World Wide Web (2021) 24:441–461
https://doi.org/10.1007/s11280-020-00844-5

Handling conditional queries and data storage on Hyperledger Fabric efficiently

Tianlu Yan[1] · Wei Chen[2] · Pengpeng Zhao[1] · Zhixu Li[1] · An Liu[1] · Lei Zhao[1]

Received: 21 March 2020 / Revised: 22 July 2020 / Accepted: 21 September 2020 /
Published online: 14 November 2020

Abstract
As a popular consortium blockchain platform, Hyperledger Fabric has received increasing attention recently. When executing transactions on such platform, it usually costs a lot of time and hardly to achieve high efficiency. Although efficiently handling transactions 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. We tackle the problem in two ways: conditional queries and data storage. In this paper, we propose the following novel methods. To improve the performance of conditional queries on Hyperledger Fabric, we use all attributes of the query to create composite keys before executing it. In order to achieve further performance improvements, we build an index called AUP in the second method, where we also study the update of AUP during transactions. To speed up data storage on Hyperledger Fabric, We create a cache for the data in the block header. 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. Finally, We implement a prototype system.

Keywords Hyperledger Fabric · Conditional queries · Data storage

1 Introduction

In recent years, blockchain technologies have attracted wide attention as they get rid of the centralized storage and can guarantee the data security. As a result, blockchain has been

This article belongs to the Topical Collection: *Special Issue on Web Information Systems Engineering 2019*
Guest Editors: Reynold Cheng, Nikos Mamoulis, and Xin Huang

✉ Lei Zhao
zhaol@suda.edu.cn

Extended author information available on the last page of the article.

widely used in many real applications. 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 [24]. 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 a transaction 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, everyone can join the network to perform transaction. In a private network, there are only a limited range of participating nodes. The access of data has strict right management, where only participants have the write permission. The consortium chain is only 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 [14] is an enterprise-grade and open-source consortium blockchain platform. Like many other blockchain systems (e.g., Ethereum [7], Parity [20]), it divides data into two states: current and historical states. Data in this system is stored in the form of key-value pairs. For a given key, the latest pair is called current state and others are called historical state. Two typical databases in the system are StateDB [23] and HistoryDB [13]. StateDB includes the collection of current states for all keys. HistoryDB includes the collection of historical states for all keys, and each key in it contains the number of a block and the number of a transaction, which 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, which is specifically defined in Definition 1). This is because, given a key, the Hyperledger Fabric will return all historical data of it, based on which we can get the results meeting the given conditions, during an API call. During the process of data storage, it performs multiple verifications, such as: MVCC and VSCC. The explanations of MVCC and VSCC are dispayed in Section 3.3. When performing these verifications, it needs to fetch the data in the block header by deserializing the block. Repeated deserialization of blocks results in a lot of time overhead. So it is significant for us to reduce the number of deserialized blocks.

Obviously, to achieve the widespread use of blockchain-based applications, it is necessary to improve the efficiency of conditional queries for historical data and increase the speed of data storage. Note that, existing studies have made great contibutions in the performance of blockchain. In terms of conditional queries, there are two main techniques: granular access control and indexes constructed based on StateDB. However, they cannot be directly used to efficiently handle conditional queries on Hyperledger Fabric. This is because, on the 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 Section 6. Having observed

these weaknesses, we propose the following novel methods: CCK and AIM. In the CCK, 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 AIM, to solve the data redundancy problem brought by CCK, we build an index called AUP for HistoryDB based on LevelDB [16], and the value of each key in AUP consists of corresponding keys of current states. In terms of data storage, most contributions focus on using different consensus. However, they have not addressed the problem of repeatedly deserializing blocks. In this paper, we propose a novel method CAM. In this method, we create a cache for the data, which need to be validated during data storage, in block header. Then, we can obtain the data in block header through cache directly, without deserializing blocks repeatedly.

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 the blockchain ledger, and o denotes the information of the publication, such as titile, time and URL. We are interested in querying all publications that are published in the venue v and belonging to the author α. Firstly, we need to query all publications belonging 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. In addition, 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 indexing capability on the data in HistoryDB. Specifically, if an publication has n authors, then the publication needs to be stored n times. As a result, a publication is stored multiple times, which causes a lot of redundancy. When the amount of data is large, this redundancy is not negligible. Due to the redundancy, it takes a lot of time to ingest the publication on the ledger. However, if we do not 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.

To tackle these challenges, we porpose two methods. In the first method CCK, we use α and v to create a 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 mutiple authors in this method, which leads to the problem of data redundancy. To solve it, in the second method AIM, we build an index called 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 $<(\alpha, v),o>$ into blockchain, it inserts $< s_\alpha, \text{“”} >$ into HistoryDB firstly, and then creates $<(\alpha, v),\varepsilon(S_\alpha)>$ in AUP for each author in S_α.

In this paper, we add more delicate processing. Compared with [26], we make the following improvements:

- To make the proposed conditional query methods more versatile, we explain them in a general way instead of the previous specific cases.
- To reduce the number of block deserializations, we develop a novel method CAM, where a cache is created to store data that need to be validated in MVCC and VSCC.

– Since Hyperledger Fabric mainly implements functions through chaincode, we introduce the design of chaincode and present the interfaces of the system we implement.

