BFT-Store: Storage Partition for Permissioned Blockchain via Erasure Coding

Xiaodong Qi, Zhao Zhang *, Cheqing Jin, Aoying Zhou
School of Data Science and Engineering, East China Normal University, Shanghai, China
xdqi@stu.ecnu.edu.cn,{zhzhang,cqjin,ayzhou}@dase.ecnu.edu.cn

***Abstract*—The full-replication data storage mechanism, as commonly utilized in existing blockchain systems, is lack of sufficient storage scalability, since it reserves a copy of the whole block data in each node so that the overall storage consumption per block is $O(n)$ with n nodes. Moreover, due to the existence of Byzantine nodes, existing partitioning methods, though widely adopted in distributed systems for decades, cannot suit for blockchain systems directly, thereby it is critical to devise a new storage mechanism. This paper proposes a novel storage engine, called *BFT-Store*, to enhance storage scalability by integrating erasure coding with Byzantine Fault Tolerance (BFT) consensus protocol. First, the storage consumption per block can be reduced to $O(1)$, which enlarges overall storage capability when more nodes join blockchain. Second, an efficient online re-encoding protocol is designed for storage scale-out and a hybrid replication scheme is employed to improve reading performance. Last, extensive experimental results illustrate the scalability, availability and efficiency of BFT-Store, which is implemented on an open-source permissioned blockchain Tendermint.**

***Keywords*—Blockchain, erasure coding, Byzantine Fault Tolerance, storage scalability**

I. Introduction

Blockchain is a distributed append-only ledger maintained by all nodes in an untrustworthy environment. In order to avoid being tampered by malicious nodes, blocks in blockchain are chained by cryptograph hash values in the block head. In general, each node holds a complete copy of blocks, and consensus protocol, such as PoW [7] in permissionless blockchain or PBFT (Practical Byzantine Fault Tolerance) [2] in permissioned blockchain, ensures data consistency among all nodes. Hence, the overall storage consumption per block is $O(n)$, where n is the number of nodes. A natural, but challenging issue arises to be: *can we lower the storage consumption but still be capable of recovering the whole blockchain at any time?*

A straightforward concern is whether the partitioning manner (e.g. hash or range partitioning), which is widely adopted in distributed systems, suits for blockchain systems or not. In this mechanism, the original data is divided into a number of partitions and dispatched to all nodes. Particularly, for high availability, each partition is replicated for several times (e.g. 3 in Hadoop [5]) to tolerate fault of single node. However, this mechanism cannot suit for blockchain due to the existence of the malicious nodes who may keep silent or tamper data deliberately. For instance, if all replicas of a block happen to be dispatched to malicious nodes, this block may be tampered, or even lost.

*Corresponding author.

Alternatively, erasure coding (EC) provides significantly lower storage overhead than multiple-replication manner at the same fault tolerance level [11]. In a nutshell, erasure coding transforms data blocks into a set of coded blocks termed as "chunks" for redundancy. Reed-Solomon coding [9], denoted by (K, M)-RS, the most widely used EC given a set of K data blocks, encodes them with M checksum blocks, termed as "parities", and the K data blocks and M parties make up the $N = K + M$ chunks. The coding is done such that any K out of N chunks are sufficient to recover original data. Although solutions based EC can handle omission node failures (e.g. fail-stop) and won't taint other nodes, they cannot deal with Byzantine failures [3]. It will recover incorrect data based on the invalid chunks sent by Byzantine nodes. Besides, EC achieves storage efficiency at the cost of high repair penalty. Specifically, the repair of a single failed chunk (either lost or unavailable) needs to read multiple available chunks for reconstruction.

Intuitively, it is not hard to devise an EC-based storage mechanism for blockchain systems. It is common that existing Byzantine fault tolerance consensus protocols require the number of malicious nodes f below a certain rate of all nodes, such as no more than 1/3 nodes are malicious for PBFT [2]. Take the PBFT protocol as an example and consider the case that $n = 3f+1$ where at least $2f+1$ nodes are honest. We can divide a block into $2f+1$ *sub-blocks* and encode every $2f+1$ them into $n = 3f+1$ chunks with f parties via a $(2f+1, f)$-RS. Then the $3f+1$ chunks are distributed over $3f+1$ nodes, thereby at least $2f+1$ chunks are stored on honest nodes, which are sufficient to reconstruct the original block. In this way the storage consumption per block is reduced from $O(n)$ to $O(1)$. However, this method is still encountered with three main challenges.

