Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.073315

ARTICLE

HATLedger: An Approach to Hybrid Account and Transaction Partitioning for Sharded Permissioned Blockchains

Shuai Zhao, Zhiwei Zhang*, Junkai Wang, Ye Yuan and Guoren Wang

School of Computer Science & Technology, Beijing Institute of Technology, Beijing, 100081, China

*Corresponding Author: Zhiwei Zhang. Email: zwzhang@bit.edu.cn

Received: 15 September 2025; Accepted: 04 November 2025; Published: 12 January 2026

ABSTRACT: With the development of sharded blockchains, high cross-shard rates and load imbalance have emerged as major challenges. Account partitioning based on hashing and real-time load faces the issue of high cross-shard rates. Account partitioning based on historical transaction graphs is effective in reducing cross-shard rates but suffers from load imbalance and limited adaptability to dynamic workloads. Meanwhile, because of the coupling between consensus and execution, a target shard must receive both the partitioned transactions and the partitioned accounts before initiating consensus and execution. However, we observe that transaction partitioning and subsequent consensus do not require actual account data but only need to determine the relative partition order between shards. Therefore, we propose a novel sharded blockchain, called HATLedger, based on Hybrid Account and Transaction partitioning. First, HATLedger proposes building a future transaction graph to detect upcoming hotspot accounts and making more precise account partitioning to reduce transaction cross-shard rates. In the event of an impending overload, the source shard employs simulated partition transactions to specify the partition order across multiple target shards, thereby rapidly partitioning the pending transactions. The target shards can reach consensus on received transactions without waiting for account data. The source shard subsequently sends the account data to the corresponding target shards in the order specified by the previously simulated partition transactions. Based on real transaction history from Ethereum, we conducted extensive sharding scalability experiments. By maintaining low cross-shard rates and a relatively balanced load distribution, HATLedger achieves throughput improvements of 2.2x, 1.9x, and 1.8x over SharPer, Shard Scheduler, and TxAllo, respectively, significantly enhancing efficiency and scalability.

KEYWORDS: Sharded blockchain; account partitioning; cross-shard transaction rate; load imbalance

1 Introduction

Blockchains [1–3] have garnered sustained attention from both academia and industry. Various sharded blockchains [4–6] have been proposed to improve system performance. In SharPer [4], each account is managed by only one shard. Each shard S_i consists of N_i nodes, among which faulty nodes F_i satisfy the condition $N_i \geq 3F_i + 1$. In this generic blockchain sharding model, system performance is constrained by the cross-shard transaction rate and load distribution. OmniLedger [7] uses hash-based sharding to evenly distribute accounts across shards, resulting in a high cross-shard transaction rate. To address this issue, existing techniques [8–10] propose account partitioning to effectively balance the cross-shard transaction rate and cross-shard load distribution.

As shown in Fig. 1, the Hash-based account partitioning method [4] is limited by high cross-shard transaction rates. Shard Scheduler [8] proposes partitioning transactions and their associated accounts across low-load shards based on the current load distribution, achieving better load balancing but still encountering high cross-shard transaction rates. In contrast, TxAllo [9] constructs a hotspot account graph based on historical transactions, and assigns hotspot accounts to the same shard to reduce cross-shard transaction rates. However, TxAllo faces challenges that shard with hotspot accounts is overloading, while other shards remain underutilized.

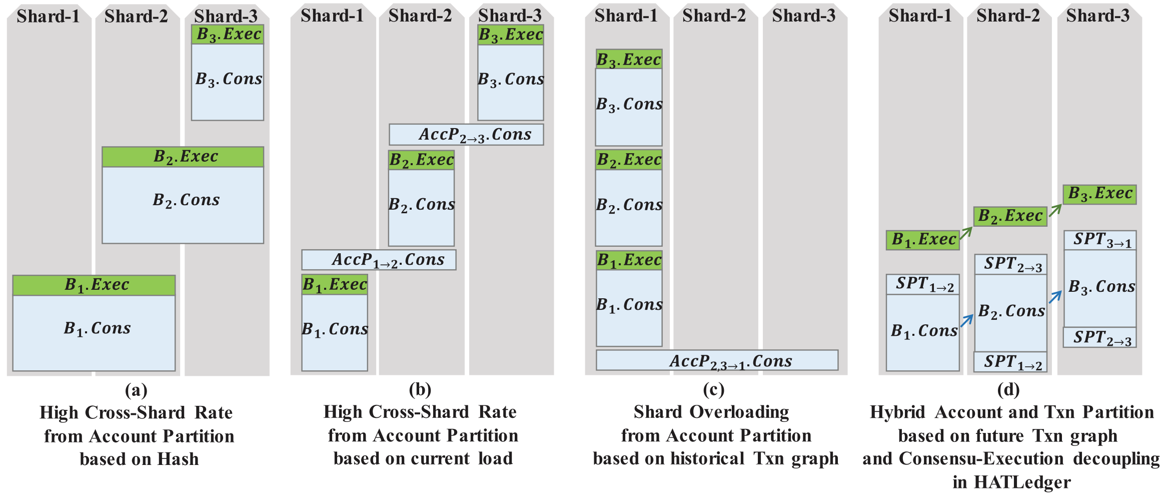
Shard-1
Shard-2
Shard-3
$B_3.Exec$
$B_3.Cons$
$B_2.Exec$
$B_2.Cons$
$B_1.Exec$
$B_1.Cons$
(a)
High Cross-Shard Rate from Account Partition based on Hash
$AccP_{2\to3}.Cons$
$AccP_{1\to2}.Cons$
(b)
High Cross-Shard Rate from Account Partition based on current load
$AccP_{2,3\to1}.Cons$
(c)
Shard Overloading from Account Partition based on historical Txn graph
$SPT_{1\to2}$
$SPT_{2\to3}$
$SPT_{3\to1}$
(d)
Hybrid Account and Txn Partition based on future Txn graph and Consensu-Execution decoupling in HATLedger

Figure 1: Account partition in different systems: (**a**) Account Partition in SharPer [4]; (**b**) Account partition in shard scheduler [8]; (**c**) Account Partition in TxAllo [9]; (**d**) Hybrid account and transaction partition in our HATLedger

As shown in Fig. 1a, the hash-based account partitioning method SharPer [4] distributes accounts randomly across shards, forcing B_1 and B_2 to undergo cross-shard consensus and execution, thus suffering from a high cross-shard transaction ratio. In Fig. 1b, Shard Scheduler [8] partitions accounts through frequent cross-shard transactions ($AccP_{1\to2}$ and $AccP_{2\to3}$) according to the current load distribution, assigning B_1, B_2, and B_3 to low-load shards to achieve a more balanced workload. However, it still faces a high cross-shard transaction ratio. In Fig. 1c, TxAllo [9] constructs a hotspot account graph based on historical transactions and performs a cross-shard partitioning ($AccP_{2,3\to1}$) to allocate hotspot accounts to the same shard, thereby reducing the cross-shard transaction ratio. Consequently, B_1, B_2, and B_3 in TxAllo are all intra-shard. However, TxAllo faces challenges that shard with hotspot accounts is overloading, while other shards remain underutilized.

We analyze the causes of high cross-shard transaction rates and shard overloading. Based on [8,10,11], the workload in blockchains can be divided into transaction consensus load and transaction execution load. Additionally, account partition can be regarded as a special type of consensus load. In a sharded blockchain system, the consensus load requires relevant nodes to reach consensus through multi-round communication protocols, such as PBFT [12]. In contrast, the transaction execution load only requires each node to locally execute and commit transactions in the consensus order. Therefore, the transaction consensus load is significantly higher than the transaction execution load. As shown in Fig. 1a–c, existing account partitioning and transaction partitioning are limited by Consensus-Execution coupling. Therefore, the target shard can only begin reaching consensus on the partitioned transactions after receiving both transactions and accounts.

However, we observe that transaction consensus only requires the relative partition order between shards, whereas transaction execution relies on account data. Moreover, the overhead associated with consensus among multiple nodes is considerably higher than that of local transaction execution on individual nodes. This creates an opportunity to decouple consensus and execution during partition. As shown in Fig. 1d, in HATLedger, the source shard can partition B_1, B_2, and B_3 across target shards. The target shard immediately begins consensus without waiting for account data. Subsequently, the source shard proceeds to send the account data in the previously specified order ($SPT_{1\to2}$ and $SPT_{2\to3}$). Therefore, transaction partitioning with Consensus–Execution decoupling can effectively address the issue of shard overloading while ensuring a low cross-shard transaction rate, significantly enhancing overall system performance.

A sharding approach aimed at lowering the cross-shard rate and mitigating load imbalance faces several challenges: (1) How to select appropriate accounts for partitioning to effectively reduce the cross-shard transaction rate. As the system load fluctuates dynamically, existing methods based on historical transactions for account selection incur significant latency, thereby failing to effectively reduce the cross-shard transaction rate. (2) How to decouple consensus and execution during transaction partitioning, thereby effectively mitigating load imbalance. (3) How to ensure the security of system in the presence of faulty nodes.

