GeoTP: Latency-aware Geo-Distributed Transaction Processing in Database Middlewares

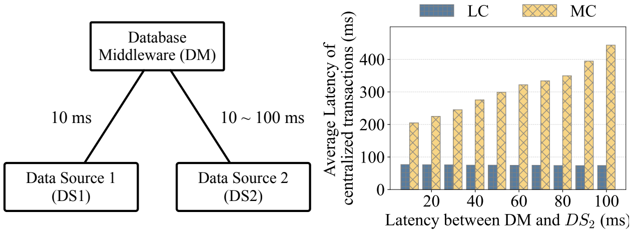
Qiyu Zhuang[†], Xinyue Shi[†], Shuang Liu[†], Wei Lu[†], Zhanhao Zhao[†]
Yuxing Chen[‡], Tong Li[†], Anqun Pan[‡], Xiaoyong Du[†]
[†] *Renmin University of China* [‡] *Tencent Inc.*
[†]{qyzhuang, xinyueshi, shuang.liu, lu-wei, zhanhaozhao, tong.li, duyong}@ruc.edu.cn
[‡]{axingguchen, aaronpan}@tencent.com

***Abstract*—The widespread adoption of database middleware for supporting distributed transaction processing is prevalent in numerous applications, with heterogeneous data sources deployed across national and international boundaries. However, transaction processing performance significantly drops due to the high network latency between the middleware and data sources and the long lock contention span, where transactions may be blocked while waiting for the locks held by concurrent transactions. In this paper, we propose GeoTP, a latency-aware geo-distributed transaction processing approach in database middleware. GeoTP incorporates three key techniques to enhance performance in geo-distributed scenarios. First, we propose a decentralized prepare mechanism to reduce network round-trips for distributed transactions. Second, we design a latency-aware scheduler to minimize the lock contention span by strategically delaying the lock acquisition. Third, heuristic optimizations are proposed for the scheduler to reduce the lock contention span further. We implemented GeoTP on Apache Shardingsphere, a state-of-the-art middleware, and extended it into Apache ScalarDB. Experimental results on YCSB and TPC-C demonstrate that GeoTP achieves up to 17.7x performance improvement.**

***Index Terms*—Transaction Processing, Geo-Distributed, Heterogeneous Databases**

I. Introduction

Globalization of enterprises is an inevitable trend in economic development. Critical global applications, such as cross-border e-commerce [1] and e-banking [2], require data to be stored in different regions for compliance with local government regulations [3], while guaranteeing atomic transaction processing among them. For instance, a global e-commerce application might store its US user account data in the US and the stock data in the warehouse location (Singapore). Furthermore, these databases are often managed by different departments, making them highly likely to be heterogeneous. Consequently, a typical product purchase requires a geo-distributed transaction that updates two heterogeneous databases in different locations, ensuring the user balance and current stock are updated atomically. To achieve this, database middlewares, such as Shardingsphere [4] and ScalarDB [5], become indispensable to connect heterogeneous databases across different regions for unified data services. Unlike distributed database systems [6]–[8] , which usually requires rebuilding databases and applications. Database middleware can provide transaction processing capabilities without modification. This facilitates easier global service constructions, leading to widespread adoption in enterprises [9], [10].



(a) Data source deployment (b) Impact of network latency

Fig. 1: A motivating example

Database middlewares (abbreviated as DMs) typically employ the eXtended Architecture (XA) Protocol, an extension of the two-phase commit (2PC), to ensure the transaction's atomicity. Databases, such as MySQL [11] and PostgreSQL [12], serve as the data sources of DMs. Specifically, the DM accepts the transactions submitted by the clients. We consider transaction T as a *centralized transaction* if it involves a single data source; otherwise, T is considered as a *distributed transaction*. For a *centralized transaction*, the DM forwards it to the relevant data source, which executes it and returns the results. Upon receiving the commit or abort command, the DM instructs the relevant data source to commit or abort directly, requiring one wide-area network (WAN) round trip. For a *distributed transaction*, whether interactive or stored procedures, the typical transaction processing protocol first executes read/write operations in the relevant data sources during the execution phase. Upon receiving the commit or abort command, the DM follows the 2PC [13], [14], including a prepare phase and a commit phase, to ensure transaction atomicity. This commit process requires two WAN round trips. *The WAN round trip time dominates transaction latency and significantly degrades performance, particularly for distributed transactions, which requires two WAN round trips for commitment. The impact is more pronounced in geo-distributed scenarios where network latencies between the DM and data sources are high* [15].

Data sources [11], [12], [16]–[18] typically use two-phase locking (2PL) or its variants for concurrency control[*]. *In addi-*

[*] We focus on serializable isolation level in this work.

tion to the overhead of WAN communications, long lock contention span—the time span between the acquisition (before reads or writes) and release (after the commitment) of a record lock—is also a critical factor for performance degradation. We explicitly design an experiment to show the impact of lock contention span on transaction performance. Figure 1 illustrates two data sources, DS_1 and DS_2. with a 10 ms WAN round-trip time (RTT) between the DM and DS_1, and varying latency between the DM and DS_2 (10–100 ms). The workload includes 80% *centralized transactions* accessing DS_1 and 20% *distributed transactions* accessing both DS_1 and DS_2. We evaluate the average latency of *centralized transactions* (on DS_1) with varying the network latency between DM and DS_2 under low-contention (LC) and medium-contention (MC) workloads. As depicted in Figure 1b, network latency between the DM and DS_2 has a more substantial impact on *centralized transactions* under medium contention than low contention, even though these transactions do not access DS_2. This is because in medium-contention workloads, *centralized transactions* are more likely to access shared records with *distributed transactions* that access DS_2. The lock contention span of *distributed transactions*, significantly affected by the network latency between the DM and DS_2, impacts the latency of *centralized transactions* due to shared record blocking. We provide more details for this in §II.

Several works are proposed to reduce WAN round trips in distributed transaction processing. Early Prepare [19] and RedT [15] reduce the network round-trips by writing logs during execution, thus eliminating the prepare phase. Carousel [20], Natto [21] and Janus [22] reduce network round trips by integrating consensus protocols with 2PC, assuming knowing the read/write sets in advance. However, they require rewriting the kernel-level protocol, making them difficult to extend to heterogeneous data sources. Another line of work has proposed delayed scheduling techniques to reduce lock contention spans. QURO [23] preprocesses the application code to reorder the read/write operations and delays the acquisition of exclusive locks for writes. However, it lacks consideration for network latency, limiting its effectiveness in geo-distributed scenarios. Chiller [24] and DAST [25] address latency differences in geo-distributed scenarios by scheduling cross-region subtransactions to follow intra-region ones, as hot records are often in the intra-region ones. However, these methods are designed for stored procedures and overlook the varied latency between cross-region nodes and execution times, potentially differing by orders of magnitude, leaving substantial room for optimizing the lock contention span.

In this paper, we present GeoTP, a latency-aware geo-distributed transaction processing approach in database middleware. We propose three key techniques to mitigate the impact of network latency and lock contention while ensuring that GeoTP continues to support general-purpose transactions, including interactive and stored procedures. Our key techniques and contributions are summarized as follows.

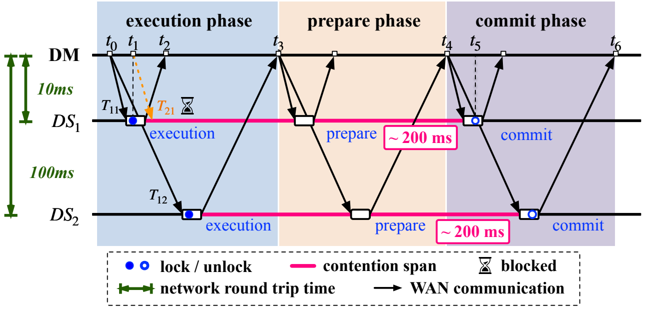


Fig. 2: Distributed transaction processing in DMs

(1) Decentralized prepare mechanism that offloads the coordination cost required for the prepare phase (§IV-A). GeoTP triggers the prepare phase implicitly at the end of the execution phase, effectively eliminating one WAN round trip and reducing the latency of distributed transactions. However, this process is challenging due to the different transaction protocols used by various data sources. To address this, we leverage annotations to mark the last statement and develop an efficient component called geo-agent to abstract differences between data sources, facilitating decentralized preparation in GeoTP. Additionally, we design an early abort mechanism that allows fault transactions to abort quickly, preventing such transactions from degrading the performance.