Three Challenges. 1) How to guarantee all blocks never become lost in Byzantine environment. When $2f+1$ sub-blocks are encoded with f parities in PBFT, it is impossible to ensure all $2f+1$ honest nodes have reserved the correct chunks. Because the agreement of a block is reached for a node when it receives $2f+1$ commit votes from others, which only promises that at least $f+1$ honest nodes will commit the block due to the existence of f fault nodes. **2)** How to support flexible scalability with addition of new nodes. The arrival of a new node means redistribution of current data, i.e, historical blocks need to be re-encoded among all nodes. However, since no node preserves entire blockchain, it is challenging to ensure that honest nodes have accomplished re-encoding in a potentially hostile environment. **3)** How to enhance reading performance on encoded partitions distributed over nodes. Compared with

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00205

full-replication manner, the reading performance of blockchain declines due to data exchange among nodes for decoding, which involves $O(n)$ network transmissions.

Contributions. Our main contributions are as follows:

(1) Design a Byzantine fault-tolerant storage engine called ***BFT-Store***, the first work breaking full-replication, via EC for permissioned blockchain system, which reduces the storage complexity per block from $O(n)$ to $O(1)$.
(2) Propose two key designs to achieve scalability and speed up reading performance: i) a four-phase re-encoding protocol based on PBFT to promise the availability of all blocks; ii) a multiple replication manner to ensure efficient access of blocks.
(3) Integrate BFT-Store into an open-source blockchain system Tendermint [1], and the extensive experiments confirm the scalability, availability and efficiency of BFT-Store.

II. Problem Settings

Assumption. The basic assumption of BFT-Store is that out of the n nodes, at most $f = \lfloor \frac{n-1}{r} \rfloor$ ($r \geq 3$) are malicious, which can misbehave in arbitrary way. The fault factor r decides the rate of Byzantine nodes, i.e, no more than $1/r$ fault nodes exist. The fault factor r is configurable in BFT-store. For example, fault factor r in classic PBFT protocol is 3. In Zilliqa, the fault nodes can be up to 1/4 of all ($n \geq 4f+1$) [6], and at most 1/5 nodes can be malicious ($n \geq 5f+1$) in Ripple [10]. According to [8], given f Byzantine faulty node, no solution can deal with a system with fewer than $3f+1$ nodes. Besides, BFT-Store adopts *partial synchrony model* [4], using a known bound Δ and an unknown Global Stabilization Time (GST). After GST, all transmissions between two honest nodes arrive within time Δ.

Goals. Basically, BFT-Store must promise all blocks can be recovered by honest nodes, which is critical to blockchain system. Although partitioning methods based on EC improve storage efficiency, they degrade reading performance for remote data access at the same time. In particular, the performance deteriorates when a node pulls sufficient chunks from others to recover a block by decoding. Hence, we exploit BFT-Store with three goals. 1) **`Availability`**: each block must be available for all honest nodes, which means no block will be lost. 2) **`Scalability`**: the storage complexity per block keeps constant and storage capability of system is improved as n grows; 3) **`Efficiency`**: under guarantee of availability and scalability, compared to full-replication manner, the decline of reading performance of BFT-store should be as small as possible.

