Towards Rich Query Blockchain Database

Yanchao Zhu[1], Zhao Zhang[1,2], Cheqing Jin[1], Aoying Zhou[1], Gang Qin[3], Yingjie Yang[3]
School of Data Science and Engineering, East China Normal University, Shanghai, China[1]
Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, China[2]
Institute of Blockchain Technology, Ouyeel Co. Ltd., Shanghai, China[3]
yczhu@stu.ecnu.edu.cn,{zhzhang,cqjin,ayzhou}@dase.ecnu.edu.cn,{qingang,yangyingjie}@ouyeel.com

ABSTRACT

In this demo, we present SEBDB, a novel blockchain database that integrates immutability and transparency properties of blockchain with modeling and query ability of relational database. In summary, SEBDB has the following advantages: First, it adopts the linked structure and full replication of data among multiple participants to guarantee immutability and transparency. Second, it introduces the relational model to blockchain without introducing extra overhead, based on which relational queries are supported. SEBDB supports SQL-like language as the general interface to support convenient application development, in which intrinsic operations are re-defined and re-implemented to suit for blockchain platform. Third, it supports rich verifiable queries based on the proposed authenticated index, thin clients can participate in the system regardless of limitations of storage, network, and computing resources. We demonstrate the usability and scalability of SEBDB using a donation system.

KEYWORDS

Blockchain database; relational semantics; query processing

ACM Reference Format:
Yanchao Zhu[1], Zhao Zhang[1,2], Cheqing Jin[1], Aoying Zhou[1], Gang Qin[3], Yingjie Yang[3]. 2020. Towards Rich Query Blockchain Database . In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3340531.3417424

1 INTRODUCTION

Recent years have witnessed the widespread increase of interest in blockchain technology. From the data management perspective, a blockchain can be viewed as a decentralized database which ensures immutability and consistency of the data recorded from each participant through cryptographic hashing and consistency protocol techniques. Blockchain has been adopted in many applications to enable trust among multiple participants, such as supply chain management, digital assets transfer, philanthropy, etc. With the wider adoption of blockchain, there is an increasing demand to perform analytic queries over blockchains [7] [9].

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3417424

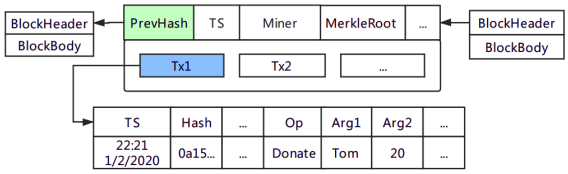

Figure 1: Block Structure

Although often referred to as decentralized database, blockchain systems are far less convenient to use than traditional databases in terms of query language and query processing. First, they cannot handle complex queries on block data in an efficient way. Block data are stored in key-value stores or raw file, both of which provide simple APIs to access data. To support complex queries, one can deploy a smart contract to store all the block data. However, it is impractical to store all data in the state database, as even the blockchain itself suffers from the huge storage overhead [5]. Another way is to import block data into an off-chain database for querying [4], however, it takes extra overhead of data migration and users have to afford the cost to maintain two systems. Second, the issue of querying on-chain and off-chain data at the same. Participants store the private data in off-chain databases for privacy, however, it makes it difficult to integrate on-chain and off-chain. Users cannot use smart contract to access off-chain data for the problems of consistency and privacy. Importing data into off-chain databases also doesn't work because the block data is repeatedly stored in different off-chain databases, taking significant storage overhead. Third, existing systems cannot handle the situation for individual users. Individual users are usually equipped with mobile phones and cannot participate in the consensus process due to limitations of storage, network, and computing resources. They have to query full nodes that store all blockchain data for query results and verify the correctness of results. Traditional blockchain systems only support simple verifiable queries, like SPV in Bitcoin. vChain [9] and Ruan et al. [7] extend current systems to support verifiable boolean range query and provenance query. Compared with outsourced database which supports rich verifiable query types [1], verifiable query on blockchain remains to be studied.

