EBTree: A B-plus Tree Based Index for Ethereum Blockchain Data

Huang XiaoJu
51174500024@stu.ecnu.edu.cn
East China Normal University
China, Shanghai

Gong XueQing*
xqgong@sei.ecnu.edu.cn
East China Normal University
China, Shanghai

Huang ZhiGang
51174500095@stu.ecnu.edu.cn
East China Normal University
China, Shanghai

Zhao LiMei
51174500061@stu.ecnu.edu.cn
East China Normal University
China, Shanghai

Gao Kun
51194501039@stu.ecnu.edu.cn
East China Normal University
China, Shanghai

ABSTRACT

The emergence of smart contract promotes the popularity of blockchain applications, leading the dramatically growth of Ethereum blockchain data size. The analysis on blockchain data is urgently needed for users, e.g., collecting statistics of tokens, monitoring the status of Ethereum blockchain. However, Ethereum could only support simple searches on blockchain data on account of its storage model.
This paper proposes the EBTree, an index for Ethereum blockchain Data, and implements it based on Ethereum client (Geth1.8). With the properties of B^+ tree, EBTree could support real-time top-k, range, equivalent search on Ethereum blockchain data. Besides, EBTree takes up relatively small storage space because it only stores the identifiers of blockchain data. Meanwhile, considering of the time intervals of mining block and synchronizing data from Ethereum network, the time of insertion in EBTree has little influence on the performance of Ethereum client. We conduct experiments to evaluate the performance of EBTree. According to the result of experiments, EBTree shows great performance on searches and insertion at low cost of storage.

CCS CONCEPTS

• **Information systems → Distributed retrieval**.

KEYWORDS

Ethereum, B^+ Tree, Blockchain, Complex Search

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASSE 2020, May 13–15, 2020, Nagoya, Japan
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7710-2/20/05...$15.00
https://doi.org/10.1145/3399871.3399892

*Corresponding author

1 INTRODUCTION

Blockchain [29] becomes popular in worldwide researchers for its properties (e.g. tamper-resistant, traceability) [28]. After introducing the smart contract [8] into the blockchain, tons of blockchain applications [26][1][20] emerge, leading the sharp rise of blockchain data size.
Ethereum [25] is currently the most mainstream development platform for blockchain applications [13]. The existing development framework and development language (e.g. Solidity [10]) on Ethereum are relatively mature. Therefore, this article takes Ethereum as the experimental platform. Ethereum network is a peer-to-peer network [18] [22], where each peer node stores all Ethereum blockchain data [5]. Besides, those data are encoded and stored in LevelDB [12]. Ethereum blockchain data have following characteristics: 1) Ethereum blockchain data are append-only; 2) Ethereum blockchain data are encoded and stored in LevelDB; 2) New block data will be produced in certain in about 15 seconds; 3) The data size is huge and grows at fast speed. For those, it is difficult to implement complex searches on Ethereum blockchain data with current Ethereum storage model.
However, complex searches on Ethereum are needed eagerly by users. On one hand, the statistics (result of complex searches) of Ethereum blockchain data can provide important information for blockchain application developers. For example, average gas price can be estimated by statistic of Ethereum blockchain data. With it, proper gas price can be chosen to ensure transactions be packed timely. On the other hand, the statistics can also reveal the abnormal conditions, which are usually triggered by hackers or bugs of smart contract [11]. For example, some hackers won the big prize of the popular Ethereum game, Fomo3d [2], by controlling the Ethereum network for less than one minute. They increased the gas price of transactions maliciously, which made transactions from other users not be packed. Therefore, the hackers jammed the Ethereum blockchain with their transactions for several blocks and won the game. In fact, it's easy to recognize such attack by monitoring the change of gas price in transactions with real-time complex searches. Top-k search on gas price could return transactions with the top-k highest gas price, which will change greatly when above attack happens.

Since B^+ tree [9] is well suited to random and sequence search on data content, introducing it into Ethereum storage model could dramatically improve the efficiency of complex searches (e.g. top-k, range, equivalent search) on blockchain data. Although the insertion of B^+ tree consumes extra time, that is much smaller than the time of mining a block [16]. The insertion of B^+ tree brings little influence to the performance of Ethereum client. Therefore, this paper proposes EBTree, a B^+ tree based index for complex searches on blockchain data. Further, due to the compact strategy used by Ethereum client, the data size of EBTree could be relatively small.

To summarize, we make following contributions in this paper: 1) We organize the researches on the analysis of blockchain data, group them, and analyze their pros and cons. 2) We describe the Ethereum storage model in detail. 3) We design an index called EBTree that can provide complex searches and implement EBTree based on Ethereum client.

In the following, we introduce the related work in section 2. Then the storage model of Ethereum is described in section 3. Section 4 describes the details of EBTree. The evaluation of EBTree is shown in section 5 before the conclusion and future work are given in section 6.

2 RELATED WORK

In order to implement the complex searches on blockchain data, scholars have proposed many solutions. These solutions can be divided into two categories based on the way of processing blockchain data:

(1) Store the blockchain data into other database (e.g. MongoDB [4], MySQL [14]). EtherQL [17] uses synchronization manager to get blockchain data from Ethereum network, parses out the fields of blockchain, stores them into MongoDB. MongoDB is an efficient and highly scalable NoSQL database, which supports complex searches. Thus, EtherQL could support relatively efficient complex searches on blockchain data.