Concepts and notations. Instead of encoding each block, BFT-Store encodes every $n-2f$ original blocks into n chunks with $2f$ parities via a $(n-2f, 2f)$-RS, and each node preserves a unique chunk and hash values of the n chunks for verification. According to the assumption of BFT, at least $n-2f$ chunks are stored on honest nodes, and all blocks can be recovered based on these chunks by RS decoding. Specifically, a node just needs to request a set of chunks from $n-2f$ nodes to recover a original block by decoding when fails to pull the block from others directly. Note that BFT-Store employs a $(n-2f, 2f)$-RS rather than $(n-f, f)$-RS, because BFT protocol just ensures at least $n-2f$ (not $n-f$) honest nodes have committed blocks. Before in-depth discussion, we give some basic concepts and notations. 1) **`Schema and version number`** s: for a (K, M)-RS, (K, M) is called as the schema of RS coding. Whenever K or M varies, a new RS schema is applied and assigned a version number s. 2) **`Coding Round`** $\mathcal{R}_e^{(s)}$: the process of encoding $n-2f$ original blocks is considered as a coding round labeled by a unique sequence number e which starts at 0. The e-th coding round with schema version number s is notated with $\mathcal{R}_e^{(s)}$. 3) **`Block, parity and chunk`**: in $\mathcal{R}_e^{(s)}$ with (K, M)-RS, the $K = n-2f$ original blocks are denoted by $\{B_0^e, \cdots, B_{K-1}^e\}$, and the $M = 2f$ parities are $\{P_0^e, \cdots, P_{M-1}^e\}$. Both blocks and parities are called chunks in this study, represented as $\{C_0^e, \cdots, C_{N-1}^e\}$ ($N = n$). 4) **`Target node`** $N_t(B)$: the target node $N_t(B)$ for a block B refers to node storing it locally. 5) **`Reachable`**: block B is *reachable* for a node if B can be read without decoding, meaning that its target node $N_t(B)$ is honest. Let $P_r(B)$ denote the probability that block B is reachable.

III. Design

A. Encoding and Reading

Encoding. To partition blocks among n nodes, each node encodes $n-2f$ original blocks into n chunks $\mathcal{C} = \{C_0^e, \cdots, C_{n-1}^e\}$ with $2f$ parities in a coding round $\mathcal{R}_e^{(s)}$ independently and preserves a unique chunk. Note that the block smaller than the maximal one is appended with '0' for alignment. However, if target node $N_t(B)$ is malicious and keeps silent, block B becomes unreachable and can only be recovered by decoding, which reduces the reading performance. Moreover, the probability of this event is $1 - P_r(B) = f/n \approx 1/r$, which cannot be ignored.

To solve above problem, we employ the multi-replication technique to raise $P_r(B)$ to improve reading performance. In $\mathcal{R}_e^{(s)}$, each chunk C_i^e is replicated over c different nodes, where c is the number of replicas. Hence, a block has c different target nodes and notation N_t refers to a set of nodes rather than single node. Then block B is not reachable only if all of its target nodes $N_t(B)$ are fault, thereby the probability that B is not reachable is calculated as follows.

$$1 - P_r(B) = \prod_{i=0}^{c-1} \frac{f-i}{n-i} \leq \prod_{i=0}^{c-1} \frac{f}{n} = \left(\frac{f}{n}\right)^c \leq r^{-c}$$

It can be seen that the probability $1 - P_r(B)$ decreases exponentially with the growth of c, and the probability $P_r(B)$ exceeds $1 - r^{-c}$, e.g., when $c = 3$ and $r = 3$, $P_r(B)$ is greater than $1 - 1/27 = 26/27 \approx 96\%$. Suppose the average size of a block is T, then the average storage overhead per block is $\frac{cnT}{n-2f} \approx \frac{crT}{r-2}$, which is a constant complexity $O(1)$.

Example 1: Figure 1 illustrates an instance of BFT-Store with schema $(2, 2)$-RS versioned by 0. In a coding round, each chunk is replicated over 3 different nodes and each node preserves three chunks. For example, in coding round $\mathcal{R}_e^{(0)}$, blocks B_0^0 and B_1^0 are encoded into 4 chunks, including two block chunks C_0^0, C_0^1 and two parity chunks C_0^2 and C_0^3. Meanwhile, node N_0 stores three chunks, C_0^0, C_3^0 and C_1^0, and chunk C_0^0 is replicated on three target nodes N_0, N_1 and N_3 separately.

With c replicas for each chunk, each node N_i has to store c different chunks, namely chunk set $\mathcal{C}_e(i)$, in $\mathcal{R}_e^{(s)}$.