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

(1) We are the first to propose methods to process conditional queries on Hyperledger Fabric historical data.
(2) To avoid the process of removing unrelated results, we propose a method—CCK to execute conditional queries on Hyperledger Fabric by creating composite key. To tackle the data redundancy problem, we design a method—AIM to build an index—AUP for HistoryDB. Furthermore, in order to achieve timely requeries, we update AUP during transactions.
(3) To speed up data storage, we create a cache for data in block header to avoid repeated deserialization of blocks. In addition, we conduct extensive experiments on a real-world dataset DBLP, and the results demonstrate that the proposed approach can achieve high performance in multiple dimensions.
(4) Finally, we implement a prototype system on Hyperledger Fabric to efficiently handle transactions.

The rest of this paper is organized as follows. In Section 2, we brefily view existing work related to the research of blockchain. Section 3 presents the background of Hyperledger Fabric. In Section 4, we formulate the problem and define notations used in this work. We introduce the methods and algorithms proposed in this paper in Section 5. In Section 6, we conduct extensive experiments on a real-world dataset. In Section 7, we show the arichitecture of our chaincode and the client interface of the system. This paper is concluded in Section 8.

2 Related work

Though blockchain is an emerging area, it has received significant attentions. There is a large number of work focuses on it. These studies are mainly divided into two categories: security and performance. In terms of security, [17] makes a survey of blockchain security issues and challenges. Lin et al. [18] discusses the applicability of blockchain to intrusion detection, and identify open challenges. Kang et al. [15] provides APIs to easily enable data privacy in both client code and chaincode. It also supports on-demand, automated auditing based on encrypted data. There is also a lot of work focuses on the performance of blockchain, including [9, 21, 24, 25]. They mainly concentrate on realizing higher throughputs and lower latencies by using different consensus algorithms, encryption methods etc.. In [4], authors analyze how fundamental and circumstantial bottlenecks in Bitcoin 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 [25] constrast 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 [5], authors first describle BLOCKBENCH, which is the first evaluation framework for analyzing private blockchains and serves as a fair means of comparsion 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 blockchain: Ethereum, Parity and Hyperledger Fabric. They measure the overall performance of the platforms and draw conclusions across the three platforms. The problem investigated in [6] is similar to [5], both of them discuss several research directions for bringing blockchain performance closer to the realm of databases. Zheng et al. [29] 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, [1] 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. Gupta et al. [11, 12], Zhang and Poslad [27], Zhang et al. [28] pay more attention to how to efficiently handle queries in the blockchain platform. [27, 28] handle the problem of flexible queries by using granular access control, both of them improve performance by optimizing encryption methods. Gupta et al. [11, 12] are the most similar work to our methods, they both propose two methods to process temporal queries on Fabric. However, they need an extra phase to updata index. Androulaki [1] introduces the impact of block size, CPU, SSD, and RAM disk on blockchain latency and throughput. Arati [2] puts Fabric through different workload sets to study Fabric's throughput and latency characteristics, and customizes a set of benchmark tests for Fabric to adjust different transaction and smart contract parameters and study how they Affects transaction latency. Ankur [22] discusses Fabric from the perspective of database research, and have observed the weakness in the trading channel. Then, the easy-to-understand database concept is transitioned to Fabric. Compared with the original version, the improved version of Fabric significantly increases the throughput of successful transactions. Gao et al. [8] propose the establishment of a Hyperledger-based food trade and traceability system, called Hyper-FTT. By bringing all providers including food storage companies, food processors, and food retailers together to reach an agreement and reach a commercial transaction on the chain. Then, an uninterrupted food supply that can form a chain to provide reliable food tracking, and implementations and experiments have been conducted to evaluate the performance of the proposed demonstration system. Gorenflo et al. [10] is similar to our method CAM, it also uses cache to store data, but it does not implement the whole transaction process.

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 Section 5. To speed up data storage, we propose the CAM method.

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, a committing node

and an orderering node. The orderering node is responsible for creating blocks. The endorsing node executes the chaincode logic [19]). 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 to invoke a chaincode function, which can perform read and write operations on shared ledger data by defining 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; (4) Commitment Phase—committing valid transaction data to the ledger.

3.1 Data storage structure

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 Figure 1. It contains StateDB, HistoryDB and block index. The StateDB stores the current state of each key and supports LevelDB and CouchDB [3]. The HistoryDB stores the historical state of each key. It records the change of each key in StateDB, which is based on 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 varity 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.