The system architecture of EthGuide [24] is similar to that of EtherQL. EthGuide synchronizes data from Ethereum network, puts them into RabbitMQ, processes the data pulled from RabbitMQ [21], stores all data into MySQL. Further, EthGuide supports the management of digital assets in the smart contract that meets ERC721 [6] standards. However, both EtherQL and EthGuide take up huge storage space for storing blockchain data.

Forkbase [23] is a new storage engine for blockchain data. It uses POS tree to index and store blockchain data. While Forkbase has to parse out the blockchain data and transfer them into needed form every time new block is produced.

(2) Preserve the statistics of blockchain data. Blocksci [15] is an analysis platform for blockchain data based on blockchain nodes. In Blocksci, the blockchain nodes are responsible for synchronizing data from the blockchain network and storing them in local database. Then, the parser in Blocksci parses the data to obtain the graph of transactions, index of blockchain data and other statistical data. The result will be stored in analysis database. Based on analysis database, Blocksci can implement complex searches.

Abujamra and Randall proposed a blockchain data framework [3] which processes data in similar way with the Blocksci. However, this framework focuses more on the analysis of external data of blockchain, such as account tagging, meta data analysis, etc. On the other hand, vChain [27] pays more attention to alleviating the cost of storage and verifying the accuracy of search result.

Even though storing blockchain data locally, these systems lack the search on the specific information of blockchain data. EQL [17] is a query language for retrieving the specific information of blockchain data. While it pays more attention to the definition of query language than the implementation details.

3 ETHEREUM STORAGE MODEL

Ethereum is a popular, open-source public blockchain platform. Users can freely join the Ethereum network through the Ethereum client. Light Ethereum client only stores the meta data of blockchain. When users want to get specific information of blockchain data, light Ethereum client requests it from Ethereum network. Full Ethereum client synchronizes, stores and manages all the Ethereum blockchain data locally. Users can send transactions to Ethereum network by Ethereum client. The Ethereum client who produces the new block will pack the transactions into block.

The storage model of Ethereum is shown in Figure 1. As we can see, the blockchain is a data set of blocks, which are organized as a linked list. Each block contains the hash of block data (bh), the number of block (bn), the set of transactions, time stamp, the hash of the previous block, the root of account state trie, the root of transaction trie, the root of receipts trie, the gas used by all transactions in this block, etc. The time stamp records the time of block being mined. The account state in Ethereum corresponds to the stateObject. StateObject contains address, account balance, nonce, the hash of smart contract code, etc. The address is the identifier of StateObject. The nonce indicates the number of transactions initiated by the stateObject. Smart contract can be seen as the script run in Ethereum network. StateObjects are organized as an account state trie in the form of Merkel Patricia trie (MPT) [7] [19]. The root of account state trie is stored in block as stateRoot. Transactions and receipts construct transaction trie and receipts trie in the form of MPT. The root of transaction trie and receipt trie are recorded to prevent the lost or tampering of transactions and receipts. Receipt is the proof for transactions that are committed successfully. When the block is contained in blockchain, the corresponding receipts will be created.

Transactions include the transferred value, gas price, gas limit, etc. Gas is the unit of computer power in Ethereum. Gas price shows the unit price of gas. Gas limit indicates the maximum gas that the initiator of transaction provides. Gas price and gas limit determine the maximum fee of executing the transaction. The receipts contain the gas used, status,

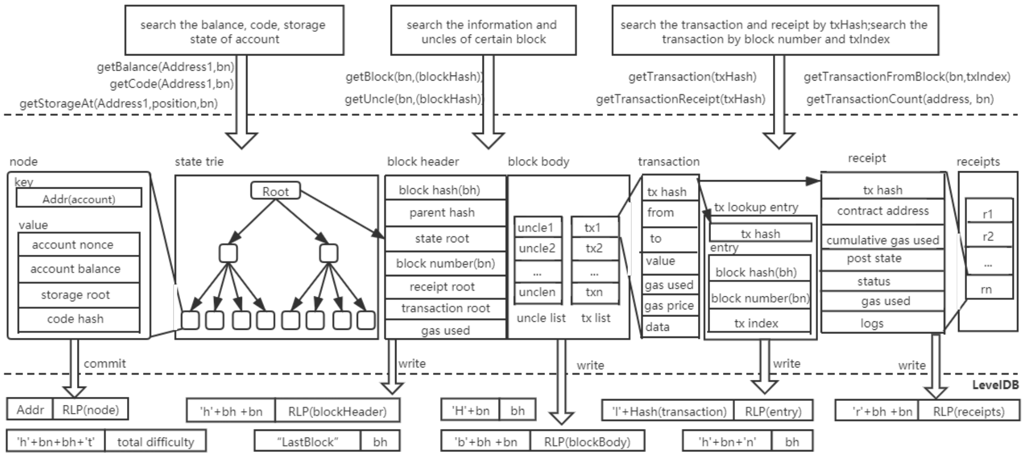

Figure 1: Ethereum storage model

logs for corresponding transactions. The transaction lookup entry is designed to get the identifier of corresponding block with transaction hash. A transaction hash is corresponded to an entry which includes bh, bn and transaction index.

