2019 IEEE 35th International Conference on Data Engineering (ICDE)

SEBDB: Semantics Empowered BlockChain DataBase

Yanchao Zhu†, Zhao Zhang†*, Cheqing Jin†, Aoying Zhou†, Ying Yan‡
†School of Data Science and Engineering, East China Normal University, Shanghai, China
‡Ant Financial, Hangzhou, China
yczhu@stu.ecnu.edu.cn,{zhzhang,cqjin,ayzhou}@dase.ecnu.edu.cn, fuying.yy@antfin.com

***Abstract*—Blockchain has been adopted in many applications to construct trust among multiple participants, such as supply chain management, digital assets transfer, philanthropy, etc. Blockchain platforms are often used as decentralized databases. However, existing blockchain platforms are far less convenient to use than traditional databases. They are lack of the capability of modelling complex tasks conveniently and efficiently, especially when both on-chain and off-chain data are involved at the same time. In this paper, we propose and implement a novel *blockchain database*, called SEBDB, which leverages the existing databases' functionality which are optimized for decades. Comparing to existing works, SEBDB is the first platform which considers both useability and scalability. Specifically, first, we add relational data semantics into blockchain platform, where each transaction is a tuple with multiple attributes in a pre-defined table. Second, we use SQL-like language as the general interface, instead of code-level APIs, to support convenient application development, in which intrinsic operations are re-defined and re-implemented to suit for blockchain platform. Third, as RDBMS has achieved great success in the past decades, our system, though not relying on RDBMS, treats it as an important component. Finally, we define a mini-benchmark to evaluate the performance of the blockchain database. Extensive experiments demonstrate the effectiveness and efficiency of our proposed system.**

***Index Terms*—Blockchain database, relational semantics, query processing, data storage**

I. Introduction

Blockchain, as a distributed ledger technology has been adopted in various applications to enable trust connection among multiple participants, such as decentralized cryptocurrency, international settlement, security trading and settlement, traceability of food ingredient, etc. Blockchain consists of a growing list of blocks to record all transactions settled by participants. To ensure the immutability, each block is affiliated with the information of the hash of the previous block, timestamp, and Merkle hash root of this block. Consensus protocol is used to ensure the consistency of the data recorded from each participant.

Although a blockchain system is defined as a decentralized database, it is far less convenient to use than traditional databases in terms of query language and query processing efficiency. Traditional databases have optimized data model, storage, indexing and query processing engine, such as Oracle [1], DB2 [2] and MySQL [3]. Blockchain systems should leverage the techniques from existing databases which have been optimized for decades, to improve their usability and scalability. To this end, we re-architect the blockchain platform from storage layout and query-transaction operators and propose SEBDB in this paper.

Ethereum [1] and Hyperledger Fabic [2] are two major platforms to support general-purpose applications, while the former suits for public blockchain scenario and the latter suits for consortium blockchain scenario. In general, transactions are stored in a file system or key-value storage, organized in the blockchain and accessed by code-level APIs. As the implementations upon two platforms are in common to some extents, we use an example upon Hyperledger Fabric and omit the other one due to space limitation.

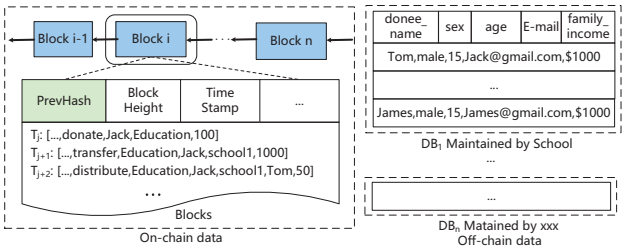

Fig. 1. A typical implementation upon Hyperledger Fabric

Example 1: A donation system needs to handle three main types of transactions, including (i) *donate*, i.e, a donor donates money to a project, (ii) *transfer*, i.e, the charity transfers money from a project to an organization, and (iii) *distribute*, i.e, an organization distributes donations to donees.

A typical solution may store data in Hyperledger Fabric and commercial RDBMS (e.g, MySQL) at the same time, as shown in Figure 1.Transactions shared by all participants (also called on-chain data) are stored in Hyperledger Fabric, while the rest private information (also called off-chain data) are managed by each local RDBMS. For example, transactions in block i refer to some events, such as "Jack donates $100 to Education", "Education transfers $1000 to School1", and "School1 distributes $50 to Tom". Local site 1 records donees' information, site 2 records donors' information, and so on. A user tends to execute a query task upon the transactions over the blockchain and other information in local sites.

From this example, three types of transactions are stored continuously in a block in free style. Although transaction data are structured, the relational operations cannot be adopted conveniently. For example, for the same operation to transaction

*Corresponding author.
[1]www.oracle.com
[2]https://www.ibm.com/analytics/us/en/db2
[3]https://www.mysql.com

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00198

types “transfer” and “donate” from two different applications, the same program must be written twice. In addition, a donor equipped with a mobile phone cannot verify the correctness of her query results. In summary, the existing blockchain systems have the following weaknesses to manage block data.

- **Weak semantics**. Although a typical transaction on blockchain is structured containing multiple attributes, the existing framework does not carefully consider this point so that it is infeasible to build complex queries upon different attributes. Without sufficient semantic description, it is infeasible to support abundant queries from many typical blockchain applications. As the relational data model achieves great success in the past four decades, adding relational semantics will empower the expression capability.
- **Insufficient operations**. Block data is either stored in the key-value database [1] or stored in flat file [2], regardless of a key-value DB or a flat file, they both have insufficient operations to support accessing block data. Moreover, we cannot adopt RDBMS to store block data directly, because of data trust and the lower writing performance. Thus, although data is structured in blockchain system, it cannot offer relational operations to access those data. In addition, existing platforms also cannot support special blockchain operations efficiently, such as track-trace, on-off chain join, etc. Therefore, we consider the current blockchain system does not have sufficient operations to support rich queries.
- **Weak authenticated query**. Public users (i.e. thin clients) in a blockchain are not able to store all block data due to computation and storage limitations. Thus they need to query data from blockchain nodes and verify the correctness of the query results through Merkle root. Existing blockchain systems only provide simple authenticated queries such as whether a transaction is in a block. No blockchain system can verify the soundness and completeness of a query result from a thin client. For example, in the donation system, a donee queries all transfer records of a specified project. To the best of our knowledge, there is currently no blockchain system capable of supporting rich authenticated queries.

To overcome the above weaknesses, some works attempt to improve data management performance of blockchain systems, such as ChainSQL [3] and BigChainDB [4]. ChainSQL maintains two loops among all participants, the inner one aims at achieving agreements of transactions among all participants while the outer one tries to store all transactions in each local commercial RDBMS, so that a user can get results by the querying engine of commercial RDBMS. The architecture of BigchainDB evolves dramatically from version 1.0 to 2.0. BigChainDB v1.0 can be treated as a distributed MongoDB [5] system, because the kernel database is monitored by all participants (a user per remote node). Only when being agreed on by a majority of participants can each transaction be recorded in the database. BigChainDB v2.0 has two rings. In the core ring, all participants attempt to make agreements on transactions. In the outer ring, each node is empowered with a MongoDB system.

However, no system can cope with the above challenges well. Firstly, none of them provide flexible query language for the applications. ChainSQL and BigChainDB adopt commercial RDBMS and key-value database, so that specific query operator of blockchain system cannot be described and processed efficiently, such as linkage. Secondly, how to deal with all kinds of data seamlessly remains unsolved. ChainSQL uses the commercial database to process queries after transfering all on-chain data to a commercial database, but this mechanism cannot guarantee the integrity of data upon the blockchain. BigChainDB does not consider how to deal with on/off-chain data together. Thirdly, data semantics are too weak to express blockchain-related queries. For example, BigChainDB only provides key-value model, insufficient to conduct queries with multiple inputs. Finally, none of them supports rich authenticated queries.