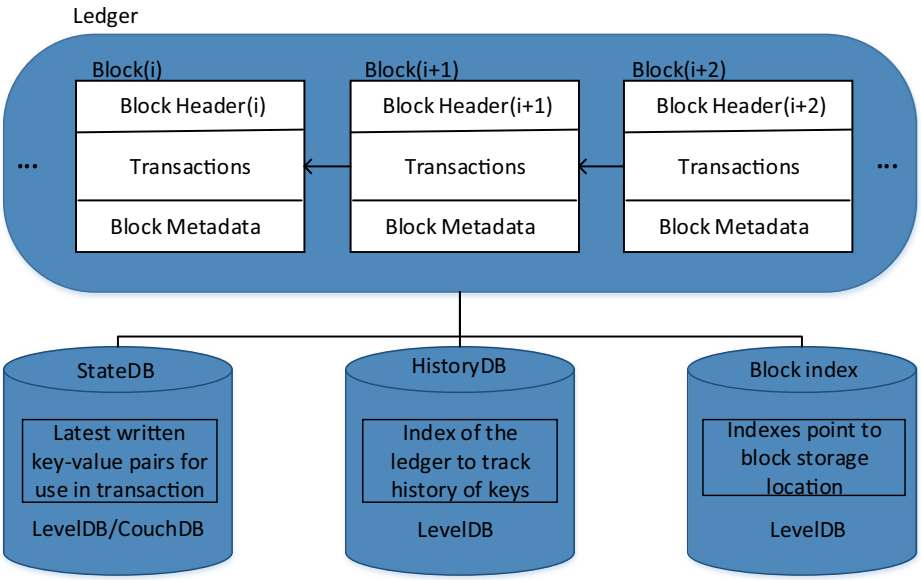

Figure 1 The architecture of ledger

If we want to add a new state or change the current state of a key, we need to initiate a transaction proposal. When executing sucessfully, 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 [12]): This is an API provided by Hyperledger Fabric to access the historical state in ledger. 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 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 accroding 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.

3.3 Validating block information

In Fabric, when a new block is created, it needs to be broadcasted to other peers and those peers will validate the block. During the validation phase, peers will validate the ordering node's signature on the new block firstly. Next, peers will deserialized the block to get all transactions. Finally, each transaction in the new block will go through VSCC validation and MVCC validation.

During VSCC(Validation System Chaincode) validation phase, peers will validate if the endorsement in the transaction is consistent with the endorsement policy. If they are consistent, the verification is passed and the transaction is valid, otherwise the transaction is invalid.

During MVCC(Multi-Version Concurrency Control)validation phase, peers will validate if the version of the key in the endorsement phase is consistent with the version of the key in the current state database. The difference of these two versions means the previous transaction has been modified and is invalid.

The data in the block is encapsulated layer by layer. In order to obtain the data that needs to be verified, the block needs to be deserialized multiple times, which takes a lot of time.

4 Problem statement

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

Definition 1 Conditional Query. Given a query q with multiple query conditions S_c= $(c_1...c_n)(n \geq 1)$, suppose the attribute c_1 corresponding to condition c has multiple values, our goal is to obtain values that satisfy all conditions.

4.1 Problem formulation

In Fabric, handling conditional queries require 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 α, and (2) this pair describes a publication which is published in a given venue v. As we can see from the example, the value of the author attribute is a collection. This is because, an article usually has multiple authors. To get this kind of data, we also need to store it into ledger efficiently firstly.

Currently, when we want to initiate a transaction to store data in the block ledger, during validation phase, it needs to deserialize block repeatedly, which will cost a large number of time. In addition, abovementioned conditional query is time-consuming on Fabric, since Hyperledger Fabric only provides restricted APIs, and does not directly support this operation. Besides, in the process of conditional query, it is necessary to deserialize blocks multiple times, and then filter the data. It is hard to directly obtain the data that meets all the conditions.

Problem Formulation. Given a conditional query, we can obtain values that satisfy all conditions by conducting the query with the proposed methods on Fabric. To obtain the

Table 1 Definitions of notations

Notation	Definition
q	A given conditional query.
c_i	The i-th query condition.
s_{c_i}	Set of values of the attribute corresponding to the query condition c_i
r	The result of the given conditional query
r_{c_i}	Conditional query results that satisfy condition c_i
s_c	The set of conditions in a conditional query
p	The collection of data records
$s^i_{c_j}$	The set of values corresponding to the jth condition of the ith data record
s_v	A data collection formed by splitting the original data
ck	Composite key formed after CK is called
s_{ckvp}	The collection of key-value pairs consisting of ck and p

values, we need to store the value in advance. By using the method we proposed, we can also speed up data storage.

5 Proposed methods

From Section 4 we know that, the conditional query is to obtain all historical values that satisfy all conditions. Besides, to better realize conditional queries, we propose a novel method to speed up data storage. Since this paper is the first to study efficient conditional queries based on Fabric, so in order to solve this problem, we have given three specific solutions, the first of which is the original basic method of designed as a baseline. 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 addition, to speed up data storage, we designs method CAM. In this section, we present proposed methods and describe problems encountered during execution.

5.1 Baseline method

Given a data record p, we split the set of attribute values S_{c_1}=$(c_1^1...c_1^m)(m \geq 1)$corresponding to c_1. We divide the set into several parts equaling to the number of elements contained in the set. We can split the set into m parts if it has m elements, but the other attribute values of the data record p are unchanged. Then we use each value split by s_{c_1} as the key, record p as the value, form key-value pairs. Finally, we form m key-value pairs S_{kvp}=$(kv_1...kv_m)$, and initiate transactions to save the m key-value pairs in the Fabric's block, and the corresponding data are also saved in the state database and the historical database.

When executing query q, we first use c_1 as the key to call GHFK to execute the query. After obtaining a data result set r_{c_1}, we then use the other query conditions in s_c to filter the data, delete the data records that do not meet the conditions, and finally obtain the result r that meets all query conditions.

