scientific reports

Check for updates

OPEN

An optimized deterministic concurrency control approach for geo-distributed transaction processing on permissioned blockchains

Zhibo Han[1], Zeshun Peng[1], Gang Wang[1], Minghe Yu[2], Xiaohua Li[1], Yanfeng Zhang[1] & Ge Yu[1]✉

Concurrency control is crucial for ensuring consistency and isolation in distributed transaction processing. Traditional concurrency control algorithms, such as locking-based protocols, usually suffer from performance degradation due to heavy transaction coordination overheads. To overcome this problem, deterministic concurrency control approaches are widely adopted in many systems since they can avoid coordination overhead by eliminating uncertainty. In these systems, every node receives identical transaction batches, orders them according to specific rules, and executes them concurrently in a determined correct sequence. However, some transactions might have to be aborted in concurrent execution, wasting expensive network bandwidth and computing resources. We find that this problem significantly lowers system performance, especially in geographically distributed settings where network communication is a bottleneck. To exploit deterministic concurrency control efficiently in geo-distributed application scenarios, this paper studies an optimized deterministic concurrency control approach GB-DCC for permissioned blockchain applications which is a new type of distributed transaction processing systems. Three general optimization strategies are proposed: deterministic pre-execution, mini-batch partitioning, and deterministic re-execution. Experiments show that under the YCSB-A benchmark workload, these strategies can reduce the distributed system's bandwidth consumption by 17.8% and improve the performance obviously.

Distributed systems[1] are widely used in modern computing for their high performance and are crucial in many important fields such as big data processing[2], cloud computing[3], scientific computing[4], and real-time processing[5]. These systems achieve scalability and availability by distributing tasks across multiple nodes and use concurrency control protocols[6] to ensure data consistency when multiple nodes concurrently access and modify shared data.

Various concurrency control models are used to ensure the ACID properties of distributed transactions. A concurrency control model like Two-Phase Locking (2PL)[6] restricts concurrent read-write access by locking data during execution. In contrast, an optimistic concurrency control model[7] resolves conflicts after execution and does not lock data until execution is finished. However, a non-deterministic model[8] requires frequent coordination among nodes during transaction execution, such as dynamic locking and state synchronization, to ensure system consistency. In distributed environments, this coordination requires high-frequency network communication, transaction waiting, and recovery operations, resulting in significant communication overhead and transaction latency, thus constraining the system's overall performance.

The deterministic concurrency control (DCC) model[9–11] avoids coordination overhead by eliminating uncertainty on the execution orders of transactions. The nodes have the same transaction batch as input and will have the same execution result. In the case of full replication where transactions are replicated to all nodes for processing, nodes use a consensus protocol to order and replicate the input transactions before executing and committing them independently. DCC can maintain consistency without requiring nodes to coordinate transaction orders during execution, thereby reducing contention and increasing transaction processing concurrency.

[1]School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China. [2]Software College, Northeastern University, Shenyang 110169, China. ✉email: yuge@mail.neu.edu.cn

Similarly to other concurrency control algorithms, DCC also has to abort transactions caused by read-write conflicts during execution to ensure serialization isolation. First, transactions are replicated to all nodes before execution, which typically involves transmitting the transaction-associated metadata across the network to all nodes. Since conflicting transactions are aborted during the execution phase, replicating these aborted transactions also wastes network bandwidth. Second, the aborted transactions are executed on all nodes, consuming unnecessary computing resources. For example, a large e-commerce platform supports transactions between buyers and sellers. The imbalance in buyers' shopping preferences can lead to severely skewed workloads and hotspot issues. This causes a drastic decrease in system throughput and an increase in latency, affecting the user experience[12]. In practice, especially under skewed workloads, the abort rate of DCC is high, significantly reducing system performance. In Aria[10], a batch-based DCC algorithm, the data show that when the proportion of reads and writes is 50% each, the transaction abort rate is 17.8%, resulting in a 17.8% waste of network bandwidth. As shown in Fig. 1, as the write skew ratio of the workload increases (YCSA-C to YCSA-B to YCSB-A), the abort rate increases. The system aborts more transactions and wastes more network bandwidth. Therefore, optimizing DCC to reduce the abort rate is important for the system to improve system throughput and fully utilize system resources.

Recently, geo-distributed systems are gaining significant attention in fields such as finance[13], commerce[14], and cloud storage[15]. When nodes are geographically distributed, they communicate over wide-area networks (WAN) to replicate data, exchange consensus messages, and synchronize states. Unlike local-area networks (LAN), WAN typically has higher latency and lower bandwidth. The transmission of invalid transactions can easily lead to bandwidth waste, which becomes a performance bottleneck for geo-distributed systems. In these applications, workloads are often highly skewed[16], resulting in a high abort rate and wasting precious WAN bandwidth. To better utilize network and computing resources, we propose three optimization strategies for DCC to reduce the number of aborted transactions:

Deterministic pre-execution. To reduce WAN bandwidth consumption caused by replicating aborted transactions, a node pre-executes its collected transactions before replicating them to other nodes. Specifically, in a given period, transactions received by a node are grouped into the same batch and executed concurrently. Any read-write conflicts within the batch (i.e., transactions are reading and writing the same data) can result in abortion. The aborted transaction go through all process, resulting in the waste of network and computing resources. By detecting and resolving these conflicts immediately after receiving transactions, the number of conflicting transactions can be reduced. Although this approach may misjudge some transactions with false positive errors, it can find all potential aborted transactions to avoid their replication and execution on all nodes, thus lowering the overall network and computing overhead.

Mini-batch partitioning. When the transaction sending rate is high, the number of transactions within a batch increases, leading to a higher possibility of conflicts. After the consensus is completed, the batch is replicated to every node. Using a deterministic partitioning strategy, a node can partition the batch into multiple mini-batches. By executing transactions within a mini-batch concurrently and executing the mini-batches sequentially, the number of transactions within each batch is reduced, thus lowering the abortion possibility, avoiding global transactions conflict and waste of resources in geo-distributed systems.

Deterministic re-execution. To further reduce the aborted transactions, a node re-executes the aborted transactions within an epoch after executing all mini-batches. In the traditional DCC approach, if a transaction is aborted, the failure result is returned to the client. The client will re-initiate the transaction, which means that the transaction will be executed and repeat the entire transaction process including making replication and consensus again. This reprocessing consumes both network and computational resources. To reduce this overhead, we append the aborted transaction into the next epoch to continue execution. The system will force a transaction to abort only when it is continuously aborted for several epochs.