In this paper, we build a novel prototype system, called SEBDB, to address such issues. SEBDB integrates relational semantics with blockchain platform, where off-chain data is stored in a RDBMS and on-chain data is modelled as a number of relations (each transaction is a tuple belongs to a certain relation). Then, some new operations are defined based on all kinds of data. Data updates are stored in blocks in an append-only manner, making full use of sequential writes of disk. Block data is also the data of data table, thus the system only maintains one copy of the data, avoiding additional storage overhead. SEBDB uses authenticated data structure (ADS) to implement authenticated query processing to ensure the correctness of client query results.

The contributions of this paper are summarized below.

- We add relational semantics to history block data, where all block data upon blockchain are treated as relations with multiple attributes. Moreover, we develop SQL-like language to help application developers to access block data friendly.
- We design three operations and three indices (i.e. block-level, table-level and layered index) for special tasks of blockchain to improve executing efficiency.
- We implement rich authenticated queries on the blockchain for thin client. Users can verify the soundness and completeness of the query results.
- We design and implement a consortium blockchain system integrated with the above technologies, named SEBDB, which can support user-defined schema and answer complex queries efficiently. Extensive experiments demonstrate the efficiency and effectiveness of SEBDB.

The rest of the paper is organized as follows. We review related work in Section II. In Section III, we introduce data model and give an overview of our system. Subsequently, Section IV - V describe key modules, including the storage and indexing mechanism, and query processing engine. We report the performance of Starchain in Section VII. Finally, we conclude the paper in Section VIII.

TABLE I
COMPARISON OF BLOCKCHAIN DATABASE SYSTEMS

Catalogy	Representative Systems	Decentralization	Relational Semantics	SQL-supported Interface	Authenticated Query	Integrating On-chain and Off-chain Data
Blockchain System	Bitcoin [6], Ethereum [1], Hyperledger Fabric [2], Ripple [7], EOS [8]	✓	Weak	×	Weak	×
Distributed Database	F1 [9], Amazon Aurora [10], SAP HANA [11]	×	Strong	✓	×	×
Blockchain + Database	ChainSQL [3], BigchainDB 1.0 [4], BigchainDB 2.0	✓	BigchainDB:weak, ChainSQL:strong	BigchainDB:×, ChainSQL:✓	Weak	×
Blockchain Database	SEBDB	✓	Strong	✓	✓	✓

II. RELATED WORK

As blockchain has been studied extensively in recent years, we review important work in this section, especially focus on blockchain data management. Dinh et al. review blockchain systems from a data processing point of view [12].

Blockchain systems: Blockchain was originally derived from Bitcoin [6], a decentralized cryptocurrency. All transactions about spending coins will be appended in a chain of blocks, and each transaction is immutable once it is recorded. After storing block data in filesystem, Bitcoin supports simple queries such as searching transactions or blocks. Ethereum [1] extends Bitcoin by adding the Turing-complete programming language to support more complex business logics. Ethereum stores block data and state data in LevelDB [13] and provides low-level APIs to access data. Hyperledger Fabric [2] stores block data in filesystem and state data in key-value database (e.g, LevelDB or CouchDB) to support more abundant queries. Obviously, such popular blockchain systems are lack of rich semantics to represent queries conveniently.

Traditional distributed databases: Blockchain systems are often regarded as distributed databases, though exactly big differences exist. A typical distributed database system has high throughput, low latency, and rich query capabilities, such as F1 [9], Amazon Aurora [10] and SAP HANA [11]. To pursuit high performance, a distributed database system will distribute a waiting task to working nodes; in addition, a big task may be divided into small ones in advance. The whole system is under control of a centralized node. Distributed database systems usually adopt non-Byzantine fault tolerant consensus protocol such as Paxos [14] or Raft [15].

Blockchain Data Management: Some works attempt to manage blockchain systems more conveniently and efficiently. ChainSQL [3] tries to combine blockchain and database by using blockchain to achieve agreements among all participants in a Byzantine environment, and process queries in a RDBMS after transferring all transactions to RDBMS. However, two copies of data replications exist and data integrity cannot be evaluated conveniently. BigchainDB [4] devises a two-loop structure to manage blockchain data, which can be treated as a key-value database monitored by all participants. Forkbase [16], a storage engine designed for blockchain and forkable applications, integrates core application properties into the storage to deliver high performance. EtherQL [17] adds an efficient query layer for Ethereum to support a set of useful analytical queries using MongoDB as the external data storage.

Some researches focus on authenticated query processing where query results from untrusted data providers can be verified by client. Existing work includes that for range query [18]–[21], aggregate query [22], and join [20], [21], [23]. Most of the above works are based on Merkle hash tree (MHT) [24].

Comparison: Table I compares current blockchain systems, traditional databases and SEBDB. Among these systems, SEBDB is the first system that researches on attaching relational semantics to block data, integrating off-chain data with on-chain data. In addition, it improves query performance of blockchain database. Furthermore, it supports rich authenticated queries for thin clients which are important for decentralized applications.

III. DATA MODEL AND ARCHITECTURE

We introduce data model and architecture in this section.

A. Data model

We add relational semantics to blockchain systems to enhance modeling capability. Transactions are structured and stored in blocks according to the time when a transaction took place. The same type of transactions have a unified semantic description inspired by relational model. We declare a table schema for all transactions of the same type, each having a set of columns. The attribute types can be string, various flavors of numbers, etc. All attributes are divided into two types, including application-level attributes and system-level attributes. The application-level attributes are defined by users explicitly, while the system-level attributes are added to the table automatically, such as transaction ID, signature, etc. *Sender* and *Tname* are important system-level attributes, where *Sender* means who sends the transaction, and *Tname* refers to the transaction type.

We propose SQL-like language to manage data, i.e, using **CREATE**, **INSERT**, and **SELECT** clauses to create a table, adding a new transaction, and getting query results respectively. It is convenient to define more clauses in future to fit for special scenarios. For example, we use **TRACE** clause to

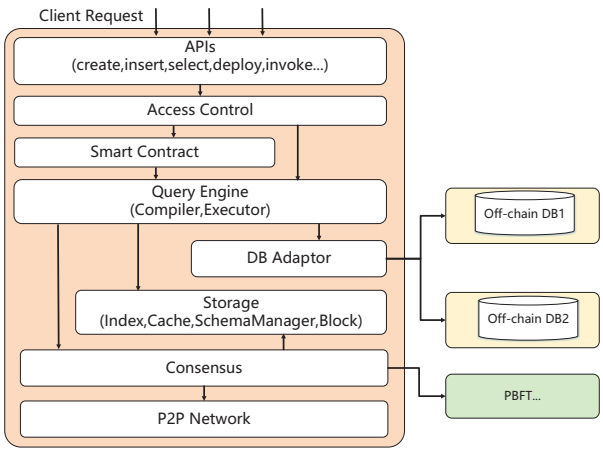

Fig. 2. Architecture of SEBDB

get the lineage of some events, and extend **JOIN** operation to integrate data from different sources.

For example, we generate three tables for three types of transactions, including *Donate(donor, project, amount)*, *Transfer(project, donor, organization, amount)*, and *Distribute(project , donor, organization, donee, amount)*. Each tuple in a table means a transaction in the blockchain. Example statements are listed below.

1) CREATE Donate (donor string, project string, amount decimal)
2) INSERT into Donate ("Jack", "Education", 100)
3) SELECT * from Donate where donor = "Jack"

Adding relational semantics enables SEBDB to simplify users' interactions by hiding the complexity from application developers. Developer can just use familiar clauses to manage blockchain data, like "INSERT", "SELECT". Particularly, since all historical transactions are stored in blockchain while query demands may focus on data in a time period, a time window can be specified for a query request.

B. Architecture

Our architecture consists of five layers, including application, query processing, storage and index, consensus, and network, as illustrated in Figure 2.