However, this method requires a data filtering process. The more query conditions that need to be met, the more filtering is required, it also leads to the missing of data obtained by access in the final result. Then with the increase of query conditions, the more blocks are accessed and the more data needs to be filtered, eventually the conditional query needs to take a lot of time to return results.

5.2 Composite key based method CCK

To address the problem of the baseline method, we design a novel method CCK, Algorithm 1 introduces the method and the detailed explanations are as follows.

In the storage phase, given a set of data records $p = (p_1...p_t)(t \geq 1)$, for each data record $p_i(1 \leq i \leq t)$, as in the first step in the baseline method, we split the attribute value set $s_{c_1}^i = (c_1^1...c_1^m)$ $(m \geq 1)$ into m parts, and the other attribute values of the data record p_i remain unchanged firstly. Then, we use each value split in $s_{c_1}^i$=$(c_1^1...c_1^m)$ and the attributes corresponding to the query conditions other than c_1^j $(1 \leq j \leq m)$ to construct a composite key by calling CK, which finally constitutes a set of composite keys with m elements s_{ck}^i=$(ck_1...ck_m)$. Next, we use each element in s_{ck} as a key, and record p_i as a

value to create a key-value pair. Then, the key-value pairs constructed in the baseline method becomes $s_{ckvp} = (c_{kv}^1 ... c_{kv}^m)$, and we initiate m transactions to save the m key-value pairs in the block ledger. Fianlly, we repeat the above steps t times to save all collections to the block ledger.

Algorithm 1: The method of creating compiste key CCK.

Input: A collection of data records p, which contains t elements($t \leq 1$). A query q contains n query conditions($c_1 ... c_n$)($n \leq 1$)

Output: Query result r

```
The stage of data storage;
Access the collection of data records p;
for i = 1 to t do
    Extract and split the attribute value corresponding to the query condition c_1 in the
    data record p_i. The result after splitting is (c_1^1 ... c_1^m);
    for j = 1 to m do
        Use c_1^j and other attribute values corresponding to the query conditions as ks,
        and call CK to create composite key ck_j;
        Use the created composite key ck_j as the key and p_i as the value to form the
        key-value pair ckv_j;
        Initiate a transaction proposal and save the key-value pair ckv_j to the block
        ledger;
    end
end
The stage of condition query;
Get conditional query q;
Set all query conditions (c_1^1 ... c_1^m) in conditional query q as ks, and call CK to create
composite key ck;
Use ck as key and call GHFK to access data in block. The result returned is r;
return r
```

In the query phase, when executing conditional query q, firstly we use all conditions in q to create composite key ck by calling CK. Then, we call GHFK(ck) to execute conditional query to get the result r. The returned set r meeting all query conditions is the final result.

Although the result r obtained by calling GHFK(ck) meets all conditions and CCK can avoid the data filtration, both baseline method and CCK have a common problem. Since the attribute value corresponding to condition c_1 is a collection of data, in order to achieve the flexibility to execute a query according to the corresponding key, the record p needs to be stored multiple times. As many elements in the attribute set corresponds to c_1, we need to save the record p many times, such a storage method brings a lot of data redundancy.

Algorithm 2: The method of building index.

Input: The collection of data records p, which contians t element($t \geq 1$). A conditional query q, which contains n query conditions $(c_1...c_n)$

Output: The query result r

```
The stage of data storage;
Access the collection of data records;
for i = 1 to t do
    Extract and split the attribute value c^i_{c_1} corresponding to the query condition c_1 in the data record p_i, and the result after the split is (c^1_1...c^m_1);
    for j = 1 to m do
        Use c^j_1 and other attribute values corresponding to the query conditions as ks, and then call CK to create the composite key;
        if the value of the key ck_j does not exist in the AUP then
            Save ck_j as key and s^i_{c_1} as values in AUP;
        end
        if the value of the key ck_j exists in the AUP then
            Query the AUP to obtain the value corresponding to c_j, and combine this value with s^i_{c_1} to form a new value ns^j_{c_1} by calling algorithm 3;
            Use ck_j and ns^j_{c_1} to create key-value pairs and save them in AUP;
        end
        Use s^i_{c_1} as a key, record p_i as a value to form a key-value pair, and then initiate a transaction proposal to save it to the block ledger;
    end
end
The stage of conditional query ;
Access conditional queries ;
Use query conditions as parameter to call CCK to create composite key ck;
Use ck as key to query AUP, and obtain the corresponding value v;
Use v as parameter to call algorithm 4 to split it, and obtain the splitted collection of values s_v;
Call GHFK with the elements in s_v as keys in order to get the result r ;
return r;
```

5.3 AUP index based method AIM

Since both baseline method and CCK bring a lot of data redundancy, in order to solve this problem, we design a novel method AIM. In Fabric, HistoryDB itself is not indexed and it is implemented by LevelDB. Therefore, it is necessary to follow a specific format when querying historical data, and flexible data query cannot be implemented well. In this method, we build an index for HistoryDB, which solves the problem of data redundancy well and make the overall data storage time greatly reduced. The implementation steps of AIM are shown in Algorithm 2 and the explanations are as follows.

