Handling Conditional Queries on Hyperledger Fabric Efficiently

Tianlu Yan[1], Wei Chen[1,2], Pengpeng Zhao[1], Zhixu Li[1], An Liu[1], and Lei Zhao[1(✉)]

[1] School of Computer Science and Technology, Soochow University, Su Zhou, China
tlyan@stu.suda.edu.cn,
{robertchen,ppzhao,zhixuli,anliu,zhaol}@suda.edu.cn
[2] Institute of Artificial Intelligence, Soochow University, Su Zhou, China

Abstract. As a popular consortium blockchain platform, Hyperledger Fabric has received increasing attention recently. When conducting queries that meet some specific conditions on such platform, we need to search ledger data which usually has multiple attributes. Although efficiently handling conditional queries can be leveraged to support various use-cases, it presents significant challenges as data on Hyperledger Fabric is organized on file-system and exposed via limited API. To tackle the problem, we propose the following novel methods in this paper. In the first one, we use all conditions of the query to create composite keys before executing it. To further improve the performance of conditional queries on Fabric, we build an index called AUP in the second method, where we also study the update of AUP during transactions. The extensive experiments conducted on the real-world dataset demonstrate that the proposed methods can achieve high performance in terms of efficiency and memory cost.

Keywords: Hyperledger Fabric · Ledger data · Conditional queries

1 Introduction

In recent years, blockchain technologies have attracted wide attention and been used in many real applications. This is because they get rid of the centralized storage and can guarantee the data security. A blockchain is a shared, distributed ledger that records transactions between different nodes in a verifiable and permanent way where nodes do not trust each other [18]. Each node in the blockchain network holds the same ledger which contains multiple blocks. A block usually has a list of transactions and encloses the hash of its immediate previous block, where transaction data can be saved in a ledger only after it has passed a series of validations. Note that, blockchain network can be divided into three categories, namely private network, public network and consortium network. In a public network, anyone can join the network to perform transactions. In a private network, there are only a limited range of participating nodes; the

R. Cheng et al. (Eds.): WISE 2019, LNCS 11881, pp. 48–62, 2019.
https://doi.org/10.1007/978-3-030-34223-4_4

access of data has strict rights management, and only participants have the write permission. The consortium chain is available for participants of a specific group. It internally specifies multiple pre-selected nodes as billers, and the generation of each block is determined by all pre-selected nodes. The consortium network is suitable for enterprise applications, each node in the network can be owned by different organizations, and enterprises can integrate the values of multiple systems without having to bring in a trusted third-party.

Hyperledger Fabric [4] is an enterprise-grade and open-source consortium blockchain platform. Like many other blockchain systems (e.g., Ethereum [3], Parity [6]), it divides data into two states: current and historical states. Data is ingested on this system in form of key-value pairs. For a given key, the latest pair is called current state and others are called historical states. Two typical databases in the system are StateDB and HistoryDB. StateDB includes the collection of current states for all keys. HistoryDB includes the collection of historical states for all keys and can be used to quickly locate the position of data in ledger. The historical data is distributed across a large number of blocks on file-system, which leads to the low efficiency of a query with multiple conditions (We refer to it as conditional query in this work). This is because, given a key, the Hyperledger Fabric will return all the historical data of it, based on which we can get the results meeting the given conditions, during an API call.

Obviously, an efficient method is necessary to conduct the conditional query in aforementioned case. Note that, although existing studies have made great contributions in blockchain query [12,13,20,21], the two main techniques, granular access control and indexes constructed based on StateDB, proposed by them can not be directly used to efficiently handle conditional queries on blockchain. This is because, on one hand, nodes are authorized to join in the Hyperledger Fabric network, then there is no need to create additional granular access control for it; on the other hand, it is time consuming to query the whole ledger data before updating the index. Assuming that a user executes a conditional query containing multiple conditions, the conventional query methods need to return all data meeting the first condition and then filter the data according to other conditions, which leads to large time cost. Additionally, conventional methods usually bring a lot of data redundancy, which is demonstrated in Sect. 6. Having observed these weaknesses, we propose the following novel methods, i.e., CCK and AIM. In the first one, we create a composite key for the given query based on the associated conditions of it. Then, we use the composite key to create a new key-value pair before executing data insertion, which can avoid the filtration of historical data. In the second one, to solve the data redundancy problem of the first method, we build an index called AUP for HistoryDB based on LevelDB [5], and the value of each key in AUP consists of corresponding keys of current states.

