Log Replaying for Real-Time HTAP: An Adaptive Epoch-based Two-Stage Framework

Jun-Peng Zhu†, Zhiwei Ye‡, Peng Cai†*, Donghui Wang†, Fengyan Zhang†, Dunbo Cai‡, Ling Qian‡
†*East China Normal University,* ‡*China Mobile (Suzhou) Software Technology Co., Ltd*
zjp.dase@stu.ecnu.edu.cn, yezhiwei@cmss.chinamobile.com, pcai@dase.ecnu.edu.cn
{donghuiwang, zhangfengyan}@stu.ecnu.edu.cn, {caidunbo, qianling}@cmss.chinamobile.com

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00167

***Abstract*—As real-time analytics become increasingly important, more organizations are deploying Hybrid Transactional/Analytical Processing (HTAP) systems. The HTAP systems, based on a primary/backup replication architecture, usually support real-time read-only queries on backup nodes for the data recently generated by OLTP applications on the primary node. This work is based on the observation that real-time analytical applications often require access to only a fraction of the latest modifications from OLTP applications. However, the state-of-the-art parallel log replay approaches treat all replicated transaction logs equally and replay the entire transaction logs with the same priority which does not take consideration into the OLAP query access pattern. This design can result in increased response latency for real-time applications.**

This paper presents AETS, an Adaptive Epoch-based Two-Stage log replay framework that implements epoch-based log replay and table group transaction commit. Simultaneously, AETS also takes full account of the table access priority in real-time HTAP workload log replay. It aims to make the data required by analytical queries visible more quickly. Furthermore, AETS includes a two-phase parallel log replay algorithm called TPLR, which achieves lower overhead compared to state-of-the-art algorithms through careful design. We also offer an adaptive fine-grained thread resource allocation method that considers changes in table access patterns over time under thread resource constraints. Our experimental results show that AETS significantly reduces visibility delay for real-time queries. And the results also show that AETS achieves significant replay throughput improvement.

***Index Terms*—HTAP, replication, parallel log replay**

I. Introduction

In recent years, hybrid transaction/analytical processing (in short, HTAP) become ubiquitous (e.g. SQL server [1], SAP HANA [2], and Greenplum [3]). These systems typically contain two types of nodes: the primary and backup (replica or secondary) database nodes. The primary database node handles read-write transactions and replicates committed transaction logs to several backup database nodes. After that, backup nodes refresh their status by parallel replaying replicated logs [2], [4], [5], [6] and provide services for analytical queries. Applications relying on real-time analytics, such as fraud detection, system monitoring, and online commerce, heavily depend on fresher data. Meanwhile, real-time analysis often requires access to only a subset of the most recent transaction data generated by OLTP applications. The expeditious replay of the replication logs by the backup node to provide the most recent data to the user has emerged as a prominent research area in modern HTAP database systems.

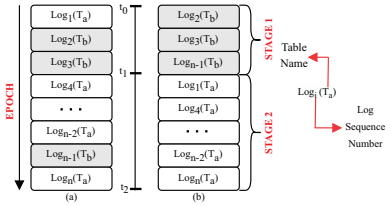


Fig. 1. A Motivating Example of Two-Stage Log Replaying. In backup nodes, T_a is a cold table, and T_b is a hot table as many queries access table T_b. Figure 1 (a) and (b) conceptually present the difference in replaying the replicated transaction logs between previous parallel log replay methods and our AETS. (a) Previous parallel log replay methods apply these logs without considering the characteristics of analytical workloads. (b) Our AETS. We divide the replicated log stream into epochs. In each epoch, AETS has priority over T_b by replaying its logs in the first stage and then replaying the logs for cold tables in the second stage. Thus, fresh data in replicated logs of T_b can be more quickly visible.

Limitations of Prior Art. As far as we know, state-of-the-art parallel log replay approaches have their limitations:

(1) **Lack of OLAP access patterns**. The state-of-the-art parallel log replay approaches [2], [4], [6], [7], [8], [9], [10], [11] do not take into account the query access pattern of the OLAP side in the backup node. Transaction workloads and real-time analytic workloads may come from different applications/businesses. Although high-throughput transaction applications generate a large amount of data, analytic queries may only require *a fraction of latest data* (c.f., Section II). Previous approaches treat all replication log entries equally and replay the entire replication log with the same priority. It can result in increased query latency. In Figure 1, we present a motivating example. There are total n log entries replicated to the backup node. Long transactions on table T_a in the primary node have generated lots of replicated log entries (i.e. $n-3$ log entries) than short transactions on table T_b (i.e. 3 log entries). The short transactions are committed after these long transactions and some of their logs have larger log sequence numbers. Figure 1(a) illustrates the hot data in T_b can be visible at t_2 after all logs have been replayed by existing parallel log replaying methods. However, better visibility should be achieved in Figure 1(b) by taking priority on replaying logs in T_b, where the fresh data of T_b can be visible at t_1.

(2) **Lack of adaptive threads resource allocation**. When

* Corresponding Author

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00167

allocating thread resources for log replaying, existing log replay approaches [4], [6], [7], [10] consider how to evenly distribute un-replayed logs over worker threads. However, in the context of real-time analytics, the table access rate emerges as a pivotal determinant. A table with a higher access rate indicates lots of queries needed to access the latest data on the hot table within a specified temporal interval. In the worst case, some time-critical tasks would be timed out if the fresh data can not be ready by quick log replaying, which is unacceptable for real-time applications. As a result, it becomes imperative to allocate more thread resources to replay logs for the table with a higher access rate. Furthermore, OLAP workloads continuously change over time, resulting in varied changes in the table access rate. We need to consider designing adaptive thread allocation strategies to cope with changes in the OLAP table access rate over time.

(3) **Lack of lower dependency analysis overhead**. The existing parallel log replay algorithms utilize two log formats: command log and value log. For algorithms using the command log, the re-execution of transactions on the backup node must follow the same order as that of the primary node. This requirement directly leads to the necessity for such algorithms to rely on complex dependency analysis, transaction, and lock management. Thus, AETS uses the value log that is the same as SiloR [12]. However, parallel log replay algorithms using value log (e.g., Kuafu [4] and ATR [6]) still incorporate dependency analysis. In Kuafu, dependency tracking needs to be guaranteed using topology graph sort. The state-of-the-art SRBR [7] based on the command log reduces dependency analysis compared to Kuafu, but it does not completely eliminate the cost due to waiting on read dependency. The ATR eliminates heavy dependency tracking compared with the above methods. However, ATR still requires a lightweight check to ensure the modifications on the same record are applied according to the modification order in the primary node. C5 [11] employs a row-based dispatch method, sending data to dedicated queues in transaction order. Each queue is serviced by a single thread responsible for log replaying. Then, a single commit thread is utilized to ensure the transaction visibility order is consistent with the primary. Indeed, it is crucial to note that none of these algorithms can support parallel log replaying based on priority.

Key Technical Challenges. As mentioned above, we summarize the challenges in the design of parallel log replaying approach for real-time analytical applications:

- The first technical challenge is how to consider workload patterns in real-time analytical applications on backup nodes when replaying the committed transaction logs replicated from OLTP applications on the primary node.
- The second technical challenge lies in designing an adaptive fine-grained thread resources allocation approach, which considers the dynamic changes in the table access rate over time.
- The third technical challenge is how to design a new parallel log replay algorithm with lower overhead than state-of-the-art that both fully capitalize on the MVCC feature of modern database systems and enable parallel log replaying based on priority.

Key Ideas. The key insight is that only a portion of the data modified by OLTP transactions will be accessed by real-time analytical applications. Replaying all replication logs strictly according to their generated order in the primary node can incur substantial latency overheads in the case that real-time analytical applications only access a fraction of the latest data. To resolve this problem, the log replay framework needs to prioritize allocating resources to replaying those replicated OLTP logs, which contain the data required by real-time queries. However, a decrement in priority does not introduce an augmentation in log replay latency for the data that has not yet been accessed by real-time analytics.