The first strategy is employed before transaction replication, it avoids transmitting transactions that conflict locally, eliminating the need to replicate these conflicts to other nodes. Additionally, the strategy is designed to minimize computational resource overhead. Since executing transactions requires computational resources, any transaction aborted after execution results in wasted computation. Therefore, it is essential to maximize the number of successfully committed transactions during execution to minimize unnecessary computational

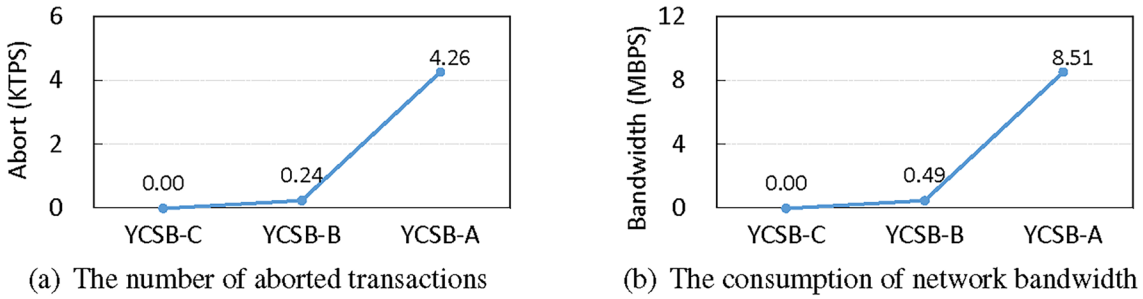


Fig. 1. Effect of aborted transactions under the YCSB workload. This indicates that as the ratio of the workload to write data increases, the number of failed transactions in the system increases, and more network resources are wasted, highlighting the importance of reducing failed transactions. Where YCSB-C, B, and A are the three workloads with increasing write data ratio.

overhead. The second and the third strategies ensure that transactions are successfully committed on the first attempt, to reduce unnecessary resource consumption.

The three strategies are different from the traditional distributed database idea, which adopts a similar idea to increase the throughput, and the transactions aborted overhead is high in the case of high conflicts, while our strategies target the limited resources in the geo-distributed environment and optimize the system by reducing the resource overhead caused by aborted transactions. We find that in the permissioned blockchain, which is usually geographically distributed, nodes communicate via the WAN, which is limited by higher network latency, and aborting the transaction increases the overhead of network resources, which can be effectively reduced by staged strategies, so that the network resources can be efficiently utilized.

The contributions of this paper are as follows:

- We analyze and evaluate the wasted network and computing resources consumed by aborted transactions when the distributed system executes transactions concurrently and propose the solutions.
- We propose three general optimization strategies: deterministic pre-execution, mini-batch partitioning, and deterministic re-execution, and design an optimized deterministic concurrency control algorithm GB-DCC for geographic-distributed permissioned blockchain.
- We implement GB-DCC in the permissioned blockchain NeuChain[17]. We evaluate GB-DCC under benchmark workload and compare it with related systems, demonstrating that it can reduce network bandwidth consumption by up to 17.8% and increase throughput by 5.1% compared to the original system.

Background and related work

This section introduces DCC and several typical systems utilizing DCC.

Deterministic concurrency control

Deterministic concurrency control (DCC)[9–11] refers to algorithms designed to manage and schedule concurrent transactions deterministically. In different systems, DCC adopts different algorithms to ensure the deterministic execution order of transactions, all of which aim to reduce the overhead of coordination.

Calvin[9] is a distributed transaction processing system. Its DCC utilizes a pre-ordering mechanism to order transactions, ensuring a deterministic execution order across all nodes. Specifically, in Calvin, each node replicates a batch of transaction requests to other nodes. Transactions are deterministically ordered, and locks are acquired in this predefined order. Once the ordering is completed, nodes execute transactions concurrently according to the predetermined sequence. The pre-ordered transaction execution prevents deadlocks and reduces conflicts by ensuring all nodes acquire locks in the same order. In the case of a conflict, Calvin detects it early and allows the conflicting transaction to be aborted, ensuring data consistency. However, re-executing conflicting transactions can improve the throughput.