Considering a use-case, an author α publishes a publication p in a venue v, a key-value pair $<\alpha, (v, o)>$ is inserted into Hyperledger Fabric ledger, and o denotes the other information of the publication, such as title, time and URL. We are interested in querying all publications that are published in the venue

v and belong to the author α. In the first method CCK, we use α and v to create composite key (α, v). By this way, we convert the above key-value pair to $<(\alpha, v), o>$. Based on this method, the processing of filtering publications that belong to α but are not published in v can be avoided. However, we need to create multiple key-value pairs for the publication with multiple authors in this method, which leads to the problem of data redundancy. To solve it, in the second method AIM, we build AUP to record all authors having relationships with the publication to be stored. The key-value pairs in AUP are in the form of $<(\alpha, v), \varepsilon(S_\alpha)>$, where S_α represents all authors of the publication p, and $\varepsilon(S_\alpha)$ denotes all authors that have co-authored with α in history. While inserting a new key-value pair $<(S_\alpha, v), o>$ into blockchain, it inserts $<S_\alpha, "">$ into HistoryDB firstly, and then create $<(\alpha, v), \varepsilon(S_\alpha)>$ in AUP for each author in S_α.

In this study, we have designed novel methods to conduct conditional queries on Hyperledger Fabric with high performance. To sum up, we make the following contributions.

- We are the first to study the problem of efficiently handling conditional queries on Hyperledger Fabric.
- To avoid the process of filtering candidates, we propose the method CCK. To tackle the data redundancy problem brought by CCK, we build an index AUP in the second method AIM.
- We conduct extensive experiments on DBLP, and the results demonstrate that the proposed approaches can achieve high performance in terms of efficiency and memory cost.

The rest of this paper is organized as follows. In Sect. 2, we brefily view existing work related to the research of blockchain. Section 3 presents the background of Hyperledger Fabric. In Sect. 4, we formulate the problem and present notations used in this work. We introduce the proposed methods in Sect. 5 and report the experimental results in Sect. 6. This paper is concluded in Sect. 7.

2 Related Work

Though blockchain analysis is an emerging area, it has received significant attention and a lot of studies have been made on it. These studies are mainly divided into two categories: security and performance. In terms of security, [14] makes a survey of blockchain security issues and challenges, [15] discusses the applicability of blockchain to intrusion detection, and identifies open challenges. There is also a lot of work focused on the performance of blockchain, including [11,17,18]. They mainly concentrate on realizing higher throughputs and lower latencies by using different consensus algorithms, encryption methods. In [8], authors analyze how fundamental and circumstantial bottlenecks in Bitcoin [1] limit the ability of its current peer-to-peer overlay network to support substantially higher throughputs and lower latencies.

2.1 Performance Modeling of Blockchain Networks

The authors of [19] contrast PoW-based blockchains to those BFT-based state machine replication and discuss proposals to overcome scalability limits and outline key outstanding open problems in the quest for the "ultimate" blockchain fabric(s). In [10], they first describe BLOCKBENCH, which is the first evaluation framework for analyzing private blockchains and serves as a fair means of comparison for different platforms and enables deeper understanding of different system design choices, and then they use BLOCKBENCH to conduct comprehensive evaluation of three major private blockchains: Ethereum, Parity and Hyperledger Fabric. They measure the overall performance of the platforms and draw conclusions across the three platforms. [9] is similar to [10], they discuss several research directions for bringing blockchain performance closer to the realm of databases. Zheng et al. [22] provide an overview of blockchain architecture firstly and compare some typical consensus algorithms used in different blockchains.

2.2 Performance Evaluation of Hyperledger Fabric

In existing work, [7] introduces the design and the architecture of Hyperledger Fabric, and presents the performance of a single Bitcoin like crypto currency application on Fabric, called Fabcoin, which uses CLI command to emulate client instead of using a SDK. [12,13,20,21] pay more attention to how to efficiently handle queries in the blockchain platform. [20,21] handle the problem of flexible queries by using granular access control, both of them improve performance by changing encryption methods. [12,13] are the most similar work to our queries, they both propose two method to processe temporal queries on Fabric.