To address the aforementioned challenges, we propose a sharded blockchain ledger based on the **H**ybrid **A**ccount and **T**ransaction partitioning, denoted as **HATLedger**. First, at the beginning of each epoch, we propose constructing a future transaction graph based on the pending transactions in each shard. The proposed future transaction graph can more effectively identify upcoming hotspot accounts. Hotspot accounts are detected, and an account sharding strategy is developed for this epoch, thereby effectively reducing the cross-shard transaction rate. Next, we propose **S**imulated **P**artition **T**ransactions to decouple consensus and execution. When an overload is imminent, the source shard first constructs two blocks, B_{source} and B_{target}. Subsequently, the source shard reaches consensus on B_{source}, while the target shard reaches consensus on B_{target}. Before this, simulated partition transactions are added to these blocks, as shown in Fig. 1d. These simulated partition transactions specify the partition order of accounts between shards. Upon receiving B_{target} and its subsequent transactions from the source shard, the target shard initiates consensus without waiting for the actual account data. Subsequently, the source shard sequentially sends the account data to the target shard in the order specified by the previously simulated partition transactions. Finally, we analyze the security of HATLedger. Consequently, HATLedger not only maintains a low cross-shard transaction rate but also resolves shard overloading issues through simulated partition transactions, significantly enhancing system performance.

In summary, this paper provides the following contributions:

Contribution 1: Compared to outdated hotspot accounts detected from historical transaction graphs, we propose constructing a future transaction graph based on pending transactions. By leveraging the identified upcoming hotspot accounts, we achieve more precise account partitioning, which effectively reduces the cross-shard transaction rate.

Contribution 2: We propose simulated partition transactions to decouple consensus and execution. We implement a sharded blockchain ledger based on the hybrid account and transaction partitioning, denoted as HATLedger. In the event of an impending overload, the source shard quickly partitions pending transactions to target shards. The target shard can simultaneously initiate consensus on the partitioned blocks without additional waiting for account data.

Contribution 3: We provide a detailed analysis showing that HATLedger ensures both safety and liveness even in the presence of malicious attacks from faulty nodes.

Contribution 4: We conduct extensive scalability experiments under real workloads. The results show that HATLedger's hybrid account and transaction partitioning effectively reduces cross-shard transaction rates and mitigates load imbalance, thereby leading to significant performance gains.

2 Background and Analysis

2.1 Background

Existing research [4–6,8,9,13] has made extensive attempts to explore sharding structures in sharded blockchains. SharPer [4], Shard Scheduler [8], and TxAllo [9] adopt a one-to-one account-to-shard mapping, where each account is managed by exactly one shard. Zilliqa [14] and LMChain [5] employ sharding technology based on a beacon chain. While each account is managed by a single shard, the beacon chain in the system acts as a scheduler, synchronizing the global state across all shards. BrokerChain [6] and SharDAG [13] adopt a one-to-many account-to-shard mapping, where some accounts can be replicated or distributed across multiple shards for concurrent processing. Therefore, to ensure greater generality and adaptability, this work focuses on the one-to-one sharding structure. Next, we introduce the system model and assumptions under the one-to-one sharding structure.

2.2 Analysis

Table 1 summarizes existing works in one-to-one blockchain sharding models. In SharPer [4], accounts are evenly distributed across shards based on account hashes. Under ideal conditions with uniformly distributed account access, the Hash-based partition strategy achieves load balancing. However, when account access becomes skewed, the shard containing hotspot accounts may experience overloading. Shard Scheduler [8] allocates pending transactions and new accounts to low-load shards based on the current load of each shard. This ensures load balancing across shards even under skewed account access by leveraging account partition and transaction partition. Nevertheless, both SharPer and Shard Scheduler face the limitation of high cross-shard transaction rates due to account dispersion. On the other hand, TxAllo [9] constructs a historical transaction graph from historical transactions and applies community detection algorithms to identify hotspot accounts, which are then assigned to the same shard. While this approach reduces the cross-shard transaction rate for transactions involving hotspot accounts, it inevitably leads to overloading of shards due to the concentration of hotspot accounts within the same shard.

By analyzing the real transaction history on Ethereum, existing studies [5,8–10] consistently reveal that a small portion of hotspot accounts dominate the majority of transactions, highlighting the skewed distribution of account access. Hu et al. [5] pointed out that approximately 14.3% of accounts are involved in nearly 90% of transactions, while the top 14.3% of accounts are not constant. Therefore, as the load dynamically changes, outdated hotspot accounts identified from the historical transaction graph cannot effectively reduce the cross-shard transaction rate in future workloads.

Table 1: Comparison of blockchain sharding mechanisms

System	Account partition	Transaction partition	Cross-shard rate	Load distribution
SharPer [4]	Hash-based	Consensus-Execution coupling	**High**	**Imbalance**
Shard Scheduler [8]	Current load	Consensus-Execution coupling	**High**	**Balance**
TxAllo [9]	Historical Txn Graph	Consensus-Execution coupling	**Low**	**Imbalance**
HATLedger	Future Txn Graph	Consensus-Execution decoupling	**Low**	**Balance**

Next, we provide a detailed analysis of how hotspot account distribution affects sharding performance. The distribution of hotspot accounts impacts system performance primarily by influencing the cross-shard transaction rate and the workload balance among shards. (1) When hotspot accounts are uniformly distributed across shards, shard workloads become balanced. However, because most transactions involve hotspot accounts stored in different shards, the cross-shard transaction rate increases substantially. Consequently, the frequency of cross-shard consensus operations—characterized by high latency and communication overhead—rises significantly, thereby degrading overall system performance. (2) When hotspot accounts are partitioned within a single shard, most transactions access hotspot accounts located in that shard. As a result, the cross-shard transaction rate drops sharply, and intra-shard consensus—characterized by low latency and minimal communication overhead—becomes dominant. However, because most transactions are processed within the same shard, the limited processing capacity of that shard leads to overload, while other shards remain idle. Consequently, overall performance deteriorates due to inefficient resource utilization.

As shown in Table 1, by comparing existing studies, we further analyze the causes that limit system performance and identify potential opportunities for improvement. As shown in Fig. 2a, we observe that the transaction partition with Consensus-Execution Coupling in existing works typically follows the sequence: **Transaction Consensus Loads and Execution Loads in the Source Shard → Transaction Partition → Transaction Consensus Loads and Execution Loads in the Target Shard**. The target shard can only begin reaching consensus on the partitioned transactions after receiving actual account data. However, the consensus phase in blockchain systems focuses on the consensus of pending transactions and their order [11]. We observe that when the target shard reaches consensus on the partitioned transactions, it only needs the source shard's ID and the hash of the previous block, without requiring the actual account data. This provides a potential opportunity to decouple consensus and execution, thereby addressing shard overloading and improving system performance. In Fig. 2b, we propose a transaction partitioning scheme with Consensus–Execution Decoupling. Pending transactions are first partitioned to the target shard, initiating the corresponding consensus workload without waiting for data. The execution workload is then triggered once the associated data have been partitioned to the target shard.

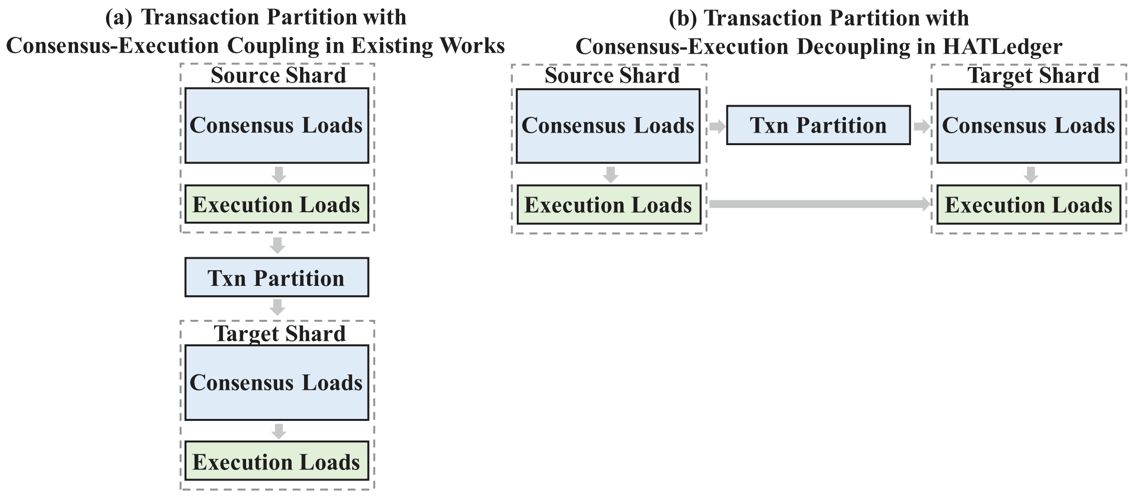

Figure 2: Transaction partition in different systems: (**a**) Transaction partition with consensus-execution coupling in existing works; (**b**) Transaction partition with consensus-execution decoupling in HATLedger

3 HATLedger

3.1 Overview

We propose **HATLedger**, a sharded blockchain ledger based on the **H**ybrid **A**ccount and **T**ransaction partitioning, to further enhance system performance.