Proposed Approach. In this paper, we present AETS, an **A**daptive **E**poch-based **T**wo-**S**tage log replay framework. To overcome the first challenge, AETS takes full account of the table access priority in parallel log replay, which is targeted towards real-time HTAP workloads with an eager requirement of data freshness. Tables are categorized into the first-class table group (i.e. hot tables) and the second-class table group (i.e. cold tables) according to the query access pattern in OLAP workloads. As the workload patterns are varying with time (e.g. cold tables change to hot, and vice visa), AETS splits the replicated log stream into different epochs. Before starting to replay logs for each epoch, AETS detects the changes in workload pattern to build the new first-class and second-class table groups if necessary. In each epoch, the replicated logs which modify the table in the first-class table group are replayed by AETS in the first stage. After replaying all hot table logs, the logs for the second-class table group are replayed in the second stage.

The goal of AETS is to decrease the latency of real-time analytic queries by eliminating the influence of log replaying over cold tables. AETS also allows the table group to be partitionable to provide adaptive fine-grained thread resource allocation, which takes into account both the size of un-replayed logs and the table access rate of real-time analytics. Our experimental results show that AETS significantly reduces response time for real-time queries.

Key Contributions. To summarize, this paper makes the following contributions:

- We observe that the data needed by real-time analytical queries usually make up a portion of the changes updated by OLTP transactions in real-time HTAP workloads. In the traditional parallel log replay approaches, the idea of treating all replicated log entries equally needs to be reconsidered. We demonstrate that this workload characteristic has a significant impact on the response latency of real-time analytics queries on the freshest data.
- We propose an adaptive epoch-based two-stage log replay framework called AETS, which implements epoch-based log replay and table group transaction commit. The visibility of fresh data in hot tables should be independent of the visibility of fresh data in cold tables.

TABLE I
OLAP QUERIES ACCESS A FRACTION OF TABLES WRITTEN BY OLTP.

Benchmark	$num(\mathcal{T})$	$num(\mathcal{A})$	$num(\mathcal{A} \cap \mathcal{T})$	$ratio$
TPC-C [13]	8	5	5	90.98%
SEATS [14]	4	8	2	38.08%
CH-benCHmark [15]	8	Q1 1	1	60.83%
		Q2 5	1	18.79%
		Q3 4	4	74.93%
		Q4 2	2	66.91%
		Q5 7	4	90.79%
		Q6 1	1	60.83%
BusTracker [16]	65	14	14	37.12%

- We present a two-phase parallel log replay algorithm called TPLR, which has a lower overhead than state-of-the-art algorithms. By employing a two-phase parallel log replay design, TPLR combines the guarantee of operation sequence on the same records and transaction committed order into one step in the second phase.
- We provide an adaptive fine-grained thread resource allocation method that considers table access pattern change over time under thread resource constraints.
- Extensive experiments show that AETS significantly reduces response time for real-time queries and improves parallel log replay performance.

II. MOTIVATION

The recently published HTAP systems [17], [18], [19] claimed they can achieve data freshness with less than one second delay. Especially, real-time analytical user-facing applications can benefit from high data freshness. The BusTracker, for example, is an HTAP workload extracted from a real application in which analytical queries are frequently issued by a machine learning task to predict the waiting time for a bus in real time. The prediction tasks usually contain small queries over fresh data for the current position of buses as the input of model inference. To achieve the best prediction accuracy, which requires maximum data freshness, even a small degree of data staleness can lead to significant prediction bias. Such a requirement is common in modern applications, particularly those that contain real-time online analytical tasks.

Typically, real-time analytic queries that require the most recent data will be routed to the primary database, putting additional strain on the primary database, however, backup nodes are frequently underutilized. Analytical queries are frequently routed to the backup database nodes to reduce the impact on the performance of the primary database in modern HTAP databases. Notably, mission-critical OLTP and real-time analytics often come from different businesses. And real-time analytical queries may only access a fraction of the modifications made by the OLTP transactions. We illustrate this HTAP characteristic by investigating several benchmarks in Table I. Considering the OLAP queries $\mathcal{A}$ and the produced log entries $\mathcal{T}$ by OLTP, we denote $num(\mathcal{A})$ as the total number of tables accessed by the OLAP queries. In TPC-C, we regard the read-only transactions such as StockLevel and OrderStatus as logical analytical queries; $num(\mathcal{T})$ as the total number of tables written by OLTP; $num(\mathcal{A} \cap \mathcal{T})$ as the number of the intersection of the tables in $\mathcal{A}$ and $\mathcal{T}$. The column *ratio* in Table I presents the ratio of log entries that need to be replayed for the OLAP workload. We define the ratio as the proportion of hot table log entries out of the total replication log entries. For the various benchmarks (excluding BusTracker in Section VI), we set the scale factor (SF) to 20 and maintained the default transaction mixed configuration. Then, we executed these transactions and generated the replication logs. Finally, we calculate the number of log entries in the hot tables divided by the total number of log entries to estimate the ratio. A larger ratio indicates that the number of log entries in the hot tables is higher, and more work is needed to replay the hot table log entries in the first stage. When analytical queries require only a portion of the data, they may have a long response time due to the lack of prioritization in replaying the necessary log entries for obtaining fresh data. For real-time analytics, our intuition is that there are still plenty of opportunities to reduce the visibility gap as much as possible of the required fresh data by prioritizing the replay of their OLTP logs.

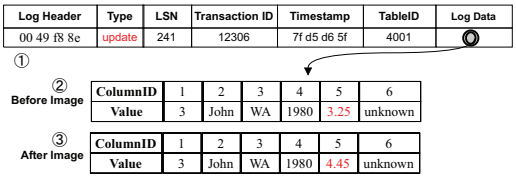


Fig. 2. The example of log format for update in AETS, which includes metadata ① and modified values ② or ③.

III. DESIGN PRINCIPLES AND ARCHITECTURE

In this section, we first introduce the database model and the logging utilized in this work. Next, we present the architecture of AETS, as depicted in Figure 3. Then, we present the method that batches transactions of the primary node into epochs. Finally, the execution workflow of AETS is introduced.

A. Database Model and Logging

We assume that the backup database system with AETS is a kind of modern main-memory database system with a high-performance main-memory storage engine (called Memtable) and multi-version concurrency control (namely MVCC). Each record in the Memtable has a version chain to link different versions committed recently. The transaction ID is monotonically increased and represents the committed order of each transaction in the primary node.

The primary node handles read-write transactions, while the backup node handles read-only transactions and analytical queries. Transaction logs generated by the primary node are replicated to the backup node, which refreshes its status by replaying the replicated logs to its Memtable. To book-keep transaction logs, the database with AETS uses a value log that is the same as SiloR [12]. Figure 2 depicts the log format of *update* operation in AETS. Each replication log entry has the following common fields except the log header:

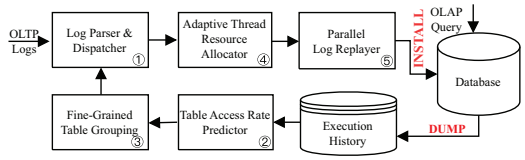


Fig. 3. Overall architecture of AETS.

- **Log Type**. The type of a log entry indicates whether it is a DML log entry or a transaction log entry with mark *BEGIN* transaction or *COMMIT* transaction. There are three type row operations, called *insert*, *update*, and *delete* respectively.
- **LSN**. The log sequence number is the unique and sequential identifier of a log entry.
- **Transaction ID**. The order of transaction ID represents the transaction commit order in the primary database.
- **Timestamp**. The create time of the log entry.
- **Table ID**. If it is a DML log entry, Table ID represents the identifier of the database table to which the DML operation is applied.
- **Log Data**. Concatenation of the pairs of column IDs and their new values.

It should be noted that we do not make any assumptions about the isolation levels employed by the primary. AETS is designed to make the modifications in the replicated log entries to be visible on the backup according to their committing order in the primary.