In spite of the great contributions made by the aforementioned studies, none of them consider conditional queries on Fabric. To tackle the problem, we propose two methods in this paper, i.e., composite key based method CCK and AUP index based method AIM, and details are presented in Sect. 5.

3 Background

A Hyperledger Fabric network contains peer nodes, ordering service nodes and clients. A peer node in the network of Fabric is divided into an endorsing node or a committing node. The endorsing node executes the chaincode (a.k.a. smart contract [16]) logic to endorse a transaction, but the committing node does not has the chaincode logic. Although they are different in this point, both of them maintain the ledger in a file system. An ordering service node participates in the consensus protocol and the process of block generation. The client can initiate a transaction proposal to invoke a chaincode function, which can perform read and write operations on shared ledger data by defined ledger APIs. Further, the transaction flow in Hyperledger Fabric consists of 4 phases, (1) Endorsement Phase - simulating the transaction on endorser nodes and collecting the state

changes; (2) Orderering Phase - ordering transactions through a consesus protocol; (3) Validation Phase - verifying the block signature and all transactions in a block; and (4) Commitment Phase - committing valid transaction data to the ledger.

3.1 Data Storage Structure

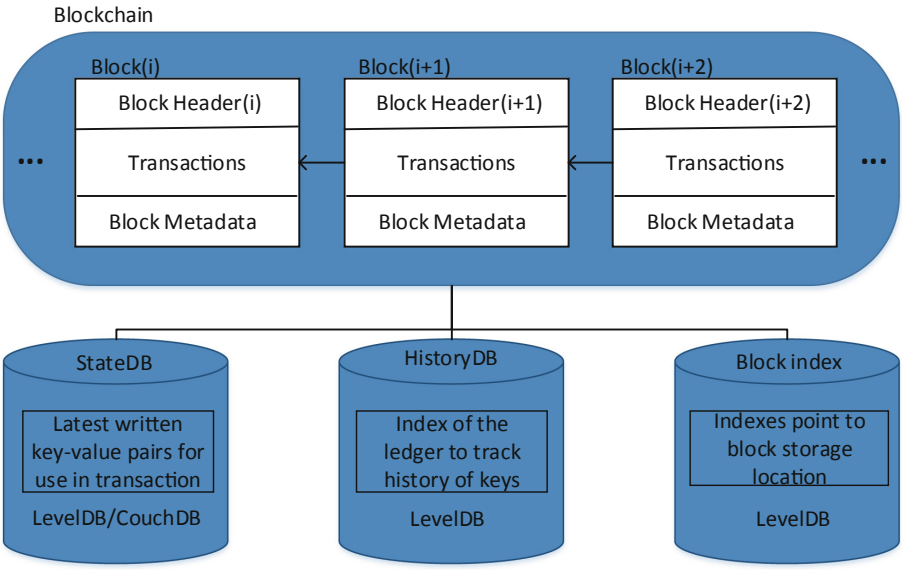

Fig. 1. The structure of data storage in a single-chain.

In Fabric, all valid transactions are stored in blocks, and all blocks are stored in the file system. A simple structure of single-chain data storage is presented in Fig. 1. It contains StateDB, HistroyDB and block index. The StateDB stores the current state of each key and supports LevelDB and CouchDB [2]. The HistoryDB stores the historical state of each key. It records the change of each key in StateDB and is implemented by LevelDB. In fact, it does not store the real value of each key and can be used to quickly locate the position of transaction in the block. Hyperledger Fabric provides a variety of block indexing methods. The content of the block index is the file location pointer, which consists of three parts: the file number, the offset within the file, and the number of bytes occupied by the block. The block index can be used to quickly find the position of blocks.

If we want to add a new state or change the current state of a key, we need to initiate a transaction proposal, executing which successfully, a new key-value pair will be added to a block. The value of the key in StateDB is changed, but the previous key-value pair is still stored in the ledger if it had the value of

the key before. Additionally, a new key-value pair will also be inserted into the HistoryDB.

3.2 Accessing Historical States

Hyperledger Fabric provides specific APIs, such as **GHFK** and **CK**, which are used in our proposed methods CCK and AIM.

GetHistoryForKey(k) (GHFK [13]**)**: This is an API provided by Hyperledger Fabric to access the historical states. For a given key k, this call returns all the past states of key k in the history.

CreateCompositeKey(ob, ks) (CK): This is an API provided by Fabric to combine the given attributes ks and object type ob to form a composite key, which can be used as a key to access historical states.