To handle the above issues, we propose SEBDB (semantic empowered blockchain database), a blockchain database that leverages techniques from existing databases that have been optimized for decades, to improve the usability and scalability of blockchain system, in our prior work [10]. Figure 1 shows the general structure of blockchain. Each block consists of a block header and a block body. A block header contains system fields such as previous block hash, timestamp, etc. A block body contains a number of transactions that denote operations to change the global state. As we can see, a transaction is structured, consisting of system fields such as hash value, timestamp, etc. and user-defined arguments like operation

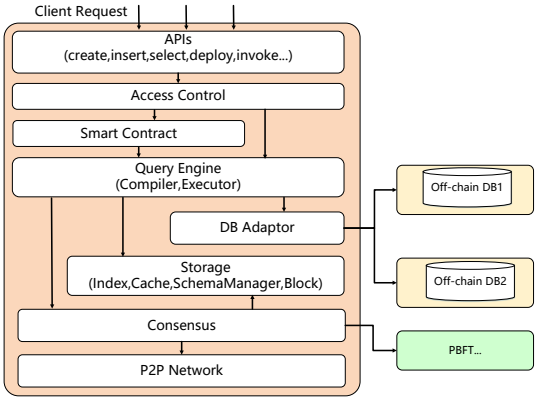

Figure 2: System Architecture

name and arguments of the operation. Based on this, SEBDB introduces relational semantics to blockchain platform, to address the above issues. Block data is modeled as a number of relations (each transaction is a tuple belongs to a certain relation). Block data is also the data of data tables, thus, the system only maintains one copy of data. Moreover, we develop SQL-like language to help application developers to access block data friendly. Then, some operators are re-defined or re-implemented for blockchain platform based on indices. Since block data is modeled by the relational model, SEBDB supports on-off chain join operation to integrate on-chain and off-chain data without introducing extra storage overhead. SEBDB takes advantage of authenticated data structure (ADS, e.g., variants of Merkle hash tree) and $t-n$ threshold signature [8] to support verifiable query processing for thin clients. In this way, thin clients need not store any data but can verify the correctness of query results. For a query, the full node constructs a *validation object* (VO) (e.g., merkle proof) for the query result. Thin clients can verify the correctness of the query result according to VO and threshold signature of the root of the ADS.

The demonstration aims to show the usability and scalability of SEBDB. In particular, we demonstrate our system via a donation system that requires transparency as well as involves various queries from multiple participants. Attendees interact with the system from the perspective of individual users or full nodes and feel the conveniences of SEBDB.

The rest of the demonstration proposal is organized as follows. Section 2 introduces the design of SEBDB. The demonstration details are presented in Section 3.

2 SYSTEM OVERVIEW

In this section, we briefly revisit the design of SEBDB, readers can refer to [10] for detailed descriptions.

2.1 Data Model and APIs

SEBDB is modeled by the relational model, a table schema is declared for transactions of the same type. The attributes are divided into two types, including application-level attributes and system-level attributes. The system-level attributes are added automatically, such as transaction id, signature, etc. The application-level attributes are defined explicitly by users, and the attributes types can be string, various flavors of numbers, etc. Basic operations in RDBMS like project, select, cartesian operations, etc. are also supported. We provide a SQL-like language to access data. Users can use CREATE, INSERT, and SELECT clauses to create a table, send a new transaction, and get query results respectively. Other operations such as GET BLOCK, etc. are also supported to query the blockchain. In particular, the system table $TransactionTable$ is created automatically to support queries on all transactions, the application-level attribute $Tdata$ of $TransactionTable$ represents all user-defined parameters. Since a query may focus on data within a time period, SEBDB supports time-window queries by specifying a time window in a query request.

2.2 System Architecture

SEBDB consists of five layers, including application, query processing, storage, consensus, and network, as shown in Figure 2.

Application Layer. This layer has two components: access control and smart contract. Access control verifies request permission before execution, where a multi-channel method is adopted to protect privacy. SEBDB supports smart contract to define a DApp (Decentralized Application), where a SQL-like language is embedded for accessing data.

Query Processing Layer. This layer aims at parsing, optimizing, and executing queries. Details are shown in Section 2.3.

Storage Layer. This layer is responsible for data storage and indices, i.e, creation and maintenance of storage structures such as index, Merkle tree, cache as well as search and insertion of data. Details are shown in Section 2.4.

Consensus Layer. As different consensus protocols are suitable for different scenarios, SEBDB uses plug-in pattern, allowing users to choose different consensus protocols according to their requirements. Currently, we support KAFKA and PBFT [2] as consensus components.

Network Layer. We adopt Gossip [6] as the basic network facility since it is widely used not only in distributed databases for failure detection and membership protocol but also in blockchain for block propagation and data recovery.

2.3 Storage and Index

2.3.1 Data Storage. Transactions are packaged into blocks for consensus. Each block contains the hash of the previous block to guarantee immutability. New blocks are appended to a file. The size of the file is configurable (by default 256MB). When the size of the file comes to the threshold, a new file is created.