System Model and Assumptions: The system model and assumptions used in this work, HATLedger, are consistent with those of SharPer [4]. HATLedger consists of a series of distributed nodes. Each shard S_i comprises N_i nodes, among which the number of Byzantine malicious nodes F_i satisfies the condition $N_i \geq 3F_i + 1$. HATLedger employs Byzantine fault-tolerant (BFT) protocols, such as PBFT [12], to ensure deterministic safety. As is common, this work assumes the *partially synchronous communication model* [12,15]. Additionally, messages might include public-key signatures and message digests [12]. We denote a message m signed by a node n_i as $\langle m \rangle_{\sigma_i}$, and the digest of a message m as $D(m)$. We also assume a strong adversary capable of coordinating malicious nodes and delaying communication to disrupt the replicated service. However, faulty nodes cannot forge messages from non-faulty nodes. Each account is managed by only one shard. We adopt a periodic trigger mechanism based on lightweight load detection. When the number of pending transactions or the transaction latency exceeds θ times the throughput or latency under the normal case ($\theta = 2$ by default), the trigger identifies the shard as approaching overload and initiates account and transaction repartitioning.

Transaction Processing Workflow: When a transaction accesses accounts within a single shard, it is classified as an intra-shard transaction and is processed solely within that shard for both consensus and execution. When a transaction accesses accounts across multiple shards, it is classified as a cross-shard transaction. For cross-shard transactions, multiple leaders from the involved shards compete to become the proposer of the transaction. This proposer coordinates the involved shards to reach consensus on the cross-shard transaction using the PBFT protocol. Existing studies [4,11] have thoroughly explored the workflows and security of transaction processing for intra-shard and cross-shard transactions. As this work focuses on account partitioning, transaction processing is beyond the scope of this paper.

Design Goals: HATLedger proposes a novel hybrid partitioning approach to address the challenges of dynamic hotspot accounts and shard overload in dynamic workloads.

System Workflow: As shown in Fig. 3, HATLedger operates in epochs. The workflow of HATLedger is as follows:

(1) At the beginning of a new epoch, HATLedger constructs a future transaction graph based on the pending transactions of all shards. Using this graph, HATLedger detects the upcoming hotspot accounts and repartitions all accounts across shards accordingly, thereby reducing the cross-shard transaction rate.

(2) Each shard then reaches consensus on transactions under the new account partition, executes the transactions, and updates the account data. When a shard detects an impending overload, it can further repartition the numerous pending transactions of hotspot accounts to different underloaded shards. Upon receiving these transactions, the underloaded shards proactively initiate intra-shard concurrent consensus.

(3) Once the source shard completes processing all relevant transactions, it transfers the updated account data to the underloaded shards. The underloaded shard then utilizes the received account data to serially execute transactions that have already achieved consensus.

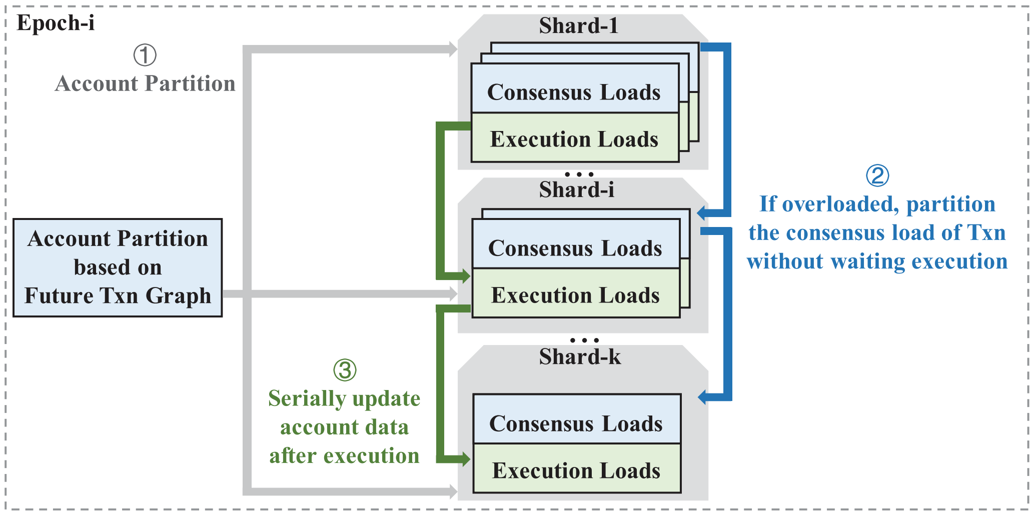
Epoch-i
①
Account Partition
Account Partition based on Future Txn Graph
Shard-1
Consensus Loads
Execution Loads
Shard-i
Consensus Loads
Execution Loads
Shard-k
Consensus Loads
Execution Loads
②
If overloaded, partition the consensus load of Txn without waiting execution
③
Serially update account data after execution

Figure 3: System workflow of HATLedger

3.2 Account Partitioning Based on Future Transaction Graph

As shown in Algorithm 1, at the beginning of a new epoch Ep_k, the shard with an identifier that satisfies the condition $(Shard_j == Ep_k \% S_{num})$ serves as the coordinator shard, which facilitates the initial account repartitioning (Line 13). Based on the accounts accessed by pending transactions in epoch Ep_k, the remaining shards construct their respective future transaction graphs. In the **future transaction graph** FTG, a vertex v represents an account ID, and a weighted edge $\langle u, v, w \rangle$ indicates that w pending transactions access both accounts u and v simultaneously. Each shard sends its future transaction graph FTG to the coordinator shard (Lines 2 to 8). After merging the $FTGs$ from all shards to rebuild the FTG, the coordinator shard utilizes a graph-based partitioning method, such as METIS [16], to generate the account partitioning proposal $Partition_k$. The coordinator shard then reaches consensus with all other shards on $Partition_k$ (Lines 13 to 18). Based on the consensus account partition $\langle Partition_k \rangle_\sigma$, each shard migrates the relevant accounts and associated pending transactions. Following the completion of account repartitioning for the new epoch Ep_k, all shards process pending transactions according to the transaction processing workflow outlined in Section 3.1 (Line 10 and Line 20). When impending overload is detected, the shard preemptively mitigates the loads using simulated partition, ensuring efficient workload distribution (Lines 11 to 12 and Lines 21 to 22).

The sharding model guarantees that each account is managed exclusively by a single shard. Suppose accounts x and y belong to $Shard_i$, while account z belongs to $Shard_j$. For an intra-shard transaction $T(x, y)$, the edge $\langle x, y \rangle$ in the future transaction graph (FTG_i) of $Shard_i$ has its edge weight incremented by 1. Consequently, edge $\langle x, y \rangle$ exists only in FTG_i and not in any other shard's FTG. For a cross-shard transaction $T(x, z)$, the edge $\langle x, z \rangle$ of FTG_i is incremented by 1, and $\langle x, z \rangle$ of FTG_j is incremented by 1. After merging FTG_i and FTG_j, the edge weight of $\langle x, z \rangle$ becomes 2. All transactions are executed under MVCC-based snapshot isolation. If a transaction reads an outdated snapshot or encounters a concurrent conflict, it is marked as invalid.

Algorithm 1: Account partitioning based on future transaction graph

```
Data: $Epk$ is the next epoch number; $Shard_{num}$ is the number of shards; $N_{all}$ is the set of all nodes in the
      system; $N_i$ is the set of nodes in $Shard_i$, where its proposer is $n_i$.
1 PROPOSER of Shard_i: Node n_i ($Ep_k\%Shard_{num}! = Shard_i$)
2     Upon Enter Next Epoch $Ep_k$ Do
3         for each $T \in PendingT_i$ do
4             for each $u$ and $v$ accessed by $T$ do
5                 $FTG_i.addEdge(u, v)$;
6                 $EWeight_i(u, v)$++;
7             $Shard_j \leftarrow Ep_k\%Shard_{num}$;
8             Send$\langle FTG_i, EWeight_i \rangle$ to $Shard_j$;
9     Upon Reach Consensus on $\langle Partition_k \rangle_\sigma$ of $Ep_k$ Do
10        Process $PendingT_i$ based on $Partition_k$;
11        When Impending Overloading Detected
12            $TxnPartition(Shard_i, PendingT_i, Partition_k)$; // Details in Algorithm 2
13 PROPOSER of Shard_j: Node n_j ($Ep_k\%Shard_{num} == Shard_j$)
14    Upon Enter Next Epoch $Ep_k$ Do
15        When Receive all FTGs from all Shard
16            $FTG_k \leftarrow Merge(FTGs)$;
17            $Partition_k \leftarrow FTG_k.Partition()$;
18            $Consensus(Partition_k, N_{all})$;
19    Upon Reach Consensus on $\langle Partition_k \rangle_\sigma$ of $Ep_k$ Do
20        Process $PendingT_j$ based on $Partition_k$;
21        When Impending Overloading Detected
22            $TxnPartition(Shard_j, PendingT_j, Partition_k)$; // Details in Algorithm 2
```