Specifically, when initiating a transaction proposal to get historical states of a given key k, we need to execute a GHFK call. During the execution of the GHFK call, it retrieves all keys in HistoryDB and each key is start with k firstly. Then it analyses all these keys to get the list of block numbers and transaction numbers. Next, it queries the block index to get the location of blocks and then deserializes all blocks to access transaction data according to transaction numbers. Finally, it extracts out all the values. That is to say, the GHFK call needs to retrieve the historical data from multiple blocks and returns an iterator in the end. The more values accessed through this iterator, the larger the number of blocks that need to be deserialized.

4 Problem Statement

In this section, we present all the notations used throughout the paper in Table 1, and then we formulate the problem.

In Fabric, handling conditional queries requires to deserialize blocks that satisfy all query conditions. For example, in DBLP, given an author α and a venue v, when we want to get all publications that belong to the author α and published in venue v, we need to deserialize blocks that satisfy these two conditions: (1) the block contains a transaction which ingests a key-value pair with key equals to α; (2) this pair describes a publication which is published in the given venue v.

Currently, abovementioned conditional query is time-consuming on Fabric, as such operation is not directly supported by Fabric. If intending to query all publications that meet those conditions, we firstly need to query all publications belong to the given author. During this process, we need to deserialize multiple blocks. Then we still need to filter publications according to the venue. Therefore, some deserialized blocks are useless. Larger the number we need to filter, more time the operation will spend. Besides, if we create a key-value pair for each author of a publication, it will lead to a large number of redundancy, since a publication usually has multiple authors and Fabric does not provide any

Table 1. Definitions of notations.

Notation	Definition
α	An author of a publication in DBLP
o	The other information of a publication in DBLP
S_α	All authors of one publication of α, $S_\alpha = \{\alpha_1, \alpha_2, ..., \alpha_n\}$
$\varepsilon(S_\alpha)$	All authors of all publications of α
v	The venue of a publication in DBLP
K	The result of creating composite key by calling CK
V	The value of a key in AUP
S_K	The collection of the results of executing CK
$\varepsilon(\alpha)$	The set of publications belonging to author α
$\varepsilon(\alpha, v)$	The set of publications belonging to α and published in v
$\varepsilon(S_\alpha, v)$	The set of publications belonging to S_α and published in v

indexing capability on the data in HistoryDB. Due to the redundancy, it takes a lot of time to ingest the publication on the ledger. However, if we don't create the key-value for each author, we can not get all information of the publication with multiple authors, when we only know an author.

Problem Formulation. Given a query, which contains multiple conditions, our goal is to obtain values that satisfy all conditions by conducting the query with the proposed methods on Fabric.

5 Proposed Methods

In this section, we present three methods to execute conditional queries and describe problems encountered during execution. The second method CCK is designed based on composite keys to avoid filtration process and the third method AIM can reduce redundancy by creating index. In order to better explain the proposed methods, we discuss the details of them based on DBLP.

5.1 Baseline Method

In this subsection, we present our baseline method for executing conditional queries on Fabric.

For each publication in $\varepsilon(S_\alpha, v)$, when we want to insert it into ledger, firstly, we need to obtain all authors in $S_\alpha = \{\alpha_1, \alpha_2, ..., \alpha_n\}$, and then the client initiates n transaction proposals to save this publication. Given a query associated with an author α and a venue v, to search all publications belonging to α and published in venue v, we firstly executes a GHFK call, then obtain the set $\varepsilon(\alpha)$. Next, we still need to remove all publications that are not published in venue v from $\varepsilon(\alpha)$. Finally, the remained publications in $\varepsilon(\alpha)$ are the results of

the query. Note that, with the increase of the number of publications that are not published in the venue v, more publications should be removed, which leads to a lot of time cost.

5.2 Composite Key Based Method CCK

To address the problem of the baseline method, we design a novel method CCK, the details of which are discussed as follows, based on composite key.

For each publication in $\varepsilon(S_\alpha, v)$, we use each author in $S_\alpha = \{\alpha_1, \alpha_2, ..., \alpha_n\}$ and a venue v to create a composite key K by calling CK firstly. Then, we create n composite keys for this publication and invoke n transactions to save it. Based on these composite keys, we can conduct the following query. For example, given a query q associated with an author α and a venue v, with the goal of obtaining all publications belonging to α and published in venue v. We firstly use the author α and venue v to create a composite key K, then execute a GHFK call, during which the key K will be compared with all composite keys generated in CCK. Note that, each GHFK call precisely accesses those blocks that contain corresponding publications belonging to α and published in venue v, on ledger. Finally, we can directly get all publications $\varepsilon(\alpha, v)$.