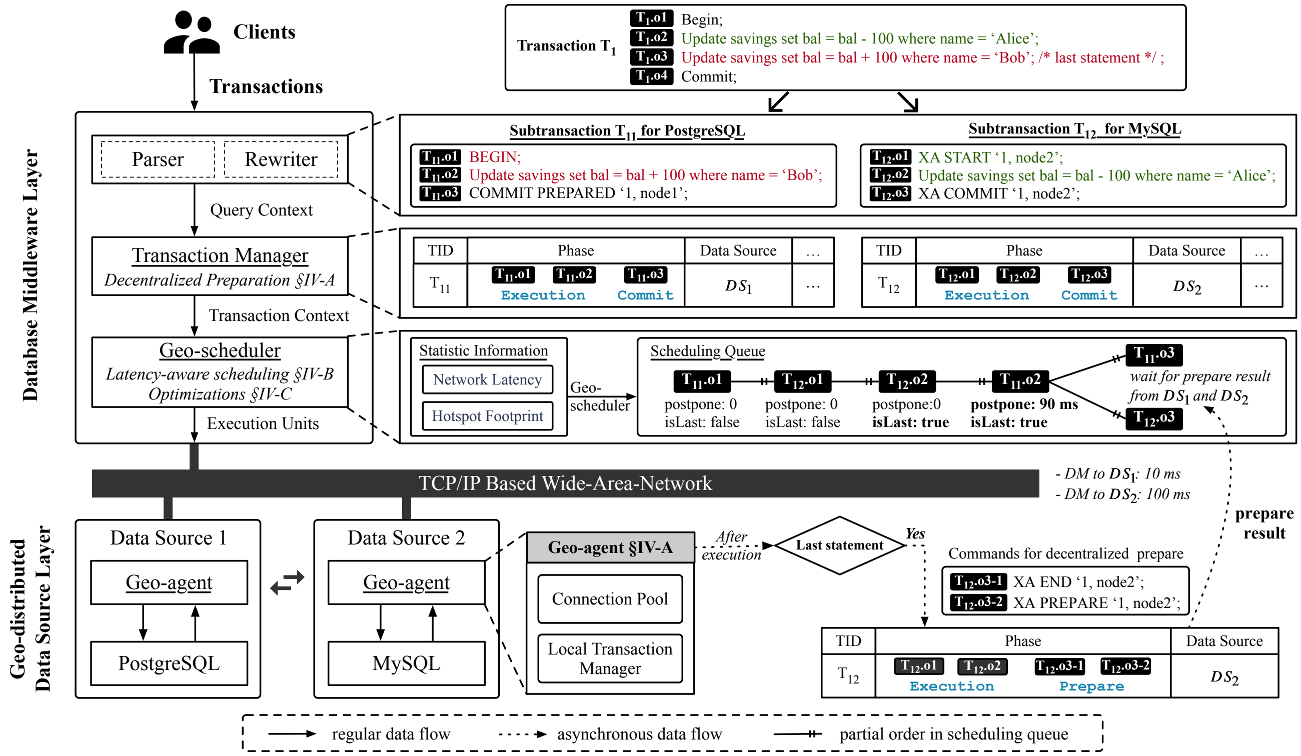
(2) Latency-aware scheduling to minimize the lock contention span (§IV-B). The lock contention span of a transaction is determined by the highest network latency involved, resulting in unnecessary lock contention. To address this, we propose a latency-aware scheduling mechanism that postpones the lock request time point for the subtransactions accessing data sources with lower network latency. Since the lock release time point remains unchanged, the lock contention span of these subtransactions is reduced. This approach minimizes the impact of *distributed transactions* on transaction concurrency, thereby improving the overall system performance.

(3) Optimized scheduling for high-contention workloads considering local execution latency (§IV-C). In high-contention workloads, the lock contention span is influenced not only by the highest network latency but also by the time subtransactions spend waiting to acquire locks. For instance, a subtransaction with lower network latency might still face significant latency if it has to wait a long time for locks on hotspots, causing its local execution latency to exceed the longest network latency and become a bottleneck. To enhance scheduling precision in such scenarios, we employ heuristic optimization, including transaction admission and local execution latency forecasting mechanisms. By doing this, GeoTP can further reduce the lock contention span.

We implement GeoTP on Apache ShardingSphere and extend our optimizations on ScalarDB. Extensive evaluations on YCSB and TPC-C show that GeoTP achieves a performance improvement of up to 17.7x over Shardingsphere and up to 3.2x over ScalarDB and offers comparable performance to distributed databases.

II. Motivation Example

In this section, we use Figure 2 as an example to motivate our work. The network latency between DM and DS_1 is 10



Clients
Transactions
Transaction T_1
T_1.o1 Begin;
T_1.o2 Update savings set bal = bal - 100 where name = 'Alice';
T_1.o3 Update savings set bal = bal + 100 where name = 'Bob'; /* last statement */ ;
T_1.o4 Commit;
Database Middleware Layer
Parser
Rewriter
Query Context
Transaction Manager
Decentralized Preparation §IV-A
Transaction Context
Geo-scheduler
Latency-aware scheduling §IV-B
Optimizations §IV-C
Execution Units
Subtransaction T_{11} for PostgreSQL
T_{11}.o1 BEGIN;
T_{11}.o2 Update savings set bal = bal + 100 where name = 'Bob';
T_{11}.o3 COMMIT PREPARED '1, node1';
Subtransaction T_{12} for MySQL
T_{12}.o1 XA START '1, node2';
T_{12}.o2 Update savings set bal = bal - 100 where name = 'Alice';
T_{12}.o3 XA COMMIT '1, node2';
TID | Phase | Data Source | …
T_{11} | T_{11}.o1 T_{11}.o2 Execution | T_{11}.o3 Commit | DS_1 | …
TID | Phase | Data Source | …
T_{12} | T_{12}.o1 T_{12}.o2 Execution | T_{12}.o3 Commit | DS_2 | …
Statistic Information
Network Latency
Hotspot Footprint
Geo-scheduler
Scheduling Queue
T_{11}.o1 postpone: 0 isLast: false
T_{12}.o1 postpone: 0 isLast: false
T_{12}.o2 postpone:0 isLast: true
T_{11}.o2 postpone: 90 ms isLast: true
T_{11}.o3
T_{12}.o3
wait for prepare result from DS_1 and DS_2
TCP/IP Based Wide-Area-Network
- DM to DS_1: 10 ms
- DM to DS_2: 100 ms
prepare result
Geo-distributed Data Source Layer
Data Source 1
Geo-agent
PostgreSQL
Data Source 2
Geo-agent
MySQL
Geo-agent §IV-A
Connection Pool
Local Transaction Manager
After execution
Last statement
Yes
Commands for decentralized prepare
T_{12}.o3-1 XA END '1, node2';
T_{12}.o3-2 XA PREPARE '1, node2';
TID | Phase | Data Source
T_{12} | T_{12}.o1 T_{12}.o2 Execution | T_{12}.o3-1 T_{12}.o3-2 Prepare | DS_2
regular data flow
asynchronous data flow
partial order in scheduling queue

Fig. 3: An overview of GeoTP

ms, while the latency between DM and DS_2 is 100 ms. There are two transactions, T_1 and T_2, arriving DM at times t_0 and t_1, respectively. T_1 is a distributed transaction accessing records in DS_1 and DS_2, while T_2 is a centralized transaction accessing a shared record r with T_1 in DS_1. We denote the subtransaction of T_i executed on data source DS_j as T_{ij}. Note that in most scenarios, network latency outweighs execution latency. For simplicity, we ignore the local execution time required in each phase without loss of generality.