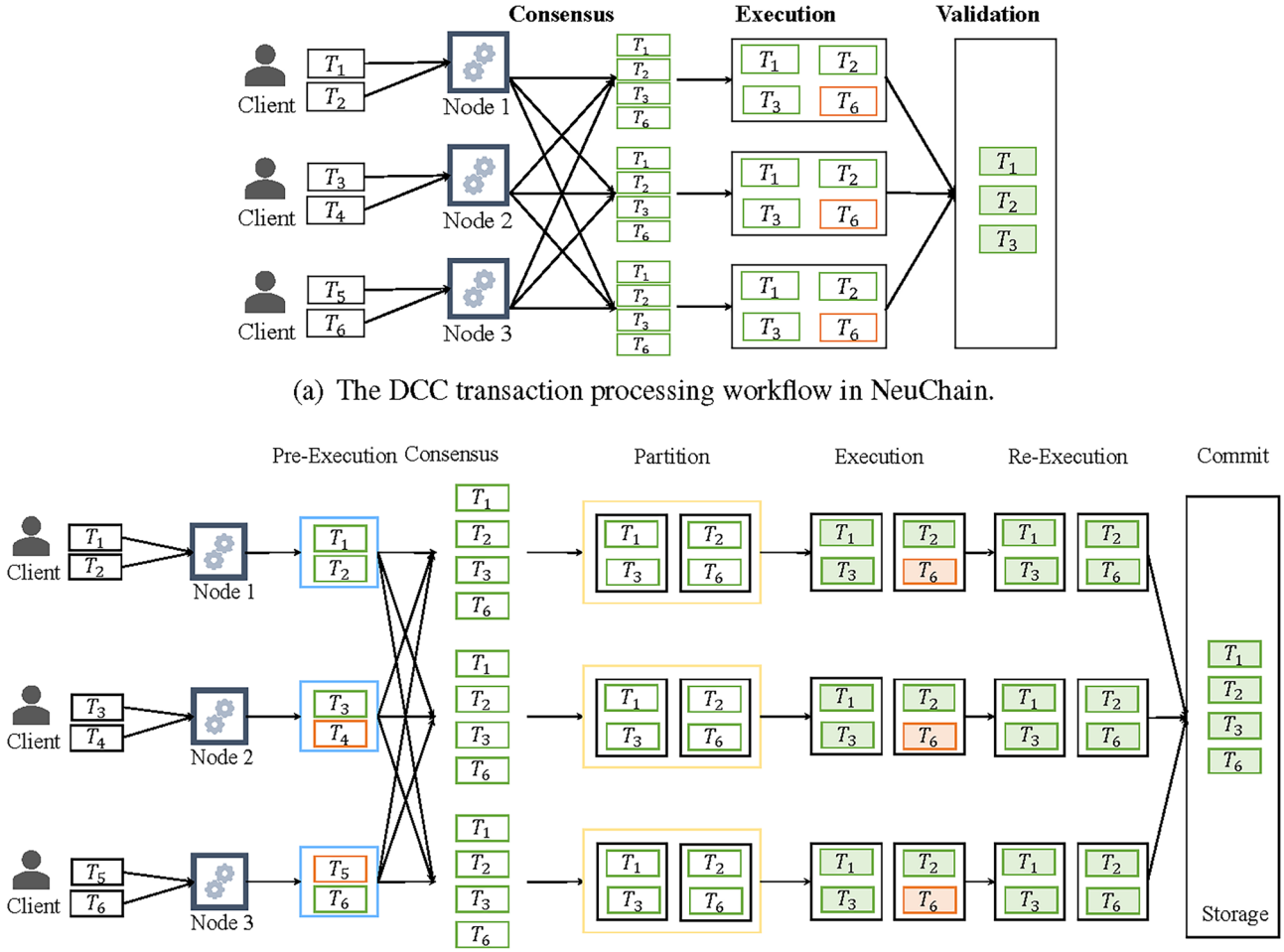
Unlike Calvin, Aria[10] employs a no-ordering mechanism. The transaction order is determined based on specific rules, thus avoiding frequent coordination between nodes. Aria's DCC includes two phases: the execution phase and the commit phase. In the execution phase, once the nodes receive all transactions in a batch, they execute them according to a determined order without directly committing to the database. The write sets of the transactions are recorded in the global write set. Only the transaction with the smaller ID is committed if two or more transactions access the same row and at least one is a write operation. If a conflict is detected and a transaction with a smaller ID exists, the one with the larger ID is aborted. Once a batch of transactions is executed, it enters the commit phase. In the commit phase, nodes analyze the dependencies of transactions based on the global write set from the execution phase; if conflicts exist, transactions are aborted; otherwise, they are committed to the database. This avoids the time required for locking and increases execution parallelism. Additionally, Aria has a reordering algorithm to modify the order of conflicting transactions, allowing more transactions to be committed within a batch. Conflicting transactions are re-executed in the next batch. However, under high data skew conditions, this reordering algorithm can cause the accumulation of aborted transactions, disrupting the system's normal operation.

Gria[11] further improves upon the mechanisms of Aria by automatically adjusting the batch size based on system load and transaction characteristics, thus reducing conflicts in low-concurrency environments. Since multiple transactions attempting to modify the same data version can lead to transaction failures and re-execution, Gria adopts a multi-version control mechanism to avoid write-after-write conflicts, allowing transactions to execute independently on different data versions. Additionally, Gria employs reordering and rechecking mechanisms to reduce conflicting transactions further. However, Gria's multi-version control and reordering mechanisms increase data replication and communication, consuming more bandwidth.

DCC-based permissioned blockchain

Blockchain[17–22], as a distributed system, requires strict concurrency control to ensure consistency among nodes. NeuChain[17], PROPHET[18], and Spectrum[19] are all blockchains that enhance throughput through DCC. They determine the execution order of transactions by consensus, avoiding the order coordination in execution phase. Take NeuChain as an example to introduce the DCC workflow, as shown in Fig. 2a, the $Node_{1-3}$ receives transactions T_1-T_6 initiated by clients, and nodes replicate transactions to other nodes. In the execution phase, nodes execute transactions according to deterministic rules. In the validation phase, nodes abort the conflicting transaction T_6 and commit all transactions.

However, blockchains are often deployed across geographic regions, resulting in slower network transmission and processing speeds. High-throughput transaction processing significantly increases the abort rate when the hot data is frequently accessed. Specifically, in NeuChain, when nodes execute transactions in a deterministic order because transactions frequently access and modify the same data, the node aborts this part of the



(a) The DCC transaction processing workflow in NeuChain.

(b) Improved transaction processing workflow in NeuChain.

Fig. 2. Comparison of the initial workflow and the improved workflow. (**a**) This is the workflow of DCC in NeuChain. The $Node_{1-3}$ receives transactions T_1-T_6 initiated by clients, and nodes replicate transactions to other nodes. In the execution phase, nodes execute transactions according to deterministic rules. In the validation phase, nodes abort the conflicting transaction T_6 and commit all transactions. In this process, the grey transactions are the initial transactions, the green transactions are successfully pre-executed or executed, and the red transactions are failed transactions. (**b**) This is the complete workflow of GB-DCC. The process begins with the client (i.e., the user) initiating a transaction T_i to the nodes. The nodes apply a pre-execution strategy and then determine the transaction order through network communication. A batch of transactions is partitioned into mini-batches before entering the execution phase. After execution, any failed transactions are re-executed, and the final results are committed to storage.

conflicting transactions, which will seriously affect the performance of the blockchain in the case of limited network resources. If the number of conflicting transactions is reduced before transaction replication or during transaction execution, it will improve the system's resource utilization efficiency.

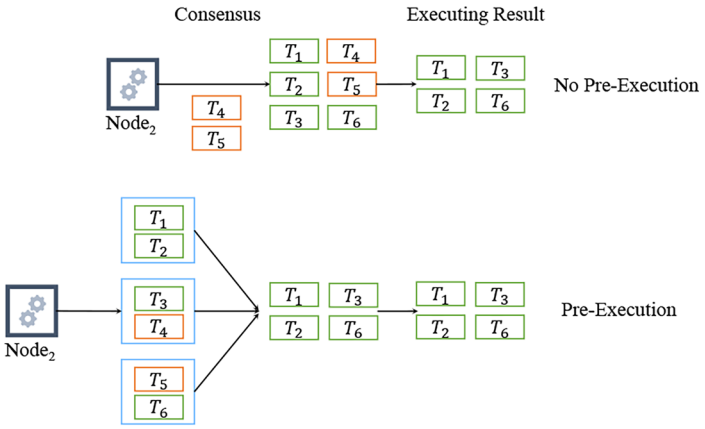
GB-DCC design

In this section, we first introduce the GB-DCC, a deterministic concurrency control applied to the geographical blockchain, and the overall workflow in the system, then introduce the processing of the strategies respectively, and finally discuss the generalization of strategies to blockchain systems.

