough problem we overcome.
Figure 7. Comparison of the time between B+Tree and HashMap to update 5000 words.

Figure 8. Comparison of the time between B+Tree and HashMap to update 50,000 words.

Figure 9. The number of blocks involved in updating 5000 words (256 kB).
5.3.3. Remote Update

In the remote update, it is ensured that the update node is a newly created node and does not contain any data. Since, in the real environment, index blocks may be distributed on different nodes, in the experiment, all blocks of an index were distributed on one, five, and 10 nodes, successively. The p2p network was dynamic, so an additional operating time needed to be considered in the remote update, such as time of finding nodes, establishing connections, and transferring data.

The results of the remote update are shown in Figures 11 and 12. For the same dataset, the more nodes the index is distributed on, the longer the update time is. This is because the blocks that are obtained from one node need to be obtained from five or more nodes. Nevertheless, the time increases slowly, which benefits from certain system characteristics, of which one is that adjacent nodes always exchange information at regular intervals, thus saving some time. In addition, the IPFS supports batch requests for blocks, so multiple blocks can be get from one node at a time, and, according to Section 5.3.2, there are fewer blocks obtained. In fact, time is still spent in the local update process.

![Figure 10](image1.png)

**Figure 10.** The number of blocks involved in updating 50,000 words (256 kB).

![Figure 11](image2.png)

**Figure 11.** The time of remote update in B+Tree (256 kB).
5.3.4. Query

Figure 13 shows the time of local query and remote query for two data structures. As expected, remote query takes more time than local query. In addition, obviously, HashMap takes less time which is due to the data structure itself. In order to reduce the height of B+Tree, the leaf nodes do not store inverted indexes. However, the buckets in the HashMap save the inverted index. Therefore, the query of B+Tree always takes one more block than HashMap. In addition, the latency of obtaining a block is increasing with the block size. In remote query, it takes extra 20 ms to get a 16 kB block, and 150 ms to get a 512 kB block.

5.3.5. Version Merge

The version merging algorithm is to merge small trees into large trees. In Figure 14, the results of merging the big trees into the small trees are also presented. The “ANC-Brown” in the legend in Figure 14 indicates that “Brown” is merged into “ANC”. It is obvious that merging small trees is more efficient. At the same time, similar to the updated results, HashMap is suitable for small amounts of data, while the B+Tree performs better in large amounts of data.
5.3.6. Network Cost

As described in Section 2.1, in the DHT-based network, a large number of XOR operations are required to map resources to the nodes with the closest logical distance. In the IPFS, each mapping operation will generate three messages, that is, each node needs to select three nodes from the routing table and then to send the resource information to them. Thus, in the DHT-based search method, the number of messages sent by each node is three times that of keywords. For instance, the process of mapping 8000 words requires 24,000 messages. It can be seen from Figure 6 that the decentralized B+Tree only generates 12 (4 × 3) to 336 (112 × 3) messages, and the decentralized HashMap generates 9 (3 × 3) to 111 (37 × 3) messages; thus, the network traffic is reduced significantly. In addition, in the querying process, DHT-based search first needs to find all nodes that contained the keyword to be queried and then sends requests to these nodes. In contrast, the proposed indexes need to request only three to four blocks to get the final results. Thus, the proposed design can greatly reduce network overhead.

![Figure 14. Comparison of the time of version merge between B+Tree and HashMap.](image)

6. Conclusions

In recent years, DSSs have developed rapidly and they have been supporting not only file storage but also streaming data storage, even providing live broadcast service [31]. However, the fact of providing only hash-based search hinders their expansion in many areas. In this paper, we propose and investigate the performance of B+Tree and HashMap combined with the inverted index respectively in DSSs. The results show that our index structures have good availability and scalability. In addition, the problem of index version control is considered for the first time, and two simple algorithms are developed to resolve this problem. In fact, delay has always been a thorny issue [32] in the p2p system. The experiments show that, when nodes are distributed in a local area network, the operation times of decentralized B+Tree and HashMap are acceptable. Therefore, our future work will be to optimize these index structures to make them suitable for wide area networks and even larger network space.

**Author Contributions:** Conceptualization, L.Z., C.X., and X.G.; methodology, L.Z. and X.G.; software, L.Z. and C.X.; validation, formal analysis and investigation, L.Z. and X.G.; resources, data curation and writing—original draft preparation, L.Z.; writing—review and editing, L.Z. and X.G.; visualization, supervision and project administration, L.Z. and C.X. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by National Natural Science Foundation of China Grant 61572194, National Natural Science Foundation of China Grant 61672233.