B. Batch Transactions into Epochs

Epoch-based techniques have gained significant popularity in transaction database systems, as evidenced by numerous studies [7], [12], [20], [21], [22], [23], [24]. These techniques involve grouping multiple transactions into one epoch, which is advanced based on various criteria such as the amount of allocated memory, the number of transactions, or fixed-length and non-overlapping time intervals. In this paper, we adopt epochs as the granularity for replaying replication logs. The replication log entries are partitioned into a set of fixed-sized and non-overlapping epochs. Specifically, the size of the epoch is determined by the number of the transactions in which a group of transactions have been committed in the primary node and are ready to be replicated to the backup node. Epochs are segmented on a transaction boundary which means log entries of a committed transaction do not belong to different epochs. These transactions are sent to the backup node in the order they are committed, ensuring a consistent replay order. Consequently, during the replay process, AETS strictly follows the order of epochs, ensuring that subsequent epochs are not processed until the replay of the previous epoch is complete.

C. The Overall Architecture of AETS

The overall architecture of AETS is shown in Figure 3. Next, we detail the core components of the architecture.

Log Parser and Dispatcher ①. The log parser simply parses all incoming OLTP log entries only. Its primary responsibility is to split all log entries of transactions and distribute them into different replay groups which take fully into consideration that tables are accessed in real-time OLAP query. Specifically, log entries belonging to the same transaction are bounded by the terms *BEGIN* and *COMMIT*. Thus, the log dispatcher finds the boundaries of a transaction by parsing the type of each log entry. In addition, it parses the table ID that each log entry tries to modify and puts the log entry into the corresponding replay group. Log entries of a transaction that updates multiple tables are actually divided into several minor pieces due that they are scattered over several replay groups. Meanwhile, this component pushes the transaction ID into the $commit\_order\_queue$ of the corresponding table group. In the backup node, the modifications of a transaction on the table group are installed according to the order of $commit\_order\_queue$ which is the same as the primary node.

Table Access Rate Predictor ②. The response latency of real-time queries over one table group is related to two factors. One factor is the total size of received and yet un-replayed replication logs in this table group and another is the table access rate in this group. The first factor seems straightforward, and we discovered that the table access rate impacts how quickly real-time analytical queries are responded to. The main reason is that the table group with a high table access rate makes it easier for many queries to have longer response times due that a large number of queries need to wait for the required record version that was modified by the primary node. The later queries may be waiting for query processing threads to schedule them, especially when earlier arriving queries have not been responded to. To tackle this challenge, we design the table access rate predictor component. Simultaneously, AETS integrates the predicted table access rate into the thread resource allocation scheme. In Section IV-A, we provide the details about this component.

Fine-Grained Table Grouping ③ and Adaptive Thread Resource Allocator ④. We may not directly assign log replay threads to each table because the number of replay threads is usually limited in order to keep enough thread resources for query processing. Furthermore, AETS aims to unrelated transaction modification operations to be committed in parallel for different table groups. For the aforementioned reasons, the AETS component incorporates a fine-grained table grouping that categorizes tables with similar access rates into the same group. AETS begins by employing a temporal graph neural network to forecast the future table access rates, leveraging the historical table access rate of each table. Subsequently, the clustering algorithm (e.g., DBSCAN) is applied to group tables with similar access rates into clusters, or each table is assigned to its own group when the number of tables is small. Furthermore, considering that the un-replayed log size and table access rate change over time, AETS establish an adaptive thread resource allocation scheme. In Section IV, we present comprehensive details of those components.

Parallel Log Replayer ⑤. Each table group is given thread resources for parallel log replaying. To further shorten the query latency of the real-time analytics requests in the backup node and increase the level of data freshness. In this paper, we

propose a two-phase parallel log replay algorithm with fewer consistency constraint checks (i.e. to ensure the same operation sequence on a record) than state-of-the-art algorithms. In the first phase, log entries are translated into several uncommitted cells parallel without considering any transaction dependency. In the second phase, AETS makes uncommitted cells committed in the order in which transactions are committed on the primary node. We give the details in Section V.

D. AETS Execution Workflow

In this section, we introduce how AETS processes incoming log entries, performs parallel log replaying on tables, and finally responds to real-time OLAP queries. AETS marks the log entries that need to be replayed and the real-time queries that need to be responded to at the beginning of each epoch. Firstly, AETS performs grouping tables which will be accessed by real-time OLAP queries. Then, considering both the predicted table access rate and the amount of un-replayed logs, AETS allocates thread resources adaptively. The *publish time* for each table group is set by AETS to make changes individually visible. For real-time analytical queries, it can be visible to the record version that had a committed timestamp prior to the *publish time* of its table group. When an analytical query comes at time qts, AETS judges which group it attempts to read. It often needs to wait until the required data has been ready (i.e. *publish time* $\geq qts$) for handling the query. The response time of a query is measured by calculating the time interval between the time of starting to handing the query whose required data version is visible and the query arrival time qts.

Figure 4 presents an example to illustrate the log replay workflow of the AETS framework. In an epoch, the backup node received replication log entries with transaction IDs ranging from 1 to 9. AETS divided log entries according to whether they belong to hot tables or not. These logs for hot table are replayed in the first stage and logs for cold tables are replayed in the second stage. In each stage, log records are distributed to the corresponding *task queue* of a table group if these records modify a table in this group. The table groups have different table access rates. Transaction IDs of these log records are stored in the *commit_order_queue*, which determines the order of being visible. Log entries in the task queue are replayed in parallel by replay worker threads detailed in Section V. After the log replaying of a transaction in the table group is completed, its transaction ID is placed in the *waiting_commit_list* to be committed in the *commit_order_queue* according to the ID order.

IV. Adaptive Threads Resource Allocation

In this section, we first present the details of the table access rate predictor component in Section IV-A. Then, an adaptive thread resource allocation method is introduced in Section IV-B.

A. Table Access Rate Predictor

1) Problem Formulation: From the execution history of OLAP queries, it is easy to obtain the accessed tables and the access timestamp of each query. For each table, we calculate the total number of queries over it in a time slot. It is worth noting that the table access rate will change over time. How to capture this table access pattern change in time is an important research content in this section. We model the demand as a temporal graph prediction problem. Essentially, the aim is to predict table access rates based on history. Additionally, there are two dimensions of information that need to be considered: (1) There are associations between tables, and these interrelated tables may be accessed together in an OLAP query. (2) The table access rate of each table changes dynamically with time.

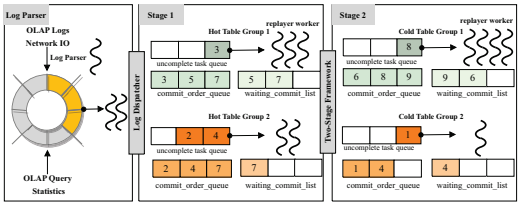


Fig. 4. An illustration execution workflow of AETS.

Many studies [25], [26] utilize recurrent neural networks and their variations to capture temporal dynamics. However, these methods often suffer from high computational costs, the training time is very long, and the limitations of recurrent neural networks in computing long sequences (e.g., gradient vanishing leads to short-term memory only). On the other hand, some other works [27], [28], [29] utilize convolution neural networks to encode sequence changes in the time dimension, which have relatively lower computational costs.

2) DTGM: **D**eep **T**emporal **G**raph **M**odel, namely DTGM, mainly consists of two modules, one is used to encode access relationships between different tables, and the other is used to encode changes in the table access rate history sequence in time. In our implementation, DTGM encodes access relationships between tables using a graph convolutional neural network (GCN) and encodes historical sequences of table access rates using a convolutional neural network (TCN).

We assume that the adjacency matrix corresponding to the original graph constructed based on the access relationships between tables is denoted as $\mathbf{C}$. The graph neural network utilized by the model is employed to capture the access relationships between tables. The specific calculation method is as follows:

$$\mathbf{Z} = \sum_{k=0}^{K} \mathbf{C^k H W_o}$$

where $\mathbf{W_0}$ is a learnable parameter, $\mathbf{H}$ is a learnable feature, and k is the power of the adjacency matrix.