Overview

We assume the exact system model as NeuChain, using DCC to process transactions. We optimize the Aria deterministic concurrency control GB-DCC on NeuChain.

The processing of GB-DCC includes three stages during concurrent execution to reduce network and computational resource consumption. As shown in Fig. 2b, the $Node_{1-3}$ receives transactions T_1-T_6 initiated by clients firstly. Before replicating these transactions to the other two nodes, the node applies a deterministic pre-execution strategy where read-write conflict dependencies use Aria's algorithm to reduce aborted transactions. Because the algorithm fits with the deterministic pre-execution algorithm of Fig. 3, which detects



Algorithm 1 Deterministic Pre-Execution.

Input: A set of transactions TS received locally.

Output: A Set of transactions broadcast to other nodes

```
Initialize reserve table TB
for T ∈ TS do
    execute(T);
    detectConflict(T);
function execute(T):
    {T.RS, R.WS} // Simulate the execution of T to get read-write set
    for rec in T.WS do
        TB[rec.key] = min(TB[rec.key], T.tid)
function detectConflict(T):
    for rec in T.WS do
        if TB[rec.key] < T.tid then
            Delete T from TS // T write data is updated
    for rec in T.RS do
        if TB[rec.key] < T.tid then
            Delete T from TS // T read data is updated
```

Fig. 3. The workflow and algorithm of deterministic pre-execution in $\boldsymbol{node_2}$. This process specifically demonstrates the effect of pre-execution in $node_2$. Compared to scenarios without pre-execution, transactions T_4 and T_5 are identified as failed transactions during pre-execution before consensus. Therefore, they do not need to be transmitted over the network to other nodes. The figure uses node two as an example. The algorithm formalizes the process.

transactions conflict by determining order and the read and write set of transactions. Only transactions that pass pre-execution are replicated to other nodes, where T_4 and T_5 are not replicated due to conflicts. After receiving the replicated transactions from other nodes, the node partitions this batch of transactions(T_1, T_2, T_3, T_6) into several mini-batches before execution, where T_1 and T_3 are in one batch and T_2 and T_6 are in the other batch, and then sequentially executes each mini-batch. After each epoch of transaction execution, the node re-executes any aborted transactions, Where T_6 is aborted in the current epoch and is successfully committed after re-execution. The internal design of the three strategies is described in the following paragraphs.

Deterministic pre-execution

In NeuChain, each node receives a set of transactions initiated by the local client and replicates them to other nodes. Once the nodes have collected all transactions from other nodes, they execute the transactions concurrently. However, this process encounters a fundamental inefficiency: conflicting transactions are inevitably aborted during execution to maintain consistency. This leads to a significant issue that replicating these transactions consumes network resources unnecessarily, as they will neither be committed nor stored after execution. Such inefficiency directly reduces system throughput and increases the latency of subsequent transactions by overburdening the network with redundant data.

To address this inefficiency, we propose a deterministic pre-execution strategy that directly targets the root cause of the problem of replicating transactions that are destined to be aborted. Specifically, our approach allows each node to simulate the execution of its local transactions before replicating them to other nodes. By pre-executing the transactions deterministically, the node identifies conflicting transactions within its local set and removes them from the replication process. This ensures that only transactions with a high probability of successful execution are shared across the network. The key mechanism lies in the simulation phase: it detects and isolates conflicts without modifying the actual data state, thereby minimizing the unnecessary overhead of replicating abort-prone transactions. The data state does not replicate such transactions just to reduce the useless overhead of network resources.

The Pre-Execution process is shown in Fig. 2. Nodes 1, 2 and 3 receive transactions T_1-T_6 from clients, and before reaching consensus, they deterministically pre-execute these transactions. In Fig. 3, using $Node_2$ as an example, we specifically illustrate the effect and algorithm of pre-execution. Without the pre-execution process, T_4 and T_5 are aborted during execution, but nodes have already replicated the two transactions through the network. This shows that the network consumption for these transactions is unnecessary. In contrast, with pre-execution, nodes do not replicate conflicting transactions to other nodes for consensus while the execution results are the same. Therefore, pre-execution reduces the waste of network resources. Moreover, the time complexity of traversing the transaction set is $O(N)$, while checking read-write conflicts and maintaining the read-write table is $O(1)$. Therefore, the total time complexity is $O(N)$, indicating that deterministic pre-execution is linear and does not significantly increase the computational overhead.

Mini-batch transaction partitioning

In the deterministic pre-execution phase, nodes evaluate only their local transactions and do not account for potential conflicts between transactions replicated from other nodes. This creates a critical challenge: when

all transactions are collected and executed concurrently in batches, conflicts between replicated transactions (e.g., read-write dependencies) can occur, especially in distributed systems like Aria. These conflicts, undetected during pre-execution, result in aborted transactions during execution, wasting computational resources and lowering system throughput. While serial execution could entirely avoid such conflicts, it comes at the cost of significantly increased execution time, making it impractical for high-throughput systems.

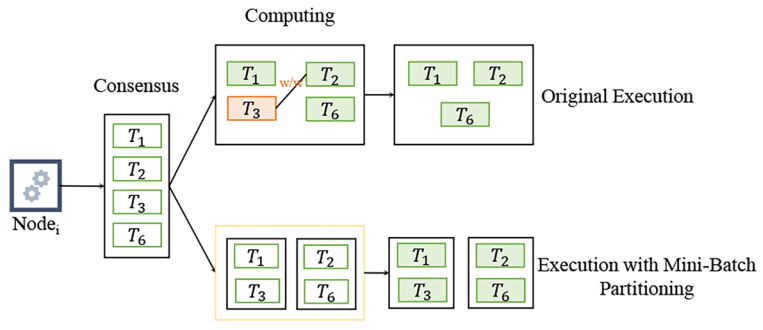
To address this challenge, we propose a mini-batch transaction partitioning strategy to balance conflict reduction and execution efficiency. This approach partitions a batch of transactions into multiple mini-batches based on transaction attributes (e.g., shared ID suffixes). Transactions within each mini-batch are executed concurrently to leverage parallelism, while the mini-batches are executed sequentially. Compared with traditional batch processing, the strategy partitions the original batch into mini-batches, both of which concurrently execute a batch of transactions during the execution phase. The difference is that traditional batch processing may have a large number of conflicts in execution, leading to a higher transaction abort rate. If you set up the system with smaller batches, although the function is similar to mini-batch, but each batch needs to go through the consensus process, which will cause additional overhead. Our strategy makes up for the shortcomings of traditional batch processing by partitioning a large batch of transactions into mini-batches during the execution phase, assigning conflicting transactions to different batches for serial execution, avoiding execution conflicts and reducing the waste of resources.