In a typical distributed transaction scenario, the DM acts as a *coordinator*, while each data source serves as a *participant*. The lifecycle of a distributed transaction, e.g., T_1, can be divided into three phases: 1) the execution phase, 2) the prepare phase, and 3) the commit phase. During the execution phase, the DM parses a transaction T_1 into subtransactions T_{11} and T_{12} and dispatches them to data sources DS_1 and DS_2 based on data distribution. The data source, e.g., DS_1, then initiates a subtransaction T_{11}, acquires locks on record r before reads or writes, and sends the execution results back to the DM. The client submits the *commit* request at t_3, triggering the prepare phase. The DM notifies DS_1 and DS_2 to verify whether the subtransactions are ready for the commitment. In response, data sources persist the transaction states and write-ahead logs and then return the prepared result. The DM collects all prepared results at t_4 and determines whether to commit or abort the transaction based on the return results from the data sources. Finally, in the commit phase, the DM dispatches the final decision to DS_1 and DS_2, which involves another WAN round trip, and the transaction is completed at t_6. The lifecycle of T_1 is from t_0 to t_6, involving three WAN round trips, which dominate the transaction latency.

As evident from Figure 2, T_{11} acquires the lock on r at t_1 and release it at t_5. The lock contention span of T_{11} on record r is around 200 ms (2 WAN round trips), determined by the network latency between the DM and DS_2. The subtransaction T_{21} arrives DS_1 at around t_2 and is blocked by T_{11} until t_5 due to its prolonged lock contention span. The DM has to await the execution results from T_{21}, which are received around t_5+5 ms, significantly increasing the execution latency of T_2. Even worse, if T_{21} acquires locks on other records, the lock contention span can transitively block other concurrent transactions. Note that, even if transaction T_2 is a *centralized transaction* without accessing any record in DS_2, the network latency between DM and DS_2 still affects the transaction latency of T_2 through the lock contention span of transaction T_1. This explains the experiment results in Figure 1b.

This motivation example highlights the substantial impacts of network latency and long lock contention spans on the transaction performance in geo-distributed scenarios.

III. Overview of GeoTP

Figure 3 provides an overview of GeoTP, which operates in the two-layer architecture. The first layer functions as the DM [26]–[28], while the second layer comprises data sources that can be geographically distributed and heterogeneous; for example, DS_1 includes a PostgreSQL instance and DS_2 includes a MySQL instance. For clarity, we assign a monotonic identifier to each operation within a transaction T. We assume that applications can use annotations, which are prefixes or suffixes on SQL statements, to pass certain operations hints to GeoTP. Given that SQL annotations are commonly used to guide and influence database query optimization [29], [30] manually, we consider this assumption reasonable.

A. Database Middleware Layer

In the first layer, similar to existing DMs, GeoTP is equipped with the parser and rewriter, which accept transactions submitted by the clients and transform them into multiple subtransactions. Despite these components, GeoTP is equipped with an enhanced transaction manager and a geo-scheduler, which differ from existing DMs. The enhanced transaction manager is responsible for coordinating the execution and handling failure recovery. The geo-scheduler is particularly crafted to calculate the optimal start time point for subtransactions, minimizing the lock contention span. Next, we use transaction T_1 in Figure 3 as an example to explain the transaction processing in GeoTP and our key techniques. Suppose Alice submits transaction T_1, which transfers \$100 from her account to Bob's account. Note that Bob's account is stored in a PostgreSQL instance (DS_1), and Alice's account is stored in a MySQL instance (DS_2).

Parser and rewriter. These components parse SQL statements received from clients and then rewrite them according to the grammar rules of target databases. For example, they translate T_1 into T_{11} and T_{12}, which are executable for PostgreSQL and MySQL, respectively. Operations $T_{11}.o1$ and $T_{12}.o1$ start an XA transaction in each data source. Operation $T_{11}.o2$ deposits \$100 into Bob's account in PostgreSQL, while $T_{12}.o2$ deducts \$100 from Alice's account in MySQL. Subsequently, $T_{11}.o3$ and $T_{12}.o3$ attempt to commit the respective subtransactions.

Transaction manager. Unlike conventional transaction managers, our enhanced transaction manager employs a decentralized prepare mechanism to eliminate the prepare phase from the critical path of XA protocol (§IV-A). In our design, the prepare phase is no longer triggered by *commit* request of clients. Instead, it is initiated after the last statement in the execution phase, explicitly specified by the client (e.g., $T_1.o3$). Upon identifying the last SQL statement, the transaction manager combines the decentralized prepare phase with the processing of this statement over the underlying data sources. For example, since there is no dependency between $T_1.o2$ and $T_1.o3$, we assume the client sends them together to the DM. The transaction manager treats them as the last SQL statement of each data source. Importantly, if some data sources are involved in the transaction but not processing the last SQL statement, the transaction manager directly notifies those data sources to initiate the prepare phase. This approach eliminates one WAN round trip required by the prepare phase for distributed transactions.

Geo-scheduler. The geo-scheduler implements the latency-aware scheduling of subtransactions by calculating each statement's optimal start time point based on the network latency and predicted transaction execution latency. As is shown in Figure 3, suppose the average network latency from the DM to DS_1 and DS_2 is 10 ms and 100 ms. We show the schedule produced by the geo-scheduler on the bottom-right of the first layer. Specifically, $T_{11}.o1$, and $T_{12}.o1$ are first scheduled to execute. Next, $T_{12}.o2$ is scheduled without postponing, while $T_{11}.o2$ is scheduled and has been postponed 90 ms for execution. Unlike traditional schedulers where $T_{11}.o2$ and $T_{12}.o2$ are sent to data sources simultaneously, resulting in a contention span of 100 ms for both operations; our scheduler adopts a prioritized strategy (details in §IV-B & IV-C). This strategy reduces the contention span of $T_{11}.o2$ to 10 ms without increasing the overall latency of T_1. The scheduling of $T_{11}.o3$ and $T_{12}.o3$ needs to wait for the prepare results from data sources DS_1 and DS_2, respectively.

B. Geo-distributed Data Source Layer

In the second layer, each data source is equipped with a geo-agent comprising two crucial components: a connection pool and a local transaction manager. The connection pool manages connections with the DM, the underlying database, and other geo-agents. The local transaction manager receives/forwards messages from/to the DM or database and notifies the database to initiate the implicit decentralized prepare phase. Following the running example, after the execution of $T_{12}.o2$, which is the last statement of T_{12}, the geo-agent instructs DS_2 to execute $T_{12}.o3$-1 and $T_{12}.o3$-2 (shown in the bottom right corner of Figure 3) to complete the prepare phase. Once $T_{12}.o3$ is received, the geo-agent only needs to await the result of $T_{12}.o3$-2 and then instructs DS_2 to commit T_{12}. In the event of an abort during execution, the geo-agent implements the *early abort* mechanism to proactively notify other data sources involved to pre-terminate other subtransactions without the coordination of DM (§IV-A).

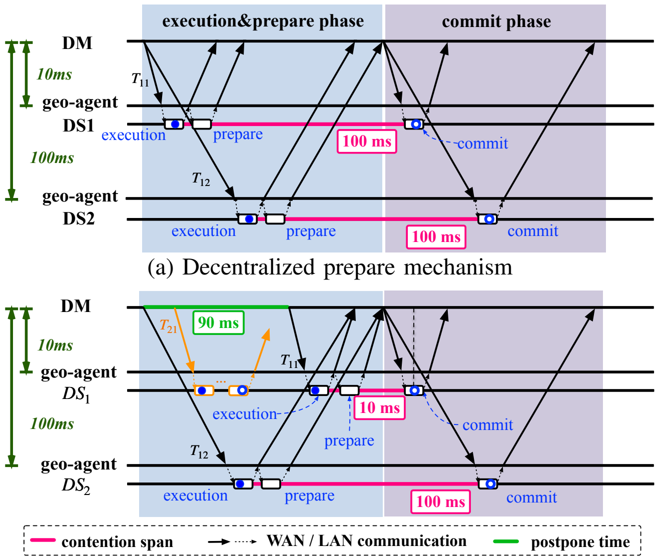
IV. Detailed Design

In this section, we introduce the key techniques of GeoTP, including the decentralized prepare mechanism (§IV-A), the latency-aware transaction scheduling mechanism (§IV-B), and further optimizations for high-contention workloads (§IV-C).