- **Application Layer**. This layer has three components: APIs, access control and smart contract. The access control verifies request permission before execution, where a multi-channel method is adopted to protect users' privacy. The system supports smart contract embedded SQL-like language to define a DApp (Decentralized Application), where SQL-like is responsible for accessing data.
- **Query Processing Layer**. This layer aims at parsing, optimizing and executing SQL-like queries. Details will be covered in Section V.
- **Storage Layer**. This layer is responsible for data storage and indices, i.e, creation and maintenance of storage structures such as indices, Merkle tree, cache as well as search, scan and insertion of block data. See Section IV for details.
- **Consensus Layer**. As different consensus protocol suits for different scenario, SEBDB uses plug-in pattern, allowing users to select different consensus protocol according to their requirements. Currently, we support KAFKA and PBFT [25].
- **Network Layer**. We choose Gossip [26] as basic network facility, since it is not only widely used in distributed databases for failure detection and membership protocol [27] [28], but also in blockchain [1] [2] [6] for block propagation and data recovery.

Since it is costly to make all nodes heavy (storing the whole data, and attending consensus protocol), some nodes will act as *thin client*s, and only keep concise information (i.e, block headers) to verify the correctness of querying results. See Section VI for details.

IV. Storage and Index

We describe data storage and indexing mechanisms in this section.

A. Data Storage

SEBDB maintains on-chain data and off-chain data at the same time. Off-chain data are managed by a local RDBMS, and accessed via an interface (ODBC, JDBC, etc.). We focus on how to deal with on-chain data in the blockchain. Each block in the blockchain consists of a *block header* and a *block body*. Block header records meta information, including *prevHash* for the hash value of previous block, *blockHeight* for the number of blocks to this block, *timestamp* for the time being packaged, *transRoot* for the root of Merkle tree on all transactions in the block, *Signature* for packager of the block, and *blockHash* for the hash of current block.

Block body contains a number of transactions, each having several attributes, including *Tid*, *Ts*, *Sig*, *SenID*, *Tname* and some user-defined application-level attributes, where *Tid* is transaction id, *Ts* records when the transaction is sent out, *Sig* guarantees unforgeability of transactions, *SenID* records the identity of transaction sender, and *Tname* denotes the name for a category of the transactions. Each transaction type is associated to a user-defined schema. Generally, the schema can be stored and maintained as a regular table. The system sends a special transaction to synchronize schema among nodes. Figure 3 illustrates the structure of a block. The transaction with *Tid =2* means *org1* transfers money from *education* project to *CangShan* at *June 8th, 2018, 10:21 PM*.

On-chain data are organized as blocks and stored physically in files on storage devices. Blocks are appended to files, and once a block is appended, it is immutable. The default size of a file is set 256MB due to the trade-off between system overhead and convenience of reading and writing. Certainly, users can configure the size of a file. Although the storage unit is a block, the cache unit is a transaction type (i.e. a table) because data belong to the same transaction type are usually visited together.

B. Indexing mechanisms

It is inefficient to process a request by scanning blocks one by one, because the tuples in a block may belong to multiple

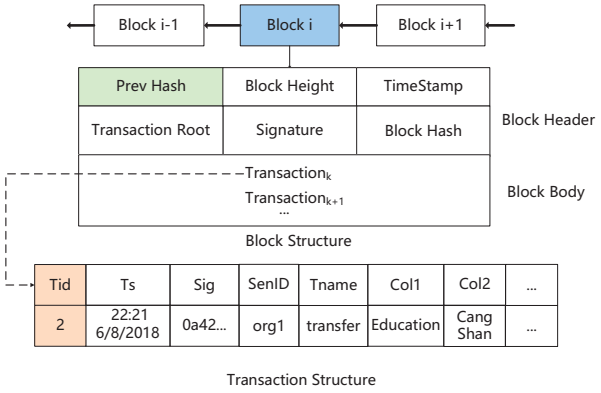

Fig. 3. Block Structure

tables. We devise indexing mechanisms to hasten data access, especially for three basic operations, including (i) getting a block when given $block\_id$, $transaction\_id$ or $timestamp$, (ii) getting tuples belong to the same transaction type, and (iii) getting transactions under some condition.

Block-level B^+-tree. To deal with the first operation, we build a B^+-tree with key (bid, tid, Ts), where bid is block id, tid is the first transaction in the block, and Ts denotes the timestamp of the block. For any two blocks, b_i and b_j, if b_i is earlier than b_j, then $(b_i.bid < b_j.bid) \wedge (b_i.tid < b_j.tid) \wedge (b_i.Ts < b_j.Ts)$ always holds. In this way, given $block\_id$, $transaction\_id$ or $timestamp$, we go from the root down to the leaf node to get the location of the target block. Similarly, given $timestamp$, we can get the block id. Moreover, since bid, tid, and Ts are incremental, leaf nodes are kept full.

Table-level Bitmap Index. To deal with the second operation, we maintain a table-level bitmap index to record table distribution to avoid scanning all blocks. Each bitmap refers to a table, and the i-th bit in a bitmap indicates whether block i contains transactions of that table or not. Hence, For a table query request, looking up bitmap index can avoid scanning all blocks. When a new table is generated, a new bitmap is added. When a new block arrives, the bitmap index is updated by setting corresponding bitmaps. The index can also be created on $SenID$ for tracking query.

Layered index. To deal with the third operation, we propose a layered index upon a table's attribute to hasten data access. The first level consists of bitmaps or entries that describes the distribution of attribute's values among blocks. For a discrete attribute, each bitmap in the first level refers to a discrete value; for a continuous attribute, we will generate an equal-depth histogram in advance, and each entry represents range of index keys of a block. The second level is a B^+-tree for that attribute within the block.

Figure 4 illustrates an example of layered index on a continuous attribute. Each bucket in the histogram represents continuous range of index key, such as $(-\infty, k_1], (k_1, k_2]...(k_p, \infty)$. The first level consists of multiple entries, each having a block id and a subset of buckets of the equal-depth histogram. A bucket occurs in the subset if corresponding block contains transactions within the range denoted by the bucket. Moreover,

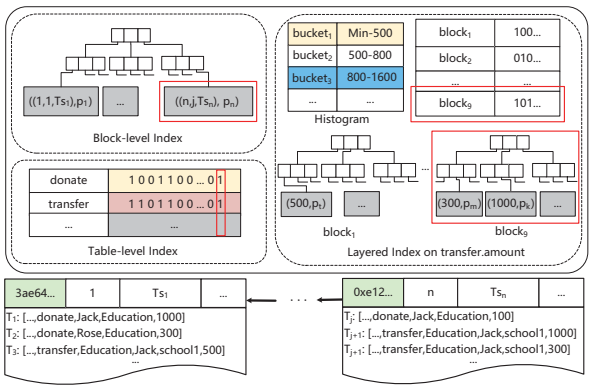

Fig. 4. Index Structures and Index updates

the subset can be represented by a bitmap, i.e, the i-th bit is set if the i-th range occurs. The second-level consists of one B^+-tree on index attribute per block.

To answer a range query, it traverses the first-level index, and executes bitwise AND operation on the subset of each entry and a range defined by the query predicate. If the result is 0, then the entry does not contain query results. In this way, blocks without query results are filtered. It is efficient to get query results through B^+-tree for the rest blocks. The equal-depth histogram is created by sampling historical transaction during index creating, the height of histogram is configurable for different precisions. Finally, a new index entry will be appended to the first level of the index and a B^+-tree will be created for the block when a new block arrives.

Example 2: Figure 4 shows structures and updates of three types of indices. There are block-level index, table-level index, and a layered index on $transfer.amount$. A new block with block id n, first transaction id j and timestamp Ts_n is appended to blockchain. Updates of indices are identified by frames. For block-level index, a new entry $((n, j, Ts_n), p_n)$ is append to the tree, p_n denotes the location of the block. For table-level index, since the block contains transaction type $donate$ and $transfer$, the n-th bits in the bitmap of $donate$ and $transfer$ are set. For layered index, the update consists of two steps. It first adds an entry to the first layer for block n, since the block contains transactions within range denoted by bucket 1, 3, they will join the entry, i.e., the first and the third bits of the bitmap are set. Then, a B^+-tree is created for indexing transactions in the block in a bulk loading way.

The layered index has three benefits: (i) convenient to batch appending without re-balancing the index structure when a new block is chained; (ii) efficient for empty query because records can be filtered earlier by the first-level of the index; (iii) easy to integrate with the block-level index to execute queries with time windows under the column-based predicate constraint. Note that the layered index supports tracking query if an index is created on a system-level column *SenID* or *Tname*, and it can support range or point query if an index is created on a application-level column.