All Ethereum storage data are stored in LevelDB in the form of $\langle k, v\rangle$ pair. The k is usually a string joined by prefix, bn and the hash of v. The v is the encoded blockchain data. Blockchain data is encoded by RLP, which makes the size of encoded data relatively small. The account state trie is also stored in LevelDB. The nodes in trie are put into the LevelDB separately as $\langle k, v\rangle$ pairs. The k is the account address. The v is the encoded node data. Also, some statistic data are stored in LevelDB, such as the total difficulty of blockchain, the hash of latest block in blockchain. For the transaction lookup entry, the k contains the prefix and transaction hash, while the v is the encoded entry. Besides, LevelDB stores the correspondence between bn and bh. Thus, once users provide the bn or bh, Ethereum client could calculate the k of block data, and return v of block data to users.

Ethereum client stores all history blockchain data. Besides, new block is produced every 15 seconds. Hence, the data size of blockchain grows larger and larger. Meanwhile, the popularity of Ethereum applications brings more data into the smart contract, which leads the sharp rise of blockchain data. To decrease the storage usage, the developers of Ethereum client apply efficient compacting strategy in LevelDB.

Based on such storage model, Ethereum client could only support some simple searches on blockchain data (e.g. get the block by block number or block hash). For example, users need to provide bn or bh to search specific block. Since the correspondence of bn and bh is recorded in LevelDB, we could get the bn and bh. Then, the Ethereum client constructs the key from them, loads the encoded block data *eblock* from LevelDB, decodes the *eblock* and returns the corresponding block data to users. Since all transactions are contained in blocks, users have to get the corresponding block in above way if they need to search specific transaction. Once the block data is given, users could get the block number and block hash by the hash of transaction easily with transaction look up entry. Then, the block is loaded to memory, and the transaction could be found by transaction index.

In the whole, it is impossible to get blockchain data if the identifiers of them are not provided. Also, the complex searches are hard to complete with Ethereum storage model.

4 EBTREE

To support the complex searches, including top-k, range and equivalent search, on Ethereum blockchain data, we propose EBTree, a B^+tree based index. Like B^+tree, EBTree can dramatically improve the efficiency of complex searches. Besides those, EBTree has following characteristics: 1) The time used in insertion is short, compared with the time intervals of mining block and synchronizing data from network; 2) EBTree provides real-time complex searches; 3) EBTree takes up small storage space.

4.1 EBTree Structure

The leaf nodes, internal nodes and meta data comprise EBTree. EBTree has one and only one root. The root could be an internal node or leaf node. Each internal node could have at most p, at least $p/2$ child nodes. The child node could be internal node or leaf node. Each leaf node has at most s data (da) and stores the identifier or pointer of next leaf node ($nePtr$). All leaf nodes compose of a linked list. The meta data in EBTree is used to record the real-time statistics of EBTree. Figure 2 shows the structure of EBTree.

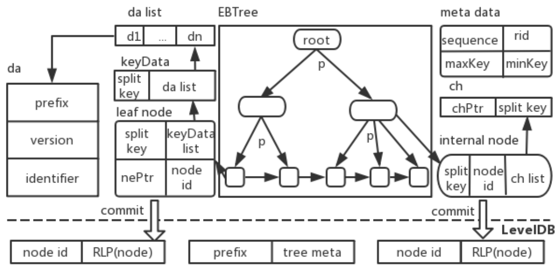

Figure 2: EBTree Structure

Similar to B^+ tree, the internal node records the node identifier, split key and ch list. An internal node identifier is

concatenated by prefix and node number (num) (e.g. txv_1). Prefix is used to identify the EBTree built for different kind of data. For example, the txv means the value of transaction, and the $txgu$ means the gas used of transaction. num is the serial number of a node in EBTree. Therefore, txv_1 means the first node produced in EBTree for transaction value. The function of split key is similar to the search key in B^+ tree. For blockchain data, the split key could be transaction value, the total gas used of block, the balance of account, etc. Each ch contains $chPtr$ and split key. $chPtr$ could be the identifier or pointer of child node.
A leaf node contains $keyData$ list, node identifier, split key and $nePtr$. Each $keyData$ includes the split key and da list. The node identifier of leaf node is produced in the same way as the internal node. The da is concatenated by prefix, version and identifier of blockchain data. For example, the $tx100\_10$ means the tenth transaction in hundredth block. The size of da list could be huge when there are many blockchain data corresponding to the same split key. For example, there are 75,889 transactions whose value is 1 ether in the first 3,000,000 blocks.