A. Decentralized Prepare Mechanism

With the traditional 2PC protocol, the DM is responsible for coordinating both the prepare and commit phases of distributed transactions, incurring two WAN round trips and resulting in expensive coordination costs. This issue often leads to prolonged distributed transaction latency, significantly impacting performance. In GeoTP, we propose (1) a decentralized prepare mechanism to eliminate one WAN round trip time for the prepare phase and (2) an early abort mechanism to pre-terminate unnecessary execution of subtransactions. Due to space limitations, we illustrate the pseudo-code of key functions in our extended version [31].

Decentralized prepare mechanism. We propose a decentralized preparation mechanism to offload the coordination cost associated with the prepare phase, i.e., one WAN round trip from the DM to data sources. In GeoTP, the prepare phase is initiated by the geo-agent, reducing the cost from one WAN round trip (i.e., from the DM to the data source) to one local-area network (LAN) round trip (i.e., from the



(a) Decentralized prepare mechanism

(b) Latency-aware scheduling mechanism

Fig. 4: Distributed transaction processing in GeoTP

geo-agent to the data source). The design hint is that once a subtransaction completes its execution phase, it can directly enter the prepare phase without waiting for the *"prepare"* messages from the DM. This approach does not compromise transaction correctness and may incur minimal additional overhead if a client proactively aborts the transaction. However, this overhead is negligible compared to the reduction in WAN round trip time. To enable decentralized preparation, we use an annotation code to explicitly mark the last SQL statement in a transaction. Upon completing the execution of the last SQL statement, the geo-agent directly initiates an implicit decentralized prepare phase. For example, if the data source is MySQL, the geo-agent executes *"XA end"* and *"XA prepare"* statements to initiate the prepare phase; for PostgreSQL, it uses the *"Prepared transaction"* statement. Thus, the DM only needs to await the results of this implicit prepare phase before proceeding to the commit phase. Then, the DM flushes the commit/abort log. If all prepared results are successful, the DM notifies involved data sources to commit; otherwise, the DM awaits the abort results from data sources. Unlike existing works on reducing round trips [15], [20], [21], GeoTP does not require modifications to the database kernel, making it suitable for database middleware with heterogeneous data sources.

As illustrated in Figure 4a, the decentralized prepare mechanism enables the completion of distributed transaction commitments in a single WAN round trip, unlike the traditional two-round trip process. The lock contention span for both subtransactions within T_1 is reduced from 200 ms (shown in Figure 2) to 100 ms, which corresponds to the longest network round trip time involved in T_1.

Early abort mechanism. In conventional DMs, subtransactions remain unaware of the execution status of their peer subtransactions, such as failures due to the lock timeout, until receiving the abort notification from the DM, resulting in resource wastage. The challenge is introduced by the inability of heterogeneous data sources to communicate directly with each other. Inspired by Guerraoui et.al. [32], GeoTP proposes the early abort mechanism to address this issue effectively. In GeoTP, the geo-agent maintains connections to other data sources in its connection pool. Once a subtransaction encounters an abort before commitment, the geo-agent proactively notifies other data sources to abort the corresponding peer subtransactions, bypassing the DM and thereby reducing half of the WAN round trip. In the previous execution process, when subtransaction T_{12} aborts in DS_2, it requires one and a half WAN round trips to abort transaction T_{11} —half a round trip for DS_2 to send the abort message to the DM and one round trip for the DM to dispatch the abort command to DS_1 and receive the abort result from DS_1. An illustration example can be found in the extended version of our paper [31].

B. Latency-Aware Scheduling Mechanism

In geo-distributed scenarios, significant differences in network latencies often lead to unnecessary lock contention spans, as we illustrated in the motivation example in Figure 2. To address this issue, we propose a latency-aware scheduling approach to optimize the start time point for each subtransaction. In this part, we assume that there is no data-conflict blocking, allowing subtransactions to acquire locks and complete their execution immediately, for the simplicity of illustration. We will incorporate the transaction execution latency in §IV-C.

Lock request timing postponing. We first formulate the lock contention span for subtransaction T_{ij} as follows:

$$LCS(T_{ij}) = \hat{t}^{T_{ij}}_{last} - \hat{t}^{T_{ij}}_{1st} \tag{1}$$

where $\hat{t}^{T_{ij}}_{1st}$ and $\hat{t}^{T_{ij}}_{last}$ represent the first lock acquisition time point and the last lock release time point of T_{ij}, respectively.

The primary objective of the DM is to minimize each subtransaction's lock contention span defined in Eq.(1). We achieve this by postponing the start time point $t^{T_{ij}}_{start}$, which is the time point when the DM dispatches subtransaction T_{ij} to the data source. For clarity and without generality, we explain our formulas under the assumptions that (1) the time point when the DM receives the transaction is 0ms, and (2) subtransactions can acquire locks immediately, meaning the first lock acquisition time point $\hat{t}^{T_{ij}}_{1st}$, can be represented as the time point the target data source receives T_{ij}. Formally, $\hat{t}^{T_{ij}}_{1st} = t^{T_{ij}}_{start} + \frac{1}{2}\tau_{ij}$, where τ_{ij} denotes the RTT between the DM and the data source where T_{ij} executes.

Since GeoTP eliminates the network round trip in the prepare phase, the last lock release time point $\hat{t}^{T_{ij}}_{last}$, can be represented as the time point when the target data source receives the *"commit"* message for T_{ij}. Formally, $\hat{t}^{T_{ij}}_{last} = \max\limits_{\forall T_{is} \in T_i} \tau_{is} + \frac{1}{2}\tau_{ij}$, with $\max\limits_{\forall T_{is} \in T_i} \tau_{is}$ representing the highest RTT from the DM to the data sources involved in T_i. Furthermore, to avoid increasing the overall transaction latency, there is a constraint that the end time point of any subtransaction's execution and prepare phase must not exceed the original end time point of the transaction's entire execution and prepare phase. Taking T_{ij} as an example, its end time point of execution and prepare phase can be represented as $t^{T_{ij}}_{start} + \tau_{ij}$,

Algorithm 1: Latency-aware scheduling mechanism

```
1  Function ScheduleTranscation($T_i$):
2      lat_max := 0, retry_cnt := 0
3      time_now := GetSystemClock()
4      for $T_{ij} \in T_i$.subtxns do
5          node := GetNode($T_{ij}$)
6          /* $\tau$ between DM and targeted data source */
7          $T_{ij}$.latency := GetNetworkLatency(node)
8      if adv_opt is true then
9          /* Further optimization §IV-C */
10         for $T_{ij} \in T_i$.subtxns do
11             p := 0, $\widehat{LEL}(T_{ij})$ := 0
12             for $r_k \in T_{ij}$.records do
13                 p := UpdatePossibility($r_k$, p) // Eq.(9)
14                 $\widehat{LEL}(T_{ij})$ := $\widehat{LEL}(T_{ij})$ + $w\_lat_{r_k}$
15             if $p < rand()$ then
16                 if retry_cnt++ < 10 then
17                     goto line 11
18                 return None /* abort */
19             else
20                 $T_{ij}$.latency := $T_{ij}$.latency + $\widehat{LEL}(T_{ij})$
21     lat_max := GetMaxSubtransactionLatency($T_i$)
22     for $T_{ij} \in T_i$.sub_txn do
23         $t_{start}^{T_{ij}}$:= time_now + (lat_max - $T_{ij}$.latency)
```

the original end time point of T_i's entire execution and prepare phase is $\max\limits_{\forall T_{is} \in T_i} \tau_{is}$. The objective function and constraint (indicated after *s.t.* in Eq.(2)) for each subtransaction are formally described as follows:

$$\begin{aligned} &\arg\min_{t_{start}^{T_{ij}}} LCS(T_{ij}) \Rightarrow \arg\min_{t_{start}^{T_{ij}}} \big(\max_{\forall T_{is} \in T_i} \tau_{is} - t_{start}^{T_{ij}}\big) \\ &\text{s.t.} \quad t_{start}^{T_{ij}} + \tau_{ij} \le \max_{\forall T_{is} \in T_i} \tau_{is} \end{aligned} \tag{2}$$

We can derive the optimal subtransaction start time to minimize each subtransaction's lock contention span in Eq.(2) as:

$$t_{start}^{T_{ij}} = \max_{\forall T_{is} \in T_i} \tau_{is} - \tau_{ij} \tag{3}$$

Notably, for transactions with multiple rounds of interactions, the optimal start time point is calculated for each round.