Compared with the baseline method, CCK is more efficient to query all publications that satisfy all conditions, since the filtration process has been avoided. However, in CCK, the number of transactions to be invoked should equal to the number of authors in a publication. That is to say, we have to save the same publication multiple times, which results in massive redundancy.

5.3 AUP Index Based Method AIM

In this part, we build an index AUP to solve the problem of redundancy. For each publication in $\varepsilon(S_\alpha, v)$, we use each author α in $S_\alpha = \{\alpha_1, \alpha_2, ..., \alpha_n\}$ and venue v to create composite key K by calling CK. As an author may publish multiple publications in a same venue, the value V of each K is also a composite key, we create it with the Algorithm 1. The composite key consists of corresponding keys of current states. We create a key-value pair for each author α in $S_\alpha = \{\alpha_1, \alpha_2, ..., \alpha_n\}$ and insert it into AUP. Although a publication may belong to multiple authors, we only need to save the same publication one times, and then we invoke a transaction to save the publication. We do not use the CK as the first parameter is meaningless in this method. In Algorithm 1, we use a separator *'#'* to split each key. For example, we add *'#'* between *key1* and *key2*, and the final result is in the form of *key1'#'key2*. The reason for choosing *'#'* as a separator is: there is no '#' in the names of author and venue. By this way, we can separate keys accurately.

Considering a example, when a new publication data need to be saved to ledger, we first use each author in S_α and v to create composite key K by calling CK, then initiate a transaction proposal to commit data to ledger. Next, we query the AUP to get the value V of key K. If V is empty, we use all authors of the publication as a key to create composite key by Algorithm 1, which used as

Algorithm 1. Creating a Composite Key

Input: keys of current states(key$_1$,key$_2$,...,key$_i$)$(0 \leqslant i \leqslant n)$
Output: composite key: V

```
Receive keys;
namespace ← '#';
for i=0 to n do
    if key_i does not contain '#' then
        V ← V+namespace+key_i;
    end
end
Return V;
```

Algorithm 2. Splitting Multiple Values

Input: V(a value in AUP) and L(the length of the value)
Output: S_K: a collection of the splited keys

```
Receive value;
namespace ← '#';
index ← 0;
for i=0 to L do
    if Value[i]==namsepace then
        components ← append(components,Value[index:i]);
        index ← i+1;
    end
    append the components into S_K;
end
Return S_K;
```

the value of K. Otherwise, we split the value V with the Algorithm 2, where we still use *'#'* as separator and get the collection S_K. If the new key is different from any key in S_K, we append the new key to S_K, and then we use S_K to create new composite key NV by Algorithm 1 and put this new key-value pair $<K, NV>$ into AUP. Otherwise, we don't need to do anything. Finally, When the transaction is completed sucessfully, the publication data is saved to ledger.

The process of a conditional query is shown in Algorithm 3 explicitly. Firstly, we use the author α and venue v to create composite key K by calling CK. Then we use K to query AUP and get the value V. Next, we need to separate V and get the collection of keys S_K. Finally, we execute a group of GHFK calls based on the keys in S_K and get the collection $\varepsilon(\alpha, v)$, which is the result that we want to get.

During the design of AUP, we use Mutex in the Go language. Mutex is a commonly used method to control shared resource access, which ensures that only one goroutine can access shared resources at the same time. For example, if we use 4000 goroutines to execute transactions, after one goroutine queries the AUP to get the value of a the given key, another goroutine updates the value of

the key, which will make the value obtained by the previous goroutine incorrect. Then, the incorrect value will lead to the loss of data. To solve the problem, we use Mutex to create the index AUP. When a goroutine writes to the AUP, other goroutines need to wait until the previous goroutine has finished writing.