In Fig. 2, we add this partitioning strategy before the concurrent execution, in which a batch of transactions is partitioned into multiple mini-batches according to the rules and then successively enters the concurrent execution phase to execute. Figure 4 illustrates the overall process and algorithm of mini-batch transaction partitioning. $Node_i$ needs to execute transactions T_1-T_3, T_6. If directly enters the execution phase to execute these transactions concurrently, transaction T_3 is aborted due to a write-write conflict with T_2. However, with mini-batch transaction partitioning, the conflict between the two transactions can be avoided. Specifically, according to the partitioning rules, T_1-T_3 and T_6 are partitioned into two mini-batches: T_1 and T_3 in one batch, and T_2 and T_6 in the other batch. These two batches then enter the concurrent execution phase successively. Since transactions within each mini-batch do not conflict during concurrent execution, both transactions can be executed successfully, resulting in the successful commit of all transactions, including T_3. The algorithm traverses the transaction set to divide batches, the time complexity is $O(N)$, Additionally, the transaction execution introduces an extra complexity of $O(K)$ due to the batch size, resulting in a total time complexity of $O(KN)$. Compared to standard DCC, which directly executes batches, this strategy incurs some computational overhead but significantly reduces aborted transaction.

Deterministic re-execution

Although the deterministic pre-execution and mini-batch transaction partitioning strategies can significantly reduce the number of read-write conflict transactions in the system, conflicts inevitably occur as long as nodes execute transactions concurrently during the execution phase. After the execution phase, the system returns the results of aborted transactions to the clients, who then resubmit these transactions. This necessitates reprocessing these transactions, which can limit the performance of geo-geographical systems. However, this re-submission of conflicting transactions by the clients can be avoided. We designed a deterministic re-execution strategy. Specifically, conflicting transactions from the current batch are placed into the next batch for continued execution, allowing them to be successfully executed and committed in the subsequent batch. This process is the same as the original execution and does not affect the ACID properties of the transactions.

In the transaction processing of no re-execution, transaction T_6 is aborted during the execution phase, and the abort result is returned to the client. The client re-initiates the transaction and restarts the transaction execution. However, with re-execution, aborted transactions will be re-executed in the next epoch. It will be successfully executed and committed since it does not conflict with the transactions in the next epoch. This process and algorithm are illustrated in Fig. 5. It is important to note that we limit the times of the re-executed



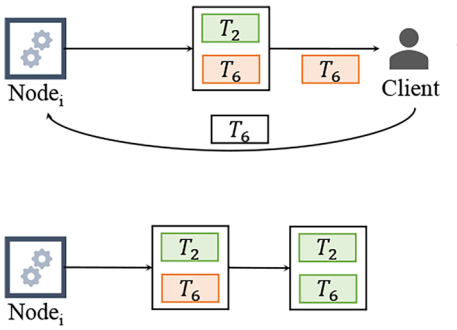
Algorithm 2 Mini-Batch Transaction Partitioning.

Input: A set of transactions TS of a batch.

Output: Transactions of multiple batches

```
1: Initialize mini-batch queue BQ and a mini-batch B
2: for T ∈ TS do
3:     if (B is not full) then
4:         B.Push(T);
5:     if (B is full) then
6:         BQ.Push(B);
7:         B.Clear();
8: return BQ
```

Fig. 4. The workflow and algorithm of mini-batch transaction partitioning in $node_i$. This also illustrates the specific role of mini-batch transaction partitioning through comparison. $Node_i$ needs to execute transactions T_1-T_3, T_6. If transactions are executed directly, one of the transactions, T_3 fail to execute. However, by applying mini-batch partitioning, all four transactions can be successfully executed and committed, improving system performance. $Node_i$ represents any node. The algorithm formalizes the process.



Algorithm 3 Deterministic Re-Execution.

Input: A aborted of transactions TS of the epoch e_i.

Output: A set of transactions of the epoch e_{i+1}.

```
for T ∈ TS do
    if (T.retryTimes > 5) then
        continue
    push T into e_{i+1} and T.retryTimes++
return e_{i+1}
```

Fig. 5. The workflow and algorithm of deterministic re-execution in $node_i$. This process demonstrates the role of re-execution through comparison. Transaction T_6 fails without re-execution, and the result is returned to the client (i.e., the user), who then resubmit the transaction. With re-execution, T6 is successfully executed in the next round. The algorithm formalizes the process..

transaction to 5. Transactions that are re-executed more than 5 times will be aborted with result returned. This is the same as conflict transactions being aborted. Although Aria has already implemented this functionality, we verify that this transaction re-execution strategy can be applied to other distributed systems, demonstrating that it can be more widely applied in concurrency control. Therefore, the time complexity of the algorithm is $O(5N)$, that is, $O(N)$, where N represents the number of transactions that are re-executed and 5 represents the maximum number of re-executions.

Discussion

We design the three strategies that integrate seamlessly with existing workflows. These strategies operate independently, using transaction data as input without altering transaction structures, and produce outputs compatible with the original processing workflow. Next, we explore the applicability and limitations of these three strategies across various blockchain architectures. And finally we discuss the security of GB-DCC against adversarial behaviors.

For DCC permissioned blockchains. The modular strategy has been integrated into blockchain systems that utilize Aria's deterministic concurrency control (DCC). For other blockchains[17–19] adopting DCC, the core mechanism is global transaction ordering, which ensures that transactions are executed in a fixed sequence, providing a consistent processing foundation across all nodes. This predetermined transaction order facilitates pre-execution and reduces the likelihood of transaction aborts. Furthermore, the strict consistency requirements of DCC enable aborted transactions to be deferred to the next cycle, thereby improving throughput while maintaining system consistency. Notably, deterministic ordering inherently clarifies transaction dependencies, allowing large transaction batches to be partitioned into smaller mini-batches for concurrent processing, which enhances execution efficiency. However, when transaction dependencies are complex, pre-execution becomes more difficult. Additionally, under high workloads, the mechanisms for mini-batch execution and re-execution of aborted transactions may further strain system resources.