Algorithm. Algorithm 1 outlines the key function of the latency-aware scheduling mechanism. *ScheduleTransaction()* is invoked by the geo-scheduler and adjusts the start time point of each subtransaction. The DM iterates through each subtransaction of the input transaction T and retrieves the network latency between the DM and the target data source (lines 4-7). In this section, we assume the *adv_opt* as false, with further optimization introduced in §IV-C (lines 9-20). After that, the DM records the latency of the slowest subtransaction (line 21) and then calculates the optimal start time point for each subtransaction based on Eq.(3) (lines 22-23).

Recall the example transaction in Figure 4a, the distributed transaction T_2 arrives at 5 ms in the DM and needs to access the same record r on DS_1 with transaction T_1. Without postponing the start time point of T_{11}, T_{21} needs to wait until 105 ms, when T_{11} releases its locks. In GeoTP, as shown in Figure 4b, our geo-scheduler postpones the start time point of subtransactions T_{11} by 90 ms, T_2 can acquire the locks ahead of T_{11} and release locks before T_{11} arrives. Consequently, the lock contention span for subtransactions T_{11} and T_{12} and T_{21} are reduced to 10 ms, 100 ms and 10ms, respectively. This postponing mechanism enhances transaction concurrency, leading to an overall improvement in system performance.

C. High-Contention Workload Optimizations

The previous discussion assumed that the transactions are under low-contention workloads. However, in high-contention workloads, the lock contention span is influenced not only by the longest RTT but also by the time required for subtransactions to acquire locks. In high-contention workloads, subtransactions often cannot immediately acquire locks. Additionally, frequent transaction waits or rollbacks due to the contention waste system resources and significantly undermine the effectiveness of predicting the time required for acquiring locks. To address the aforementioned challenges, we propose a heuristic *local execution latency forecasting* mechanism to improve latency-aware scheduling using real-time statistical information. Additionally, we introduce a *late transaction scheduling* mechanism to manage access to hot records.

Hotspot statistics collecting. Since the *local execution latency forecasting* and *late transaction scheduling* rely on real-time hotspot statistics, we first introduce how GeoTP collects this information. The geo-scheduler uses the hotspot footprint to maintain statistics for hot records of data sources, including four fields: (1) w_lat_r, the weighted average latency of subtransactions completing operations on the record r; (2) t_cnt_r, the total number of transactions that have accessed the record r; (3) c_cnt_r, the number of committed transactions that have accessed the record r; (4) a_cnt_r, the number of transactions currently accessing the record r. We update these fields after the completion of each subtransaction T_{ij} within transaction T_i. Specifically, to update the w_lat_r for the record r that T_{ij} has accessed, the DM uses a weighted average approach as formulated in Eq.(4). Since the latency of T_{ij} accessing a specific record r cannot be directly collected (due to data record granularity), we calculate it using a weight $w_r = \frac{w\_lat_r}{\sum_{\forall r_k \in T_{ij}.records} w\_lat_{r_k}}$, the proportion of w_lat_r relative to the sum of access latency of all records accessed by T_{ij}. The latency of T_{ij} accessing the record r is then estimated by $LEL(T_{ij}) \cdot w_r$, with $LEL(T_{ij})$ denoting the local execution latency of T_{ij}. Then we use this latency to update w_lat_r, with the weighted update coefficient α.

$$w\_lat_r = \alpha \cdot w\_lat_r + (1 - \alpha) \cdot LEL(T_{ij}) \cdot w_r \tag{4}$$

To enhance efficiency, we organize these hot records using an AVLTree in memory, which ensures that both point and range access have a time complexity of $O(\log n)$. Additionally, we implement an LRU list to evict cold data, allowing GeoTP to dynamically update hot records during operation. This approach not only reduces the memory overhead but also minimizes the CPU overhead for latency estimation.

Local execution latency forecasting. As formulated in Eq.(5), based on the collected statistical information on hot records, we estimate the local execution latency of subtransaction T_{ij} by accumulating the value of w_lat_r of each hot record that T_{ij} need to access. To distinguish from the actual local execution latency of T_{ij}, we use $\widehat{LEL}(T_{ij})$ to represent the forecasted local execution latency.

$$\widehat{LEL}(T_{ij}) = \sum\nolimits_{\forall r_k \in T_{ij}.records} w\_lat_{r_k} \tag{5}$$

Then we incorporate the forcasted local execution latency into Eq.(1). The $\hat{t}_{1st}^{T_{ij}}$ is updated by adding $Req(r_{1st})$, the time span from the lock request to the lock acquisition on T_{ij}'s first accessing record r_{1st}. The $\check{t}_{last}^{T_{ij}}$ is updated by adding the forecasted local execution latency to the execution and prepare phase of subtransactions. The updated $\hat{t}_{1st}^{T_{ij}}$ and $\check{t}_{last}^{T_{ij}}$ are formulated in Eq.(6).

$$\begin{aligned} \hat{t}_{1st}^{T_{ij}} &= t_{start}^{T_{ij}} + \tfrac{1}{2}\tau_{ij} + Req(r_{1st}) \\ \check{t}_{last}^{T_{ij}} &= \max_{\forall T_{is} \in T_i} (\tau_{is} + \widehat{LEL}(T_{is})) + \tfrac{1}{2}\tau_{ij} \end{aligned} \tag{6}$$

Finally, we generate the new objective function and constraint for the lock contention span using Eq.(6).

$$\begin{aligned} &\arg\min_{t_{start}^{T_{ij}}} LCS(T_{ij}) \\ \Rightarrow &\arg\min_{t_{start}^{T_{ij}}} [\max_{\forall T_{is} \in T_i} (\tau_{is} + \widehat{LEL}(T_{is})) - (t_{start}^{T_{ij}} + Req(r_{1st}))] \\ \text{s.t.} \quad & t_{start}^{T_{ij}} + \tau_{ij} + \widehat{LEL}(T_{ij}) \leq \max_{\forall T_{is} \in T_i} (\tau_{is} + \widehat{LEL}(T_{is})) \end{aligned} \tag{7}$$

Since $Req(r_{1st})$ is contained in $\widehat{LEL}(T_{ij})$, it can be considered as a constant without affecting the optimal solution. Therefore, the optimal start time point can be formulated as follows:

$$t_{start}^{T_{ij}} = \max_{\forall T_{is} \in T_i} (\tau_{is} + \widehat{LEL}(T_{is})) - (\tau_{ij} + \widehat{LEL}(T_{ij})) \tag{8}$$

Moreover, discrepancies between predicted and actual latency do not always degrade performance. According to Eq.(8), if the predicted latency $\widehat{LEL}(T_{ij})$ is lower than the actual latency $LEL(T_{ij})$, Eq.(7) may not achieve the minimal value, but still perform better than execution without latency-aware scheduling. However, if $\widehat{LEL}(T_{ij})$ exceeds $LEL(T_{ij})$, performance may suffer if the delayed subtransaction becomes the new bottleneck. In cases of inaccurate runtime predictions, we can scale down the predicted latency before incorporating it into calculations to mitigate any negative impact.