Analysis: (1) The account partition based on the future transaction graph has the same cost and security guarantees as TxAllo's method, which uses the historical transaction graph. Through global consensus, all shards adopt a consistent account partition. Moreover, the future transaction graph enables more precise detection of upcoming hotspot accounts, leading to more effective account partitioning and significantly reducing the cross-shard transaction rate. Experiments in Section 5 validate this observation. (2) HATLedger maintains both security and liveness even in the presence of malicious behavior. The potential malicious actions include incomplete FTGs sent by proposers from other shards and inappropriate account partitioning schemes proposed by the coordinator shard's proposer. However, consensus protocols such as PBFT [12] ensure that all shards reach an agreement on a consistent account partitioning scheme, guaranteeing the system's security. Additionally, the coordinator shard rotates the coordinator role and repartitions accounts at the beginning of each new epoch, effectively mitigating the long-term impact of inappropriate partitioning and enhancing system robustness. While a malicious proposer from the coordinator shard may cause temporary performance degradation by proposing an inappropriate partitioning scheme, other shards can promptly identify and report such behavior. Furthermore, the transaction partitioning techniques introduced in the next section effectively address shard overloading caused by inappropriate partitioning schemes. Therefore, HATLedger maintains both security and liveness.

3.3 Transaction Partitioning Based on Consensus–Execution Decoupling

As discussed in Section 2, assigning hotspot accounts to the same shard effectively reduces the cross-shard transaction rate. However, it introduces a critical issue: overloading the associated shard and underutilizing other shards. To address this issue, we propose a transaction partitioning approach with Consensus–Execution decoupling.

As described in [8,9], the transaction partitioning in existing works, which is based on **Consensus–Execution coupling**, is as follows: consensus and execution can begin in a shard only when it holds both the partitioned account data and the corresponding assigned pending transactions.

In contrast, our proposed transaction partitioning in HATLedger, which is based on **Consensus–Execution decoupling**, is as follows: upon receiving the partitioned pending transactions, a shard promptly initiates consensus without the corresponding account data. The shard then executes the agreed transactions once the corresponding account data becomes available.

Therefore, we present the following theorem:

Theorem 1: *Both the transaction partitioning in existing works, which is based on Consensus–Execution coupling, and the transaction partitioning in HATLedger, which is based on Consensus–Execution decoupling, achieve equivalent results for transaction consensus and execution, thereby ensuring system consistency.*

Proof: Assume the source shard contains a partitioned account $\langle \mathrm{u}, \mathrm{v}_0 \rangle$, a related transaction $\mathrm{T}_1(\mathrm{u})$ pending consensus on the source shard, and another transaction $\mathrm{T}_2(\mathrm{u})$ designated for partitioning to the target shard. Since account data is not involved during transaction consensus, the outputs of both the existing workflow and HATLedger are the same. The source shard first reaches consensus on $\mathrm{T}_1(\mathrm{u})$, executes it, and updates the account $\langle \mathrm{u}, \mathrm{v}_1 \rangle$. Subsequently, the target shard reaches consensus on $\mathrm{T}_2(\mathrm{u})$ and executes $\mathrm{T}_2(\mathrm{u})$, updating $\langle \mathrm{u}, \mathrm{v}_1 \rangle$ to $\langle \mathrm{u}, \mathrm{v}_2 \rangle$.□

Therefore, as shown in Fig. 4, when $Shard_1$ in HATLedger faces impending overload, transaction partitioning can be performed continuously. $Shard_1$ selects the partitioned accounts $PAccSet\{acc\}$ and their associated pending transactions $PTxnSet$ from the pending transaction pool. Subsequently, two blocks, B_1 and B_2, are constructed from $PTxnSet$, where B_1 contains the simulated partition-out transaction $SPT_{out}Shard_1 \rightarrow Shard_2, h(B_2)$ and B_2 contains the simulated partition-in transactions $SPT_{in}Shard_1 \rightarrow Shard_2, h(B_1)$. $Shard_1$ then starts consensus on B_1 and simultaneously sends B_2 along with the remaining $PTxnSet$ to $Shard_2$. Upon receiving B_2 and $PTxnSet$, $Shard_2$ immediately begins consensus on B_2 without waiting for the actual account data $PAccSet\{A\}$. Similarly, $Shard_2$ can construct block B_3 from $PTxnSet$, where B_2 contains the simulated partition-out $SPT_{out}Shard_2 \rightarrow Shard_3, h(B_3)$ and B_3 contains the partition-in transactions $SPT_{in}Shard_2 \rightarrow Shard_3, h(B_2)$. This enables $Shard_1$, $Shard_2$, and $Shard_3$ to independently start consensus on B_1, B_2, and B_3 without waiting for one another. Once $Shard_1$ completes consensus and execution of B_1, it sends the updated set $\{PAccSet\{A, v_1\}, h(B_1), h(B_2)\}$ to $Shard_2$. Likewise, $Shard_2$ sends the updated $\{PAccSet\{A, v_2\}, h(B_2), h(B_3)\}$ to $Shard_3$.

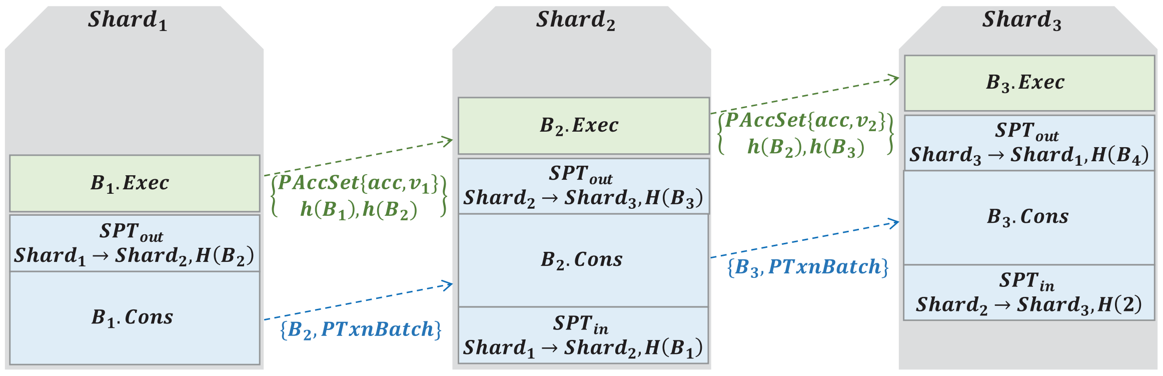
$Shard_1$
$B_1.Exec$
SPT_{out} $Shard_1 \to Shard_2, H(B_2)$
$B_1.Cons$
$\{PAccSet\{acc, v_1\}$ $h(B_1), h(B_2)\}$
$\{B_2, PTxnBatch\}$
$Shard_2$
$B_2.Exec$
SPT_{out} $Shard_2 \to Shard_3, H(B_3)$
$B_2.Cons$
SPT_{in} $Shard_1 \to Shard_2, H(B_1)$
$\{PAccSet\{acc, v_2\}$ $h(B_2), h(B_3)\}$
$\{B_3, PTxnBatch\}$
$Shard_3$
$B_3.Exec$
SPT_{out} $Shard_3 \to Shard_1, H(B_4)$
$B_3.Cons$
SPT_{in} $Shard_2 \to Shard_3, H(2)$

Figure 4: Transaction partitioning based on consensus–execution decoupling in HATLedger

Algorithm 2: Transaction partitioning based on consensus–execution decoupling

```
  Data: $Epk$ is the next epoch number; $Partition_k$ is the account partition in epoch $K$; $N_i$ is the set of nodes
        in $Shard_i$, where its proposer is $n_i$.
1 PROPOSER of $\mathbf{Shard_i}$: Node $\mathbf{n_i}$
2     When Impending Overloading Detected
3         $PAccSet, B_{source}, B_{target}, PTxnSet, Shard_{target} \leftarrow TxnPartition(Shard_i, PendingT_i,$
          $Partition_k)$;
4         Send $\langle B_{target}, PTxnSet \rangle$ to $Shard_{target}$;
5         $Consensus\ (B_{source}, Shard_i)$;
6         $PAccSet \leftarrow Execute(PAccSet, B_{source})$;
7         Send $\langle PAccSet, h(B_{source}), h(B_{target}) \rangle$ to $Shard_{target}$;
8 PROPOSER of $\mathbf{Shard_j}$: Node $\mathbf{n_j}$
9     Upon Receive $\langle B_{target}, PTxnSet \rangle$ from $Shard_{source}$ Do
10        $PendingT_j.push(PTxnSet)$;
11        $Consensus\ (B_{target}, Shard_j)$;
12        When Receive $\langle PAccSet, h(B_{source}), h(B_{target}) \rangle$ from $Shard_{source}$
13            $Validate\ (h(B_{source}), h(B_{target}))$;
14            $Execute\ (PAccSet, B_{source})$;
15     When Impending Overloading Detected
16         $TxnPartition\ (Shard_j, PendingT_j, Partition_k)$;
17 Function: $TxnPartition\ \ (Shard_{source}, PendingT_{source}, Partition_k)$
18     $Shard_{target} \leftarrow LowLoadShard\ (Partition_k)$;
19     $PAccSet, PTxnSet \leftarrow PendingT_{source}.pop()$;
20     $B_{source} \leftarrow PTxnSet.pop()$;
21     $B_{target} \leftarrow PTxnSet.pop()$;
22     $SPT_{in} \leftarrow \langle h(B_{source}), Shard_{source} \rangle$;
23     $SPT_{out} \leftarrow \langle h(B_{target}), Shard_{target} \rangle$;
24     $B_{source}.append\ (SPT_{out})$;
25     $B_{target}.append\ (SPT_{in})$;
26     return $PAccSet, B_{source}, B_{target}, PTxnSet, Shard_{target}$;
```