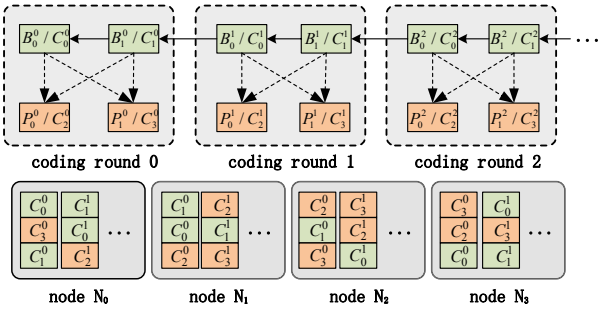


Fig. 1: Example of BFT-Store with $(2, 2)$-RS coding and 3 replicas for each chunk.

In BFT-Store, each node N_i computes its chunk set $\mathcal{C}_e(i)$ independently without any information exchange. To achieve this, all nodes are sorted by their public keys and the position in this list is treated as the global index of each node. The sorted node list is identical among all honest nodes because each node knows all others' public keys. For simplicity, we use number k to refer to the chunk C_k^e in a coding round. To construct $\mathcal{C}_e(i)$, node N_i with index i uses hash values of all blocks of coding round $\mathcal{R}_e^{(s)}$ as seed to generate c different numbers ranging from 0 to $n-1$, represented by $A = \{a_1, \cdots, a_c\}$. Then the chunk set $\mathcal{C}_e(i)$ is notated with $A + i = \{a_j + i \bmod n : 1 \le j \le c\}$ $(0 \le i \le n-1)$. The chunk set of each node can be verified by others since it is generated based on committed blocks. Besides, this approach guarantees the chunk sets of different honest nodes must be different, which ensures at least $n - 2f$ different chunks are available to recover all original blocks.

Example 2: When $n = 3 \times 1 + 1$ and $c = 3$, we generate 3 different numbers $A = \{0, 1, 3\}$. Then the four chunk sets of all nodes are: $\mathcal{C}_e(0) = A + 0 = \{0, 1, 3\}, \mathcal{C}_e(1) = A + 1 = \{1, 2, 0\}, \mathcal{C}_e(2) = A + 2 = \{2, 3, 1\}$ and $\mathcal{C}_e(3) = A + 3 = \{3, 0, 2\}$, shown as follows.

$\mathcal{C}_e(0)$	$\mathcal{C}_e(1)$	$\mathcal{C}_e(2)$	$\mathcal{C}_e(3)$
0	1	2	3
1	2	3	0
3	0	1	2

Note that each node preserves 3 different chunks while each chunk is replicated over 3 different nodes.

Reading. First, when receives a request for B_t with height h, node N_c returns the target block directly if B_t is stored locally. Otherwise, node N_c try to fetch B_t from the target nodes $N_t(B_t)$ by sending a block message $\langle \text{BLOCK}, h, e, s \rangle$ to them. Surely, each node can cache blocks received from others to improve reading performance, e.g. *Least Recently Used cache* (LRU). Second, when node N_i receives block message from N_c for block with height h, it sends back target block B_t to N_c if B_t is stored locally. Node N_c returns B_t when receives it from target nodes. Third, if N_c receives no valid block from target nodes $N_t(B_t)$ in time, it broadcasts a decode message $\langle \text{DECODE}, h, e, s \rangle$ to randomly selected $n-f$ nodes to pull chunks for decoding. Forth, upon receiving decode message from N_c, node N_i sends its corresponding chunk set $\mathcal{C}_e(i)$ to N_c of coding round $\mathcal{R}_e^{(s)}$. Last, node N_c recovers all original blocks of coding round $\mathcal{R}_e^{(s)}$ by decoding when it receives $n - 2f$ different chunks.