Late transaction scheduling. To restrict the number of concurrent transactions on hot records and improve prediction accuracy, the DM first predicts the abort rate of transactions before distributing them to data sources, blocking those with high abort rates. Specifically, a transaction will be aborted if it cannot acquire locks on any of the records due to lock timeout. Therefore, the transaction's abort rate, denoted as $Pr(T_i)$, is equivalent to 1 minus the probability of the transaction successfully acquiring locks on all required records. This prediction is conducted with the hotspot footprint. We observe that if a transaction is blocked by another transaction with a high abort rate, other transactions waiting for the blocked transactions are also likely to be aborted. Therefore, we predict the probability that a transaction can successfully acquire the lock on a record by calculating the probability that all preceding transactions in the waiting queue can successfully acquire locks. The number of transactions in the waiting queue can be represented by $a\_cnt_{r_k} - 1$ and each transaction has a $\frac{c\_cnt_{r_k}}{t\_cnt_{r_k}}$ probability of successfully acquiring the lock on r_k without being blocked. Given that, we formalize the abort rate for transaction T_i in Eq.(9):

$$Pr(T_i) = 1 - \prod_{\forall r_k \in T_i.records} (\frac{c\_cnt_{r_k}}{t\_cnt_{r_k}})^{\max\{a\_cnt_{r_k}-1,0\}} \tag{9}$$

Algorithm. We integrate the late transaction scheduling and local execution latency forecasting mechanisms for further optimization, as illustrated in Algorithm 1. When the adv_opt is true, we consider the local execution latency via *PredictLatency()*. Specifically, the DM traverses the keys accessed by the subtransactions and predicts the abort rate and local execution latency based on Eq.(5) and Eq.(9) (lines 13, 20). Transactions with a high abort rate are blocked (line 17). Otherwise, the DM calculates the optimal start time point for each subtransaction based on the network latency and predicted local execution latency (line 23). Additionally, the transactions that have been blocked multiple times are aborted (line 18).

D. Discussion

The high-contention optimization estimates the local execution latency. When a user specifies a predicate on the primary key or a key from a secondary index, we can identify hot records cached in the memory that match the predicate, enabling us to estimate the local execution latency of subtransactions. In certain cases, such as when there is no index on the predicate key, inferring hot records from the statements becomes challenging and inefficient, which may limit the effectiveness of this technique. However, other optimizations continue to enhance performance. Additionally, dependencies between operations from different subtransactions may necessitate multiple rounds of interactions. However, within each round, the geo-scheduler can optimize the scheduling of query execution. Due to space limitations, we discuss the integrity constraint problems in our extended versions [31].

V. Correctness and Recovery

In this section, we first describe the failure recovery mechanism of GeoTP. Then, we provide proofs of the atomicity and isolation correctness in GeoTP.

A. Failure Recovery

The failure recovery process of GeoTP includes three key aspects: (1) identifying which transactions require recovery, (2) determining where to collect the necessary information for recovery, and (3) deciding how to recover these transactions. We discuss the recovery process for both the DM failures and data source failures. Note that our recovery process relies on the following common settings in popular databases [11], [12]: ❶ if the DM fails and disconnects, the data sources abort all subtransactions not completed the prepare phase; and ❷ if a data source fails, it automatically aborts subtransactions that have not completed the prepare phase after it restarts.

When the DM encounters a failure, GeoTP would abort uncommitted transactions that have not entered the commit phase and then complete uncommitted transactions that have once it restarts. On the other hand, when a data source, such as DS_i, fails along with its geo-agent. In this case, the DM is responsible for recovering all uncommitted distributed transactions that accessed this failed data source. The DM aborts the distributed transaction if its subtransaction in DS_i has not completed the prepare phase. Otherwise, the DM continues to execute the distributed transaction after the data source is reconnected. The detailed recovery process can be found in our extended versions [31].

B. Atomicity Correctness

To ensure atomicity, we must guarantee that the final state of the transaction is either committed or aborted, with all subtransactions achieving the same status. GeoTP guarantees the atomicity correctness following two steps. First, GeoTP initiates the decentralized prepare phase after the execution phase. A transaction can be committed only if all subtransactions complete the prepare phase and vote *Yes*; otherwise, the transaction will be aborted. Second, once the final status of the transaction is determined, GeoTP flushes the commit/abort log into the disk, ensuring that the decision cannot be reversed. In the case of failures, we ensure that the DM and the underlying data sources eventually reach a unique and consistent decision after failure recovery (§V-A).

C. Isolation Correctness

GeoTP is an effective distributed transaction processing approach that can be applied to multiple middleware systems. GeoTP postpones the execution of subtransactions but does not modify the concurrency control algorithm, thereby maintaining the isolation properties of the original middleware system. Take the 2PL algorithm as an example, latency-aware scheduling postpones the acquisition of locks but still requires the acquisition of a lock before its operation, as enforced by the concurrency control mechanism of each data source. The commit protocol of GeoTP ensures that a transaction can release locks only after commitment, thereby preserving serializability. Thus, GeoTP does not compromise the isolation guarantees the original database middleware provided.

VI. IMPLEMENTATION

We implement GeoTP on the codebase of Apache Shardingsphere-v5.0 [26], a popular open-source DM, involves about 5k lines of Java code modifications and is available at [31]. GeoTP does not require any modifications to data sources. As a result, databases supported by Shardingsphere (possibly other DMs) can leverage the capabilities of GeoTP.

Database middleware layer. First, we enhance the *sqlParse()* to recognize the annotation code provided by the client and convey this information via *QueryContext*. Then, we enhance the statement handler to support latency-aware scheduling facilitated by the geo-scheduler. The statistical data required for scheduling is derived from two sources: (1) a dedicated thread that continuously monitors the network latency between the DM and data sources, utilizing the *ping* command at 10ms intervals, and (2) the hotspot footprint is recorded by *LockMetaTable* and updated by *MultiStatementsHandler.feedback()*. Lastly, we implement the decentralized prepare mechanism in *XAShardingSphereTransactionManager.commit()* to eliminate one WAN round trip for distributed transactions.

Data source layer. We have implemented the geo-agent in the data source layer, which incorporates an enhanced connection pool and a local transaction manager. The connection pool is designed to interface with other geo-agents, thereby supporting functionalities such as *SendRollbackMsg()* and *SendAsyncMsg()*. The transaction information is systematically organized within the local transaction manager using a *ConcurrentHashMap*. Before the return of *CommandExecutorTask.run()*. We invoke the *AsyncPrepare()* in an asynchronous thread upon identifying that the 'last' flag is true and then start the prepare phase of the corresponding subtransaction.

VII. PERFORMANCE EVALUATION

A. Experiment Setup

We conduct experiments on an in-house cluster of up to 6 separate machines, each equipped with 16 vCPUs and 32 GB of DRAM, running CentOS 7.4.

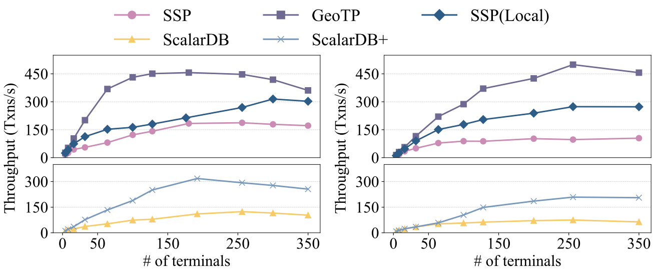
1) Baselines: In our experiments, we compare GeoTP with state-of-the-art DMs and a distributed database.

Database middlewares. ① Shardingsphere (abbreviated as SSP) is a state-of-the-art DM that supports geo-distributed transaction processing over relational database systems via XA protocol. ② SSP (local), a mode provided by SSP without atomicity guarantees, which we employ to show the peak performance of SSP. Succinctly, it employs a decentralized commit protocol but allows transactions to be committed when data sources return different votes. ③ ScalarDB [5], another state-of-the-art DM, supports geo-distributed transaction processing without specific requirements for underlying data sources. ④ ScalarDB+, a variant of ScalarDB, we implemented by integrating the latency-aware transaction scheduling mechanism and the heuristic optimization. We use ScalarDB+ to study the scalability of our proposed approach.