For **T**emporal **C**onvolution **N**etwork, namely TCN, a gating mechanism based on temporal convolutional neural network is utilized. The specific calculation method is as follows:

$$\mathbf{Z} = tanh(\mathbf{\Theta_1} * \mathbf{H} + \mathbf{b_1}) \odot sigmoid(\mathbf{\Theta_2} * \mathbf{H} + \mathbf{b_2})$$

where $\mathbf{\Theta_1}$ and $\mathbf{\Theta_2}$ are learnable parameter, $\mathbf{b_1}$ and $\mathbf{b_2}$ are bias, $\odot$ is Hadamard product. In the overall model structure,

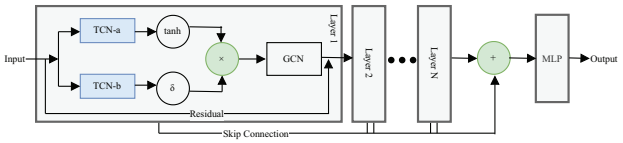


Fig. 5. Deep Temporal Graph Model. Each layer contains gated TCN and GCN.

the GCN is equivalent to adding a pooling layer after the temporal encoder. Both encoders stack multiple layers to form the final model structure. For better model performance, DTGM also incorporates residual connections and skip connections between each layer. Figure 5 shows the details of the model, which references Wu et al. [27].

3) Loss: In training the model, the average absolute error is used as the loss function to guide the optimization of the model parameters. In particular, for a given forecast result $\hat{\mathbf{Y}} \in \mathbb{R}^{T_f * N * D}$ and the corresponding actual data $\mathbf{Y} \in \mathbb{R}^{T_f * N * D}$, the loss function for prediction is computed as follows:

$$\mathcal{L}_{pred}(\hat{\mathbf{Y}}, \mathbf{Y}, \mathbf{\Theta}) = \frac{1}{T_f N D} \sum_{i=1}^{T_f} \sum_{j=1}^{N} \sum_{k=1}^{D} |\hat{\mathbf{Y}}_{i,j,k} - \mathbf{Y}_{i,j,k}|$$

where T_f is the predicted time window size, N is the number of tables, D is the dimension of output, Θ is model parameters.

AETS may also employs a clustering algorithm (e.g., DBSCAN) for table grouping in real-world applications, or each table is assigned to its own group. The output of table grouping subdivides the single table group into several groups where tables in each group have a similar table access rate.

The DTGM model can adapt to workload changes effectively, which helps overcome the limitation of calculating table access rates just based on historical data. This adaptability is also confirmed through experiments in Section VI-F. It is worth noting that training cost of the offline DTGM model is significantly lower. Retraining is only necessary if there are substantial changes in the business.

B. Dynamic Threads Resource Allocation

Given a limited total number of replaying threads, the number of replaying threads allocated to a table group is determined adaptively by two factors. The first factor is the total size of log entries routing to each group. The second factor is the degree of urgency for processing the analytical requests in this group as we discussed at the end of Section III. The number of replay threads in each replay group also needs to be varied when the table access rate changes. Therefore, we introduce an equation to acquire the number of replay threads for each group, i.e., t_{gi}:

$$\begin{cases} \lambda_{gi} * \dfrac{n_{gi}}{t_{gi}} = \lambda_{gj} * \dfrac{n_{gj}}{t_{gj}}, \text{for each group pair } (gi, gj) \in G \\ \displaystyle\sum_{i=1}^{|G|} t_{gi} = T, T \text{ is the total number of replay workers} \end{cases}$$

where n_{gi} is the total size of un-replayed log entries in the table group i and λ_{gi} is the urgency factor that is decided by the table access rate of OLAP requests on this table group i. In this paper, r denotes the table access rate of table group, and λ_{gi} is the $\log(r)$ that can guarantee numerical stability. This approach is adopted because the number of thread replays for different tables is directly proportional to the size of the log. Furthermore, in this paper, our goal is to ensure that tables heavily accessed by OLAP are replayed as quickly as possible. Therefore, the number of threads is also adjusted in proportion to the urgency factor. For instance, if the access rate of a table group is $1,000$ (i.e. tables in this group will be accessed by $1,000$ queries in the next moment) and λ_{gi} is denoted as r, this implies that the thread resources allocated to this table group are expanded by a factor of $1,000$. With our method, when $log(10^3)$ equals 3, the thread resources are increased by a factor of 3. This method offers two advantages: (1) it is more intuitive and easier to interpret, and (2) it ensures numerical stability. In Section VI-F, we conduct the experiment to illustrate the benefits of this method for reducing visibility delay. Note that with the increase of table access rate, λ_{gi} becomes larger and more resources are allocated for replaying logs in this group.

Algorithm 1: Commit a transaction.

```
1  Function CommitATxn(committing_txnid):
2      while true do
3          min_txnid = waiting_commit_list.min_txnid();
4          if min_txnid = committing_txnid then
5              remove the min_txnid from the
                waiting_commit_list;
6              break;
7          end
8      end
9      for cell_info in committing_txnid.context do
10         memnode = cell_info→node;
11         exclusive_lock(memnode);
12         memnode.value_list.append(cell_info);
13         exclusive_unlock(memnode);
14     end
15     tg_cmt_ts = committing_txnid.commit_ts;
```

V. Parallel Log Replay

This section introduces TPLR, a two-phase parallel log replay algorithm. Then, we present the visibility at backup in Section V-B.

A. TPLR: A Two-Phase Parallel Log Replay Algorithm

Different from transaction execution in the primary database, there are no rollback operations when replaying logs in backup replicas. That means all of the logs must be replayed and visible successfully in the same order as the primary database. The main problem addressed by previous works [4], [6], [7], [8] on parallel log replaying is to avoid the violation of

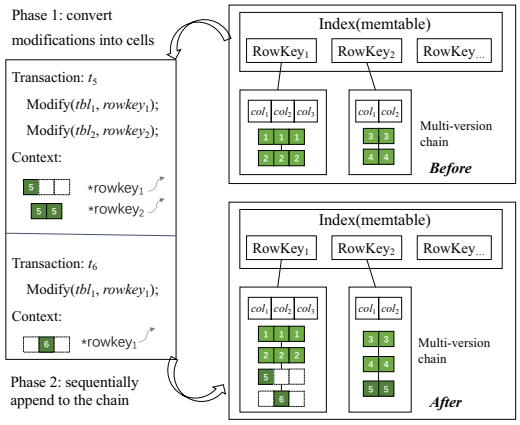


Fig. 6. Two-phase log replay in the multi-version Memtable, where each node of the Memtable has a transactionID-based version chain. The number on the green square (i.e. cell) is the ID of the transaction modifying this column.

transaction dependency in the replicated logs. If two log entries are from different transactions modifying the same record, they need to be committed and visible according to their order in the primary database.

However, parallel log replay schemes are too rigorous if the transaction dependency is not violated during these two phases. Actually, data consistency between the primary and backup databases can be guaranteed if the visibility order of replayed data on the backup is consistent with the committed order in the primary for real-time analytics. As a result, we designed a two-phase parallel log replaying approach called TPLR. The first phase is to parallel replay all log entries *without considering transaction dependencies*, and the second phase is called the commit phase to allow the replayed data to be committed.

In the first phase, the replay workers pop tasks from the task queue of a table group and then parse the data modifications. It looks up the Memtable to find the node that the transaction wants to modify. After that, logs are translated into several uncommitted cells, which are temporarily stored in the transaction context. Each cell has a pointer pointing to the actual node in the Memtable. Translating won't directly install modifications into the Memtable and therefore has no need to acquire any locks. Figure 6 presents an example of a two-phase log replaying for two transactions, where both of them write $rowkey_1$. The first phase which converts modifications into uncommitted cells is totally parallel without considering any transaction dependency. In the transaction context, we store the addresses of nodes in the Memtable and the converted cell information.

Algorithm 2 presents the main steps of the second phase. After log entries of a transaction in one table group have been replayed, its transaction ID will be pushed into the $waiting\_commit\_list$. In the second phase, the commit thread pops the first element from $commit\_order\_queue$ as $committing\_txnid$, and starts to commit the transaction. If the minimal transaction ID stayed in the *waiting_commit_list* equals $committing\_txnid$, the transaction

Algorithm 2: The commit phase.

```
1 Function Commit():
2     while !thread_.stop() do
3         committing_txnid =
            commit_order_queue.first();
4         commit_order_queue.pop();
5         CommitATxn(committing_txnid);
6     end
```

with $committing\_txnid$ can be committed safely as shown Algorithm 1. Their uncommitted cell information can be appended to the value list of a record in the Memtable. Unlike executing transactions on the primary database, the backup log replay process only involves installing the *new-version* values from the log entries to their respective addresses, without the need for the *old-version* of these values. As shown in Figure 6, in the first phase, transactions t_5 and t_6 keep the value of col_1 ($rowkey1$) and col_2 ($rowkey1$) in their transaction contexts, respectively. In the second phase, these changes are appended to the version-list of $rowkey1$ in accordance with the primary committing order. Locks are required in this part, but as we can see, the holding lock time is short. A lock is acquired to synchronize the read operations of the readers (line 11 in Algorithm 1).