For other permissioned blockchains. Permissioned blockchains[20,21] employ alternative concurrency control, they do not necessarily execute transaction based on read and write sets. For non-deterministic concurrency control, such as two-phase locking, a strict locking mechanism is used to ensure the serialization of transactions, with a low degree of concurrency and a significant decrease in the efficiency of transaction execution, but the abort rate is 0, the three strategies can not be applied to it; and optimistic concurrency control, a typical blockchain is Fabric[20], which allows transactions to be executed concurrently, but conflict verification is required before commit, which can lead to large aborted transactions in high-conflict scenarios, affecting throughput. The Pre-execution reduces conflicted transactions in advance, while mini-batch and re-execution is batch processing and needs to be considered based on Fabric characteristics Compared with non-deterministic concurrency control, GB-DCC reduces aborted transactions through three strategies, which improves system throughput and ensures concurrency. In addition, for multi-version concurrency control(MVCC), The first two strategies of GB-DCC are not applicable, pre-execution and mini-batch require adjustments to version management, leading to increased complexity. In contrast, re-execution strategies can handle aborted transactions with relatively low overhead under light workloads.

For public blockchains. For public blockchains[22], their large network scale, decentralized architecture, and prioritization of correctness and consistency over performance typically result in the serial execution models. Consequently, pre-execution and mini-batch execution are not suitable for public blockchain systems. Furthermore, re-execution may lead to transaction backlogs under high-load conditions due to slower execution speeds. Frequently deferred transactions can result in wasted block space resources and even increased transaction fees, ultimately impacting the user experience. Therefore, these three strategies are unsuitable for public blockchain systems employing serial execution. Moreover, in a real-world blockchain environment, transactions are complex and volatile, and the network fluctuates and is uncertain. GB-DCC needs to statically analyze the read and write set of transactions, and is temporarily limited by a fixed node distribution, unable to consider dynamic node joining and leaving, which limits the promotion of GB-DCC to the real blockchain

For other systems. GB-DCC's strategies are originally designed for geo-distributed blockchain systems, showing strong potential for broader applicability in software-defined environments. In SD-IoT and Software-Defined Multimedia IoT, where low latency and high throughput are crucial, GB-DCC can reduce transaction conflicts and ensure concurrency execution, thereby improving QoS, reducing jitter, and enhancing real-time performance under dynamic workloads. Furthermore, in Software-Defined Networking-based systems, GB-DCC's partitioning aligns well with flow-aware forwarding and load balancing. By integrating GB-DCC with SDN controllers, we envision hybrid models where transaction priority and conflict profiles inform dynamic flow management, leading to improved resource utilization, lower latency, and better scalability.

Security. In permissioned blockchains, authenticated nodes reach consensus on the transactions order, preventing manipulation. In short, the strategies are designed for transaction execution and cannot be attacked under a deterministic order.

Performance evaluation

In this section, we implement GB-DCC based on the open source blockchain system NeuChain, evaluate the system throughput, latency, and network bandwidth consumption, and compare it to state-of-the-art systems, including NeuChain[17], a high-performance permissioned blockchain, Hyperledger Fabric[20], a permissioned blockchain using MVCC with optimistic concurrency control, and ResilientDB[21], a geo-distributed and high-performance blockchain. The consensus protocol used in NeuChain and Fabric is Raft. In contrast, ResilientDB uses PBFT as the consensus protocol.

Experimental setup. We deploy three virtual machines in the same region, each equipped with an 8 vCPUs, 16GB of RAM, and running Ubuntu 22.04. We set the batch size to 100 and the epoch size to 50 ms. To simulate a geo-distributed environment, we limit the network bandwidth between each virtual machine to 100 Mbps. Each batch is partitioned into 16 mini-batches by default.

Workload. We use the universal distributed system workload YCSB[23] and SmallBank. They are widely adopted in distributed systems, such as Calvin and Aria, as well as open source blockchain systems like Hyperledger Fabric, and ResilientDB. YCSB simulates typical blockchain application scenarios and allows us to configure test parameters, such as ratios of read/write operations and transaction sizes. The table size is set to 1,000,000 records, each with 10 attributes, and each attribute is 100 bytes in size. Read operations retrieve the entire record, while write operations update only one record attribute at a time. To simulate a high contention scenario, the data skewness is set to 0.99, meaning a few hotkeys have a high access frequency, while most keys have a lower access frequency. We use SmallBank workload to evaluate the effect of different numbers of mini-batches. SmallBank is a benchmark for Online Transaction Processing (OLTP) workloads, simulating various fundamental transactions in banking applications. It is configured with 100,000 accounts, and the transactions follow a uniform distribution.

Overall performance

Since network bandwidth cannot be directly evaluated through experiments, we calculate bandwidth consumption using the throughput and the average size of each transaction. In NeuChain, Aria is a multi-leader replication architecture. Each node places the complete data copy. Unlike leader-follower, all nodes act as leader nodes to exchange transactions, resulting in equal bandwidth consumption between nodes. For ResilientDB, which uses PBFT, and Fabric, which employs Raft, we calculate the bandwidth consumed by the consensus leader. Assuming the size of each transaction is S_t and the total number of systems nodes is M where the network bandwidth consumed by each node for replicating transactions to other nodes is $S_t * (M-1)$. The total bandwidth represents the bandwidth generated by all transactions (i.e. $N_a + N_c$) the node receives. The bandwidth calculation for a (leader) node to commit or abort transactions is shown in Equation 1, where N_c represents the number of committed transactions and N_a represents the number of aborted transactions.

$$Bandwidth\ \ consumption = \frac{N_i * S_t * (M-1)}{Total\ \ Bandwidth} = \frac{N_i}{(N_a + N_c)} \quad (N_i = N_c, Na) \tag{1}$$

Figure 6a uses a bar chart to show the overall performance of the different systems. GB-DCC achieves higher throughput and zero abort rates compared to other systems by pre-executing to eliminate local conflicted transactions, mini-batches partitioning to reduce concurrent conflicted transactions and re-executing conflicted transactions, which is in line with the design purpose of this research. Figure 6b shows the bandwidth utilization of these systems with the bar chart. GB-DCC has zero aborted transactions, whereas NeuChain has 17.8% of its bandwidth consumed by aborted transactions. It is worth noting that the total bandwidth is less than that of NeuChain, because the pre-execution eliminates conflicting transactions, reducing the bandwidth wasted by consensus. In addition, Fig. 6c uses a line chart to show the performance under different skewed workloads, YCSB-B (5% write, 95% read), YCSB-C (100% read). Under uniform workloads, GB-DCC and NeuChain exhibit similar performance, indicating that our optimization strategies introduce minimal additional overhead. Because the strategies are effective in reducing the number of aborted transactions, GB-DCC is better suited for highly write-skewed workloads, as uniform workloads have a lower abort rate, making abort rate reduction less important.