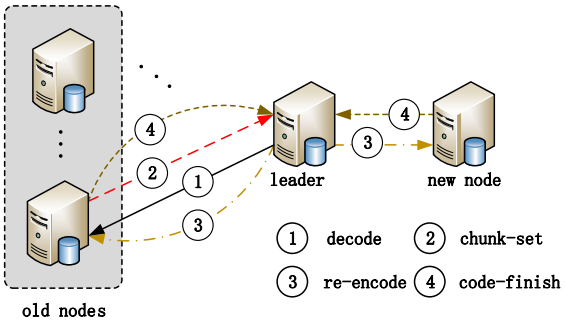


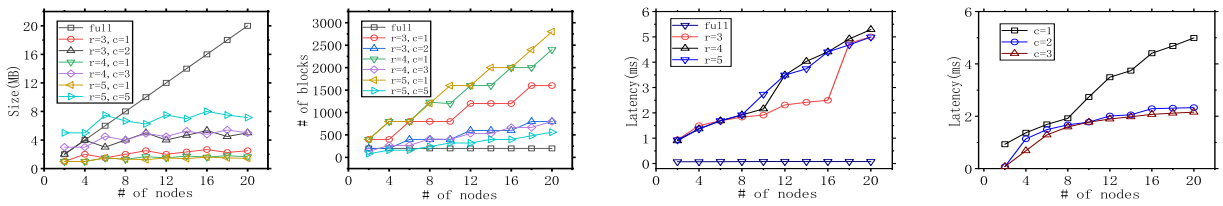
Fig. 2: Example of schema update.

B. Block Immigration and Re-encoding

It is critical to consider addition of new node in permissioned blockchain. Due to schema update for node addition, all historical blocks need to be re-encoded and re-distributed over all nodes to utilize the storage of new node. BFT-Store arranges a node as leader L to coordinate the process of re-encoding. The nodes move through a succession of configuration called *windows*, where a node is the leader and the others are *backups*. With addition of new nodes, the number of all and fault nodes become n' and f' respectively, and the version number for new schema is $s+1$. Figure 2 illustrates the process of re-encoding in a window. There are four phases for re-encoding blocks of a round $\mathcal{R}_e^{(s+1)}$ in a window. Each phase corresponds with a type of message transmitted between leader and backups, including decoding, chunk-set, re-encoding and code-finish. The four phases are detailed as follows.

1) Leader sends decode messages to backups. The leader L broadcasts a decode message to old nodes to request chunks of coding round $\mathcal{R}_e^{(s+1)}$.
2) Backups send their chunk-set messages to leader. Each backup N_i responds to leader L with its chunk set for old schema s when receives decode message.
3) Leader sends re-encode messages to backups. Leader L recovers all original blocks $\mathcal{B}$ of coding round $\mathcal{R}_e^{(s+1)}$ based on $n - 2f$ different chunks received from backups. Then, it broadcasts a re-encode message $\langle \text{RE\_ENCODE}, \mathcal{B}, \mathcal{V}, s+1, e \rangle$ to all backups, where $\mathcal{V}$ is a set of $n' - 2f'$ finish-code messages explained later.
4) Backups send coding-finish messages to leader. When backup N_i receives a re-encode message from the leader, it re-encodes blocks $\mathcal{B}$ into n' chunks. N_i stores its chunk set $\mathcal{C}_e(i)$ in coding round $\mathcal{R}_e^{(s+1)}$. Then, node N_i sends a code-finish message $\langle \text{CODE\_FINISH}, e, s+1 \rangle$ back to leader L indicating the accomplishment of re-encoding for coding round $\mathcal{R}_e^{(s+1)}$.

Note that when the leader L receives $n' - f'$ code-finish messages for previous coding round $\mathcal{R}_e^{(s+1)}$, it starts to re-encode blocks for the next coding round $\mathcal{R}_{e+1}^{(s+1)}$. To ensure all backups realize that honest nodes have accomplished re-encoding for coding round $\mathcal{R}_e^{(s+1)}$, leader L broadcasts received $n' - f'$ code-finish messages along with the re-encoding message of next coding round $\mathcal{R}_{e+1}^{(s+1)}$. A backup removes chunks with old schema when ensures that $n' - f'$ nodes have accomplished re-encoding by verifying signatures in $\mathcal{V}$. By this method, all blocks still keep available before re-encoding



(a) storage overhead per block (b) storage capability (c) latency for request, $c = 1$ (d) latency for request, $r = 5$

Fig. 3: Performance of BFT-Store against number of nodes.

for coding round $\mathcal{R}_{e+1}^{(s+1)}$. During re-encoding, the blockchain system is working properly and new blocks are generated continuously. The newly generated blocks are encoded with new RS schema among n' nodes and old blocks not re-encoded can be accessed through old schema. Therefore, even in the process of re-encoding, each node in BFT-Store can provide users with reading service. Besides, the leader may become crashed or be malicious in the process of re-encoding in a window. The leader rotation, which is similar to that of PBFT, is triggered by timeouts that prevent backups from waiting for re-encode message carrying original blocks to re-encode from the leader.