During the data storage phase, given the data record p=(p_1...p_t)($t \geq 1$), just like the method CCK, for each record p_i ($1 \leq i \leq t$), we firstly spilt the collection $s^i_{c_1}$ to get each element (c^1_1 ... c^m_1), and then we use each element and other query conditions to create composite key by calling CK. We create the collection s^i_{ck}=(ck_1...ck_m) which contians m composite keys in the end. Nextly, we use each element in s^i_{ck} as key and $s^j_{c_1}$ ($1 \leq j \leq m$) as value to create key-value pairs. All those key-value pairs are stored in AUP, which is implemented with LevelDB. Note that, if the value corresponding to the key does not exist in the AUP, we can insert it directly; if it does, we need to update the AUP. During the updating phase, we extract the original value, and then use it with s_{c_1} as a parameter to call Algorithm 3 to create the new value $ns^j_{c_1}$. Nextly, we insert the new value $ns^j_{c_1}$ into AUP and the index database AUP is created. This update method is to solve the problem that the constructed composite key ck_j is same, but the corresponding value $s^j_{c_1}$ is different. Then we use $s^j_{c_1}$ as the key, record p_i as the value to construct a new key-value pair and initiate transactions to store p_i in block ledger. Finally, we repeat the above steps t times to save all data records p into the ledger.

Algorithm 3: The process of data connection.

Input: The collection of data s, which contains n elements
Output: A value consisting of elements in the data set v

```
Get the collection of data;
Set "#" as delimiter;
Set v to be empty;
for i=1 to n do
    if The ith element in the set s does not contain a "#" then
        Combine ith element with v via "#" ;
    end
end
return v;
```

During the data query phase, given a conditional query q, we use all query conditions as parameters to create a composite key ck, and then we use it as key to query AUP to get the value v. Nextly, we use v as a parameter to call Algorithm 4 and get the result collection $s_v = (v_1...v_x)$. Fianlly, we use each element in s_v as key to call GHFK and get the result r.

In the Algorithm 3, we use "#" to join the data sets and combine them into a value for return. In the Algorithm 4, we use "#" to split the value and return a data collection.

5.4 Data storage method CAM

As we can see from Section 3.3, the process of data validation requires us to repeatedly deserialize blocks, which cost a large number of time. To solve the problem, we propose to create a cache for the data that needs to be validated. Block signature, transaction endorsements and transaction version information in the block are stored in the cache. When executing validation, we can get data directly by accessing the cache without deserializing blocks. If the data that need to be validated can not be obtained by accessing cache, for example, the cache is crashed and the data in it is lost, we can also obtain it by deserializing blocks.

Algorithm 4: The process of data separation.

Input: A data character v whose length is L
Output: A collection of data s

```
Access data v;
Set "#" as delimiter;
Set comp to be empty;
Set index to 0;
for i=0 to L do
    if v[i]=="#" then
        Use v from index to ith characters to form an element and add it to comp;
        index++;
    end
    Add comp to the data set s;
end
return s
```

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 available 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 2.60GHz, 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 the DBLP data. The total number of publications in DBLP is 4146645. As each publication in DBLP always 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 belonging to α, j represents the number of publications belonging to α and published in v). Groups are shown in Table 2. In this paper, we measure the performance of conditional query methods using the following metrics—(1) Query execution times—time

Table 2 Grouping by proportion

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

Table 3 Grouping by data size

Group	1	2	3	4	5	6	7
Size(byte)	0-600	600-650	650-700	700-750	750-800	800-900	≥ 900
Total Number	268127	758668	1225755	1047611	530999	270709	44778

taken to execute the conditional queries. (2) Insertion times—time taken to insert data into Fabric ledger. (3) Memory cost—memory size occupied by all data.

Because the size of the data affects the speed of storage, in order to better highlight the experimental results, we group the data in DBLP according to size. The grouping results are shown in Table 3. In this experiment, different transaction data sizes are controlled to compare the original Fabric and CID, and the comparison mainly compares the time required to complete these transactions.

6.3 Experimental evaluation

Table 4 shows the performance of three methods:baseline, CCK and AIM. In order to show the result vividly, we have drawn Figure 2. We randomly select 1000 records from each group to execute 1000 queries at a time. We execute each query 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 6 shows the performance of CAM. We use 4000 go routines to submit transaction data, and the experiment is divided into 10 groups. Each group randomly selects 10000 data from each group in the data set to invoke transactions, and then uses the average result of these 10 groups of experiments as the final result.