Algorithm 3. Process of a Conditional Query

Input: an author(α)and the venue(v)
Output: $\varepsilon(\alpha,v)$:All publications belong to α and published in v

```
Receive α and v;
K← use α and v to create composite key by calling CK;
V← query AUP with K;
S_K ← split V with Algorithm 2;
L ← get the length of keys;
for i=0 to L do
    call GHFK with the i-th key in S_K;
    append the result of GHFK to ε(α,v);
end
Return ε(α,v);
```

6 Experiment

6.1 Fabric Instance

We use Hyperledger Fabric v1.3 and the implemented network consists of a single organization. The organization contains three nodes, a CA node, an endorsing node and an ordering service node with one public channel avaliable for communication. The endorsing node is configured to use CouchDB as the StateDB. We use Fabric SDK to emulate clients and run the entire system by using docker containers on a server. The server is equipped with 24 Intel(R) Xeon(R) CPU E5-2630 v2 processors at 2.60 GHz, for a total 256 GB of RAM. We keep all nodes turned on and use all default configuration settings to run our experiments.

6.2 System Workload

We carry out our experiment evaluation using DBLP. The total number of publications in DBLP is 4146645. As each publication in DBLP usually has multiple authors, we create a record with the same publication for those authors respectively. Finally, the total number of records is 12508891, in which 8362245 records are redundant. The total number of different authors publishing publications in different venues is 7843756. We divide all these data into 7 groups according to the ratio $r(r=j/i$, i represents the number of publications belong to α, j represents the number of publications belong to α and published in $v)$. Groups are shown in Table 2. In this paper, we measure the performance of methods using the following metrics - (1) Query execution times - time taken to execute the conditional query. (2) Insertion times - time taken to insert data into Fabric ledger. (3) Memory cost - memory size occupied by all data.

Table 2. All groups and the number of members of each group.

Group	1	2	3	4	5	6	7
$r(\%)$	100 - 18	18 - 15	15 - 12	12 - 9	9 - 6	6 - 3	3 - 0
Total number	3218585	274878	421421	512382	612054	1034199	1770226

6.3 Experimental Evaluation

Table 3 shows the performance of three methods: baseline, CCK and AIM. We randomly select 1000 records from each group to execute 1000 queries at a time, which we execute 1000 times and take the average query time as the result. The query time is calculated from the time when the query transaction proposal is initiated until the response information is received.

Table 3. Query time of each method.

Group	Query time of baseline	Query time of CCK	Query time of AIM
1	29.03(s)	12.77(s)	9.57(s)
2	50.14(s)	12.31(s)	9.52(s)
3	55.44(s)	11.73(s)	8.51(s)
4	70.84(s)	11.70(s)	8.27(s)
5	80.92(s)	11.46(s)	7.93(s)
6	109.20(s)	10.85(s)	7.29(s)
7	174.74(s)	9.87(s)	6.07(s)

6.4 Time Cost of Baseline

As we can see from the Table 3, with the ratio r decreases, the baseline method takes more time. This is because as the ratio r decreases, the author we used to query has more publications. When we want to get all the publications that meet the conditions, we need to call the GHFK. The Fabric firstly queries the HistoryDB to get all keys that satisfy the conditions. The key in HistoryDB consists of the key of a current data, block number and transaction number. Then it uses block numbers to query block index to get all blocks and deserializes the content of these blocks. Next, it uses transaction numbers to get transactions and extracts out the values inserted. Finally, the GHFK call returns an iterator and we get values from the iterator. The more values are accessed through this iterator, the more blocks are deserialized. Therefore, given an author, the more publications belong to the authors, the more blocks need to be deserialized, the more time we will take to execute query transaction.

Consider the query in baseline method, it needs to get all blocks that contain publications belong to author α. It hence deserializes all these blocks and need to remove publications that are not published in venue v. As the number of publications that are not published in venue v increases, it needs to deserialize more and more blocks and removes more and more publications that do not satisfy the conditions. The bottleneck of the first method is that to retrieve publications belong to author α and published in venue v, we need to deserialize all blocks containing publications belongs to author α. Larger the number of publications that are not published in venue v, worse is hence the performance of baseline method.