Meta data includes $sequence$, $maxKey$, $minKey$ and rid. The $sequence$ is equal to the number of nodes in EBTree. With $sequence$, we could easily calculate the num when a new node produced. Thus, the uniqueness of node identifier is ensured. The $minKey$ and $maxKey$ record the minimum and maximum split key in EBTree, which enable the fast response for searching the latest maximum or minimum split key in EBTree. The rid records the node identifier of root in EBTree. With it, we could load the root from LevelDB. Since the LevelDB of Ethereum client use efficient compacting strategy, we use the same database to store EBTree for users' convenience and high storage efficiency. Hence we can load blockchain data and EBTree data with the same database instance. Each node is encoded and stored as a $\langle k, v\rangle$ in LevelDB. The k is the identifier of node, and the v is the output of RLP which is used to encode node. We choose the RLP to encode the node because the result of RLP takes up relatively small storage. To increase the write efficiency, the node data is put into the LevelDB in batches. The meta data also is stored in LevelDB. For example, the $\langle txv, RLP(meta)\rangle$ records the meta data of EBTree for transaction value.

4.2 Insertion in EBTree

To adapt different data set, EBTree allows us to adjust the depth and width by setting the capacity of internal node and leaf node. Usually, we make the size of node close to the size of data block in LevelDB as much as possible, enabling the highest read/write efficiency. Hence the appropriate capacity of internal node and leaf node can be calculated. As the Algorithm 1 depicts, we should load the root of EBTree from the LevelDB before insertion. Firstly, the identifier(rid) of root in EBTree ebt could be loaded from LevelDB through the corresponding prefix (e.g. txv is the prefix of EBTree for transaction value. Then, the root of EBTree could be loaded from LevelDB with. Furthermore, the need to be replaced by the pointer of root for reuse. There are three important steps during insertion:

Algorithm 1 Insertion in EBTree

```
Input: The blocks needed be processed $Blocks$, The database instance $db$;
Output: The identifier($rid$) of root in EBTree $ebt$;
1: Load $rid$ of $ebt$ from the $db$;
2: Load the root of $ebt$ $rt$ by function $resovleNode(rid)$;
3: Replace the $rid$ with the pointer to root;
4: $k = 0$;
5: repeat
6:     Pre-process the blockchain data;
7:     Start from the $rt$ to find the target leaf node $tle$ that $Blocks[k]$ should belong to;
8:     Insert the data into $tle$;
9:     Split $tle$ if necessary;
10:    $k++$;
11: until ($k => len(Blocks)$)
12: Update the $rid$;
13: Commit the changed node into the $db$;
14: Write $rid$ back to the $db$;
15: return $rid$;
```

(1) First, the block data should be pre-process before insertion. We need to collect the split key, version and identifier of blockchain data. The block number in blockchain data is used as version. The version and identifier are concatenated as one variant, da (described in section 4.1). The da, instead of specific blockchain data, is stored in EBTree, leading the small storage usage of EBTree.

(2) Second, the da and split key should be stored into the EBTree. This process may include three procedures: 1) Find the target leaf node (tle). In order to find the tle, we choose the appropriate ch (depicted in section 4.1) of root to search by split key. If the $chPtr$ (described in section 4.1) is an identifier, we need to load the corresponding child node from LevelDB. Then, the $chPtr$ should be replaced with the pointer of child node. Furthermore, we continue to search in that child node in the same way, until we get the tle. 2) Add the da into the tle. We need to traverse the da list and find the appropriate location to put da. 3) Split the tle if necessary. According to Figure 3, after this step, there will be a sub-tree of EBTree in memory. In common case, the subtree only includes one path from root to tle. However, once the split occurred, the new nodes produced in this insertion are included.

(3) Third, the node in sub-tree should be committed to database. Before written to LevelDB, the node should be collapsed and encoded. To collapse node, we replace the pointer in node with node identifier, which makes the node separate from others. Then the collapsed node should be encoded by RLP and written

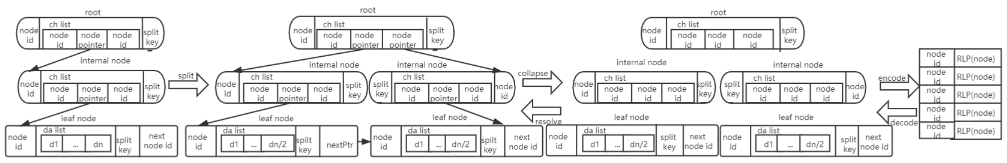

Figure 3: Insertion and reconstruction in EBTree

back to LevelDB. The commit is bottom-up: only when all changed child nodes are committed can the parent node be committed. We also could postpone the commit if the blockchain data arrived in batches (e.g. the transactions in one block).

4.3 Search in EBTree

EBTree supports top-k, range and equivalent search on blockchain data. The equivalent search could be considered as a kind of range search. There are two main steps in process of those searches:

(1) Find the first target leaf node (tle). In the process of top-k search, we start to search in root of EBTree. When the internal node is met, we choose to search in the first child node of the internal node until the tle is found. While for range search, this step is identical to the step in section 4.2.
(2) Traverse the linked list composed of leaf nodes from tle and assemble the result set (rs). For top-k search, we traverse the linked list from tle and add the da to rs until the length of rs is larger than or equal to k. While for range search, we stop scanning leaf nodes in linked list when the split key in leaf node is larger than the upper limit of given range. Meanwhile, the da in leaf nodes is added to rs.
(3) Return the rs.