Cost Analysis for select operation. Assuming each block is stored in a file. Consider a table r with p tuples satisfying predicate P distributed among k blocks. Let t_T denote the

transmission time per disk block, t_S denote the average disk block-access time, f denote the size of a packaged block in blockchain, and n denote the current block chain height. We assume that the size of each disk block is b, and a tuple of table r can be stored in a block.

$$C_{No-index} = n \cdot t_S + \frac{f \cdot n}{b} \cdot t_T \quad (1)$$

$$C_{bitmap-index} = k \cdot t_S + \frac{f \cdot k}{b} \cdot t_T (k \leq n) \quad (2)$$

$$C_{layered-index} = p \cdot t_S + p \cdot t_T \quad (3)$$

Equations (1), (2) and (3) represent the cost of adopting scan, bitmap-index and layered index respectively. From the above equations we know that a scan algorithm often has the highest cost compared with bitmap-index scanning and layered-index scanning, while the cost comparison of bitmap-index scanning and layered index scanning are decided by the distribution of tuples of table r and the selectivity of predicate P on table r. If the size of query result is large, using table-level bitmap index may outperform layered index since random I/O is slow. We will show the performance in SectionVII.

V. Query processing

As is mentioned before, each transaction type corresponds to a table logically. Hence, basic operations in RDBMS, such as project, select, Cartesian operations, etc. can also be supported by SEBDB. However, unlike the traditional RDBMS, data in the same table are not physically stored continuously and may be distributed into multiple non-adjacent blocks physically. Thus, executions of basic operations are very inefficient based on storage pattern of blockchain. Fortunately, we can employ the block-level index, the table-level index and the layered index proposed in Section IV-B to improve the performance of basic operations. Specifically, we re-implement the basic operations based on the above three indices. We omit implementation details due to space limitations.

Actually, SEBDB defines and implements some special operations towards blockchain, i.e. *track-trace, on-chain join and on chain and off chain join (on-off join)*. The track-trace operation is an unary operation aiming at tracking from two dimensions, i.e. (i) who sends transactions, and (ii) which transaction is sent, in a time window. For example, tracking details of all transactions sent by "org1". The on-chain join operation is a binary operation, whose goal is traceability among transactions. For example, we trace the flow of a donation donated by "Jack", which is from the donor "Jack" to a certain project, and then to a specific donee. The on-off join operation is also a binary operation which integrates on-chain data with off-chain data. For example, we try to know personal information of a donee who accepts a certain donation. The algorithms of three operations are as follows.

A. On-chain Tracking

Provenance is critical for blockchain applications to reveal the effect and evolution of interested events. Usually, we need to track the history from two dimensions, i.e. *operator* and *operation*, which means who sends the transaction and which

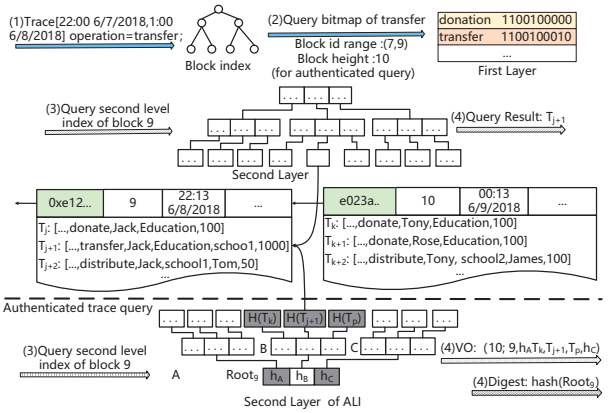

Fig. 5. An example of trace query

transaction is done. Thus, three kinds of typical provenance are listed below. Given the set R contained all tables in SEBDB, each table has columns SenID, Tname, ..., Ts, and a time window $[s, e]$. The tracking operation returns $\{t | \exists r \in R(t[SenID] = o \wedge t[Tname] = p \wedge s \leq t[Ts] \leq e \wedge t \in r)\}$ if given an operator o and an operation p. Note that operator or operation is optional if we want to track from a single dimension.

Algorithm 1: Tracking operation

Input : time window (c, e), operator o, operation p, block index BI, layered indices I_d on SenID and I_n on Tname ;

Output: Query result: Ω;

1 $B \leftarrow BI(c, e)$;
2 $B' \leftarrow First\_level\_bitmap(I_d(o))$;
3 $B'' \leftarrow First\_level\_bitmap(I_n(p))$;
4 $B \leftarrow B \& B' \& B''$;
5 $M \leftarrow$ *blocks positioned by* B;
6 **foreach** $b_r \in M$ **do**
7 $\quad P_o \leftarrow second\_level(I_d, b_r)$;
8 $\quad P_p \leftarrow second\_level(I_n, b_r)$;
9 $\quad$ **foreach** $p \in (P_o \cap P_p)$ **do**
10 $\quad\quad t \leftarrow read(p)$;
11 $\quad\quad \Omega \leftarrow \Omega \cup t$;
12 $\quad$ **end**
13 **end**
14 **return** Ω ;

To support tracking operation, the layered indices on column *SenID* and *Tname* are pre-created. Note that the the two indices are created on all tables for all historical transactions. Algorithm 1 describes how to deal with the tracking task from two dimensions. At line 1, it searches block index and get a bitmap B, the i-th bit is set if block i is in the time window. Then, it searches first level index of I_d and I_n and get bitmaps B', B''. The j-th bit of B' or B'' is set if block j contains transactions sent by o or belonging to p respectively (at lines 2 to 3). Then, it performs AND operation on B, B' and B'' to get blocks that contain transactions of o and p within time window (at lines 4 to 5). Finally, although through above strategies it can get blocks that may contain query result. It is inefficient to

scan all target blocks. Thus, it searches the second-level index in order to get target transactions effectively. For each block, it searches second-level indices on $SenID$ and $Tname$ and get intersection of resulting pointers to transactions. Finally, all resulting transactions are read from disk (at lines 6 to 13).

Example 3: The upper half of Figure 5 shows a simple example of track query from operation dimension. Consider there is a layered index on Tname. Firstly, it searches block index to get block ids within time window and generates a bitmap. The 7-th to 9-th bit is set (means block 7 to 9 is in the time window) (step 1). It then gets bitmap of $transfer$ from first level index and then performs AND operation on the two bitmaps. The i-th bit in the result is set if block i contain query results. In this example, is block 9 (step 2). Finally, it queries second-level index on block 9 to get query result T_{j+1} (step 3, 4).

B. On-chain Join

As is mentioned in Section IV-B, a block may contain different types of transactions. In addition, a block in blockchain is usually much larger than that of relational database system. Thus, traditional join algorithms such as block nested loop join can not be adopted directly due to repeated readings of blocks. Since traceability queries are usually executed by equi join. We implement an one pass scan hash join. Initially, we scan all blocks and construct partitions of r and s using hash function h. For each partition s_i of s, we build a hash index on it. For each tuple t_r in r_i, probe the hash index to locate all transactions s_r such that $t_s[attr]$=$t_r[attr]$ for join results. With table-level index, the performance can be improved for only blocks containing transactions of r or s are read.

Furthermore, if we happen to have layered index on the join attribute, the performance can be further optimized. Algorithm 2 shows the join processing with layered indices on continuous attributes. Firstly, it searches block index and get a bitmap B, the i-th bit is set if block i is in the time window (at line 2). Secondly, it searches first level index of I_r and I_s and get bitmaps B', B''. The j-th bit of B' or B'' is set if block j occurs in index entries of r or s respectively (at lines 3 to 4). Then, it performs AND operation on B and and B' or B'' to get blocks that contain transactions of r or s within time window (at lines 5 to 7). Let e_{r_i} and e_{s_j} denote entries of block i in I_r and block j in I_s. And let k and m denote a bucket with upper and lower bounds (l, u) respectively. If $intersect(b_r, b_s)$ returns TRUE (at line 10), i.e. $\exists\ k \in e_{r_i}(m \in e_{s_j} \wedge \neg(k.u < m.l \vee k.l > m.u))$ holds, then sort merge join of b_r and b_s is executed. For discrete attribute, the intersection result of a pair of blocks depends on whether there are join results of each bitmap key.