**Conflicts of Interest:** The authors declare no conflict of interest.
References

1. Benet, J. IPFS—Content Addressed, Versioned, P2P File System. *arXiv* 2014, arXiv:1407.3561.
2. Storj. Available online: https://storj.io/storj.pdf (accessed on 19 October 2020).
3. Siad. Available online: http://www.siacoind.com/ (accessed on 10 October 2020).
4. Hou, L.Y.; Tang, T.Y.; Liang, T.Y. IOTA-BT: A P2P File-Sharing System Based on IOTA. *Electronics* 2020, 9, 1610. [CrossRef]
5. Huang, S.; Chang, S.; Wu, Y. A Secure File Sharing System Based on IPFS and Blockchain. In Proceedings of the 2020 2nd International Electronics Communication Conference, New York, NY, USA, 8–10 July 2020; pp. 96–100.
6. Stoica, I. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE ACM Trans. Netw.* 2003, 11, 17–32. [CrossRef]
7. Maymounkov, P.; Mazieres, D. Kademia: A peer-to-peer information system based on the XOR metric. In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 02), Cambridge, MA, USA, 7–8 March 2002; pp. 53–65.
8. Ethereum. Available online: https://ethereum.org/en/ (accessed on 27 September 2020).
9. Halkes, G.P.; Poudewse, J.A. Verifiable Encryption for P2P Block Exchange. In Proceedings of the 2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P), Delft, The Netherlands, 25–27 August 2010; pp. 1–4.
10. Yang, J.; Shen, H. Blockchain Consensus Algorithm Design Based on Consistent Hash Algorithm. In Proceedings of the 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Gold Coast, Australia, 5–7 December 2019; pp. 461–466.
11. Yu, M.; Sahraei, S.; Li, S.; Avetisyan, A.S.; Kannan, S.; Viswanath, P. Coded Merkle Tree: Solving Data Availability Attacks in Blockchains. *arXiv* 2019, arXiv:1910.01247.
12. Garcés-Erice, L.; Felber, P.A.; Biersack, E.W.; Urvoy-Keller, G.; Ross, K.W. Data indexing in peer-to-peer dht networks. In Proceedings of the 24th IEEE International Conference on Distributed Computing Systems, Hachioji, Tokyo, Japan, 24–26 March 2004; pp. 200–208.
13. Pankowski, T. Keyword Search in P2P Relational Databases. In *Agent and Multi-Agent Systems: Technologies and Applications*; Jezic, G., Howlett, R., Jain, L., Eds.; Springer: Cham, Switzerland, 2015; pp. 32–58.
14. Ghanbari, S.; Amiri, N.; Mahboobi, B. Distributed Search of Logical Expressions in DHT Storage Systems. In Proceedings of the 2019 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), Victoria, BC, Canada, 21–23 August 2019; pp. 1–5.
15. Khudhur, N.; Fujita, S. Siva—The IPFS Search Engine. In Proceedings of the 2019 Seventh International Symposium on Computing and Networking (CANDAR), Nagasaki, Japan, 26–29 November 2019; pp. 150–156.
16. Ziliang, L.; Chris, L.; Eric, L.; Ben, K.; Siu-Ming, Y. Decentralized Search on Decentralized Web. *arXiv* 2018, arXiv:1809.00939.
17. Ngom, B.; Makpangou, M. Summary prefix tree: An over DHT indexing data structure for efficient superset search. In Proceedings of the 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, 30 October–1 November 2017; pp. 1–5.
18. Varyani, N.; Nikhil, S.; Shekawat, V.S. Latency and Routing Efficiency Based Metric for Performance Comparison of DHT Overlay Networks. In Proceedings of the 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA), Crans-Montana, Switzerland, 23–25 March 2016; pp. 337–342.
19. Harjula, E.; Koskelo, T.; Ylianttila, M. Comparing the performance and efficiency of two popular DHTs in interprocess communication. In Proceedings of the 2011 IEEE Wireless Communications and Networking Conference, Cancun, Mexico, 28–31 March 2011; pp. 2173–2178.
20. Search Engine for the Interplanetary Filesystem. Available online: https://github.com/ipfs-search/ipfs-search (accessed on 10 October 2020).
21. Li, Y.; Zheng, K.; Yan, Y.; Liu, Q.; Zhou, X. EtherQL: A Query Layer for Blockchain System. In Proceedings of the International Conference on Database Systems for Advanced Applications, Suzhou, China, 27–30 March 2017; pp. 556–567.
22. Łukasz, M. Distributed Wiki Search. Available online: https://github.com/magik6k/distributed-wiki-search (accessed on 11 October 2020).

23. Distributed Wiki Mirror. Available online: https://github.com/ipfs/distributed-Wikipedia-mirror (accessed on 16 November 2020).

24. Politou, E.; Alepis, E.; Patsakis, C. Delegated content erasure in IPFS. *Future Gener. Comput. Syst.* 2020, 112, 956–964. [CrossRef]

25. Apache Tika—A Content Analysis Toolkit. Available online: http://tika.apache.org/ (accessed on 13 November 2020).

26. Hiemstra, D. A probabilistic justification for using tf x idf term weighting in information retrieval. *Int. J. Digit. Libr.* 2000, 3, 131–139. [CrossRef]

27. Lin, X.; Li, S.; Shi, W.; Teng, J. A Scalable Version Control Layer in P2P File System. In Proceedings of the International Conference on Grid and Cooperative Computing, Beijing, China, 30 November–3 December 2005; pp. 996–1001.

28. Nizamuddin, N.; Salah, K.; Ajmal Azad, M.; Arshad, J.; Rehman, M.H. Decentralized document version control using ethereum blockchain and ipfs. *Comput. Electr. Eng.* 2019, 76, 183–197. [CrossRef]

29. Wiki Mirror. Available online: https://wiki.kiwix.org/wiki/Content_in_all_languages (accessed on 14 October 2020).

30. Go-ipfs-API. Available online: https://github.com/ipfs/go-ipfs-api (accessed on 14 October 2020).

31. PPIO. Available online: https://www.pp.io/zh (accessed on 15 October 2020).

32. Shen, J.; Li, Y.; Zhou, Y.; Wang, X. Understanding I/O Performance of IPFS Storage: A Client’s Perspective. In Proceedings of the 2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS), Phoenix, AZ, USA, 24–25 June 2019; pp. 1–10.

**Publisher’s Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).