AETS supports parallel commit between different table groups, distinguishing it as one of the features of AETS when compared to the state-of-the-art approaches. A single thread on each group is used to make the modifications to be committed according to the primary committing order using Algorithm 1. For read queries which may access several table groups on backup, AETS must ensure that the transaction visibility order matches that of the primary, and this is guaranteed by Algorithm 3 in Section V-B. AETS separates the process of parallel log replaying from transaction visibility to readers, allowing parallel log replaying algorithms to make the most of the available multi-thread resources. The decoupling also enables priority-based parallel log replaying.

B. Visibility at Backups

In the backup node, if a query needs to read the latest data version committed before the query arrival time, it first gets the latest snapshot timestamp value from the primary. Then, it requires the query to access the latest record version with the commit timestamp smaller than the snapshot timestamp [2], [6], [30]. If this condition is not met, the query needs to wait until its required fresh data are visible. The ultimate aim of AETS is to reduce the visibility gap between the primary and backup nodes and thus shorten the response time of OLAP queries without relaxing consistency restrictions. When a query starts, first of all, we need to locate the groups it tries to access, and then find out whether the data version satisfies the requirement of the visibility rule for this query. For each replay group, we maintain one additional

Algorithm 3: Visibility at backup.

```
Function Visibility(Q):
    qts = get_timestamp();
    gids = get_gid(Q);
    while true do
        min_tg_cmt_ts = minimum(g[gid].tg_cmt_ts,
          gid ∈ gids);
        if min_tg_cmt_ts ≥ qts or global_cmt_ts ≥ qts
          then
            break;
        end
        Wait until the replaying of required log entries
          is completed;
    end
    Search for the latest record version V_i that
      satisfies: V_i.commit_ts ≤ qts;
```

timestamp tg_cmt_ts, which represents the timestamp of the latest committed transaction in the table group. We update the value of tg_cmt_ts in the commit phase of log replaying at the last step of Algorithm 1. Algorithm 3 presents the main steps of how a query finds the latest data committed before the query arrival time. Given an incoming analytical query, if the minimum commit timestamp of all its accessed groups $min\_tg\_cmt\_ts$ is less than the query's arrival timestamp qts, it means that there may be a newer version that has not been replayed. Therefore, only after the $min\_tg\_cmt\_ts$ is greater than the query's timestamp, all required data be confirmed to have been replayed. After that, AETS search for the latest record version V_i that is no larger than the query's timestamp.

Some groups may not receive new OLTP logs for a long period, so their tg_cmt_ts would not be triggered to be refreshed, leading to the problem that the tg_cmt_ts in Algorithm 3 remains low and the query cannot proceed. To resolve this, we also compare the query's timestamp with the global commit timestamp $global\_cmt\_ts$ which is used to track the maximum commit timestamp of replayed transactions. In order to avoid the situation that $global\_cmt\_ts$ is not updated for a long time, which means there are no new log entries replicated from the primary. We insert an empty log with a dummy transactionID into all groups regularly just like the heartbeat mechanism. The log dispatcher is capable of carrying out such operations. If it finds that there is no new log in the task queues of all table groups within a specific time due that the primary being idle (e.g., 50 milliseconds), it will generate a dummy log. In this way, $global\_cmt\_ts$ will eventually be bumped.

VI. Experimental Evaluation

In this section, we empirically evaluate the design of AETS using real-world benchmark. In particular, we mainly focus on the following research questions (RQs):

- **RQ1:** Is the AETS framework efficiently designed for real-time HTAP?

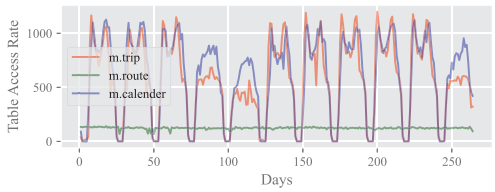


Fig. 7. An illustration of table access rate of BusTracker.

- **RQ2:** How is the multi-core scalability performance of AETS?
- **RQ3:** How much additional overhead does the design of AETS generate?
- **RQ4:** How is the effect of epoch size on visibility delay?
- **RQ5:** What impact does the adaptive thread allocation approach have on visibility delay?
- **RQ6:** How accurate is the prediction of DTGM model?

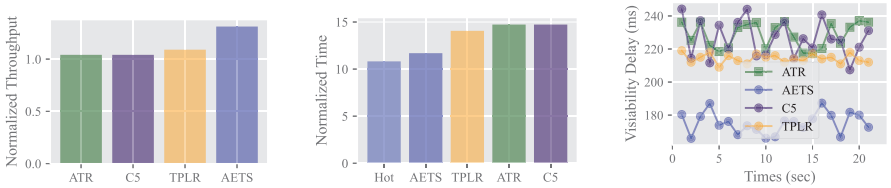
A. Experimental Setup

1) System: We implemented an in-memory backup database prototype from scratch with AETS, which consists of 11817 lines of C++ code. The Memtable utilizes a B+Tree as the in-memory storage engine for the backup database. Additionally, each record in the Memtable maintains multiple versions. AETS can parse the value log entries generated by the primary database, which is MySQL 8.0 community server [31].

2) Environment: We conducted extensive experiments on a three servers cluster, with each server equipped with a 64-core 2.50GHz Intel Xeon Processor (Cascadelake) and 128GB of memory. The servers are connected via a 10 Gbps Ethernet network. These servers run 64-bit Ubuntu 20.04.5 LTS with Linux kernel 5.4.0. One server runs the primary node, and the other servers run backup node.

3) Benchmarks: We use the following benchmarks to evaluate AETS:

- **TPC-C**. We run TPC-C's read-write transactions (Payment, NewOrder, and Delivery, default mixed configuration) on the primary database as the OLTP workload, and run read-only transactions on the backup database as the analytical workload. We set the scale factor (SF) of the dataset to 20. The proportion of log entries generated by hot tables is 90.98%. The epoch size is set to 2048 transactions. Hot tables accessed by read-only requests can be classified into two groups. One group consists of the *district*, *stock*, *customer* and *order* tables. Another group consists of the *order_line table*. The access rate for the *order_line* table is twice that of the other four tables. The remaining tables are considered cold tables, and each table is allocated to its own group.
- **BusTracker**. This is a real-world workload derived from and published by QB5000 [32], and it exhibits a comprehensible trend in access rates. This makes it a valuable workload for evaluating the performance of our log replaying mechanism. BusTracker contains schemas such as *m.app_state_log*, *m.screen_log*, and others that



(a) Normalized Replay Throughput (b) Normalized Replay Time (c) Visibility Delay

Fig. 8. TPC-C performance comparison under 32 threads. (a) normalized replay throughput (b) normalized replay time (c) visibility delay

undergo frequent modifications, although users rarely access this data. Once we had a comprehensive understanding of the BusTracker table schema, we generated a synthetic workload to evaluate the performance of AETS. The hot tables in BusTracker encompass a wide range of tables, including *m.trip*, *m.calendar*, *m.estimate*, *m.agency*, *m.stop_time*, *m.route*, *m.stop*, *m.messages*, *m.region_agency*, etc. The log entries associated with the hot tables constitute 37.12% of the total. Additionally, the epoch size remains at 2048 transactions. The grouping is determined dynamically. The table access rates for the three typical tables is shown in Figure 7.

- **CH-benCHmark**. CH-benCHmark [33] is a well-known HTAP workload. The OLTP originates from the TPC-C workload as shown in the above TPC-C. The OLAP part contains 22 independent queries [15]. In this case, 93.72% of the log entries are generated by hot tables. Each table is assigned to its own group. The other configurations remain consistent with those used in TPC-C.