The rs only includes the identifier of blockchain data. Once users need specific information of blockchain, we could load the needed encoded blockchain data from LevelDB by the identifiers. Besides, we could support the search on specific data version with the version stored in da. For example, even with 3,000,000 blocks, we could search the top-k transaction value in 50,000 blocks.
In the whole, loading nodes from LevelDB one by one is friendly to memory on account of the huge data size of blockchain. But it will decrease the efficiency of search in some way. However, once the p is set appropriately, the depth of EBTree is low, the number of internal nodes needed be loaded is small. Therefore, the efficiency of search can be ensured.

4.4 Efficiency Analysis

Ethereum blockchain data take up huge volume of storage. Therefore, the analysis of the insertion efficiency, search efficiency and storage usage is necessary.
As the section 4.2 described, the time consumed by preprocessing data could be ignored for it is simple enough. Assuming that there are n $keyData$ (described in section 4.1) in EBTree, the time complexity is of inserting data into leaf nodes is $O(log_p n)$ (p means the capability of internal node). Considering the time of collapsing and encoding is a constant, the total time consumed by commit is proportion to $log_p n$.

Table 1: Storage Usage of Node's field (Byte)

Split key	Node identifier	*ch*		*da*		
		split key	chPtr	prefix	version	identifier
8	8	8	8	1	8	16

To analyze the search efficiency of EBTree, we need to calculate the efficiency of following parts: find the target leaf node and collect result. Similar to B^+ tree, the cost for finding the node is $log_p n$. The time of collecting result differs in different kind of searches. For top-k search, the efficiency of collecting result is $O(k/s)$. Usually the k is much smaller than $log_p n$, so the total time complexity of top-k search is $log_p n$. For range search, the size of result set len_{rs} depends on the blockchain data size. So, it is hard to estimate the precise tendency of range search efficiency. In the whole, the total cost of range search could be expressed as $O(log_p n + len_{rs})$. For equivalent search, the cost of collecting result could be ignored in common case, since the time of loading the leaf node and adding da into result set is tiny. Therefore, the total cost of equivalent search is $O(log_p n)$. However, when the length of result set is too long, the cost of collecting result still should be counted.
EBTree needs extra storage space to store the nodes and meta data. The storage space used by EBTree is related with the size of the blockchain data. Assuming that there are t blockchain data needed to be indexed, according to the EBTree structure described in section 4.1 and the Table 1, the da list takes up $(17 * t)$ bytes, then the total size of leaf nodes is about $(17 * t)$ bytes for the relatively small number of split key. Also, the ch list takes up much more storage space than split key in internal node. So, the total size of a internal node is nearly equal to the size of ch list, $16 * len_{ch}$ bytes. Since there are a lot of blockchain data corresponding to the same split key for every EBTree, the len_{ch} is smaller than t/s (s means the capability of leaf nodes). Therefore, we could estimate that the storage usage of EBTree is basically in the same magnitude with $17 * t$ bytes since the size of internal nodes is much smaller than leaf nodes. Besides, using the efficient strategy of data compression, the EBTree takes up much smaller storage space than the result we calculate.

4.5 New Storage Model and APIs

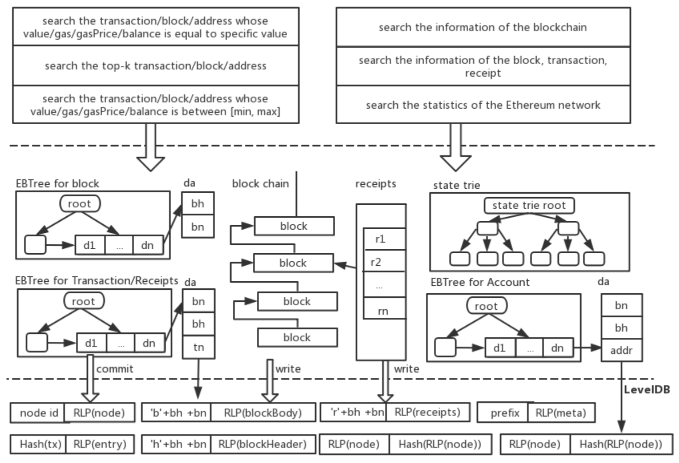
search the transaction/block/address whose value/gas/gasPrice/balance is equal to specific value
search the top-k transaction/block/address
search the transaction/block/address whose value/gas/gasPrice/balance is between [min, max]
search the information of the blockchain
search the information of the block, transaction, receipt
search the statistics of the Ethereum network
EBTree for block
root
d1
...
dn
da
bh
bn
block chain
block
receipts
r1
r2
...
rn
state trie
state trie root
EBTree for Transaction/Receipts
tn
EBTree for Account
addr
commit
write
write
LevelDB
node id
RLP(node)
'b'+bh +bn
RLP(blockBody)
'r'+bh +bn
RLP(receipts)
prefix
RLP(meta)
Hash(tx)
RLP(entry)
'h'+bh +bn
RLP(blockHeader)
RLP(node)
Hash(RLP(node))
RLP(node)
Hash(RLP(node))

Figure 4: New Storage Model

The new storage model of Ethereum client with EBTree is shown by Figure 4. We search EBTree by split key and get *da* from the result set. Using *da*, the corresponding encoded blockchain data (*endata*) is loaded from LevelDB. Then, we decode the *endata* and obtain the complete blockchain data. For different types of blockchain data, we could build EBTree for each of them. All of those data are stored in LevelDB. With the new storage model, the new Ethereum client could support the real-time top-k, range, equivalent search on Ethereum blockchain data.