Distributed databases. ⑤ YugaByteDB, an advanced distributed database that supports intelligent data partitioning and geo-distributed transactions.

Transaction scheduling techniques. ⑥ QURO, a transaction preprocessing technique, reorders write operations as late as possible within the transaction. ⑦ Chiller, a distributed transaction protocol, eliminates the prepare phase and schedules the cross-region subtransactions after the intra-region ones are complete. For a fair comparison, we implemented both techniques on the same platform as GeoTP.

2) Benchmarks: We adopt the following two benchmarks. **YCSB** [33] generates synthetic workloads that simulate large-scale internet applications. We use the YCSB transactional



(a) Scalability - YCSB (b) Scalability - TPC-C

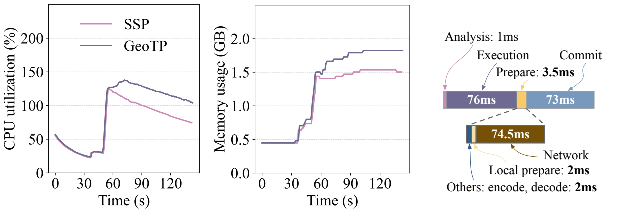
Fig. 5: Overall performance comparison

variant adopted in related works [15], [34], where each transaction has 5 operations by default, each with a 50% probability of being a read or write. We run the workloads on a table partitioned with 1 million records per data node. Each record consumes 1KB, culminating in a total of 4GB of data hosted by the table. We control the distribution of accessed records using the parameter *skew_factor*, where a higher *skew_factor* results in greater contention. We set the skew factor to 0.3, 0.9, and 1.5 for low, medium, and high-contention workloads. **TPC-C** [35] is a popular OLTP benchmark modeling a warehouse order processing application. The workloads consist of 9 relations, with each warehouse being 100 MB in size. By default, each data node hosts 16 warehouses. In our experiments, we use the standard TPC-C with 5 types of transactions by default. Following previous works [36], [37], we exclude 'think time' and user data errors that cause 1% of NewOrder transactions to abort.

3) Default Configuration: We use one machine to serve as the client, generating transaction requests by Benchbase [38]. By default, we run 64 client terminals. For YCSB, we use the medium contention by default. The default ratio of distributed transactions is set to 0.2. For the remaining 5 machines, we deploy database middleware on 1 machine, while the other 4 machines are data nodes. We emulate the default geo-distributed network environment via *tc* command [39]. The client, GeoTP, and a data node are located in Beijing, with the other data nodes in Shanghai, Singapore, and London. Based on the network evaluation conducted in ECS [40], the average latency between the corresponding data nodes and GeoTP are 0ms, 27ms, 73ms, and 251ms, respectively. Unless otherwise specified, we use the default settings to conduct our experiments. The remaining 4 nodes host MySQL v8.0.22 and PostgreSQL v15.2 as data nodes. Except for §VII-F, where we specify otherwise, all data nodes run MySQL v8.0.22 as data sources. By default, we set the isolation level to serializable and configure the buffer pool size to 24GB and the lock-wait timeout to 5s. The rewriter in middleware will replace *SELECT* with *SELECT...FOR SHARE* to add an explicitly shared lock for PostgreSQL's read operations.

B. Overall Performance of GeoTP

We first compare GeoTP with state-of-the-art DMs using YCSB and TPC-C benchmarks with varying numbers of client terminals. As shown in Figure 5, GeoTP outperforms SSP(Local), SSP, and ScalarDB by up to 2.65x, 5.14x, and 7.15x, respectively. This throughput improvement is attributed to the latency-aware and late transaction scheduling of GeoTP, which effectively reduces lock contention span and enhances concurrency. ScalarDB, on the other hand, does not rely on the transactional capabilities of underlying data sources but solely on DM nodes for concurrency control, which limits its scalability and performance. Moreover, we can observe that ScalarDB+ achieves up to 3.16x and 3.22x throughput gain over ScalarDB under YCSB and TPC-C, respectively. This demonstrates the general applicability of the proposed techniques in GeoTP. As the number of terminals increases, all approaches experience a decline in performance. This decline is attributed to system resource competition and lock contention within the databases.



(a) CPU (b) Memory (c) Breakdown

Fig. 6: Resource Utilization over YCSB

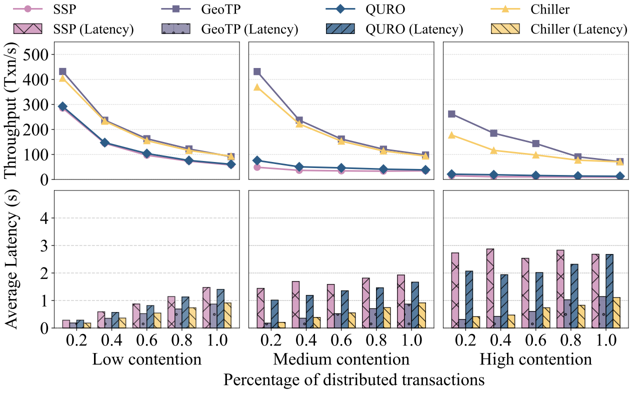


Fig. 7: Impact of distributed transactions over YCSB

We analyze the overhead of GeoTP, with results shown in Figure 6. Figure 6a presents CPU utilization, where GeoTP achieves around 30% higher CPU efficiency due to network latency detection and latency-aware scheduling. Figure 6b shows memory usage, with GeoTP requiring approximately 300MB more than SSP as it maintains metadata for hot records in memory. Lastly, Figure 6c provides a breakdown of module costs in a single transaction lifecycle. Here, analysis overhead remains under 1 ms, with a 3.5 ms wait to enter the commit phase, facilitated by the decentralized prepare mechanism. Compared to the entire transaction latency, the overhead incurred by GeoTP is negligible, while GeoTP achieves 66.6% lower average latency compared to SSP.

C. Impact of Distributed Transactions

We now compare GeoTP against other baselines by varying the percentage of distributed transactions.

YCSB: We control the ratio of distributed transactions by generating keys assigned to different data nodes. We evaluate

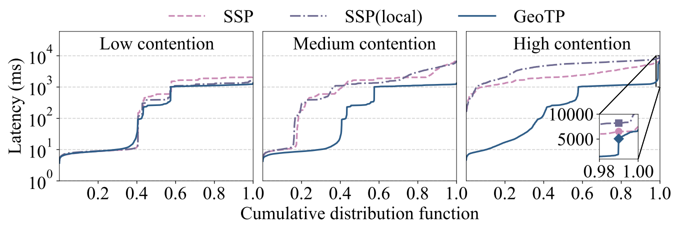
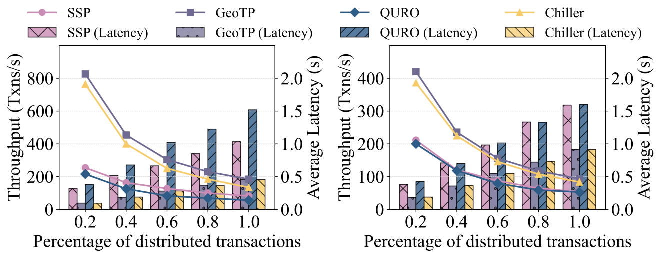


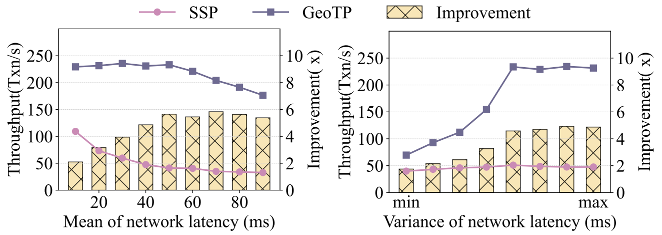
Fig. 8: Analysis of latency CDF over YCSB



(a) Payment

(b) Neworder

Fig. 9: Impact of distributed transactions over TPC-C



(a) Fixed STD

(b) Fixed mean

Fig. 10: Impact of network latency configurations over YCSB

GeoTP under three levels of contention as used in Figure 7. GeoTP outperforms in all three scenarios. As the proportion of distributed transactions increases, although the performance of both systems declines, the advantages of GeoTP are still pronounced. GeoTP outperforms Chiller up to 1.6x and other baselines up to 8.9x. The evaluation results show that while QURO performs better than SSP, it still falls short compared to other methods because it doesn't consider network latency, making it unsuitable for geo-distributed scenarios. Chiller merges the prepare phase with execution and executing inner-region subtransactions after outer-region ones. This approach reduces the lock contention span in low and medium workloads, allowing Chiller to achieve comparable performance. However, GeoTP includes specific optimizations for high-contention workloads, further enhancing its performance. It is aligned with our findings discussed in § VII-E.