4) Evaluation Metrics: We evaluate AETS with four aspects of metrics: (1) replay throughput, (2) visibility delay, (3) replay time, and (4) management overhead. The experimental data presented is an average of measurements collected from all backup nodes.

5) Baseline Methods: We compare the implementation of **AETS** with the following methods:

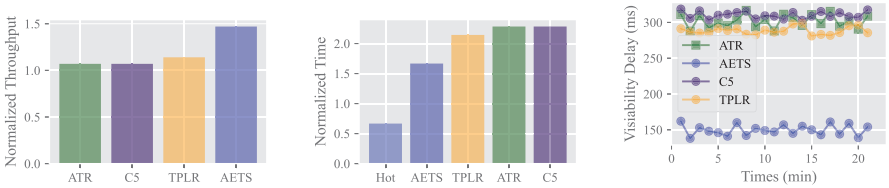
- **ATR** [6]: We implement the log replay method of the ATR based on the log format of our experiments. Compared with AETS, ATR uses the transactionID-based log dispatch method. In addition, the operation sequence is determined by comparing the before-image and the after-image of the value log entries. If an operation sequence error exists, the threads need to be synchronized.
- **C5** [11]: C5 employs the row-based dispatch method. Each row on the backup has its dedicated queue, which is distributed among different workers to ensure sequential log replay. In our implementation, C5 periodically (i.e., 5ms) used the timestamp of the smallest completed LSN among all dedicated queues as the current snapshot timestamp. This process ensures that all writes up to a certain log entry are included, allowing clients to read from it. The row-based dispatch method incurs significantly higher parsing costs compared to ATR and AETS. In our experiments, both ATR and AETS only need to parse the log metadata, while C5 must parse the entire log data image for dispatch.
- **TPLR**: It differs from AETS in that log entries are not grouped in this context. Hot and cold tables belong to a single group. Only two phases of log replay are retained.

B. Performance Comparison (*RQ1*)

TPC-C. The normalized replay throughput divided by primary throughput using 32 threads on TPC-C is shown in Figure 8(a) and normalized replay time divided by the replay time of the AETS(cold) table is shown in Figure 8(b) using different log replaying approaches.

We observed that the log replay throughput and replay time of ATR and C5 were nearly identical. During log replay in ATR, the transactionID-based dispatch method is employed. It conducts the operation sequence check to ensure the primary order of multi-version data. ATR uses a single thread to ensure that transaction visibility order in the same order as on the primary. Similarly, C5 schedules the all modifications of a row to a dedicated queue based on the transaction order. Each queue is served by a thread responsible for replaying the log entries. In C5, even though it maintains the visibility order of records in the same sequence as the primary using the row-based dispatch method. Nevertheless, employing the row-based dispatch method in our experimental setup results in increased parsing costs. This is because, in our experiments, with ATR and AETS, we only need to parse the log metadata, while with C5, we must parse the entire log data image. Our experimental results demonstrate that the cost of parsing the entire log data image for C5 and performing the operation sequence check for ATR is similar in a 32 threads experiment setting. In summary, the replay throughput and replay time performance of both approaches are comparable.

The replay throughput of AETS is $1.2\times$ greater than that of ATR and C5. This performance difference can be attributed to the table group approach employed by AETS, which involves grouping hot and cold table log entries based on their table access rates. This grouping allows AETS to replay these log entries in parallel across different groups. Simultaneously, AETS enables parallel commits between multiple table groups by utilizing a single commit thread for each table group. This feature is the most significant distinction from ATR and C5. It is important to note that the parallel commit based on table groups significantly reduces operation sequence checking in the commit phase. It directly reduces the number of transactions that need to be validated for commit order.



(a) Normalized Replay Throughput (b) Normalized Replay Time (c) Visibility Delay

Fig. 9. BusTracker performance comparison under 32 threads. (a) normalized replay throughput (b) normalized replay time (c) visibility delay

AETS facilitates parallel commits between groups, which sets it apart from ATR and C5's ungrouped single-thread commit approach. AETS maximizes both intra-group and inter-group parallelism to enhance parallel log replaying performance. Consequently, the parallel commit of multiple table groups can further reduces parallel replay time, resulting in an improved log replay throughput.

To illustrate the benefits of the table-group commit method used by AETS, we also compared the replay throughput and replay time of AETS and TPLR. Clearly, without grouping, TPLR slightly outperforms ATR and C5. This is due to TPLR which defers the operation sequence check to the second phase (i.e., the commit phase). This enables TPLR to achieve better parallel performance in the first phase (i.e., the parallel replaying phase) without the need to ensure operation sequence order consistency with the primary. Nevertheless, the performance of TPLR is still constrained by the ungrouped single-thread commit mechanism, just like ATR and C5, since TPLR is not grouped. In summary, TPLR demonstrates slightly improved performance compared to ATR and C5. However, it is important to note that TPLR still lags behind AETS in terms of replaying throughput and time.

Furthermore, inter-group parallel commits across multiple groups of log entries can disrupt the visibility order of transactions, potentially leading to inconsistencies in the transaction visibility order compared to the primary. To address this issue, we propose Algorithm 3 to ensure that the transaction order visible to readers is exactly the same as on the primary.

In summary, it is important to emphasize that AETS separates log replaying process from the visibility operation for OLAP queries. This decoupling allows parallel log replaying algorithms to fully exploit the parallelism offered by multiple threads (i.e., intra-group and inter-group parallelism). Simultaneously, this decoupling also provides an opportunity for table group transaction commit, which enables priority-based log replaying.

We also evaluate the level of data freshness provided by different parallel log replay approaches by observing the visibility delay of OLAP queries under TPC-C workloads. Given an incoming query at qts, its visibility delay is calculated according to the time of waiting for the required data with a commit timestamp larger than qts to be replayed. We observe that the average visibility delay of ATR is $1.3\times$ than AETS as shown in Figure 8(c). On one hand, AETS effectively hides the delay of cold tables by replaying log entries in the cold tables in the second stage. On the other hand, AETS facilitates table group parallel log replaying and commit to further reduce visibility delay significantly. AETS also employ adaptive thread resource allocation approach for table groups with varying table access rates to further improved the visibility delay.

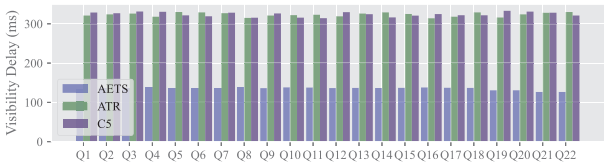


Fig. 10. Visibility delay on CH-benCHmark.

BusTracker. In the BusTracker experiments, as illustrated in Figure 9, we observed a significant decrease in the overall replay time for the hot table (i.e., Hot in Figure 9(b)) when the size of the hot table's log comprised a portion (37.12%) of the total log size. This is the most crucial feature of the AETS method, as it prioritizes to replay log entries on hot tables, leading to faster responses to OLAP queries.

CH-benCHmark. Figure 10 presents the experimental results, showing the visibility delay for each query. Our experiments have shown that the visibility delay of ATR and C5 is greater than that of AETS. This is mainly because AETS implements a fine-grained table grouping strategy which each table is assigned to its own group. This allows parallel commits between various table groups, leading to enhanced performance. The visibility delay of cold tables is also hidden due to the design of AETS, even though the proportion of cold tables is minimal. On the other hand, we observe that for 22 queries, their visibility delays are very close. The main reason is that queries that span multiple table groups require mutual waiting because Algorithm 3 is used to ensure that the visibility order matches that of the primary.

C. Multi-Core Scalability (*RQ2*)

To assess the multi-core scalability of the parallel log replay in AETS, we conducted experiments using the TPC-C benchmark. To begin, we generated AETS log entries from the primary server during a 30 seconds warm-up phase, followed by a subsequent 5-minute high-load phase. This ensured that the log entries captured a realistic workload scenario. Once the log entries were generated, we replicated them into the main memory of the replica in epoch mode, simulating a real-time environment. We varied the number of replay threads on the

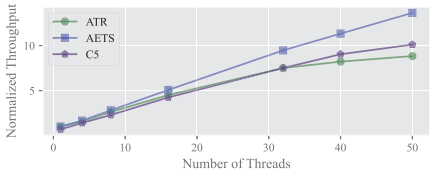


Fig. 11. Normalized replay throughput scalability on TPC-C over the number of threads, which is divided by single-thread throughput of ATR.