In Algorithm 2, we present the details of transaction partitioning upon detecting an impending overload. We propose using **simulated partition-out transactions** SPT_{out} and **simulated partition-in transactions** SPT_{in} to provide transaction partition information to the source and target shards, respectively, instead of actual account data. Specifically, when the source shard ($Shard_{source}$) detects impending overload, it selects a low-load target shard ($Shard_{target}$) according to the account partitioning ($Partition_k$). Subsequently, the source shard identifies the accounts to be partitioned ($PAccSet$) and all their associated pending transactions ($PTxnSet$) from the pending transaction pool ($PendingT_{source}$) (Lines 17 to 19). Two blocks, B_{source} and B_{target}, are then constructed from $PTxnSet$. B_{source} continues consensus in the source shard, while B_{target} and the remaining $PTxnSet$ are partitioned to the target shard for consensus. To replace actual account data, HATLedger requires consensus in the source shard regarding the target shard information, and vice versa. To achieve this, a **simulated partition-out transaction** SPT_{out} is inserted into B_{source}, containing the target shard's ID $Shard_{target}$ and the hash of the next related block B_{target} to be agreed upon in the target shard. Similarly, a **simulated partition-in transaction** SPT_{in} is inserted into B_{target}, containing the source shard's ID $Shard_{source}$ and the hash of the previous related block B_{source} agreed upon in the source shard (Lines 20 to 26).

Subsequently, the source shard sends the partitioned block B_{target} and the remaining $PTxnSet$ to the target shard (Line 4). Upon receipt, the target shard immediately initiates consensus on B_{target} without waiting for actual account data, as B_{target} contains the necessary partition information via the simulated partition-in transaction (SPT_{in}) (Lines 8 to 11). Meanwhile, the source shard initiates consensus on B_{source}, which contains account partition-out information in SPT_{out} (Line 5). After the source shard completes consensus and execution of B_{source}, it sends the updated account data to the target shard specified in the simulated partition-out transaction SPT_{out} of B_{source} (Lines 6 to 7). The target shard, upon receiving the account data, validates the hash and executes block B_{target}, followed by consensus and execution of the remaining pending transactions in $PTxnSet$ (Lines 12 to 14).

As a result, source and target shards can leverage the simulated partition to begin consensus on related transactions without waiting for the execution involving actual account data. This innovative transaction partitioning strategy allows HATLedger to mitigate shard overloading while significantly improving transaction throughput and overall system efficiency.

Analysis: (1) In terms of performance, HATLedger achieves a lower cross-shard transaction rate while effectively addressing shard overloading. Specifically, HATLedger constructs a future transaction graph to better identify upcoming hotspot accounts, enabling more effective account partitioning to reduce the cross-shard transaction rate. When impending overload is detected, the shard performs fine-grained transaction partitioning to efficiently reach consensus on pending transactions without requiring the source and target shards to wait on each other. (2) In terms of security, both the source shard and the target shard independently reach consensus on the simulated partition-out and partition-in transactions, ensuring consistency in the transaction partition order and the hashes of B_{source} and B_{target}. Additionally, the source shard transmits $h(B_{source})$ and $h(B_{target})$ alongside the actual account data during transaction partitioning. This mechanism ensures that if a proposer maliciously sends inconsistent B_{source} or B_{target}, the non-faulty nodes within the shard can immediately detect and report the behavior. These approaches ensure the security of HATLedger.

4 Safety and Liveness

In this section, we analyze the safety and liveness of HATLedger. Specifically, HATLedger adopts the same system model and assumptions, as well as the intra-shard and cross-shard transaction processing workflows of SharPer [4]. Consequently, it inherits the same level of security and liveness as SharPer in these aspects. Next, we analyze whether the solutions proposed by HATLedger—account partitioning based on the future transaction graph and transaction partitioning based on Consensus–Execution decoupling—may be impacted by malicious behavior and how they may affect its safety and liveness.

When faulty nodes act as followers in shards, Byzantine fault-tolerant protocols such as PBFT [12] are able to maintain the safety and liveness of HATLedger. Consequently, this discussion emphasizes scenarios where shard proposers are faulty nodes and examines their potential impact on the system.

Safety: (1) The account partitioning based on the future transaction graph does not affect the safety of HATLedger. This is because the account partitioning scheme proposed by the coordinator shard's proposer is agreed upon by all nodes in all shards through the consensus protocol. (2) To mitigate shard overloading, we propose the transaction partitioning method. In this process, we construct a block B_{source}, which reaches consensus in the source shard, and a block B_{target}, which reaches consensus in the target shard. Simulated partition-out and partition-in transactions are appended to these blocks, respectively. These transactions include the simulated partition-out and partition-in order, as well as the hashes of the preceding and succeeding blocks ($h(B_{source})$ and $h(B_{target})$). When the proposer of either the source shard or the target shard behaves maliciously, mismatches between B_{target} and $h(B_{target})$ or B_{source} and $h(B_{source})$ may occur. For example, the proposer of $Shard_{target}$ may propose a mismatched B_{target} to intra-shard nodes, which does not match $h(B_{target})$ specified in the SPT_{out} of B_{source}. Similarly, the proposer of $Shard_{source}$ may propose a mismatched B_{source} to intra-shard nodes or send a mismatched B_{target} to $Shard_{target}$'s proposer. At this point, through the hashes ($h(B_{source})$ and $h(B_{target})$) sent during the subsequent account data transfer in Algorithm 2, other nodes in both the source shard and the target shard can verify the hash or consensus signature to determine whether the proposer has acted maliciously. Only after successful verification will the other nodes execute and commit the transactions. Otherwise, the nodes can promptly report the malicious behavior and invoke the view-change mechanism of the consensus protocol to replace the proposer and re-initiate consensus and execution on the correct block. Therefore, HATLedger ensures safety.

Liveness: Consistent with existing sharded blockchain systems [4,8,9], when a malicious proposer intentionally delays initiating a proposal, HATLedger ensures liveness through the timeout detection mechanism of the consensus protocol's view-change process, which promptly replaces the proposer. At the beginning of a new epoch, if the proposer of the coordinator shard is a faulty node and proposes an inappropriate account partitioning scheme, it does not compromise the security of HATLedger, as previously discussed. However, it may lead to overloading in certain shards. To address this, the proposed transaction partitioning method effectively mitigates the impact of shard overloading. Furthermore, when a shard detects that its performance is significantly constrained by the inappropriate account partitioning, it can preemptively request a transition to the next epoch. Entering a new epoch involves a different shard serving as the coordinator, which proposes a new account partitioning scheme based on the future transaction graph. This mechanism guarantees that any adverse effects of inappropriate account partitioning are transient, ensuring long-term system stability and liveness. Therefore, HATLedger ensures liveness.

5 Evaluation

5.1 System and Setup

System: We implement HATLedger in C++. As shown in Table 2, we compare the following systems: SharPer [4], Shard Scheduler [8], TxAllo [9], HATLedger, and HATLedger*. Specifically, SharPer, Shard Scheduler, TxAllo, and HATLedger adopt different account partitioning strategies under Consensus–Execution Coupling. Compared to these systems, HATLedger* employs transaction partitioning based on Consensus–Execution decoupling, as described in Section 3.3.