C. On-off chain Join

Since off-chain data are stored in RDBMS, we can execute join query to integrate on-chain and off-chain data. Let r be the on-chain table and s be off-chain table to be joined on $attr$. First, we get data of s from RDBMS via interface such as ODBC, JDBC. Then blocks containing transactions belonging to r are read with the help of bitmap index. Finally, we execute hash join for join results.

Algorithm 2: On-chain Join

Input : join tables r and s, join attribute $attr$, time window (c, e), block index BI, index I_r on $r.attr$, index I_s on $s.attr$;
Output: query result Ω;

1 $\Omega \leftarrow \emptyset$, $result \leftarrow \emptyset$, $M_R \leftarrow \emptyset$, $M_S \leftarrow \emptyset$;
2 $B \leftarrow BI(c, e)$;
3 $B' \leftarrow First\_level\_bitmap(I_r)$;
4 $B'' \leftarrow First\_level\_bitmap(I_s)$;
5 $B' \leftarrow B\&B'$; $B'' \leftarrow B\&B''$;
6 $M_r \leftarrow blocks\ positioned\ by\ B'$;
7 $M_s \leftarrow blocks\ positioned\ by\ B''$;
8 **foreach** $b_r \in M_r$ **do**
9 **foreach** $b_s \in M_s$ **do**
10 **if** $intersect(b_r, b_s)$ **then**
11 $result \leftarrow SortMergeJoin(b_r, b_s)$;
12 $\Omega \leftarrow \Omega \cup result$;
13 **end**
14 **end**
15 **end**
16 **return** Ω ;

Algorithm 3: on-off-chain Join

Input : on-chain table and off-chain table r and s, join attribute $attr$, time window (c, e), block index BI, index I_r on $r.attr$;
Output: query result Ω;

1 $\Omega \leftarrow \emptyset$, $result \leftarrow \emptyset$;
2 $B \leftarrow BI(c, e)$;
3 $s_{min} \leftarrow min(s.attr)$;
4 $s_{max} \leftarrow max(s.attr)$;
5 $B' \leftarrow First\_level\_bitmap(I_r)$;
6 $B' \leftarrow B\&B'$;
7 $M \leftarrow blocks\ positioned\ by\ B'$;
8 **foreach** $b_r \in M$ **do**
9 **if** $intersect(b_r, (s_{min}, s_{max}))$ **then**
10 $result \leftarrow SortMergeJoin(b_r, s)$;
11 $\Omega \leftarrow \Omega \cup result$;
12 **end**
13 **end**
14 **return** Ω ;

Moreover, the performance can be further optimized if there is a layered index on $r.attr$. Algorithm 3 shows join processing on continuous attributes. Firstly, it searches block index and get a bitmap B, the i-th bit is set if block i is in the time window (at line 2). Secondly, it get range of off-chain table, i.e. (s_{min}, s_{max}) (at lines 3 to 4). The range is used to filter blocks that contain no join result (at lines 5 to 7). Block i takes part in join processing if $intersect(b_r, (s_{min}, s_{max})$ returns TRUE, i.e. $\exists\ k \in e_{ri}(\neg(k.u \leq s_{min} \vee k.l \geq s_{max}))$ holds (at lines 9-10). It executes sort merge join between block i and s taking advantage of the second level index (at lines 8 to 13). For discrete attribute, it first queries off-chain database for

unique values of join attribute, and then execute OR operation on bitmaps of unique keys to get blocks for join processing.

Note that we first load all off-chain data to blockchain to avoid repeated data transfer and set a threshold for caching off-chain data. In addition, the query results from off-chain data are sorted on join attribute. Thus, for each block, we can run sort merge join since leaf nodes of second-level index are sorted.

VI. Thin client

Thin clients only store block headers to verify the correctness of data from an untrusted node, like SPV node in bitcoin. Reconstructing the root of Merkle Hash tree of visited blocks can verify the correctness of data conveniently if we query data by scanning all blocks. However, the scan approach is too inefficient, for example, for a range query, all blocks are returned to client in order to guarantee $soundness$ and $completeness$ of query results. In this section, we design an authenticated index structure based on our layered index and propose the authenticated query processing based on the index.

Merkle B-tree (MB-tree) [21] is a combination of B^+-tree and Merkle Hash Tree (MHT) [29], where each leaf node contains the hash value of record, and each internal node stores the hash of the concatenation of its children. In addition, the MB-tree guarantees soundness and completeness of the query results visited by the index. For a range query, a server first searches the MB-tree to find the first record in the range. The digests of left sibling nodes on the path from the root to the record are added to a verification object (VO). Next, leaf nodes are scanned following the pointers between leaf nodes until right boundary of the query range. Then it searches the MB-tree to find the right boundary of the range. All the digests on the right of the path are inserted to VO. Finally, the VO is returned to a client. The client verifies soundness and completeness of query results by reconstructing root of MB-tree with VO and compares it with a correct root which is encrypted by a trusted data owner.

A thin client, needs to verify query results by other nodes. However, data of different nodes are inconsistent due to different speeds of transactions execution. Thus, we cannot adopt MB-tree directly because it cannot support snapshot verification for multiple versions of Merkle root. To support authenticated queries, we modify our layered index by replacing B^+ tree in the second layer with MB-tree and design a 2-phase approach. We call this index ALI (Authenticated Layered Index). Since each block maintains the second level index, each block height corresponds to a snapshot. The authenticated query flow is as follows.

The first phase of query processing includes three steps: (i)A thin client sends a query to a randomly selected full node. (ii) The full node executes the query based on the ALI and adds verification information along with block height h to VO. (iii) The full node returns VO to the thin client.

The second phase also has three steps, which tries to verify the VO received from the first phase. (i) The client sends the query along with h received from the first phase to another randomly selected node called *auxiliary full node*. (ii) The auxiliary full node gets blocks based on the query and h (i.e. only blocks smaller than h are visited), and generates a digest according to the roots of MB-trees the query visited. The digest is returned to thin client. (iii) The client verifies the soundness and completeness of query results using VO and the digest. Actually, VO and generation of digest differ for different queries. For example, for tracking query, the VO consists of one VO each MB-tree the query visited. The auxiliary full node generates digest by hashing the concatenation of merkle roots of second level index in blocks that the query needs to visit.

$$p_w = p * \sum_{i=0}^{m-1} C_{m-1+i}^{i} * p^{m-1} * (1-p)^i \tag{4}$$

$$p_r = (1-p) * \sum_{i=0}^{m-1} C_{m-1+i}^{i} * (1-p)^{m-1} * p^i \tag{5}$$

$$\theta = \begin{cases} \frac{p_w}{p_w+p_r} & (m+i \le n \ \&\& \ m \le max) \\ 0 & (m > max) \end{cases} \tag{6}$$

Certainly, the auxiliary node can also be untrusted, to handle this situation, clients can reduce the risk by sampling from multiple nodes. Suppose the ratio of Byzantine nodes to the number of summary points is p (may be different under different consensus protocol), the client sends out n request to n auxiliary nodes and receive m identical digests. Consider there are at most max byzantine nodes. The probability θ that the digest is wrong is described by equation 6. A user can adjust n and m to achieve different credibilities. Then, the client can re-construct the digest with VO it received and compare it with the correct one to check query result.

It is convenient to modify Algorithm *1-3* to support Track-trace and Join based on the ALI and authenticate query algorithms in [21] [23]. For a specified node, it runs the corresponding algorithms to answer the query request from a thin client. We omit the details of the algorithms due to space limitation.