In large-scale blockchains like NeuChain, these three strategies remain effective with linear time complexity. Deterministic pre-execution detects conflicts locally. Regardless of the number of nodes, it effectively reduces unnecessary network communication and preventing excessive conflicts under extreme workloads. Mini-batch partitioning adapts dynamically to workload changes, lowering conflicts and improving throughput. Deterministic re-execution minimizes transaction restarts, optimizing resource utilization in high-conflict scenarios.

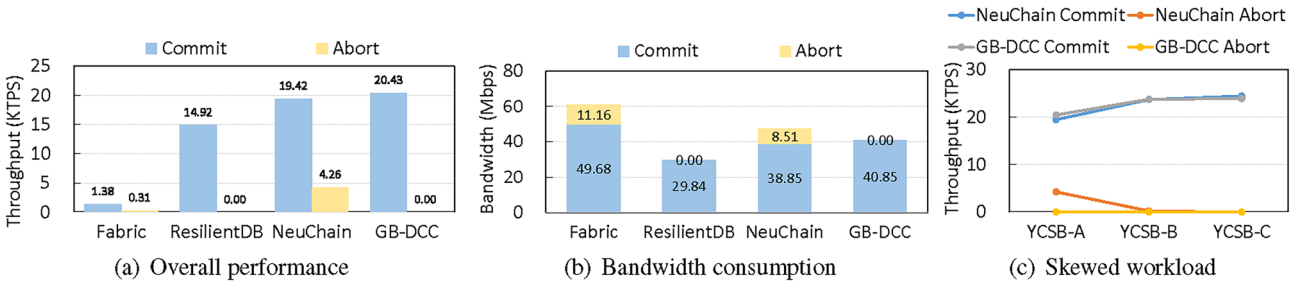


Fig. 6. Performance comparison under the YCSB workload. (**a**) This compares the number of committed and failed transactions in different systems. Green represents successfully executed transactions, while red represents aborted (failed) transactions, measured in kilo transactions per second(KTPS). (**b**) This is a comparison of network bandwidth usage in the systems. Green and red represent the network resources consumed by successful and failed transactions, respectively, measured in megabits per second (Mbps). (**c**) This is a comparison of system performance at different skewed workload, showing that GB-DCC works significantly for high skewed workload.

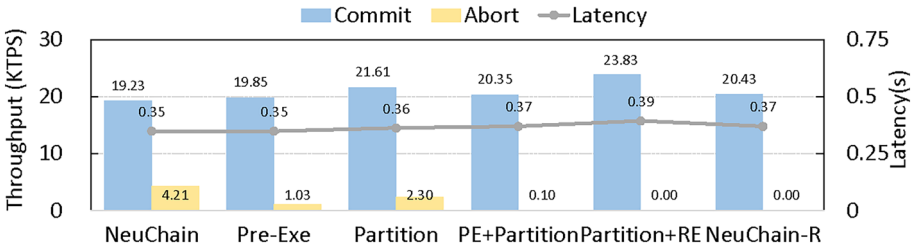


Fig. 7. Effect of different strategies, where Pre-Exe and PE are deterministic pre-execution, Partition is mini-batch transaction partitioning, and RE is deterministic re-execution. This shows the combined effort of various strategies proposed in this paper. The x-axis denotes different combination methods, the left y-axis represents KTPS, and the right y-axis represents latency. The red triangles show the latency values, while the blue line illustrates the trend.

Performance breakdown

To better validate the effect of three strategies, we experiment with each strategy and combination, as shown in Fig. 7. The deterministic pre-execution reduces the number of aborted transactions, because PE identifies conflicting transactions in advance, primarily reducing ineffective network transmission. Mini-batch transaction improves throughput by partitioning conflicting transactions into mini-batches and executing them sequentially. But conflicting transactions may still exist, incorporating pre-execution partitioning further reduces the aborted transactions.

And by re-executing the aborted transactions, the system's abort rate can be reduced to zero. Additionally, the partition and re-execution achieves high throughput. Compared to mini-batch transaction partitioning alone, this effectively converts the 2.30 KTPS aborted transactions into throughput because without the pre-execution, conflicting transactions are eventually re-executed and committed. However, this combination consumes more computational time, resulting in the highest system latency.

It is noted that we do not conduct a standalone deterministic re-execution experiment because in high-conflict scenarios such as YCSB-A, aborted transactions continuously accumulate in the final phase, preventing the system from functioning normally.

Effect of different numbers of mini-batches

Finally, we explore the effect of different numbers of mini-batches under different workloads. Figure 8a shows the system performance for different numbers of batches under the SmallBank workload. Due to the read-intensive nature of the SmallBank workload and the low number of conflicting transactions, the abort rate of the system is low. As the batch size increases, the aborted transactions gradually decrease.

Figure 8b shows the system performance under the YCSB-A workload. It can be seen that the abort rate is inversely proportional to the number of batches. Due to the high data skewness, the abort rate is high when transactions are not partitioned (1 batch). The partitioning strategy alone can only reduce a portion of the abort rate. Compared to SmallBank, the overall abort rate remains high. Based on the partitioning characteristics, the number of transactions executed in parallel is reduced and conflicts are reduced, the abort rate approaches zero as the number of batches approaches one transaction per batch, meaning the system executes transactions serially. But the latency of the system also increases gradually. Based on the evaluation of the two different workloads, it can be concluded that when the transactions contain more write operations, a higher number

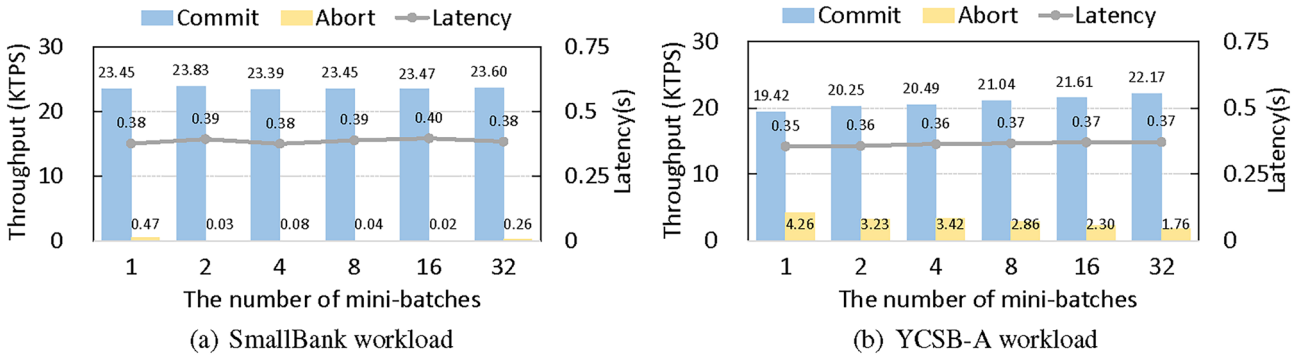


Fig. 8. The effect of different batches, where 1, 2, ... represent the number of mini-batches. (**a**) This shows the testing of different batch sizes in mini-batch partitioning under the SmallBank workload. (**b**) This shows the testing of different batch sizes in mini-batch partitioning under the YCSB-A workload. Their x-axis denotes the number of batches, and the y-axis has the same meaning as in point 6.

of mini-batches can be chosen to reduce the abort rate. Conversely, fewer batches can be selected to increase execution speed.