Besides APIs provided by Geth1.8, we add APIs for above complex searches as the Table 2 shows. In every API, we use a input parameter e to declare which EBTree we want to search. With the e, the k (key in LevelDB) for meta data of EBTree could be obtained. Then the root of corresponding EBTree will be loaded from the LevelDB. Furthermore, the search can start from the root to the target leaf node.

Table 2: APIs for Complex Searches

Top-k Search	$topKVSearch(e, k, bn)$	$topKDSearch(e, k, bn)$
Range Search	$rangeVSearch(e, begin, end, bn)$	
Equivalent Search	$equivalentVSearch(e, key, bn)$	
Meta Data Search	$getStatistics(e)$	

For the convenience of description, we take the EBTree for transaction value as example to explain to meaning of APIs. *topKVSearch* returns the top-k transaction values of all transactions in blockchain and the corresponding *da* (defined in section 4.1) list. While *topKDSearch* returns first k *da* in EBTree. The result of *rangeVSearch* will include all transfer value of transactions between given range, and the corresponding *da* list. *equivalentVSearch*, as a special kind of range search, finds all transactions whose value is equal to the given input parameter *key*. For above search, users could specify the data version that we want to search on by setting *bno*. If users choose to search on latest blockchain data, the *bno* could be set to 0. *getStatistics* returns the statistics of EBTree. (e.g., the maximum and minimum split key, the total number of *da* and *keyData*, the total number of leaf nodes, the sequence of EBTree)

5 EXPERIMENTS

We expand the function of Geth1.8 to implement EBTree in about 6k lines of golang code, which is open-source at GitHub [1]. For the convenience of description, we use Geth-EBTree to represent the Geth with EBTree. In this section, we evaluate the performance of Geth-EBTree in terms of insertion efficiency, search efficiency and storage consumption. To facilitate analysis, we choose to conduct the experiments on the EBTree for transaction value.

Experiments are running on a Server (Intel E5-2620 CPU, 2.4GHZ and 32GB memory on Ubuntu18.04LTS). We set the cache to 2GB when the Geth-EBTree is started.

The data set we used in the experiments includes the first 3,000,000 blocks in public Ethereum network. There are 15,362,853 transactions in the data set. Since transactions in block are packed in batches, we record the experiment results for every 50,000 blocks for the convenience of collecting data and accuracy of experiment data. Besides, the capability of internal nodes (p) is set to 128, and the capability of leaf nodes (s) is set to 16 according to the analysis given in section 4.4.

5.1 Efficiency of Insertion in EBTree

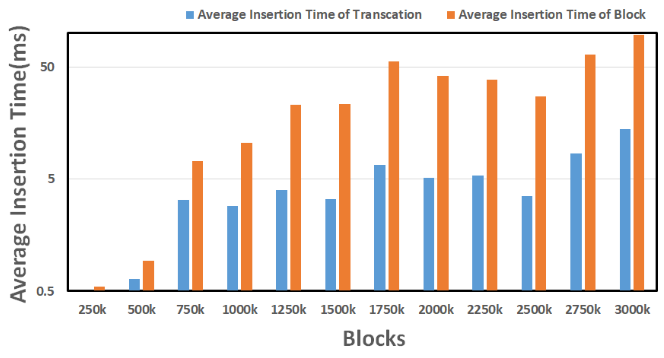

Figure 5: The Average Insertion Time

To remove the influence brought by speed of network, we used *copydb* (a built-in function in Geth1.8) to import the blockchain data from local database into the Geth-EBTree. Once the block is imported, we call the *InsertEBTree* function to insert the identifier and version of all transactions in this block into the EBTree. Besides, since all transactions in one block are imported and processed in batches in Ethereum client, we postpone the commit (the third step of insertion) until all transactions in one block are added into the EBTree.

[1] https://github.com/julia2804/go-ethereum/tree/mimota-0.1

Figure 5 shows the average insertion time of one block and one transaction on different data size. According to the experiment results, the average insertion time of block and transaction are short even the number of transaction reaches to 15,362,853. Since loading data from cache is much quicker than loading from disk, the insertion time could be greatly influenced by the cache hit ratio, which can explain the fluctuation of insertion time in picture.

There are two main reasons for the high efficiency of average insertion time: 1) Since the commit is postponed, once the first transaction in block is inserted, we can get a path from root to leaf node *tle* as the Figure 3 shows. If the following transaction in the same block is also inserted to the *tle*, the insertion time of this transaction is tiny, which will apparently decrease the average insertion time. Even though the paths are not completely identical, there are at least one common node (root) for all transactions. Since the depth of EBTree is lower than 5, the time of finding the *tle* for *tn* decreases apparently. Thus, the average of insertion time also will reduce. 2) After all transactions in one block are inserted, all dirty nodes that changed in insertion are written into LevelDB in batches, which will highly decrease the average commit time of block and transaction.