2.3.2 Index Mechanisms. The link structure of blocks ensures the immutability of data. However, transactions belonging to a table are distributed in blocks according to the time of occurrence, which is unfriendly to relational queries. We devise indexing mechanisms to hasten data access for three types of tasks: (i) getting blocks, (ii) getting transactions of the same transaction type, and (iii) getting transactions under some condition.

Block Index. The block index is used to handle the first type of task, which is a B^+-tree with key $< bid, tid, T_s >$, where bid is the block id, tid is the first transaction id in the block, and T_s represents the timestamp when the block was packed. The leaf node stores a pointer to the location of the block. Since the values of bid, tid, and T_s are incremental (i.e., bid of $block_i$ is less than that of $block_{i+1}$, and the same for tid and T_s), so the indexing supports querying from any of these three dimensions.

Table Index. Table index can be used to handle the second type of task. A bitmap index is maintained to keep track of the

distribution of tables to avoid scanning all blocks. Each bitmap refers to a table, and the i-th bit in the bitmap indicates whether block i contains transactions for that table. Thus, looking up the bitmap index avoids scanning all the blocks to query the table.

Compacted Layered Index (CLI). CLI is used to handle the third task when the result size is small. We adopt the structure of LSM-tree so that the updates of CLI have little impact on the write performance. The first layer of CLI consists of bitmaps to filter blocks that contain no query results. An equal-depth histogram is generated by sampling historical data, with buckets representing continuous ranges of the index key attribute. Each block is denoted by an index entry with bucket ids representing ranges of transactions in the block. The second level index is an LSM-structured component. C_0 is the memory component, consisting of one B^+-tree for each block. C_1 is the disk component, which is a B^+-tree with key $< attr, timestamp >$. When the size of C_0 comes to a threshold θ, C_0 is merged into C_1 to reduce disk IOs when the index size is large. For a query request, it firstly searches the first layer index, most blocks that contain no query result can be filtered earlier by the first-level. The rest blocks are visited by the second layer index.

2.3.3 Authenticated Index. SEBDB supports creating authenticated indices to support verifiable query processing. We modify CLI and propose **authenticated compacted index** (ACI) by replacing C_0 and C_1 with MB-tree [3] to support verifiable queries. MB-tree can be used to support verifiable queries where thin clients can validate the correctness of query results. MB-tree is a combination of B^+-tree and Merkle hash tree where each leaf node contains the hash values of records, and each internal node stores the hash of the concatenation of its children's hash values. We replace C_0 of CLI with an MB-tree with index key $< attr, timestamp >$ to reduce the size of VO returned to the client. Updates of ACI are executed in a copy-on-write way so that read operations are not blocked when maintaining the index. As there is no trusted node in the system, thin clients have to query multiple nodes to ensure the correctness of query results, which takes extra overhead. To handle this situation, we take advantage of *t-n threshold signature* to reduce the overhead. In the schema of *t-n threshold signature*, if the threshold signature of a message m is validated, then t of n participants have signed the m. We generate threshold signatures for the roots of ACI. We integrate the generation of threshold signatures with the consensus protocol of PBFT. When a block b_i reaches consensus, the threshold signatures of ACI for block b_{i-1} is generated. In this way, thin clients can query one node and validate the correctness of query results according to the signatures.

2.4 Query Processing

There are two kinds of queries in SEBDB, query processing for full nodes, and verifiable query processing for individual users (i.e., verifiable query processing).

2.4.1 Query Processing. SEBDB supports basic operations of RDBMS as well as specific operations for the blockchain database. In order to improve query performance, we have re-implemented some relational operations based on the index structures above to avoid scanning the entire blockchain, such as Select, Join, etc. As block data is modeled by the relational model, SEBDB supports integrating on-chain and off-chain data through on-off chain join operation. Off-chain data is queried through the DB Adaptor component,

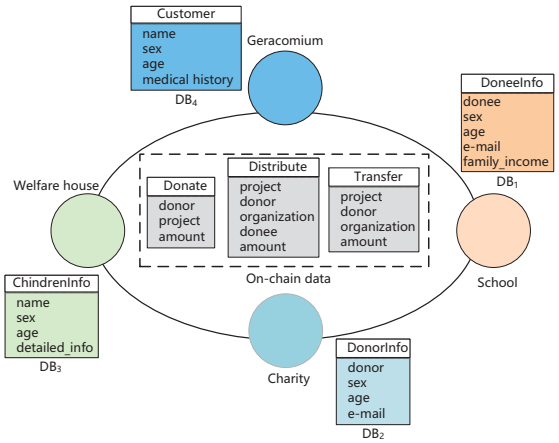

Figure 3: Database Schema

which maintains the off-chain database's Connections. Users can adjust the configuration file to connect to different databases. In this way, SEBDB supports the integration of on-chain and off-chain data without introducing additional storage overheads. On-chain and off-chain operations are performed locally, so no privacy issues are introduced.