Table 2: Systems for comparative experiments

System	Account partition	Transaction partition
Sharper	Hash-based (AccountID%$Shard_{num}$)	Consensus–Execution coupling
Shard scheduler	Present load	Consensus–Execution coupling
LB-Chain	Historical load of hot accounts	Consensus–Execution coupling
TxAllo	Historical Txn Graph	Consensus–Execution coupling
HATLedger	Future Txn Graph	Consensus–Execution coupling
HATLedger*	Future Txn Graph	Consensus–Execution decoupling

Setup: Our experiments are conducted on Alibaba Cloud, where the default cluster consists of 8 clients and 4 shards. Each shard is composed of 8 nodes, totaling 32 nodes across the 4 shards. The parameters for the scalability experiments are listed in Table 3, where Number of total nodes = Number of nodes per shard * Number of shards. Each node is equipped with a 4-core Intel Xeon (Sapphire Rapids) Platinum 8475 B processor and 16 GB memory. The operating system is 64-bit Ubuntu 20.04 LTS. We adopt a transaction-countbased epoch configuration, where the coordinating shard ($Shard_j = Ep_k\%Shard_{num}$) initiates a transition to the next epoch after committing 100,000 transactions in $Shard_j$. Both account and transaction repartitioning between the source and target shards are encapsulated as a cross-shard transaction, making the cost of one migration equivalent to that of a single cross-shard transaction. All systems employ PBFT [12] as the consensus protocol within each shard, and the experiments are conducted on Alibaba Cloud's internal network with a bandwidth of 1.2 Gbps.

Table 3: Setup for comparative experiments

Number of nodes per shard	Number of shards	Number of total nodes
8	4, 6, 8, 10, 12, 14	32, 48, 64, 80, 96, 112
8, 10, 12, 14, 16, 18	4	32, 40, 48, 56, 64, 72

Workload: Consistent with the workloads used in existing studies [4,9,11], we employ three types of workloads: ETH-Workload, Zipfian-Workload, and Hotspot-Workload. By default, all systems create 100,000 accounts. Specifically, ETH-Workload uses the Ethereum transactions between block heights of 13 to 13.2M (Aug.–Oct. 2021), as accounts were particularly active during this period [17]. The default parameter for the Zipfian-Workload (Zipf = 0.99) reflects common workload distributions in distributed systems. For the Hotspot-workload, 1% of the accounts are designated as hotspot accounts, and 90% of the transactions access these hotspot accounts, following a distribution similar to that described in [5]. For each test, we repeat the experiment five times and take the average as the final result.

5.2 Overall Performance

As shown in Table 3, we conducted extensive sharding scalability experiments to compare the performance of different systems. Using the default parameters underlined in Table 3 and the Eth-Workload, we evaluated the overall performance of all systems. As illustrated in Fig. 5, HATLedger* achieved the highest throughput of 37,398 tps, followed by HATLedger (32,317 tps), LB-Chain (23,703 tps), TxAllo

(20,771 tps), Shard Scheduler (19,675 tps), and SharPer (16,929 tps). Compared to SharPer, Shard Scheduler, TxAllo, and LB-Chain, HATLedger* demonstrated significant throughput improvements of 2.2x, 1.9x, 1.8x, and 1.6x, respectively. Meanwhile, as shown in Table 4, HATLedger and HATLedger* exhibited the lowest average latency, attributable to the significant reduction in the cross-shard transaction rate. Across all experiments (Figs. 6–9), HATLedger* consistently achieved the best performance. Additionally, the results of the ablation experiments validated the efficiency of the techniques proposed by HATLedger* for account partitioning based on the future transaction graph and transaction partitioning based on Consensus–Execution decoupling.

Table 4: The average latency across different systems

System	Sharper	Shard scheduler	TxAllo	LB-Chain	HATLedger	HATLedger*
Latency	380 ms	362 ms	238 ms	365 ms	187 ms	185 ms

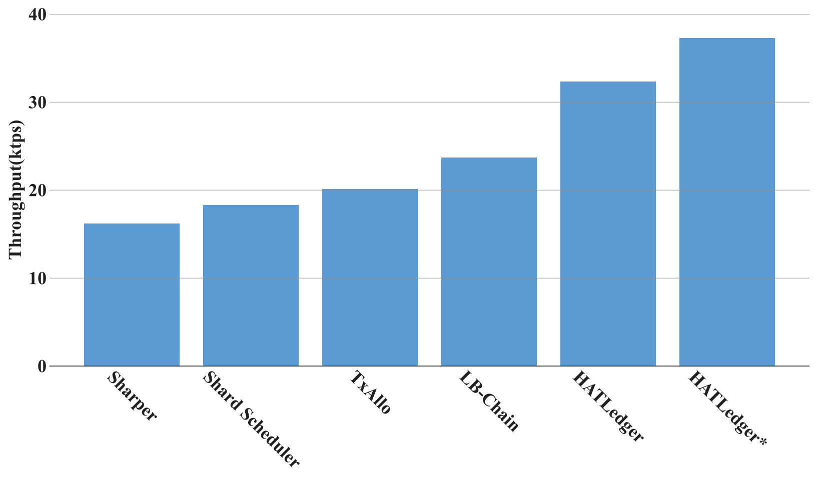

Figure 5: The overall performance of different systems

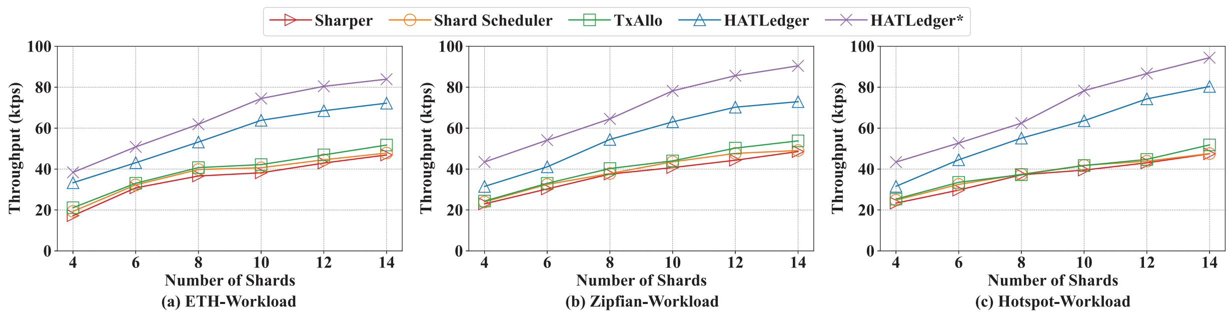

Figure 6: The throughput with different numbers of shards under (**a**) ETH-Workload, (**b**) Zipfian-Workload, and (**c**) Hotspot-Workload

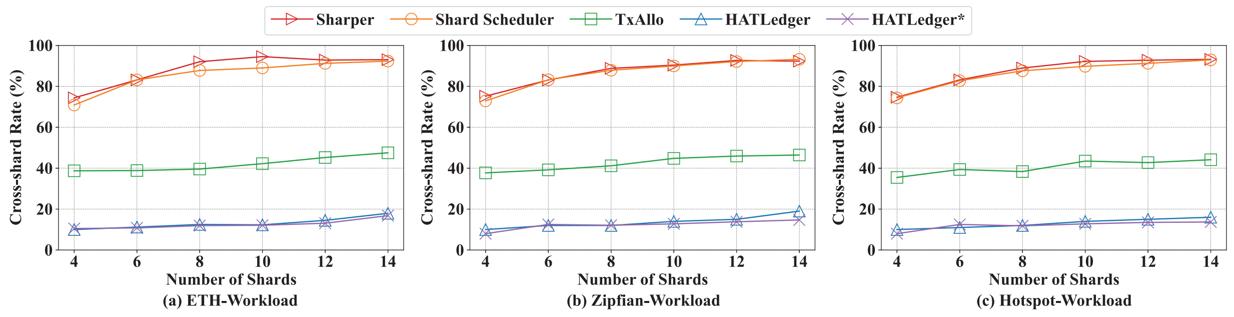

Figure 7: The cross-shard rate with different numbers of shards under (**a**) ETH-Workload, (**b**) Zipfian-Workload, and (**c**) Hotspot-Workload

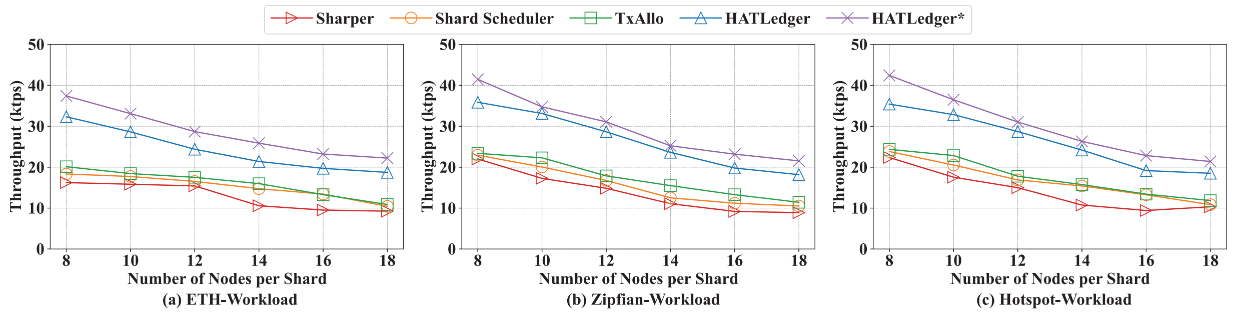

Figure 8: The throughput with different numbers of nodes per shard under (**a**) ETH-Workload, (**b**) Zipfian-Workload, and (**c**) Hotspot-Workload