Example 4: The bottom half of Figure 5 shows a simple example of authenticated track-trace query with an authenticated index on Tname. Assuming there are four full nodes with consensus protocol of PBFT [25], the result is guaranteed if two identical responses are returned. At the first phase, a thin client sends a query request to a randomly selected full node. At the full node side, the first 2 steps are similar to Example 3 except that current the block height 10 is added to VO. Since only block 9 contains query result, the server then traverses MB-tree on block 9 (step 3). During the traversal, left sibling nodes along the search path are added to *VO*, h_A in this example. At the leaf level, T_{j+1} along with left boundary T_k and right boundary T_p is added to *VO*. Finally, the server traverses MB-tree with right boundary and adds right sibling node along the search path to *VO*, (h_C in this example) (step 4). For auxiliary node, it receives block height 10 and query from client, the first 2 steps are similar to example 3. At step

TABLE II
BENCHMARKING QUERIES

Q1	INSERT INTO $donate$ VALUES(?,?,?);
Q2	TRACE OPERATOR = "$org1$";
Q3	TRACE $[start, end]$ OPERATOR = "$org1$", OPERATION= "$transfer$";
Q4	SELECT * FROM donate WHERE $amount$ BETWEEN ? AND ? ;
Q5	SELECT * FROM $transfer$, $distribute$ ON $transfer.organization = distribute.organization$;
Q6	SELECT * FROM $onchain.distribute$, $offchain.donorinfo$ ON $distribute.donee = donorinfo.donee$;
Q7	GET BLOCK ID=?;

3, since only block 9 contains query result, it hashes $Root_9$ for digest d and returns it to client (step 4).

At the client side, it sends query to the full node and auxiliary full nodes to receive *VO* and digest d. Once it receives two identical digests. It reads *VO* and reconstructs root of MB-tree on block 9 using $(h_A, T_k, T_{j+1}, T_p, h_C)$. Boundary transactions T_k and T_p are included in *VO*, thus the client can verify completeness of result. It then hashes the reconstructed root of block 9 using the same hash function as auxiliary full node for digest. Since it is identical to d, thus query result of block 9 is correct.

VII. Experimental Evaluation

All experiments were conducted upon a cluster, where each node, equipped with two 16-core 2.10GHz Intel Xeon CPU, 96GB RAM, 3TB Raid 5 disk space and 1 Gbps network throughput, was running CentOS version 7 operation system. All codes are written in C++, and MySQL Community Server 5.7.21 is used to store off-chain data. We adopt KAFKA[4] 1.0.0 and Tendermint[5] 0.19.3 as consensus components.

A. BChainBench: a mini benchmark for Blockchain Database

Since there exists no benchmark for blockchain database from the system point of view till now, we define a new mini benchmark, called BChainBench, to suit for this scenario. Recall that BLOCKBENCH mainly focus on consortium blockchains [30].

Blockchain Database Schema: Figure 6 introduces a schema that consists of 7 tables in a donation system, where *Donate*, *Transfer* and *Distribute* are main on-chain tables, and the rest four are off-chain tables. *Donate*, *Transfer* and *Distribute* record donation, money transfer and donation distributions respectively. Off-chain tables store private information of corresponding participants. Table *DonorInfo*, maintained by charity, stores detailed information of donors. Table *DoneeInfo*, maintained by a school, stores private data of donees,

[4]http://kafka.apache.org/
[5]https://tendermint.com/

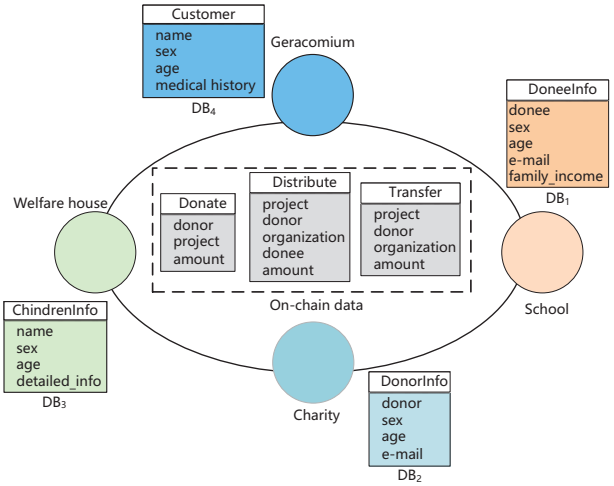

Fig. 6. Blockchain Database Schema

such as family income, etc. Table *ChildrenInfo*, maintained by welfare, stores private data of orphans. Table *Customer*, maintained by the nursing home, stores private data of old people. These tables are used for different applications. Since the processing of on-off chain join are similar, we take the case of querying detailed info of donees for illustrating performance of on-off chain join.

Data Generator: We implement a data generator to simulate real scenario from two dimensions, including time dimension and the dimension of data distribution in attributes. The time dimension describes the frequency of transactions, i.e., the physical distribution in blocks of a transaction (i.e. a tuple). The data distribution in attributes determine the size of a query result. This data generator supports uniform and Gaussian distribution of transactions.

Workload: The workload includes seven queries, listed in Table II. Q1 sends transactions with type *Donate* to test write performance. Q2 and Q3 test performance of tracking query. Q2 tracks all transactions of charity $org1$ from one dimension while Q3 tracks $org1$'s money transfer operations in a time window from two dimensions. Q4 and Q5 test performance of querying tables. Q4 is a basic range query on *Donate*. Q5 joins on-chain tables to show how donations are dispatched. Q6 integrates on-chain and off-chain data to show detailed information of donees. Q7 is a basic operation of blockchain which finds a specified block.

Metrics: Write throughput is measured as completed transactions per second. Query latency is measured as the response time for a query. Since the authenticated query is completed at server side and client side, it is measured by VO size, running time at client side and running time at server side.

Data Set: We change result size or blockchain size for testing. To test the performance under different blockchain size, we fix the result size and vary number of blocks from 500 to 2,500. To test the performance under different result size, we fix blockchain size to 1000 blocks and vary result size. Resulting transactions either follow uniform or Gaussian distribution (with mean equals to the middle of block and

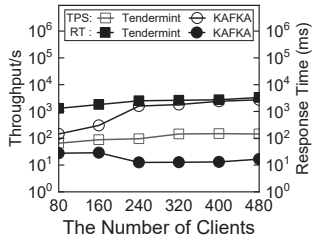

Fig. 7. Write Throughput

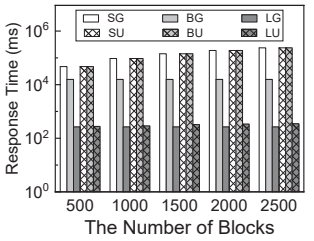

Fig. 8. Varying Tracking Data Size

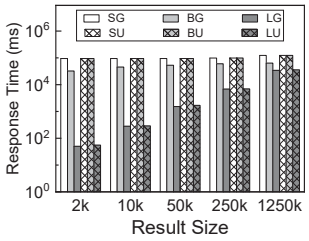

Fig. 9. Varying Tracking Result Size

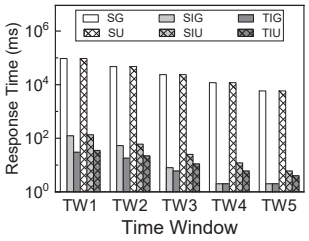

Fig. 10. Varying Time Window

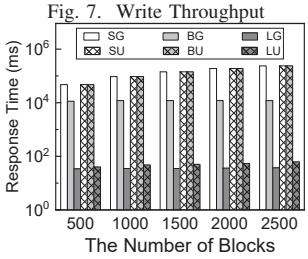

Fig. 11. Varying Range Data Size

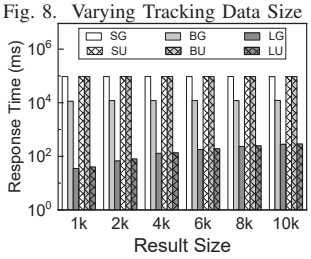

Fig. 12. Varying Range Result Size

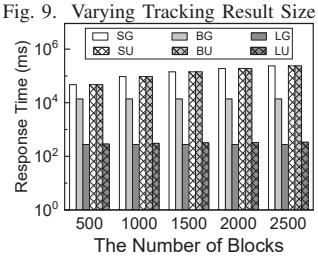

Fig. 13. Varying On-chain Data Size

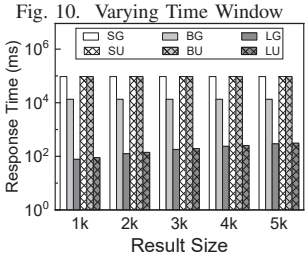

Fig. 14. Varying On-chain Result Size

variance set to 20) among blocks to simulate different data distributions. For authenticated query test, result size is fixed to 10,000, and transactions follow uniform distribution among blocks. We vary number of blocks from 500 to 2,500 to test the performance of authenticated query. To test cache, we fix the blockchain size to 1,000 blocks.