2.4.2 Verifiable Query Processing. The query processing layer also supports verifiable query processing based on authentication indexes. SEBDB supports a variety of verifiable query processing. We use a range query as an example to illustrate verifiable query processing. ① A client sends a range query $l \leq t.attr \leq u$ to a full node and waits for VO which contains query results and validation information. ② Upon receiving the query request, a full node executes range query on C_0 and C_1 of ACI as the way described for MB-tree to get VO. Then, it adds threshold signatures of C_0 and C_1 to VO. ③ Then, it returns VO to the client. ④ Upon receiving VO, the client reconstructs the roots of C_0 and C_1 according to VO and compares them to the signed ones. If they are the same, then the query result is correct.

2.4.3 Time-window Query Processing. Queries with a time window can be done efficiently based on the indices. For a time-window query, first, it searches the block index for the set B containing block ids within the time window. Only the blocks in B participate in the query processing. Then, the query is executed on the blocks involved in the time window. For verifiable queries, C_0 or C_1 will be searched only if they overlap with the time window.

3 DEMONSTRATION

In our demonstration, we show how SEBDB supports various query demands from different users in the scenery of a donation system.

There are five participants in the application, including charity organization, school, welfare house, geracomium, and auditing department. Each participant stores its private data in an off-chain database, public users interact with the system through thin clients. There are various query demands from both participants and public users. For example, a school wants to query detailed information of a donee of a project (i.e., integrate on-chain and off-chain data), a donor who maintains a thin client wants to query how his donations are dispatched, and the auditing department wants to track all transactions of a specified participant, etc. These queries are easily handled by SEBDB without additional development efforts.

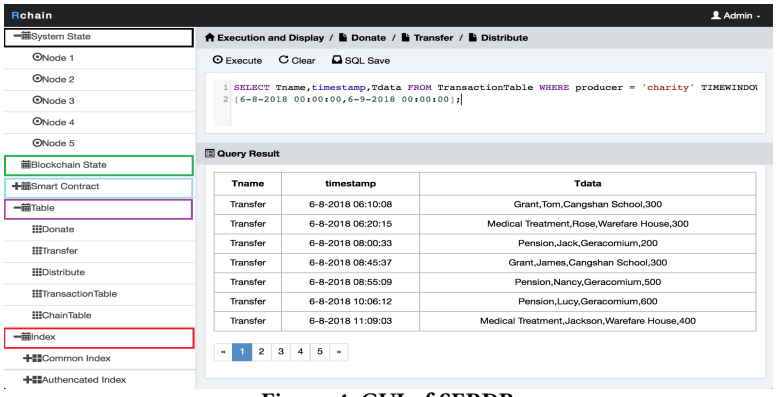

Figure 4: GUI of SEBDB

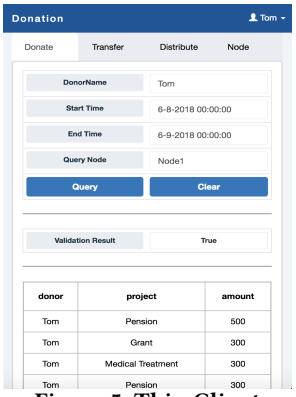

Figure 5: Thin Client

To illustrate the benefits of SEBDB, we present to the audience the use of SEBDB for both participants that maintain full nodes and thin clients equipped with a smartphone.

Figure 4 visualizes operations from participants that maintain full nodes. Initially, users see a navigation bar at the left part of the web page, including (i) System State, showing blockchain nodes status in the current system; (ii) Blockchain State, showing linked blocks in the system; (iii) Smart Contract, showing smart contracts in the system; (iv) Table, showing tables; (v) Index, showing indices and authenticated indices. Users can click these functions for detailed descriptions. At the right part of the web page, users can input SQL-like language and click the execute button, then query results will be shown at the bottom of the page.