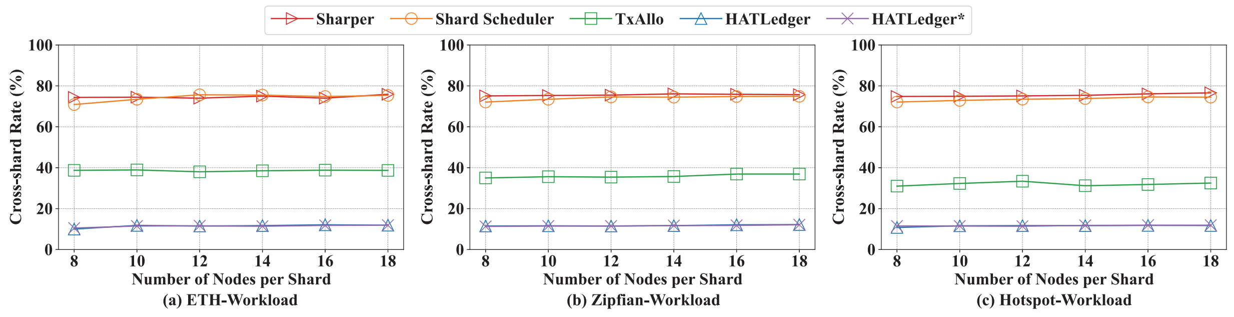

Figure 9: The cross-shard rate with different numbers of nodes per shard under (**a**) ETH-Workload, (**b**) Zipfian-Workload, and (**c**) Hotspot-Workload

5.3 The Impact of Different Numbers of Shards

First, we investigate the impact of varying the number of shards on system performance. As shown in Table 3, we fix the number of nodes per shard at 8 and incrementally increase the number of shards from 4 to 14. Under the Eth workload, the throughput of HATLedger* increases from 37,398 to 83,927 tps as the number of shards grows, while HATLedger's throughput rises from 32,317 to 72,196 tps. In contrast, other systems exhibit a smaller improvement, with throughput increasing from 20,000 to 50,000 tps. Moreover, the cross-shard transaction rates of HATLedger and HATLedger* remain around 10%, significantly lower than those of TxAllo (40%) and other systems (80%). These results demonstrate that both HATLedger and HATLedger* enhance performance by accurately identifying upcoming hotspot accounts through future transaction graph analysis, thereby reducing the cross-shard rate. Additionally, HATLedger* outperforms

HATLedger due to its execution-consensus decoupled transaction partitioning strategy. As the number of shards increases, the probability that transaction-accessed accounts are distributed across different shards also rises. The results in Fig. 7 validate this observation. As the number of shards increases, the cross-shard transaction rates of SharPer and Shard Scheduler sharply rise from 70% to 90%. This trend occurs because SharPer ignores account partitioning, while Shard Scheduler allocates transactions to currently underloaded shards—achieving load balance but sacrificing cross-shard rate. In comparison, TxAllo's cross-shard rate increases more gradually from 40% to 50%, as it detects hotspot accounts using historical transaction graphs and groups them into the same shard to reduce cross-shard transactions. However, hotspot detection based on historical graphs exhibits delayed adaptation. In contrast, HATLedger and HATLedger* exploit the future transaction graph to more precisely identify forthcoming hotspot accounts, thereby maintaining a stable cross-shard rate between 10% and 20%.

All systems show slightly better performance under the Zipfian and Hotspot workloads than under the Eth workload. The Eth workload, being more random in hotspot distribution and transactions, leads to a small performance drop in all systems.

5.4 The Impact of Different Numbers of Nodes per Shard

As shown in Table 3, we fixed the system at four shards and gradually increased the number of nodes per shard from 8 to 18. As shown in Fig. 9, when the number of shards remains constant but the number of nodes per shard increases, all systems exhibit noticeable performance degradation. The throughput of HATLedger* declines from 37,398 to 22,173 tps, while HATLedger drops from 32,317 to 18,721 tps. SharPer's throughput decreases from 16,929 to 9251 tps. During this process, the cross-shard transaction rates of all systems remain stable. This result indicates that increasing the node number per shard only increases the consensus cost without affecting the cross-shard rate. Moreover, since cross-shard consensus incurs higher overhead than intra-shard consensus, SharPer—with the highest cross-shard rate—suffers the most significant performance decline. In contrast, HATLedger* effectively alleviates shard overload through transaction partitioning based on Consensus–Execution decoupling, achieving the most stable performance among all systems.

5.5 The Workload Distribution of Different Systems

Each shard consists of 8 nodes, and we evaluated load distribution across different systems in a setup with 8 shards. As shown in Fig. 10c,d, both TxAllo and HATLedger partition hotspot accounts to the same shard (Shard 1), which reduces cross-shard transaction rate but leads to workload imbalance, as Shard 1 becomes significantly overloaded. In contrast, as shown in Fig. 10e, HATLedger* achieves a more balanced workload distribution across all shards. This improvement is attributed to the consensus–Execution decoupled transaction partitioning mechanism, which redistributes pending transactions from Shard 1 to other relatively idle shards. As a result, the load across shards in HATLedger* remains relatively balanced.

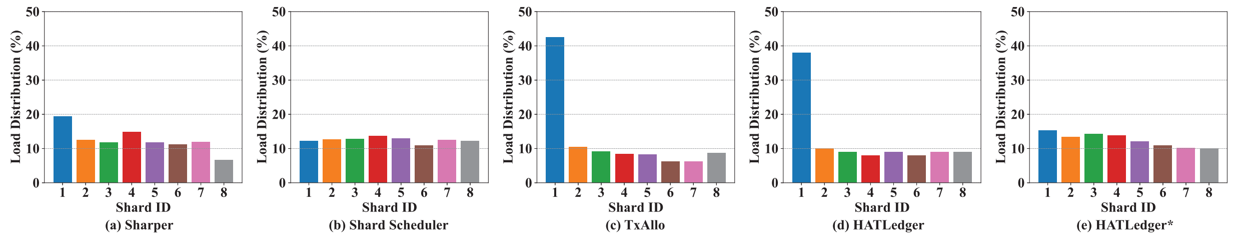

Figure 10: The workload distribution of each shard in (**a**) Sharper, (**b**) Shard scheduler, (**c**) TxAllo, (**d**) HATLedger, and (**e**) HATLedger*

5.6 The Performance under Overloads

We simulated an overload DDoS attack by increasing the client transaction sending rate from 10,000 to 100,000 tps. In Fig. 11a, the throughput of HATLedger* and HATLedger initially increases with the growth of the sending rate and eventually stabilizes at 37,398 and 32,317 tps, respectively. When the sending rate is below 30,000 tps, HATLedger* and HATLedger exhibit similar performance because both systems have sufficient capacity to process all transactions. When the sending rate exceeds 30,000 tps, the performance of HATLedger approaches saturation. HATLedger* reach saturation when the sending rate exceeds 40,000 tps. This demonstrates that HATLedger* handles sudden overloads more effectively through execution-consensus decoupled transaction partitioning mechanism, thereby providing better resilience against DDoS attacks.

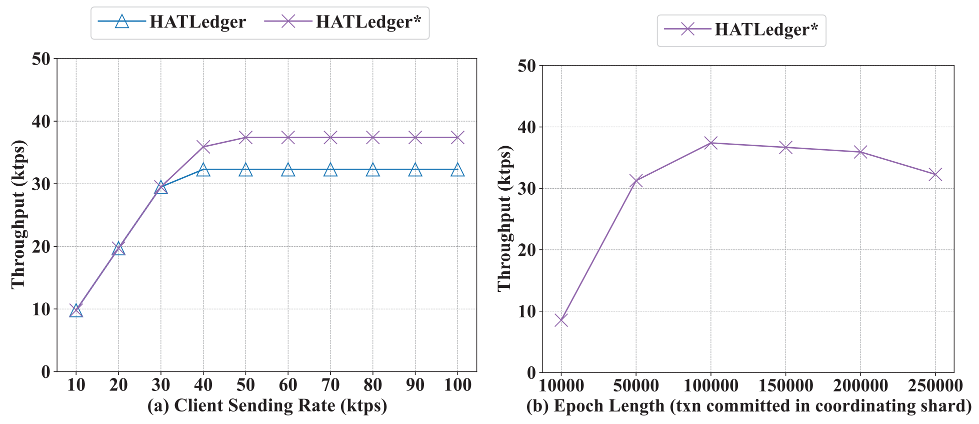

Figure 11: The performance under different scenarios: (**a**) The performance under different client sending rates; (**b**) The performance under different epoch lengths

5.7 The Performance under Different Epoch Lengths

We adopt a transaction-count-based epoch configuration, in which the epoch length is controlled by the number of transactions committed in the coordinating shard ($Shard_j = Ep_k \% Shard_{num}$). As shown in Fig. 11b, the throughput of HATLedger* rises sharply at first and then declines slowly. When the epoch length is 100,000, HATLedger* reaches its optimal performance (37,398 tps). Specifically, the coordinating shard $Shard_j$ transitions to the next epoch after committing 100,000 transactions in $Shard_j$. A shorter epoch length results in frequent account and transaction repartitioning, consuming system resources and interfering with normal transaction execution, thereby reducing throughput. Conversely, an excessively long epoch length lowers the repartitioning frequency. However, as hotspot accounts evolve over time, the increased cross-shard transaction rate ultimately leads to performance degradation.