IV. Experimental Evaluation

Implementation. We implement BFT-Store in Tendermint, an open source platform adopted PBFT protocol for consensus, which is used to build permissioned blockchain by some systems, e.g. [12]. At the network layer, each node maintains a TCP connection with its peers, so that all nodes can communicate with each other via P2P protocol. All experiments are conducted upon a cluster of 4 machines, where each node is equipped with 16 CPU cores with 2.10GHz, 96GB RAM and 3TB disk space. The network bandwidth is 1Gbps. To test the scalability, we run multiple (up to 5) Tendermint instances (nodes) on each machine to simulate a middle-size network with up to 20 nodes. Note that this testing way is also adopted in lots of latest works [6].

Storage consumption. We employ full-replication manner as the baseline method, and study the situations where the block size is fixed to 1MB. Figure 3(a) reports the average storage consumption per block under different cases against the number of nodes. In comparison, in our novel manner, the storage consumption per block increases very slightly when more nodes join. For example, when $r = 4$ and $c = 3$, the storage consumption per block is 5MB, even there are 20 nodes. Figure 3(b) shows the overall storage capability of the blockchain system under different mechanisms. Assuming each node only owns 200MB storage space, the overall storage capability keeps merely 200 blocks for full-replication manner, while that of BFT-Store increases significantly as the network grows. For example, when $r = 5$ and $c = 1$, more than 2600 blocks can be stored in system.

Request latency. Figure 3(c) reports the latency for block reading as r varies from 3 to 5 when $c = 1$. For full-replication manner, the latency for block access keeps about 0.076ms due to no data transmission between two nodes. In comparison, the latency increases significantly in our manner as more nodes join system because the amount of data transmitted for decoding increases. When $n = 20$ and $r = 4$, the latency for block access even reaches 5.2ms. Figure 3(d) reports the latency of request when $r = 5$ while c rises from 1 to 3. As expected, with more replicas for each chunk, the latency of block access decreases because the probability $P_r(B)$ that block B is reachable grows. In Figure 3(d), when the number of nodes exceeds 10, there is a significant improvement of latency while c transfers from 1 to 2, because the network transmissions of chunks for decoding become the bottleneck.

V. Conclusions

Full-replication manner without any scalability leads to the huge storage consumption, which is a serious problem in permissioned blockchain. The combination of PBFT and erasure coding offers a chance to partition blocks over all nodes in blockchain system. Our contributions include: (i) exploit erasure coding for blockchain to reduce storage complexity per block from $O(n)$ to $O(1)$; (ii) propose a hybrid erasure coding and replicating to improve reading performance over encoded partitions; (iii) design a re-encoding approach to deal with addition of new nodes.

Acknowledgments

This work is partially supported by National Key R&D Program of China (2018YFB1003303), National Science Foundation of China (U1811264, 61972152), and ECNU Academic Innovation Promotion Program for Excellent Doctoral Students (YBNLTS2019-021).

References

[1] Tendermint. https://tendermint.com/.

[2] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[3] H. Chen, H. Zhang, et al. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *TOS*, 13(3):25, 2017.

[4] C. Dwork, N. Lynch, et al. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, 1988.

[5] S. Ghemawat, H. Gobioff, et al. The google file system. In *SOSP*, pages 29–43. ACM, 2003.

[6] L. Luu, V. Narayanan, et al. A secure sharding protocol for open blockchains. In *CCS*, pages 17–30. ACM, 2016.

[7] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[8] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *JACM*, 27(2):228–234, 1980.

[9] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial Applied Mathematics*, 8(2):300–304, 1960.

[10] D. Schwartz, N. Youngs, A. Britto, et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5, 2014.

[11] H. Weatherspoon et al. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[12] Y. Zhu, Z. Zhang, et al. Sebdb: Semantics empowered blockchain database. In *ICDE*, pages 1820–1831. IEEE, 2019.