Conclusion

In this paper, we identify the shortcomings of distributed systems deployed across geographical regions when executing transactions concurrently. Transactions aborted due to read/write conflicts consume network and computational resources, and this wasteful consumption severely affects system performance. We introduce three strategies to reduce the overhead of unnecessary network and computational resources in distributed systems. We propose and implement GB-DCC, an optimized deterministic concurrency control on the geographic blockchain, which integrates these three strategies into the permissioned blockchain NeuChain. We design the three strategies and discuss their generality. Finally, we conduct comprehensive evaluations, showing that GB-DCC effectively reduces the number of aborted transactions compared to NeuChain, efficiently utilizing network and computational resources.

Data availibility

The datasets generated and analyzed during the current study are available from the corresponding author upon reasonable request.

Received: 4 February 2025; Accepted: 28 April 2025
Published online: 02 June 2025

References

1. Van Steen, M. & Tanenbaum, A. S. *Distributed systems* (Maarten van Steen Leiden, The Netherlands, 2017).
2. Wang, J., Xu, C., Zhang, J. & Zhong, R. Big data analytics for intelligent manufacturing systems: A review. *J. Manuf. Syst.* **62**, 738–752 (2022).
3. Duan, S. et al. Distributed artificial intelligence empowered by end-edge-cloud computing: A survey. *IEEE Commun. Surveys Tutorials* **25**, 591–624 (2022).
4. Lavin, A. et al. Simulation intelligence: Towards a new generation of scientific methods. arXiv preprint arXiv:2112.03235 (2021).
5. Jiang, Y. et al. Reliable distributed computing for metaverse: A hierarchical game-theoretic approach. *IEEE Trans. Veh. Technol.* **72**, 1084–1100 (2022).
6. Harding, R., Van Aken, D., Pavlo, A. & Stonebraker, M. An evaluation of distributed concurrency control. *Proc. VLDB Endowment* **10**, 553–564 (2017).
7. Ye, C., Hwang, W.-C., Chen, K. & Yu, X. Polaris: Enabling transaction priority in optimistic concurrency control. *Proc. ACM Manag. Data* **1**, 1–24 (2023).
8. Harding, R., Van Aken, D., Pavlo, A. & Stonebraker, M. An evaluation of distributed concurrency control. *Proc. VLDB Endowment* **10**, 553–564 (2017).
9. Thomson, A. et al. Calvin: fast distributed transactions for partitioned database systems. In Proceedings of the 2012 ACM SIGMOD international conference on management of data, 1–12 (2012).
10. Lu, Y., Yu, X., Cao, L. & Madden, S. *Aria. Proceedings of the VLDB Endowment* **13**, 2047–2060 (2020).
11. Wang, X., Peng, Y. & Huang, H. Gria: An efficient deterministic concurrency control protocol. *Frontiers Comput. Sci.* **18**, 184204 (2024).
12. Zhang, J. et al. Esdb: Processing extremely skewed workloads in real-time. In Proceedings of the 2022 International Conference on Management of Data, 2286–2298 (2022).
13. Chen, Y. & Bellavitis, C. Decentralized finance: Blockchain technology and the quest for an open financial system. Stevens Institute of Technology School of Business Research Paper (2019).
14. Satwika, I. K. S. & Andika, I. G. Performance analysis of an e-commerce website using distributed servers (case study: Ecommerce bumdes sarining kukuh winangun). *J. Comput. Netw. Architecture High Perform. Comput.* **6**, 1390–1398 (2024).
15. Yang, C.-T., Shih, W.-C., Huang, C.-L., Jiang, F.-C. & Chu, W.C.-C. On construction of a distributed data storage system in cloud. *Computing* **98**, 93–118 (2016).
16. Yang, Y. & Zhu, J. Write skew and zipf distribution: Evidence and implications. *ACM Trans. Storage (TOS)* **12**, 1–19 (2016).
17. Peng, Z. et al. Neuchain: A fast permissioned blockchain system with deterministic ordering. *Proc. VLDB Endowment* **15**, 2585–2598 (2022).

18. Hong, Z. et al. Prophet: Conflict-free sharding blockchain via byzantine-tolerant deterministic ordering. In IEEE INFOCOM 2023-IEEE Conference on Computer Communications, 1–10 (IEEE, 2023).
19. Chen, Z. et al. Spectrum: Speedy and strictly-deterministic smart contract transactions for blockchain ledgers. *Proc. VLDB Endowment* **17**, 2541–2554 (2024).
20. Androulaki, E. et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proceedings of the thirteenth EuroSys conference, 1–15 (2018).
21. Gupta, S., Rahnama, S., Hellings, J. & Sadoghi, M. Resilientdb: Global scale resilient blockchain fabric. arXiv preprint arXiv:2002.00160 (2020).
22. Böhme, R., Christin, N., Edelman, B. & Moore, T. Bitcoin: Economics, technology, and governance. *J. Econ. Perspectives* **29**, 213–238 (2015).
23. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R. & Sears, R. Benchmarking cloud serving systems with ycsb. In Proceedings of the 1st ACM symposium on Cloud computing, 143–154 (2010).

Author contributions
Z.H. conducted experiments, analyzed the results, and drafted the manuscript. Z.H. and Z.P. developed the system and revised the manuscript. All authors formulated a plan, reviewed the manuscript, and provided revision suggestions.

Funding
This work was funded by the National Natural Science Foundation of China (62372097), the Fundamental Research Funds for the Central Universities (N2416003).

Declarations

Competing interests
The authors declare no competing interests.

Additional information
Correspondence and requests for materials should be addressed to G.Y.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.