6 Related Work

Liu et al. [18] provided a comprehensive review of existing consensus protocols and proposed the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) to systematically assess their respective advantages and limitations. Luo et al. [19] proposed the Symbiotic Blockchain Consensus (SBC), an energy-efficient sharding mechanism designed for wireless networks to satisfy the low-power demands of 6G systems. Xiong et al. [20] proposed a group-based approach to improve the efficiency of blockchains. Chien et al. [21] proposed that making effective predictions is very important for resource allocation. A more advanced step was made by OmniLedger [7], which introduced state sharding so that each shard only manages a portion of the global ledger. RapidChain [22] was developed to substantially mitigate high reconfiguration costs. SharPer [4] targets scalability by dynamically redistributing shards across network clusters. Shard Scheduler [8] proposes to partition transactions and their associated accounts to low-load shards based on current load distribution, achieving better load balancing but still suffering from high cross-shard transaction rates. In contrast, TxAllo [9] constructs a hotspot account graph based on historical transactions and partitions hotspot accounts to the same shard, thereby reducing high cross-shard transaction rates but leading to load imbalance.

7 Conclusion

HATLedger is an efficient sharded blockchain ledger built on **H**ybrid **A**ccount and **T**ransaction partitioning. Leveraging the future transaction graph, HATLedger identifies upcoming hotspot accounts and effectively reduces the cross-shard transaction rate. To address potential shard overloading, HATLedger supports transaction partitioning based on Consensus-Execution decoupling, eliminating the need for waiting between source and target shards. Experimental results demonstrate that HATLedger achieves up to a 2.2× improvement in throughput compared to existing sharded blockchains.

Acknowledgement: We sincerely thank Beijing Institute of Technology for providing support for the completion of this research.

Funding Statement: This research was funded by the National Key Research and Development Program of China (Grant No. 2024YFE0209000), the NSFC(Grant No. U23B2019).

Author Contributions: Shuai Zhao: Investigation, Methodology, Software, Validation, Visualization, Writing—original draft; Zhiwei Zhang: Formal analysis, Funding acquisition, Resources, Writing—review & editing; Junkai Wang: Data curation, Formal analysis, Writing—review & editing; Ye Yuan: Funding acquisition, Project administration; Guoren Wang: Funding acquisition, Project administration, Resources. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are openly available in the transaction history of Ethereum with block heights of 13 to 13.2M (Aug.–Oct. 2021) at https://ethereum.org (accessed on 12 October 2025).

Ethics Approval: Not applicable.

Conflicts of Interest: Prof. Guoren Wang, one of the co-authors, currently holds the position of Editor-in-Chief at Computers, Materials & Continua. The submission was made via the journal's regular online system. In line with the author guidelines, manuscripts involving an editor of the journal as a co-author are overseen by a different editor who has minimal potential conflicts of interest, thereby ensuring an impartial review process. Furthermore, since this manuscript is submitted to a special issue, we confirm that no conflicts of interest exist with any of the guest editors associated with this issue.

References

1. Bitcoin Project [Internet]. 2024 [cited 2025 Aug 26]. Available from: http://bitcoin.org.
2. Ethereum Foundation [Internet]. 2025 [cited 2025 Aug 26]. Available from: https://ethereum.org.
3. Hyperledger Project [Internet]. 2024 [cited 2025 Aug 26]. Available from: http://github.com/hyperledger/fabric.
4. Amiri MJ, Agrawal D, Abbadi AE. SharPer: sharding permissioned blockchains over network clusters. In: Li G, Li Z, Idreos S, Srivastava D, editors. SIGMOD '21: international conference on management of data. New York, NY, USA: ACM; 2021. p. 76–88. doi:10.1145/3448016.3452807.
5. Hu D, Wang J, Liu X, Li Q, Li K. LMChain: an efficient load-migratable beacon-based sharding blockchain system. IEEE Trans Comput. 2024;73(9):2178–91. doi:10.1109/TC.2024.3404057.

6. Huang H, Peng X, Zhan J, Zhang S, Lin Y, Zheng Z, et al. Brokerchain: a cross-shard blockchain protocol for account/balance-based state sharding. In: IEEE INFOCOM 2022-IEEE conference on computer communications. Piscataway, NJ, USA: IEEE; 2022. p. 1968–77. doi:10.1109/INFOCOM48880.2022.9796859.
7. Kokoris-Kogias E, Jovanovic P, Gasser L, Gailly N, Syta E, Ford B. Omniledger: a secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE symposium on security and privacy (SP). Piscataway, NJ, USA: IEEE; 2018. p. 583–98. doi:10.1109/SP.2018.000-5.
8. Król M, Ascigil O, Rene S, Sonnino A, Al-Bassam M, Rivière E. Shard scheduler: object placement and migration in sharded account-based blockchains. In: Proceedings of the 3rd ACM Conference on Advances in Financial Technologies; 2021 Sep 26–28; Arlington, VA, USA. New York, NY, USA: ACM. p. 43–56. doi:10.1145/3479722.3480989.
9. Zhang Y, Pan S, Yu J. Txallo: dynamic transaction allocation in sharded blockchain systems. In: 2023 IEEE 39th international conference on data engineering (ICDE). Piscataway, NJ, USA: IEEE; 2023. p. 721–33. doi:10.1109/ICDE55515.2023.00390.
10. Li M, Wang W, Zhang J. LB-Chain: load-balanced and low-latency blockchain sharding via account migration. IEEE Trans Parallel Distrib Syst. 2023;34(10):2797–810. doi:10.1109/TPDS.2023.3238343.
11. Ruan P, Dinh TTA, Loghin D, Zhang M, Chen G, Lin Q, et al. Blockchains vs. distributed databases: dichotomy and fusion. In: Proceedings of the 2021 international conference on management of data. New York, NY, USA: ACM; 2021. p. 1504–17. doi:10.1145/3448016.3452789.
12. Castro M, Liskov B. Practical byzantine fault tolerance. In: OSDI ’99: proceedings of the third symposium on operating systems design and implementation. Berkeley, CA, USA: USENIX Association; 1999. p. 173–86.
13. Cheng F, Xiao J, Liu C, Zhang S, Zhou Y, Li B, et al. Shardag: scaling dag-based blockchains via adaptive sharding. In: 2024 IEEE 40th international conference on data engineering (ICDE). Piscataway, NJ, USA: IEEE; 2024. p. 2068–81. doi:10.1109/ICDE60146.2024.00165.
14. The ZILLIQA Team. The ZILLIQA Technical Whitepaper [Internet]. 2017 [cited 2025 Aug 26]. Available from: https://docs.zilliqa.com/whitepaper.pdf.
15. Dwork C, Lynch N, Stockmeyer L. Consensus in the presence of partial synchrony. J ACM. 1988;35(2):288–323. doi:10.1145/42282.42283.
16. Karypis G, Kumar V. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-reducing Orderings of Sparse Matrices. Technical Report. Minneapolis, MN, USA: University of Minnesota. 1997 [cited 2025 Aug 26]. No. 97-061. Available from: https://hdl.handle.net/11299/215346.
17. InPlusLab. Ethereum BlockChain Datasets [Internet]. 2025 [cited 2025 Aug 26]. Available from: https://xblock.pro/#/.
18. Liu J, Liu C, Lin M, Xu G. Comprehensive survey of blockchain consensus mechanisms: analysis, applications, and future trends. Comput Netw. 2025;272:111661. doi:10.1016/j.comnet.2025.111661.
19. Luo H, Sun G, Chi C, Yu H, Guizani M. Convergence of symbiotic communications and blockchain for sustainable and trustworthy 6G wireless networks. IEEE Wireless Commun. 2025;32(2):18–25. doi:10.1109/mwc.001.2400245.
20. Xiong H, Jin C, Alazab M, Yeh KH, Wang H, Gadekallu TR, et al. On the design of blockchain-based ECDSA with fault-tolerant batch verification protocol for blockchain-enabled IoMT. IEEE J Biomed Health Inform. 2021;26(5):1977–86. doi:10.1109/JBHI.2021.3112693.
21. Chien WC, Lai CF, Chao HC. Dynamic resource prediction and allocation in C-RAN with edge artificial intelligence. IEEE Trans Ind Inform. 2019;15(7):4306–14. doi:10.1109/TII.2019.2913169.
22. Zamani M, Movahedi M, Raykova M. Rapidchain: scaling blockchain via full sharding. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. New York, NY, USA: ACM; 2018. p. 931–48. doi:10.1145/3243734.3243853.