6.4 Time cost of baseline

As we can see from the Table 4, 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 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 numbers and transaction numbers. 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

Table 4 Query time of each method

Group	Ratio	Query time of baseline	Query time of CCK	Query time of CI
1	100-18	29.03(s)	12.77(s)	9.57(s)
2	18-15	50.14(s)	12.31(s)	9.52(s)
3	15-12	55.44(s)	11.73(s)	8.51(s)
4	12-9	70.84(s)	11.70(s)	8.27(s)
5	9-6	80.92(s)	11.46(s)	7.93(s)
6	6-3	109.20(s)	10.85(s)	7.29(s)
7	3-0	174.74(s)	9.87(s)	6.07(s)

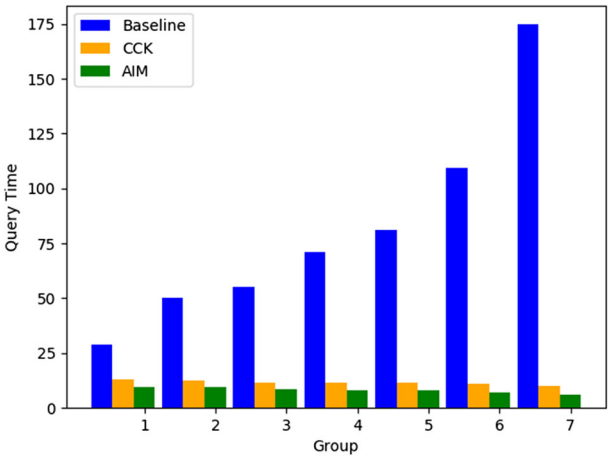

Figure 2 Histogram of query time of three methods

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 belonging to the authors, the more blocks need to be deserialized, the more time it will take to execute a query transaction.

Consider the query in baseline method, it needs to get all blocks that contain publications that belong to author α. It deserializes all these blocks and needs 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 to retrieve publications belonging to author α and published in venue v, we need to deserialize all blocks containing publications belonging to author α. Larger the number of publications that are not published in venue v, worse is the performance of baseline.

6.5 Time cost of CCK

The third column of Table 4 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 belonging to the author α and published in venue v becomes smaller, and the block number that we need to deserialize is smaller. We are able to achieve this improvement by using CCK because we can exactly know which block contains publications belonging to author α and published in venue v. That is to say, we just need to get blocks that contain publications belonging 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($x \geq y$) 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 better

the performance of CCK. The time-cost by using CCK is much smaller than that by using baseline method.

6.6 Time cost of AIM

We next analyze the time-cost of using AIM to execute conditional queries, it is similar to CCK. This is because in AIM, we also create composite key and we exactly know which block contains the data that meet 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 it to create composite key (α,p), and we need to take (α,p) and 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 leads to the size of ledger created by using CCK is larger than the ledger created by using AIM and the cardinality of the ledger data that performs conditional queries becomes larger. That is why AIM is better than CCK in query performance. Besides, the redundancy makes 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 method, and we execute these transactions with 4000 goroutines. Both baseline method and CCK cost more than 13h to finish these transactions. However, when we use AIM, the total number of transactions is 4146646 and it costs 5h29m to finish these transactions. By using AIM, we save more than 2 times running time, which we can see from Table 5. 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 cannot get the new data immdiately because it has not yet been saved to the index. Besides, without this method, we need to spend a lot time querying the ledger first if we want to construct an index.

6.7 Memory cost of the tree methods

In addition, with the use of index, 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|). In baseline and CCK, the total size of all data is 12508891|P|. In CI, 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, CI saves more data storage space than baseline and CCK.

6.8 Time cost of CAM

As we can see from Table 6, with the increase of transaction data, the time spent on the overall transactions also increases. This is because the transaction data increases, the data needs to spend more time in data transmission during the communication process, as presented in Figure 3.

Table 5 The data ingestion time for different methods

Methods	Method 1	Method 2	Method 3
Transaction Number	12508891	12508891	4146646
Data Ingestion Time	13h8m	13h12m	5h29m

Table 6 Insertion time of fabric and CAM

Group	1	2	3	4	5	6	7
Size(byte)	0–600	600–650	650–700	700–750	750–800	800–900	≥900
Fabric(s)	40.1	48.4	50.2	59.4	70.1	89.1	100.2
CAM(s)	30.2	35.3	40.2	46.6	59.1	67.2	80.1

Comparing Fabric with CAM, it can be seen that no matter how large the transaction data is, CAM takes less time than Fabric. By using the CAM method, the process of block deserialization is omitted in the transaction process, so it saves some time. When the block size continues to increase, the time spent on Fabric and CAM transactions continues to improve, but the speed of CAM improvement is slower than that of Fabric. Although it takes some time to build the cache, compared to the time to deserialize the block, this is trivial and can be ignored. Therefore, the speed of saving blocks to the file system is improved.

7 System implementation

We design a prototype system by using the proposed method. In the following, we present the design of chaincode and the interface of the system.

7.1 Chaincode design

In Fabric, chaincodes are divided into two categories, system chaincodes and user-defined chaincodes. User-defined chaincodes are important part in Fabric, and users can use it to implement custom system functions. So, in our system, we use user-defined chaincodes to achieve the required functions. In our system, we use Go to implement chaincode. Functions that chaincodes can implement include data storage, query of world state data and query of historical data. Figure 4 shows the structure of chaincode.