Scenario 1: Application Deployment. Three tables are created to record operations from participants, i.e. donate, transfer and distribute using **CREATE** clause, each representing a transaction type, as shown in Figure 3. *Donate* denotes donate operation from a donor. $Transfer$ denotes transfer of money from a project to an organization and *Distribute* denotes distribution of donations from organizations to donees. Tables are updated by sending transactions to the system. For example, when a donor donates to a project, a new transaction of donate is sent to the system using the **INSERT** clause. As we can see, it is easy to deploy an application on SEBDB through the SQL-like language. More complex business logic can be supported using smart contract.

Scenario 2: Query Processing. Users can input query requests using **SELECT** clause to get query results. An example of tracking queries which tracks all transactions of participant *charity* within time window [6-8-2018 00:00:00, 6-15-2018 00:00:00] is shown in Figure 4. This corresponds to a relational query on $TransactionTable$. Query results are shown at the bottom of the web page. SEBDB supports rich queries through the SQL-like query language. Users can create indices to improve query performance. Indices can be created dynamically using the **CREATE [AUTHENTICATED] INDEX** clause, which takes both convenience and efficiency for application development. Users can choose different depths of the histogram for different precisions. In this demonstration, we create authenticated indices on *donor* of *donate*, *organization* of $transfer$ and *donee* of *distribute* to support verifiable queries. As we can see, it is convenient to optimize query performance by creating index dynamically in SEBDB.

Scenario 3: Thin Client. Figure 5 shows the thin client on a smartphone. We provide a user-friendly interface to users and hide SQL-like language behind. Three functions are provided to users, i.e. (i) *Donation*, showing donation transactions of a donor. The function corresponds to a verifiable query on *donate* table; (ii) $Transfer$, showing transfer operations to the specified organization. The function corresponds to a verifiable query on $transfer$ table; (iii) *Distribution*, showing distributions to an organization. The function corresponds to a verifiable query on *distribute* table.

For the *Donation* function. A user can input the donor name and time window to query donate operations. The authenticated query processing is handled after clicking the query button. Then the VO received from the full node will be returned. After that, thin clients reconstruct the roots of MB-trees and compare them to the signed ones to validate the integrity of the result. In this demonstration, it queries donate transactions of donor Tom within time window [6-8-2018 00:00:00, 6-9-2018 00:00:00]. Since the validation result is true, the query result is correct. As we can see, thin clients need not store anything but can verify the correctness of query results, which expands the application scenarios of blockchain.

4 ACKNOWLEDGMENTS

This work is partially supported by National Science Foundation of China (U1811264, U1911203, 61972152 and 61532021), Guangxi Key Laboratory of Trusted Software (kx202005) and ECNU Academic Innovation Promotion Program for Excellent Doctoral Students. Cheqing Jin is the corresponding author.

REFERENCES

[1] Sumeet Bajaj and Radu Sion. 2013. CorrectDB: SQL Engine with Practical Query Authentication. *Proc. VLDB Endow.* 6, 7 (2013), 529–540.
[2] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In *OSDI*.
[3] Feifei Li, Marios Hadjieleftheriou, George Kollios, et al. 2006. Dynamic authenticated index structures for outsourced databases. In *ACM SIGMOD*.
[4] Yang Li, Meihui Zhang, Ying Yan, et al. 2017. EtherQL: A Query Layer for Blockchain System. In *DASFAA*.
[5] X. Qi, Z. Zhang, C. Jin, and A. Zhou. 2020. BFT-Store: Storage Partition for Permissioned Blockchain via Erasure Coding. In *ICDE*. 1926–1929.
[6] Robbert Van Renesse and Dan. 2008. Efficient reconciliation and flow control for anti-entropy protocols. In *The Workshop on Large-Scale Distributed Systems & MIDDLEWARE*.
[7] Pingcheng Ruan, Gang Chen, et al. 2019. Fine-grained, secure and efficient data provenance on blockchain systems. *Proc. VLDB Endow* 12, 9 (2019), 975–988.
[8] Victor Shoup. 2000. Practical Threshold Signatures. In *International Conference on Theory Application of Cryptographic Techniques*.
[9] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vChain: Enabling verifiable boolean range queries over blockchain databases. In *ACM SIGMOD*.
[10] Yanchao zhu, ZhaoZhang, Cheqing Jin, et al. 2019. SEBDB:Semantics Empowered BlockChain Database. In *ICDE*. IEEE Computer Society.