In the whole, the average insertion time for a block with 200 transactions takes up only 2s when there are 15,362,853 transactions indexed. Taking account of the time for mining block and synchronizing data from network, the insertion of EBTree has little influence on the other functions of Geth-EBTree.

5.2 Efficiency of Searches in EBTree

To test the efficiency of searches, we call the APIs of top-k, range, equivalent search, and record the response time.

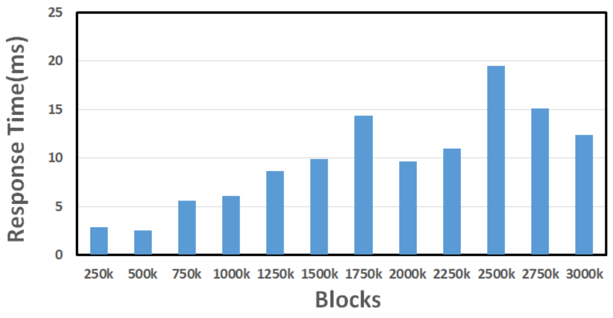

Figure 6: Efficiency of Top-1000 search

The Figure 6 indicates the performance of Geth-EBTree on top-k search. For the convenience of analysis and frequency of usage, we set k to 1000. Apparently, the response time of top-k search increases slowly as the number of transactions goes up. The response time is still relatively short even when there are more than 15,000,000 transactions. The main reason for it is: there are few transactions whose value is in top-k transaction value. For example, there are only 1,634 transaction identifiers in the result set of top-k search. Thus, the data size of leaf nodes scanned is small. In the whole, the short response time illustrates the high efficiency of top-k search.

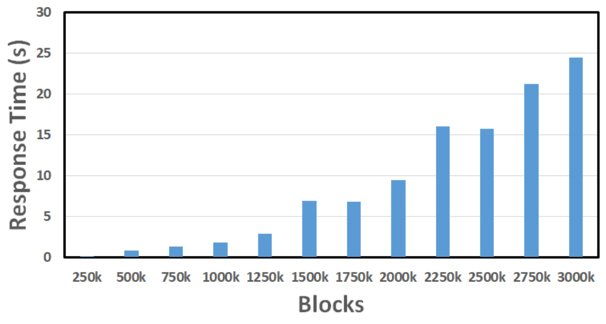

Figure 7: Efficiency of range (1 $\sim$ 1001 ether) search

The cost of range search is described by Figure 7. We choose the common transaction value, 1 ether, as the begin of range, and test the response time for range r (1 $\sim$ 1001 ether), which covers the common value of most transactions. According to the experiment results, considering of the large size of result set, EBTree shows relatively great performance on range search.

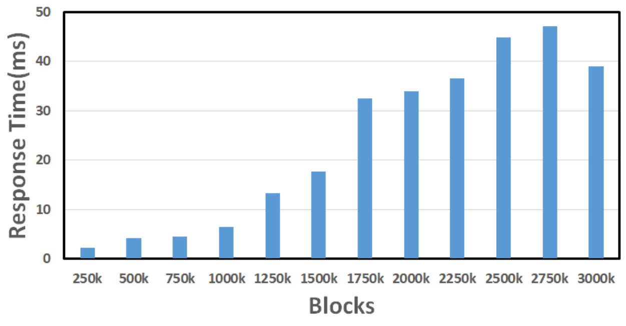

Figure 8: Efficiency of Equivalent Search for 1 ether

Figure 8 depicts the efficiency of equivalent search. Apparently, the response time of equivalent search is nearly proportion to $log_p n$. Besides, the response time of equivalent search is relative short since the result set of it contains more than 70,000 transaction identifiers.

5.3 Storage Space Consumption

To evaluate the storage space used by the EBTree, we measure the data size of Geth1.8 and Geth-EBTree, as the Figure 9 shows. The size of EBTree grows slowly as the number of blocks grows in the whole. Compared to the storage space used by the Geth1.8, the EBTree takes up little storage space. According to the result of above experiments, EBTree can provide high search efficiency with relatively small cost of storage space and insertion time. In the meanwhile, since the

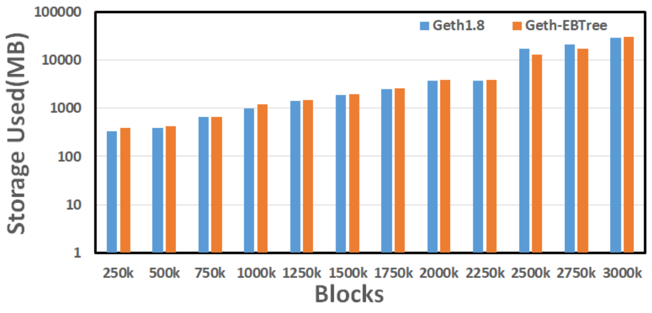

Figure 9: Efficiency of Storage

Ethereum network produces a new block every 15 seconds, the time needed by inserting data into EBTree has few influence on the whole performance of Ethereum client.