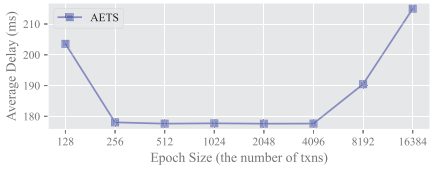


Fig. 12. Average visibility delay on TPC-C benchmark.

TABLE II
MANEGEMENT OVERHEAD OF AETS.

Datasets	dispatch	replay	commit
TPC-C	0.37%	99.47%	0.16%
BusTracker	0.80%	98.44%	0.76%
CH-benCHmark	0.72%	99.08%	0.20%

replica, allowing us to observe the impact of parallelism on the replay throughput. The experimental results presented in Figure 11, clearly depict the scalability of AETS.

As the number of replay threads increases, the normalized replay throughput of AETS consistently surpasses the ATR and C5, which is divided by single-thread through of ATR. This demonstrates the ability of AETS to effectively utilize multiple cores for parallel log replay. Interestingly, we observed that when the number of replay threads was fewer than 4, the replay throughput performance of AETS and ATR was very similar. This can be attributed to the fact that ATR has fewer operation sequence collisions, resulting in fewer synchronization requirements between threads. It's also worth mentioning that the throughput of C5 is slightly lower than ATR's when the number of threads is less than 32. In this case, the row-based dispatch cost of C5 is higher compared to ATR's operation sequence check cost, which leads to the difference in throughput. C5 exhibits better scalability than ATR when the number of threads exceeds 32. We can also observe that the ATR log replay throughput increment decreases when the number of threads exceeds 16 as shown in Figure 11. This decrease is mainly due to an increase in thread synchronization operations caused by operation sequence checks as the number of threads increases. These additional thread synchronization overhead limit ATR's ability to scale with multiple cores.

D. AETS Overhead (*RQ3*)

In this experiment, we aimed to estimate the overhead introduced by AETS during the log replay process. To achieve this, we conducted a time breakdown analysis, calculating the proportion of time spent on the dispatch phase, replay phase, and commit phase in relation to the total time required for replaying the transactions. The results of the time breakdown are presented in Table II. The analysis reveals that AETS spends approximately 1% of the total time in determining the appropriate group to dispatch a log entry to. This dispatch phase overhead is lower, indicating that AETS efficiently handles the task of assigning log entries to the relevant replay groups. This efficiency allows more than 98% of the total replay time to be dedicated to performing parallel log replay.

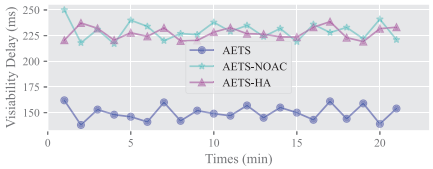


Fig. 13. Average visibility delay on BusTracker.

E. The Effect of Epoch Size on Visibility Delay (*RQ4*)

In this section, we examine how varying epoch sizes impact the average visibility delay on TPC-C. The experimental setup (32 threads) remains consistent with that of RQ2 in Section VI-C. We adjust the epoch size, and the visibility delay is depicted in Figure 12.

We observed that the visibility delay is increased when the epoch size is set up in a small value (less than 256) or a large value (greater than 4096). Log replay is performed epoch by epoch. If the epoch size is too small, log entries on hot tables in the subsequent epoch can not be replayed earlier than many log entries on cold tables. However, if the epoch size is too large, AETS needs to wait for assembling enough log entries. This also caused the visibility delay to increase. In this paper, we empirically set epoch size to 2048 which is the size of a batch of replicated log entries from the primary. The result of BusTracker has almost the same.

F. Adaptive Threads Allocation Approach (*RQ5*)

As demonstrated in Figure 13, we conducted experiments to assess the impact of various thread allocation approaches on visibility delay using BusTracker. We record that visibility delay during 30 minutes (5 minutes warm-up) of running BusTracker, and compute the average visibility delay every minute. These methods include: (1) AETS-HA: This method utilizes the historical table access rate from last five minutes as predicted table access rate to allocate thread resources of next one minute; (2) AETS-NOAC: This method does not take into account the table access rate and only considers the log size on the table group when allocating thread resources. The experimental results clearly indicate that the visibility delay of AETS is lower than that of AETS-NOAC. This is primarily due to the adaptive thread allocation policy implemented by AETS, which allocates more threads to table groups with higher table access rates, resulting in better visibility. The results also indicate that forecasting the table access rate method based on historical data does not impact the average visibility delay significantly. This is mainly due to the limited ability of AETS-HA to adapt to changing workloads. This emphasizes the significance of accurately predicting the table access rate. In Section VI-G, we delve into the accuracy of DTGM, as used in this paper, through experimental analysis.

TABLE III
PERFORMANCE COMPARISON.

Model	15 mins	30 mins	60 mins
HA	30.30%	30.30%	30.30%
ARIMA	18.66%	21.50%	27.90%
QB5000	18.12%	19.70%	20.50%
DTGM	16.80%	18.18%	19.76%

TABLE IV
COMPARISON BETWEEN OUR APPROACH AND ITS VARIANT.

Model	MAPE
w/o gcn	16.96%
DTGM	16.80%

G. Overall Comparison of DTGM (*RQ6*)

We evaluate the variants with **MAPE**: it is calculated as follows:

$$\textbf{MAPE}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{|n|}\sum_{i=1}^{n}|\frac{\mathbf{x}_i - \hat{\mathbf{x}}_i}{\mathbf{x}_i}|$$

where $\mathbf{x} = x_1, x_2, ..., x_n$ represent real access rate, $\hat{\mathbf{x}} = \hat{x}_1, \hat{x}_2, ..., \hat{x}_n$ represent prediction access rate. The results in Table III were compared based on the MAPE for predictions at three different time intervals: the next 15 minutes, 30 minutes, and 60 minutes in this experimental setting. We compare the **DTGM** using BusTracker with the following methods:

- **HA**: It uses historical average table access rate from last 60 minutes to predict future access rate.
- **ARIMA [34]**: The combination of autoregressive and moving average models is one of the time series prediction methods.
- **QB5000 [32]**. QB5000 generates forecasts by equally averaging the results of LR, LSTM, and KR.

1) Experimental Setting: The hidden layer of all networks is set to 48. During model training, the batch size is 64, the Adam optimizer is used, the initial learning rate is $1e-3$, and the learning rate decays by 0.1 after every 20 epochs. The L2 penalty factor is $1e-5$, and the dropout ratio is 0.3.

2) Prediction Performance: In order to demonstrate the effectiveness of the proposed DTGM, a thorough comparison was conducted with arts. The results obtained from this comparative analysis are presented in Table III. It evidents DTGM consistently outperformed the traditional methods.

Although DTGM has better accuracy than QB5000, the small difference in accuracy does not affect the replay performance and visibility delay. In this case, the allocation of thread resources is almost the same for table groups. However, due to the need to train three models for the QB5000, retraining is expensive. In particular, LSTM demands more resources and presents challenges in the training process.

3) Ablation Study: We delve into an ablation study to thoroughly investigate the effectiveness of different DTGM components. To provide a comprehensive analysis, we compare DTGM with its variant **w/o gcn**, which excludes the GCN component. The experimental result indicates DTGM outperforms the compared method as shown in Table IV. By effectively leveraging graph-based representations, DTGM can improve predictive performance.

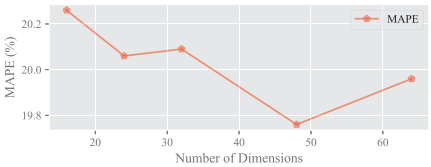


Fig. 14. Hidden layer dimension hyper-parameter.

4) Hyper-parameter analysis: This subsection studies the impact of the model hyper-parameters. The experimental result is shown in Figure 14, which depicted the current optimal hidden layer dimension as 48.