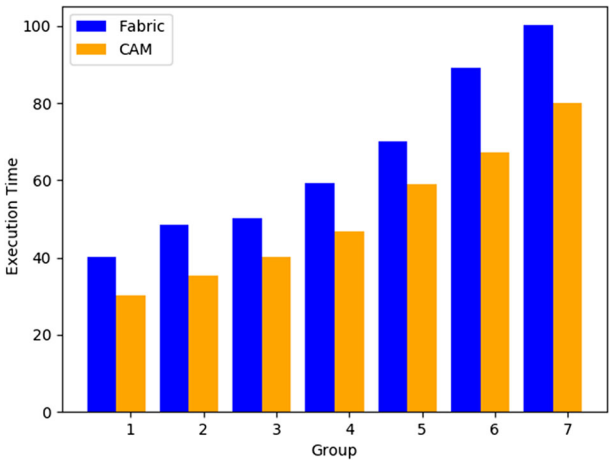

Figure 3 Histogram of execution time of fabric and CAM

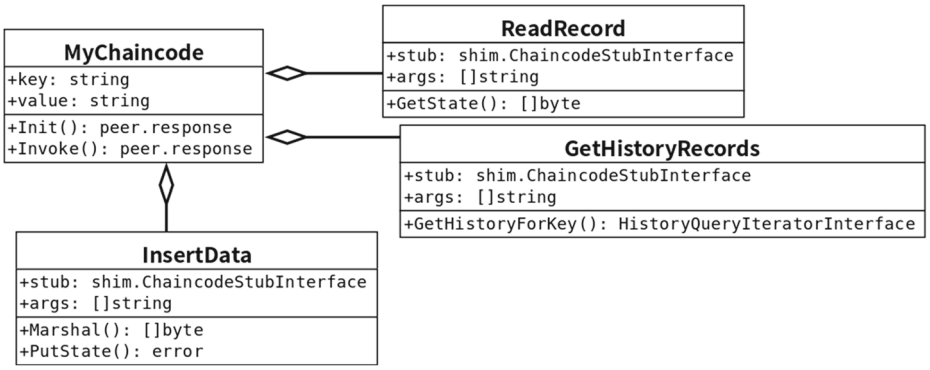

Figure 4 Chaincode design structure diagram

7.2 System interface

Figure 5 shows the interface of query, users can obtain related information by giving the name of an author and the name of a venue. System will show the information related to the given author and venue. When a user wants to inert data into ledger, he can enter the author, journal name and other information, then click the submit button, the system will store the data in the system. If the data is successfully stored in the system, the system will return a transaction id to user. Figure 6 is the interface of data storage.

8 Analysis

From the above three conditional query methods, we can see that the AIM has the best performance. It solves the problem of redundancy, improves the efficiency a query and data insertion. We get two conclusions. Firstly, when we execute conditional query, and the key which we want to use has a large number of unrelated values need to be removed,

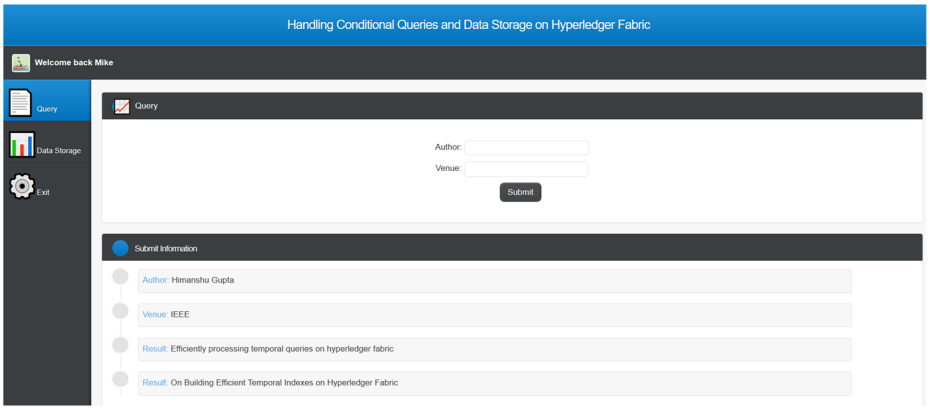

Figure 5 The interface of conditional query

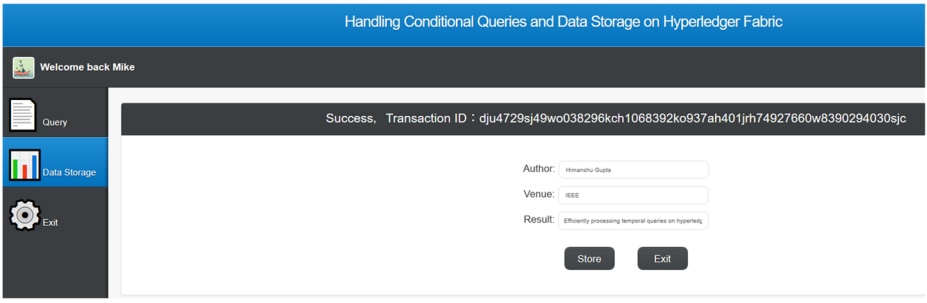

Figure 6 The interface of data storage

the best method is to use all conditions to create 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 ingestion data have a significant improvement.

From the method CAM, we can see that the speed of data storage has been accelerated. This is because, by using cache, we reduce number of blocks to be deserialized and we can obatin data directly by accessing cache.

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 the medical history of a patient in a certain department.

9 Conclusions

In this paper, we present three conditional query methods to handle the request of executing 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 conduct a comprehensive study to understand and analyse the conditional query performance of 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 addition, to efficiently handle data storage, we use cache to store data that need to be validated during MVCC validation and VSCC validation. It can effectively save the time of data storage.