Important parameter settings include: The page size of MB-tree implementation is 4 KB, SHA256 is used for authenticated index structure, the block size is 4 MB, and the transaction size is 300 Byte.

In the rest of this paper, we use *SU, SG, BU, BG, LU, LG* to denote a run of query with scan, bitmap index and layered index following uniform and Gaussian distribution respectively. We use *SIU, SIG, TIU* and *TIG* to denote track with single or two indices under uniform and Gaussian distribution.

B. Write Performance

We evaluate write performance of SEBDB with different consensus components. We deploy 4 servers and vary the number of clients connected to them. We use default settings for Tendermint. For KAFKA, we start 1 broker and create a $trnsaction$ topic with 1 partition. The block size is set to 200 transactions and timeout for packaging is set to 200 ms. A client works as follows: It first sends a transaction to system, and then waits for a response from the system before it sends next transaction. Each client sends 100 transactions.

Figure 7 shows the throughput and response time of the system. The system has higher throughput with KAFKA than with Tendermint. The response time of Tendermint keeps almost unchanged when the number of clients is small, because the block size for packaging is set to 10,000, thus the system package the block according to timeout. Each transaction sent to Tendermint is first checked by and then delivered to SEBDB in a serial manner, which is a slow process. Thus, the throughput of Tendermint is limited. And the response time increases with the increment of clients, due to resources competing. For KAFKA, the throughput increases as clients increase, it comes to a threshold at 400 clients for a single thread is responsible for packaging and appending block to disk. Actually, by decoupling the processes, the throughput can be higher. The response time decreases when the number of clients is 240. This is because when the number of clients is small, the packaging time depends on the timeout and when it rises, blocks are filled and packaged quickly.

C. Tracking Performance

We test the performance of tracking query under three methods, including (i) scanning all blocks, (ii) using bitmap index, and (iii) using layered index. Figure 8 and 9 report the results when executing Q2. In Figure 8, the result size is fixed to 10,000. Layered index method significantly outperforms the other two methods under all situations, since it can find results with the help of an index. Moreover, *BG* behaves better than *SG* for it reads fewer blocks. *LG* behaves better than *LU* for fewer blocks are searched. In Figure 9, the result size rises from 2,000 to 1,250,000. Since the result size if large, we set the variance to 50 for Gaussian distribution. When the number of resulting transactions rises, the difference among three methods will be reduced. The response time increases for all methods due to disk IO, serialization overhead and network transmission.

We next evaluate the performance of Q3. *SIU* and *SIG* use an index on operator, and *TIU* and *TIG* uses two indices on operator and operation. The blockchain contains 10,000 *transfer* transactions and 10,000 transactions operated by $org1$. Moreover, there exist 1,000 resulting transactions, i.e, *transfer* transactions sent by $org1$. Q3 accepts a pair of window parameters $(start, end)$. Let $start = 0$, and change $start$ to timestamp of block $(1000 - 1000/2^{i-1})$ for TW_i. Figure 10 shows the performance of Q3. We observe that *TI* outperforms the other methods for it directly finds all resulting transactions. When the window size decreases, all methods will run faster.

D. Range Query Performance

We then test the performance of range query under *scan*, *bitmap* and *layered index* (the depth of histogram is set to 100) methods.There exist 10,000 *donate* transactions. Figures 11 and 12 report the results for such methods by executing Q4. In Figure 11, the number of resulting transactions is fixed

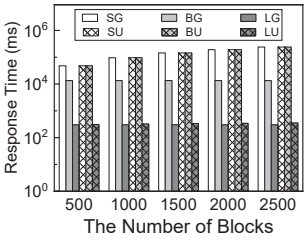

Fig. 15. Varying On-off Data Size

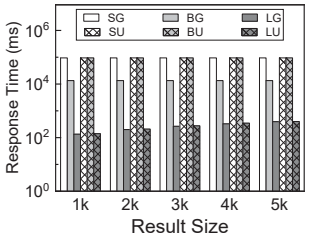

Fig. 16. Varying On-off Result Size

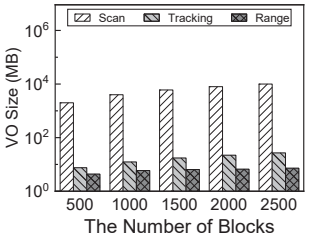

Fig. 17. VO size

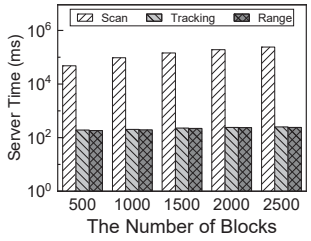

Fig. 18. Running Time at Server Side

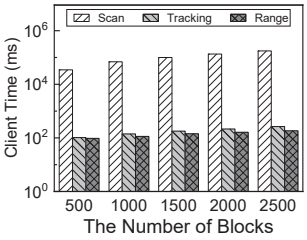

Fig. 19. Running Time at Client Side

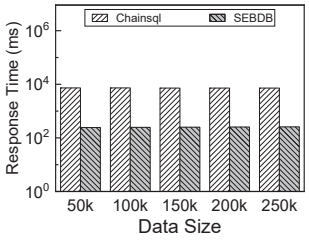

Fig. 20. One-dimension Tracking

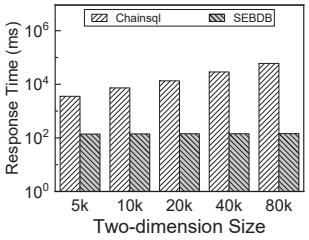

Fig. 21. Two-dimension Tracking

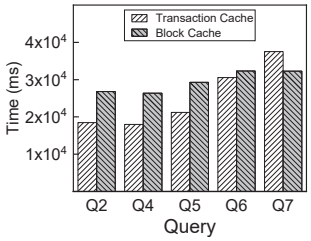

Fig. 22. Cache

to 1,000. *Layered index* method significantly outperforms the other two methods under all situations, since it can find the resulting transactions with the help of an index. Moreover, *BG* method behaves better than *SG* for resulting transactions are distributed in fewer blocks. In Figure 12, the number of resulting transactions rises from 1,000 to 10,000. When the number of resulting transactions rises, the difference between three methods decreases. Moreover, other than layered index method, *bitmap* and *scan* are insensitive to the result set, i.e, the running time remains almost unchanged since the result size is small.

E. Join Performance

We evaluate the performance of on-chain join and on-off chain join under three different methods, including (i) Hash join with scan (all blocks are scanned) (ii) Hash join with bitmap index (blocks containing transactions of corresponding tables are scanned), and (iii) nested loop join with layered index (blocks may produce join results are read by index). In each set of experiments, we change either blockchain size or result size. Each table has 10,000 transactions, the result size is set to 5,000 for testing performance under different number of blocks.

Figures 13 and 14 report the performance of Q5. Layered index method significantly outperforms other two methods for the following reasons: (i) it only compares block pairs that may produce join result; (ii) taking advantage of the second level index, it avoids scanning the whole block; (iii) since transactions are sorted at the leaf level, it can execute sort merge join between two pairs quickly. *BG* outperforms *SG* since fewer blocks are read. As the number of blocks increases, the response time of *LU* increases for more pairs of blocks are compared. As the result size gets greater, more blocks take part in join processing and more transactions are read from disk, thus the performances of *LU* and *LG* decrease.

Figures 15 and 16 report the results for on-off chain join by executing Q6. Layered index method outperforms the other two methods, because it only reads blocks that may produce results by second-level index. The performance of *BG* outperforms *SG* for it reads fewer blocks for joining. Similar to on-chain join, the response time of *LU* increases as the number of blocks increases. The performances of *LU* and *LG* decrease as the result size increase.