6.5 Time Cost of CCK

The third column of Tabel 3 presents the performance of CCK. When we execute queries in group 1, CCK takes 12.77s which takes 16.26s less time than the baseline method. When we execute queries in group 3, CCK takes 11.73s which takes 43.71s less time than baseline. As the ratio decreases, the performance of CCK method becomes better. This is because with the decrease of ratio, the number of publications belong to the author α and published in venue v becomes smaller, and the number of blocks that we need to deserialize also becomes smaller. We are able to achieve this improvement by using CCK because we exactly know which block contains publications belong to author α and published in venue v. That is to say, we just need to get blocks that contain publications belong to author α and published in venue v. This effect becomes more severe, when we execute queries in group 7. Considering the case when an author has total x publications, in which y publications published in venue v and the data of each publication is stored in different blocks. When we execute queries with the baseline method, we need to deserialize x blocks and remove $x - y(\mathrm{x} \geq \mathrm{y})$ publications from the result. However, if we use CCK, we only need to deserialize y blocks. The larger $x - y$, the higher the performance of CCK. This is equivalent to the smaller ratio, the higher the performance of CCK. The time-cost by using CCK is much smaller than that by using the baseline method.

6.6 Time Cost of AIM

We next analyze the time-cost of using AIM to execute conditional queries, it is not much different from CCK. This is because in the AIM, we also create composite key and we exactly know which block contains the data that meets our conditions. So the number of block we need to deserialize is same. However, CCK has a big problem, it brings a lot of redundancy. We need to use each author in a publication and the venue of the publication to create composite key (α, v), and we need to take (α, v) and the other information o as a key-value pair to insert into the ledger. So if a publication has n ($n \geq 1$) authors, it will generate n key-value pairs and wherein $n - 1$ are duplicates, which lead to the size of ledger created by using CCK is bigger than the ledger created by using AIM and the cardinality of the ledger data that performs conditional

queries becomes larger. That is the reason why AIM is a little better than CCK in query performance. Besides, the redundancy causes us to spend a lot of time inserting these key-value pairs into ledger. In our experiment, we ingest a publication in one transaction. So the total number of transaction is 12508891 by using CCK and baseline methods, and we execute these transactions with 4000 goroutines. Both baseline method and CCK method cost more than 13 h to finish these transactions. However, when we use AIM method, the total number of transaction is 4146646 and it costs 5 h 29 m to finish these transaction. By using AIM method, we save more than 2 times time, which we can see from Table 4. We build the index during the process of a transaction. In fact, the data is continuously streaming in. If we do not build the index during the process of a transaction, when we execute queries, we may can not get the new data immediately because it has not yet been saved to the index. Beside, if we do not use this method, when we want to construct index, we will need to querying ledger before, which will cost a lot of time.

6.7 Memory Cost of the Three Methos

In addition, by constructing the index AUP, we also save data storage space. Specifically, let us use $|P|$ and $|I|$ to denote the average size of a transaction data in block and the key-value pair in AUP ($|P| > |I|$) respectively. In baseline method and CCK, the total size of all data is $12508891|P|$. In AIM, the total size of all data is $4146646|P| + 2234392|I|$. The difference between these two values is $8362245|P| - 2234392|I|$, and $8362245|P| - 2234392|I| > 0$. Therefore, AIM saves more data storage space than baseline method and CCK.

Table 4. The data insertion time of different methods.

Methods	Baseline	CCK	AIM
Transaction number	12508891	12508891	4146646
Data insertion time	13 h 8 m	13 h 12 m	5 h 29 m

6.8 Analysis

From the above three methods, we can see that the AIM has the best performance. It solves the problem of redundancy, improves the efficiency of queries and data insertion. Then, we get two conclusions. Firstly, when we execute conditional queries, and the key which we want to use has a large number of unrelated values need to be removed, the best method is to use all conditions to create a composite key. Then we can use this composite key to execute queries, which can help deserialize a small number of blocks and directly find blocks containing values that we want to get without the process of filtration. Secondly, when multiple keys have a same value, we can create index to reduce the time of data insertion and reduce redundancy. Just like the use-case in our experiment,

multiple authors have a same publication, we reduce the time of inserting the publication into the ledger by creating an index AUP. By combining the method of creating composite key and building index, the performance of both queries and inserting data have a significant improvement.

In addition, methods presented in this paper can also be generalized to other conditional queries. For example, we can use the proposed methods to get a medical history of a patient in a certain department in the medical field.

7 Conclusion and Future Work

In this paper, we present three methods to handle conditional queries on Hyperledger Fabric. We use the first method as our baseline, both CCK and AIM easily outperform the baseline. We benchmark these three methods and we also conduct a comprehensive study to understand and analyse the conditional queries performance on Hyperledger Fabric by creating composite keys and building an index. Besides, the process of building index is included in an transaction. Not only does it saves more time during the process of insertion data, but also we can get data in a timely manner.