6 CONCLUSION

In this paper, we analyze the characteristics of Ethereum storage model, and propose EBTree, the index for Ethereum blockchain data. The nodes of EBTree are stored in LevelDB separately, which leads the small usage of memory. We implement EBTree based on Geth1.8, depict the new storage model of it and provide APIs for top-k, range and equivalent search. The result of experiments shows the high efficiency and small storage usage of EBTree on the Ethereum blockchain data. To sum up, EBTree enables the efficient top-k, range, equivalent searches with relatively small extra storage and insertion time cost.

For the further study, we plan to focus on completing insertion concurrently in EBTree, since the blockchain data are produced in batches. Meanwhile, we will change the structure of EBTree slightly to support the joint searches on blockchain data.

7 ACKNOWLEDGMENTS

This work is supported by the National Science Foundation of China under grant(No.61572194).

8 REFERENCES

[1] Ramzi Abujamra and David Randall. 2019. Blockchain applications in healthcare and the opportunities and the advancements due to the new information technology framework. *Role of Blockchain Technology in IoT Applications* 115 (2019), 141.

[2] Aitor. [n.d.]. Fomo3D, the most widely used application on the Ethereum network. https://en.todoicos.com/ico-news/fomo3d-the-most-used-on-the-ethereum-network/.

[3] Massimo Bartoletti, Stefano Lande, Livio Pompianu, and Andrea Bracciali. 2017. A general framework for blockchain analytics. In *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 7.

[4] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. 2012. MongoDB vs Oracle–database comparison. In *2012 third international conference on emerging intelligent data and web technologies*. IEEE, 330–335.

[5] Santiago Bragagnolo, Henrique Rocha, Marcus Denker, and Stéphane Ducasse. 2018. Ethereum query language. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. ACM, 1–8.

[6] tomhschmidt BrandonRelay. [n.d.]. Fomo3D, the most widely used application on the Ethereum network. http://erc721.org/.

[7] Johannes Buchmann, Erik Dahmen, and Michael Schneider. 2008. Merkle tree traversal revisited. In *International Workshop on Post-Quantum Cryptography*. Springer, 63–78.

[8] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3 (2014), 37.

[9] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[10] Chris Dannen. 2017. *Introducing Ethereum and Solidity*. Springer.

[11] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*. Springer, 79–94.

[12] Andy Dent. 2013. *Getting started with LevelDB*. Packt Publishing Ltd.

[13] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1085–1100.

[14] Paul DuBois and Michael Foreword By-Widenius. 1999. *MySQL*. New riders publishing.

[15] Harry Kalodner, Steven Goldfeder, Alishah Chator, Malte Möser, and Arvind Narayanan. 2017. BlockSci: Design and applications of a blockchain analysis platform. *arXiv preprint arXiv:1709.02489* (2017).

[16] Aggelos Kiayias, Elias Koutsoupias, Maria Kyropoulou, and Yiannis Tselekounis. 2016. Blockchain mining games. In *Proceedings of the 2016 ACM Conference on Economics and Computation*. ACM, 365–382.

[17] Yang Li, Kai Zheng, Ying Yan, Qi Liu, and Xiaofang Zhou. 2017. EtherQL: a query layer for blockchain system. In *International Conference on Database Systems for Advanced Applications*. Springer, 556–567.

[18] Zupeng Li, Daoying Huang, Zinrang Liu, and Jianhua Huang. 2003. Research of peer-to-peer network architecture. In *International Conference on Communication Technology Proceedings, 2003. ICCT 2003.*, Vol. 1. IEEE, 312–315.

[19] Abram Magner and Wojciech Szpankowski. 2018. Profiles of PATRICIA tries. *Algorithmica* 80, 1 (2018), 331–397.

[20] Ahmed S Musleh, Gang Yao, and SM Muyeen. 2019. Blockchain applications in smart grid–review and frameworks. *IEEE Access* 7 (2019), 86746–86757.

[21] Pivota. [n.d.]. RabbitMQ is the most widely deployed open source message broker. https://content.pivotal.io/rabbitmq.

[22] Rüdiger Schollmeier. 2001. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings First International Conference on Peer-to-Peer Computing*. IEEE, 101–102.

[23] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. 2018. Forkbase: An efficient storage engine for blockchain and forkable applications. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1137–1150.

[24] XU Wen-Qi, HUANG Xiao-Ju, ZHAO Li-Mei, and GONG Xue-Qing. 2018. EthGuide: Search and analysis system of the Ethereum. In *2018 National Database Conference*. CCF.

[25] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[26] Huang XiaoJu, Xu WenQi, Zhang Tao, and Gong XueQing. [n.d.]. Blockchain-based Personal Information Management. *Software Engineering* 21, 10 ([n. d.]), 34+38–41.

[27] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vChain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 141–158.

[28] YUAN Yong and WANG Fei-Yue. [n.d.]. Blockchain: The State of the Art and Future Trends. *ACTA AUTOMATICA SINICA* v.42, 04 ([n. d.]), 3–16.

[29] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. 2018. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services* 14, 4 (2018), 352–375.