VII. RELATED WORK

Parallel Log Replay. KuaFu [4] improves the parallelism by tracking transaction write-write dependencies. Taurus [8] introduces a lightweight parallel logging scheme to speed up the logging process. Lee et al. [2] proposed a SessionID-based log dispatch method and RVID-based dynamic detection of operation sequence error. Many works log logical executed operations instead of logging physical execution results [35], [7], [36]. Adaptive logging [37] improves recovery for command logging by generating a graph based on data dependencies between transactions. Dai Qin et al. [7] proposed a deterministic record-replay strategy instead of transmitting the modified data as the replicated log. Stefan Halfpap et al. [9] proposed to only replay a subset of the data to speed up the query on the replication. Juchang Lee et al. [10] presented a novel concept of an asymmetric partition replication. Query Fresh [30] takes advantage of RDMA and NVM to achieve fast log shipping. C5 [11] employs the row-based dispatch method.

Workload Analysis. QB5000 [32] forecasts the query arrival rate by using linear regression and recurrent neural networks. Ganapathi et al. [38] used kernel canonical correlation analysis to predict performance metrics. Meduri et al. [39] built a next-query prediction framework. Higginson et al. [40] used time series analysis and supervised learning to model the OLTP and OLAP workloads for database capacity planning.

VIII. CONCLUSION

In this paper, we presented AETS, an adaptive epoch-based two-stage log replay framework. AETS separates log replaying and committing processes from the visibility operation for readers through table grouping. Simultaneously, this decoupling also provides an opportunity for table group transaction commit, which enables priority-based parallel log replaying. Compared with the state-of-the-art approaches, AETS can significantly reduce response time for real-time queries by minimizing the time gap between the receipt of replication logs and log replay.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Natural Science Foundation of China under Grant No. U22B2020, and in part by PingCAP. Peng Cai is the corresponding author.

REFERENCES

[1] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1243–1254.

[2] J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W. Han, "Hybrid garbage collection for multi-version concurrency control in SAP HANA," in *SIGMOD*, 2016, pp. 1307–1318.

[3] Z. Lyu, H. H. Zhang, G. Xiong, G. Guo, H. Wang, J. Chen, A. Praveen, Y. Yang, X. Gao, A. Wang *et al.*, "Greenplum: a hybrid database for transactional and analytical workloads," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2530–2542.

[4] C. Hong, D. Zhou, M. Yang, C. Kuo, L. Zhang, and L. Zhou, "Kuafu: Closing the parallelism gap in database replication," in *ICDE*, 2013, pp. 1186–1195.

[5] F. Färber, N. May, W. Lehner, P. Große, and et al., "The SAP HANA database – an architecture overview," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012.

[6] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, W. Han, C. G. Park, H. J. Na, and J. Lee, "Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1598–1609, 2017.

[7] D. Qin, A. Goel, and A. D. Brown, "Scalable replay-based replication for fast databases," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2025–2036, 2017.

[8] Y. Xia, X. Yu, A. Pavlo, and S. Devadas, "Taurus: Lightweight parallel logging for in-memory database management systems," *Proc. VLDB Endow.*, vol. 14, no. 2, pp. 189–201, 2020.

[9] S. Halfpap and R. Schlosser, "Workload-driven fragment allocation for partially replicated databases using linear programming," in *ICDE*, 2019, pp. 1746–1749.

[10] J. Lee, H. Lee, S. Ko, K. H. Kim, M. Andrei, F. Keller, and W. Han, "Asymmetric-partition replication for highly scalable distributed transaction processing in practice," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3112–3124, 2020.

[11] J. Helt, A. Sharma, D. J. Abadi, W. Lloyd, and J. M. Faleiro, "C5: cloned concurrency control that always keeps up," *Proceedings of the VLDB Endowment*, vol. 16, no. 1, pp. 1–14, 2022.

[12] W. Zheng, S. Tu, E. Kohler, and B. Liskov, "Fast databases with fast durability and recovery through multicore parallelism," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2014, p. 465–477.

[13] "tpcc," https://www.tpc.org/tpcc/.

[14] "Seats," https://hstore.cs.brown.edu/projects/seats/.

[15] "Ch-benchmark," https://db.in.tum.de/research/projects/CHbenCHmark/?lang=en.

[16] "Bustracker," http://www.cs.cmu.edu/~malin199/data/tiramisu-sample/.

[17] J. Chen, Y. Ding, Y. Liu, F. Li, L. Zhang, M. Zhang, K. Wei, L. Cao, D. Zou, Y. Liu, L. Zhang, R. Shi, W. Ding, K. Wu, S. Luo, J. Sun, and Y. Liang, "Bytehtap: Bytedance's htap system with high data freshness and strong data consistency," *Proc. VLDB Endow.*, p. 3411–3424, 2022.

[18] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang, "Tidb: A raft-based htap database," *Proc. VLDB Endow.*, p. 3072–3084, 2020.

[19] J. Wang, T. Li, H. Song, X. Yang, W. Zhou, F. Li, B. Yan, Q. Wu, Y. Liang, C. Ying *et al.*, "Polardb-imci: A cloud-native htap database system at alibaba," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–25, 2023.

[20] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, 2015.

[21] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 677–689.

[22] Y. Lu, X. Yu, L. Cao, and S. Madden, "Epoch-based commit and replication in distributed oltp databases," *VLDB Endowment*, 2021.

[23] J. Böttcher, V. Leis, T. Neumann, and A. Kemper, "Scalable garbage collection for in-memory mvcc systems," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, pp. 128–141, 2019.

[24] S. Shen, R. Chen, H. Chen, and B. Zang, "Retrofitting high availability mechanism to tame hybrid transaction/analytical processing," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 219–238.

[25] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," *arXiv preprint arXiv:1707.01926*, 2017.

[26] S. Guo, Y. Lin, N. Feng, C. Song, and H. Wan, "Attention based spatial-temporal graph convolutional networks for traffic flow forecasting," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 922–929.

[27] Z. Wu, S. Pan, G. Long, J. Jiang, and C. Zhang, "Graph wavenet for deep spatial-temporal graph modeling," *arXiv preprint arXiv:1906.00121*, 2019.

[28] C. Song, Y. Lin, S. Guo, and H. Wan, "Spatial-temporal synchronous graph convolutional networks: A new framework for spatial-temporal network data forecasting," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 01, 2020, pp. 914–921.

[29] S. Bloemheuvel, J. v. d. Hoogen, D. Jozinović, A. Michelini, and M. Atzmueller, "Multivariate time series regression with graph neural networks," *arXiv preprint arXiv:2201.00818*, 2022.

[30] T. Wang, R. Johnson, and I. Pandis, "Query fresh: Log shipping on steroids," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 406–419, 2017.

[31] "Mysql 8.0," https://dev.mysql.com/doc/relnotes/mysql/8.0/en/.

[32] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 631–645.

[33] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K. Sattler, "Scaling up mixed workloads: A battle of data freshness, flexibility, and scheduling," in *Performance Characterization and Benchmarking. Traditional to Big Data - 6th TPC Technology Conference, TPCTC*, 2014, pp. 97–112.

[34] B. M. Williams and L. A. Hoel, "Modeling and forecasting vehicular traffic flow as a seasonal arima process: Theoretical basis and empirical results," *Journal of transportation engineering*, vol. 129, no. 6, pp. 664–672, 2003.

[35] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker, "Rethinking main memory OLTP recovery," in *ICDE*, 2014, pp. 604–615.

[36] Y. Wu, W. Guo, C. Chan, and K. Tan, "Fast failure recovery for main-memory dbmss on multicores," in *SIGMOD*, 2017, pp. 267–281.

[37] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu, "Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases," in *SIGMOD*, 2016, pp. 1119–1134.

[38] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *ICDE*, 2009, pp. 592–603.

[39] V. V. Meduri, K. Chowdhury, and M. Sarwat, "Evaluation of machine learning algorithms in predicting the next SQL query from the future," *ACM Trans. Database Syst.*, vol. 46, no. 1, pp. 4:1–4:46, 2021.

[40] A. S. Higginson, M. Dediu, O. Arsene, N. W. Paton, and S. M. Embury, "Database workload capacity planning using time series analysis and machine learning," in *SIGMOD*. ACM, 2020, pp. 769–783.