Acknowledgements This work was supported by the National Natural Science Foundation of China (GrantNo. 61572335,61572336, 61902270), and the Major Program of Natural Science Foundation, Educational Commission of Jiangsu Province, China (GrantNo. 19KJA610002), and the Natural Science Foundation, Educational Commission of Jiangsu Province, China (Grant No.19KJB520052, 19KJB520050), and the Priority Academic Program Developmentof Jiangsu Higher Education Institutions.

References

1. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al: In: Proceedings of the Thirteenth EuroSys Conference, p. 30. ACM (2018)
2. Baliga, A., Solanki, N., Verekar, S., Pednekar, A., Kamat, P., Chatterjee, S.: In: Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20–22, 2018, pp. 65–74 (2018). https://doi.org/10.1109/CVCBT.2018.00013
3. Couchdb. https://couchdb.apache.org/.Lastaccessed10Jun2019
4. Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Sirer, E.G., et al.: In: International Conference on Financial Cryptography and Data Security, pp. 106–125. Springer (2016)
5. Dinh, T.T.A., Wang, J., Chen, G., Liu, R., Ooi, B.C., Tan, K.L.: In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1085–1100. ACM (2017)
6. 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 (2018)
7. Ethereum. https://www.ethereum.org/.Lastaccessed10Jun2019
8. Gao, K., Liu, Y., Xu, H., Han, T.: In: Blockchain and Trustworthy Systems–First International Conference, BlockSys 2019, Guangzhou, China, December 7–8, 2019, Proceedings, pp. 648–661 (2019). https://doi.org/10.1007/978-981-15-2777-7_53
9. Gervais, A., Karame, G.O., Wüst, K., Glykantzis, V., Ritzdorf, H., Capkun, S.: In: Proceedings of the 2016 SIGSAC Conference on Computer and Communications Security, pp. 3–16. ACM (2016)
10. Gorenflo, C., Lee, S., Golab, L., Keshav, S.: CoRR arXiv:1901.00910 (2019)
11. Gupta, H., Hans, S., Aggarwal, K., Mehta, S., Chatterjee, B., Jayachandran, P.: In: IEEE 34th International Conference on Data Engineering (ICDE), pp. 1489–1494. IEEE (2018)
12. Gupta, H., Hans, S., Mehta, S., Jayachandran, P.: In: IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 294–301. IEEE (2018)
13. HistoryDB. https://github.com/hyperledger/fabric/tree/master/core/
14. Hyperledger fabric. https://www.hyperledger.org/projects/fabric.Lastaccessed10Jun2019
15. Kang, H., Dai, T., Jean-Louis, N., Tao, S., Gu, X.: In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 543–555 (2019). https://doi.org/10.1109/DSN.2019.00061
16. LevelDB. https://github.com/syndtr/goleveldb/
17. Lin, I.C., Liao, T.C.: A survey of blockchain security issues and challenges. IJ Network Secur. **19**(5), 653 (2017)
18. Lin, I., Liao, T.: A survey of blockchain security issues and challenges. International Journal of Network Security **19**(5), 653–659 (2017)
19. Omohundro, S.: Cryptocurrencies, smart contracts, and artificial intelligence. AI Matters **1**(2), 19 (2014)
20. Parity. https://www.parity.io/.Lastaccessed10Jun2019
21. Pongnumkul, S., Siripanpornchana, C., Thajchayapong, S.: In: 2017 26th International Conference on Computer Communication and Networks (ICCCN), pp. 1–6. IEEE (2017)
22. Sharma, A., Schuhknecht, F.M., Agrawal, D., Dittrich, J. CoRR arXiv:1810.13177 (2018)
23. StateDB. https://github.com/hyperledger/fabric/tree/master/core/ledger/
24. Thakkar, P., Nathan, S., Viswanathan, B.: In: IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 264–276. IEEE (2018)
25. Vukolić, M.: In: International Workshop on Open Problems in Network Security, pp. 112–125. Springer (2015)
26. Yan, T., Chen, W., Zhao, P., Li, Z., Liu, A., Zhao, L.: In: WISE, Lecture Notes in Computer Science, vol. 11881, pp. 48–62. Springer (2019)
27. Zhang, X., Poslad, S.: In: IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2018)
28. Zhang, X., Poslad, S., Ma, Z.: In: IEEE Global Communications Conference (GLOBECOM), pp. 1–7. IEEE (2018)
29. Zheng, Z., Xie, S., Dai, H., Chen, X., Wang, H.: In: IEEE International Congress on Big Data (BigData Congress), pp. 557–564. IEEE (2017)

Affiliations

Tianlu Yan[1] · Wei Chen[2] · Pengpeng Zhao[1] · Zhixu Li[1] · An Liu[1] · Lei Zhao[1]

Tianlu Yan
tlyan@stu.suda.edu.cn

Wei Chen
robertchen@suda.edu.cn

Pengpeng Zhao
ppzhao@suda.edu.cn

Zhixu Li
zhixuli@suda.edu.cn

An Liu
anliu@suda.edu.cn

[1] School of Computer Science and Technology, Soochow University, Su Zhou, China

[2] Institute of Artificial Intelligence, Soochow University, Su Zhou, China