F. Authenticated Query Performance

We then test authenticated trace query and range query on three metrics, including VO size, computation overhead at the client side and query processing time at the server side. The cost at the auxiliary sever side is small for it only reads roots of MB-trees. We compare the performance of authenticated index with a basic approach where all blocks are transferred to the client and the client checks transactions by reconstructing transactions merkle roots for each block.

For ALI, the VO size depends on (i) the number of blocks that contain query results, (ii) VO size of a block (contains query results and the corresponding sibling hash values in the searching path). The processing time at the server side depends on (i) the number of blocks that contain query results (ii) the cost to read transactions by ALI. The computation cost at the client side depends on the number of blocks returned from the server and the cost to reconstruct the root for the MB-tree on a block. For the basic approach, the VO size depends on the number of blocks in the system. The processing time at the server side depends on the number of blocks in the system and the cost to scan a block. Computation cost at the client side depends on number of blocks and the cost to reconstruct root of the transaction merkle root.

We evaluate the performance of Q2 and Q4. There are 10,000 resulting transactions for both queries. There are 100,000 transactions of $donate$ distributed uniformly among blocks. The VO size of ALI is always smaller than the basic approach since the verification object on a block is smaller, as shown in Figure 17. The VO size of Q4 is smaller than that of Q2 for the ALI of Q2 (which indexes all transactions) is greater than that of Q2. The query processing time at the server side for ALI is smaller for less disk IO with the help of index, as shown in Figure 18. The computation time at the client side for ALI is smaller since the verification of ALI is

more efficient than reconstructing the transaction merkle tree, as shown in Figure 19.

G. Tracking Performance Comparision

We compare the performance of SEBBD with Chainsql by executing Q2 and Q3. Since Chainsql [3] is a multi-active database along with the data-level disaster recovery backup and audibility features based on Ripple [7]. It does not optimize the performance of tracking specially. For Chainsql, we implement the tracking query using the *GET_TRANSACTION* api. All resulting transactions are distributed uniformly among blocks. Firstly, we vary the blockchain size and fix the result size to 10,000 to test Q2. Both SEBDB and Chainsql are insensitive to the blockchain size since they use indices, as shown in Figure 20. We then test the performance of Q3. The blockchain size is fixed to 100,000 transactions and the result size is fixed to 5,000. We vary number of transactions of $org1$ from 5,000 to 80,000 and fix the number of transactions of $transfer$ to 5000. The performance of SEBDB almost keeps unchanged since it get resulting transactions with the help of optimized tracking operation. For Chainsql, all transactions of $org1$ are returned and filtered by client, thus the latency increases as the number of transactions of $org1$ increases, as shown in Figure 21.

H. Block cache vs. Transaction cache

We compare the performance of query with block cache and transaction cache where block cache stores recently read blocks and transaction cache stores recently read transactions by index. We set the cache size to 2GB and adopt LRU for both caches. We run these queries for 10 minutes before test for cache warming. We test performance of Q2, Q4, Q5, Q6 and Q7 with layered index. There are 10,000 transactions of $donate$, $transfer$ and $distribute$. For tracking query and range query, the result size is fixed to 10,000. For join query, the result size is fixed to 5,000. We start a client for each query, each client sends 100 requests. Figure 22 shows the processing time of these queries. For Q2, Q4, Q5 and Q6, the performance of transaction cache is significantly better than block cache since recent queried transactions are cached. Although performance of block cache for Q7 is better for recently read blocks are cached, the query is unusually used.

VIII. Conclusion

Since convenience and efficiency of data management are critical challenges in existing blockchain systems, we propose and implement a novel blockchain database, called SEBDB, to integrate database characteristics and blockchain system. Currently, existing works mainly focus on some concrete aspects of data management such as data storage and query optimization, while ours is the first piece of work that deals with data management issue from system perspective. We make following innovations in this paper, (i) adding relational semantics into blockchain system, (ii) using SQL-like language as the interface to avoid code-level APIs, (iii) redefining several representative operations to suit for blockchain system, and (iv) proposing a small benchmark to evaluate blockchain databases. In future, we will continue to enrich query language, optimize data storage and query processing.

Acknowledgments

Our research is supported by the National Key Research and Development Program of China (2016YFB1000905), NSFC (U1811264, U1501252, 61532021). Zhao Zhang is the corresponding author. The authors would like to thank the anonymous reviewers for their valuable feedback.

References

[1] "Ethereum," https://www.ethereum.org.
[2] E. Androulaki, A. Barger, V. Bortnikov, and e. Cachin, "Hyperledger fabric: A distributed operating system for permissioned blockchains," 2018.
[3] "Chainsql," http://www.chainsql.net/.
[4] T. McConaghy, R. Marques, A. Müller *et al.*, "Bigchaindb: a scalable blockchain database," *white paper, BigChainDB*, 2016.
[5] "Mongodb," https://www.mongodb.org.
[6] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Consulted*, 2008.
[7] "Ripple," https://ripple.com.
[8] "Eos," https://eos.io/.
[9] J. Shute, R. Vingralek, B. Samwel *et al.*, "F1: A distributed sql database that scales," *Proceedings of the Vldb Endowment*, vol. 6, no. 11, pp. 1068–1079, 2013.
[10] A. Verbitski, A. Gupta, D. Saha *et al.*, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *SIGMOD*, 2017.
[11] F. Frber, K. C. Sang, J. Primsch *et al.*, "Sap hana database: Data management for modern business applications," *Acm Sigmod Record*, 2012.
[12] T. T. A. Dinh, R. Liu, , M. Zhang *et al.*, "Untangling blockchain: A data processing view of blockchain systems," *TKDE*, pp. 1366–1385, 2018.
[13] "Leveldb," https://github.com/google/leveldb.
[14] J. Guo, J. Chu, P. Cai, M. Zhou, and A. Zhou, "Low-overhead paxos replication," *Data Science and Engineering*, pp. 169–177, 2017.
[15] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC*, 2014.
[16] S. Wang, T. T. A. Dinh, Q. Lin *et al.*, "Forkbase: An efficient storage engine for blockchain and forkable applications," *PVLDB*, 2018.
[17] M. Y. Y. Yang Li andand Zhang *et al.*, "Etherql: A query layer for blockchain system," in *DASFAA*, 2017.
[18] H. Pang and K. Tan, "Authenticating query results in edge computing," in *ICDE*. IEEE Computer Society, 2004, pp. 560–571.
[19] M. Narasimha and G. Tsudik, "Authentication of outsourced databases using signature aggregation and chaining," in *DASFAA*, 2006.
[20] H. H. Pang, A. Jain, R. Krithi *et al.*, "Verifying completeness of relational query results in data publishing," in *ACM SIGMOD*, 2005.
[21] F. Li, M. Hadjieleftheriou, G. Kollios *et al.*, "Dynamic authenticated index structures for outsourced databases," in *ACM SIGMOD*, 2006.
[22] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Authenticated index structures for aggregation queries," *Acm Transactions on Information & System Security*, 2010.
[23] Y. Yang, D. Papadias, S. Papadopoulos *et al.*, "Authenticated join processing in outsourced databases," in *ACM SIGMOD*, 2009.
[24] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*, 1989.
[25] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999.
[26] R. V. Renesse and Dan, "Efficient reconciliation and flow control for anti-entropy protocols," in *The Workshop on Large-Scale Distributed Systems & MIDDLEWARE*, 2008.
[27] G. Decandia, D. Hastorun, M. Jampani *et al.*, "Dynamo:amazon's highly available key-value store," *ACM SIGOPS*, 2007.
[28] A. Lakshman and P. Malik, "Cassandra:a decentralized structured storage system," *Acm Sigops Operating Systems Review*, pp. 35–40, 2010.
[29] R. C. Merkle, "A certified digital signature," in *on Advances in Cryptology*, 1989, pp. 218–238.
[30] T. T. A. Dinh, J. Wang, G. Chen *et al.*, "BLOCKBENCH: A framework for analyzing private blockchains," in *ACM SIGMOD*, 2017.