In our future work, we can further improve the performance of conditional queries in Hyperledger Fabric by using different methods of creating composite key and building index. As the static structure of LevelDB consists of six main parts and keys with the same prefix are adjacent in the file.

Acknowledgements. This work was supported by the National Natural Science Foundation of China (Grant No. 61572335, 61572336, 61902270), and the Major Program of Natural Science Foundation, Educational Commission of Jiangsu Province, China (Grant No. 19KJA610002), and the Natural Science Foundation, Educational Commission of Jiangsu Province, China (Grant No. 19KJB520052, 19KJB520050), and Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

1. Bitcoin. https://bitcoin.org/en/getting-started/. Accessed 10 June 2019
2. Couchdb. https://couchdb.apache.org/. Accessed 10 June 2019
3. Ethereum. https://www.ethereum.org/. Accessed 10 June 2019
4. Hyperledger fabric. https://www.hyperledger.org/projects/fabric. Accessed 10 June 2019
5. LevelDB. https://github.com/syndtr/goleveldb/. Accessed 10 June 2019
6. Parity. https://www.parity.io/. Accessed 10 June 2019
7. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, p. 30. ACM (2018)
8. Croman, K., et al.: On scaling decentralized blockchains. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 106–125. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_8

9. Dinh, T.T.A., Liu, R., Zhang, M., Chen, G., Ooi, B.C., Wang, J.: Untangling blockchain: a data processing view of blockchain systems. IEEE Trans. Knowl. Data Eng. **30**(7), 1366–1385 (2018)
10. Dinh, T.T.A., Wang, J., Chen, G., Liu, R., Ooi, B.C., Tan, K.L.: Blockbench: a framework for analyzing private blockchains. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1085–1100. ACM (2017)
11. Gervais, A., Karame, G.O., Wüst, K., Glykantzis, V., Ritzdorf, H., Capkun, S.: On the security and performance of proof of work blockchains. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 3–16. ACM (2016)
12. Gupta, H., Hans, S., Aggarwal, K., Mehta, S., Chatterjee, B., Jayachandran, P.: Efficiently processing temporal queries on hyperledger fabric. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1489–1494. IEEE (2018)
13. Gupta, H., Hans, S., Mehta, S., Jayachandran, P.: On building efficient temporal indexes on hyperledger fabric. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 294–301. IEEE (2018)
14. Lin, I.C., Liao, T.C.: A survey of blockchain security issues and challenges. IJ Netw. Secur. **19**(5), 653–659 (2017)
15. Meng, W., Tischhauser, E.W., Wang, Q., Wang, Y., Han, J.: When intrusion detection meets blockchain technology: a review. IEEE Access **6**, 10179–10188 (2018)
16. Omohundro, S.: Cryptocurrencies, smart contracts, and artificial intelligence. AI Matters **1**(2), 19–21 (2014)
17. Pongnumkul, S., Siripanpornchana, C., Thajchayapong, S.: Performance analysis of private blockchain platforms in varying workloads. In: 2017 26th International Conference on Computer Communication and Networks (ICCCN), pp. 1–6. IEEE (2017)
18. Thakkar, P., Nathan, S., Viswanathan, B.: Performance benchmarking and optimizing hyperledger fabric blockchain platform. In: 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 264–276. IEEE (2018)
19. Vukolić, M.: The quest for scalable blockchain fabric: proof-of-work vs. BFT replication. In: Camenisch, J., Kesdoğan, D. (eds.) iNetSec 2015. LNCS, vol. 9591, pp. 112–125. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39028-4_9
20. Zhang, X., Poslad, S.: Blockchain support for flexible queries with granular access control to electronic medical records (EMR). In: 2018 IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2018)
21. Zhang, X., Poslad, S., Ma, Z.: Block-based access control for blockchain-based electronic medical records (EMRs) query in ehealth. In: 2018 IEEE Global Communications Conference (GLOBECOM), pp. 1–7. IEEE (2018)
22. Zheng, Z., Xie, S., Dai, H., Chen, X., Wang, H.: An overview of blockchain technology: architecture, consensus, and future trends. In: 2017 IEEE International Congress on Big Data (BigData Congress), pp. 557–564. IEEE (2017)