We further analyze the latency distribution of transactions (with 60% of distributed transactions) in three scenarios using Cumulative Distribution Function (CDF) plots, as depicted in Figure 8. GeoTP consistently reduces the latency of distributed transactions across all three scenarios. We use **turning point** to describe the point where transaction latency experiences a significant increase. In the low-contention workload, with a turning point at 0.4, the latency of 40% centralized transactions remains unaffected by distributed transactions. As contention increases in the medium-contention workload, the turning point of SSP and SSP(local) occurs at about 0.2, which suggests that around 20% of the execution latency in centralized transactions experiences an increase due to the distributed transaction. In contrast, the turning point of GeoTP remains around 0.4, with a 99th-percentile (p99) latency lower than the baselines up to 35.9%. This improvement is due to our latency-aware scheduling, which mitigates lock contention and lessens the impact of distributed transactions on centralized transactions. In high-contention workloads, SSP and SSP(local) show a turning point near 0, reflecting severe latency spikes in centralized transactions. However, GeoTP exhibits no distinct turning point; instead, its latency rises gradually and remains considerably lower overall. Although GeoTP maintains lower p99 latency, its p99.9 advantage diminishes due to the optimization in § IV-C, which can increase latency by introducing blocks and aborts.

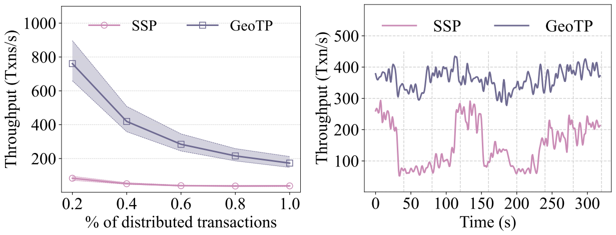
TPC-C: We control the ratio of distributed transactions in Payment and NewOrder by generating warehouseIDs and itemIDs in different data nodes. Figure 9 demonstrate that GeoTP achieves 2.81x (2.04x) higher throughput and a 66.6% (53%) reduction in latency for Payment (NewOrder) transactions. It slightly outperforms Chiller due to the relatively low contention levels in our TPC-C workloads. Compared to other baselines, the performance improvement for NewOrder transactions is less significant than for Payment transactions, as Payment transactions tend to incur more contention. This observation indicates the effectiveness of GeoTP's heuristic optimization for high-contention workloads.

D. Impact of Dynamic Network Latency

In this subsection, we evaluate GeoTP using YCSB under various network configurations, simulating by adjusting the network latency between the DM and data sources.

Mean and standard deviation: Figure 10 shows the result by varying mean and variance of network latency. For example, when the mean latency is set to 20ms, the latencies between the middleware and three data nodes are 10ms, 20ms, and 30ms, respectively. First, by fixing the standard deviation of network latency from DM to data sources, as the average latency increases, both GeoTP and SSP throughputs decrease, but the relative advantage of GeoTP over SSP becomes more pronounced. Then, by fixing the network mean between nodes, as the standard deviation increases, SSP performance remains relatively unchanged, while GeoTP's performance continues to improve. This indicates that if only a few machines experience occasional latency spikes, it has a significant and severe impact on SSP but has minimal impact on GeoTP.

Random latency: We conduct experiments to evaluate GeoTP with random network latencies by YCSB. As shown in Figure 11a, the solid line represents the average performance of running the experiment 20 times, while the portions filled with the shadow indicate the performance variations under different network environments. GeoTP outperforms SSP in all scenarios with distributed transaction ratios ranging from 0.2 to 1.0. Further, when the network latency randomly fluctuates by a factor of 1.5 for some nodes, the performance jitter remains



(a) Random latency (b) Dynamic latency

Fig. 11: Impact of random network latency over YCSB

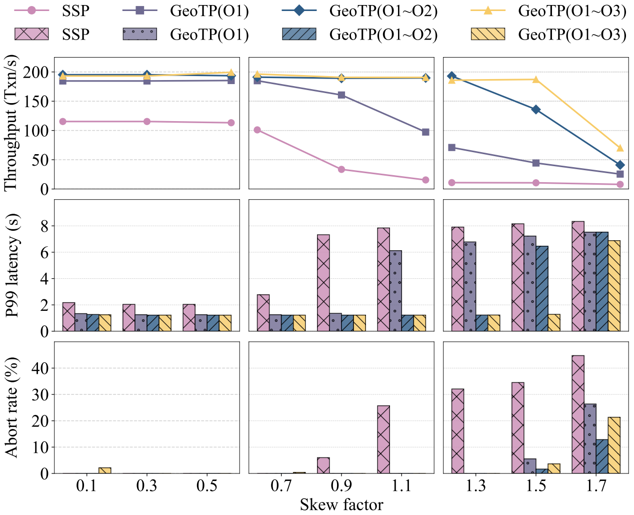


Fig. 12: Impact of optimizations over YCSB

within 22.5%. In the medium-contention workload, the average performance gains range from 4.5x to 9.1x.

Online adaptivity: We evaluate GeoTP with an online dynamic network, adjusting the network latency every 40 seconds over a 320-second period. In Figure 11b, GeoTP outperforms SSP in all scenarios and exhibits less sensitivity to dynamic network environments compared to SSP. This capability of GeoTP is attributed to its real-time network monitoring and latency-aware scheduling. In GeoTP, we utilize the exponential weighted moving average algorithm [41] when we update the network latency. This helps GeoTP balance temporary impacts and changes in trends. Over time, GeoTP demonstrates performance improvements ranging from 1.1x to 10.5x.

E. Ablation Study

We now study the effectiveness of the three optimizations: (1) **O1**: the decentralized prepare mechanism as detailed in §IV-A, (2) **O2**: the latency-aware scheduling mechanism as detailed in §IV-B, and (3) **O3**: the high-contention workload optimization as elaborated in §IV-C. Then, we use both O1 and O2 in **GeoTP (O1 $\sim$ O2)**. Similarly in **GeoTP (O1 $\sim$ O3)**, we use O1, O2 and O3. We compare GeoTP and SSP with 50% distributed transactions and a variety of skew factors (theta) as shown in Figure 12. The x-axis is partitioned into three segments denoting low (theta: 0.1 $\sim$ 0.5), medium (theta: 0.7 $\sim$ 1.1), and high (theta: 1.3 $\sim$ 1.7) contention scenarios. On the other hand, the y-axis illustrates throughput, p99 latency, and abort rate. The results demonstrate that GeoTP achieves significantly higher throughput, reaching up to 17.7x greater than SSP. Meanwhile, GeoTP reduces the abort rate by up to 32.1% and p99 latency up to 84.3% when compared to SSP.

TABLE I: Impact of heterogeneous deployments with various distributed transaction ratios (dr) over YCSB – Throughput (Txn/s) and Average latencies (ms)

		dr=25%		**dr=75%**	
		Throughput	Average latency	Throughput	Average latency
S1	**SSP**	58.7	1441.8	33.3	1815.8
	GeoTP	437.8	176.0	123.5	689.6
S2	**SSP**	74.0	1069.9	35.5	2192.9
	GeoTP	340.7	220.8	131.8	650.1
S3	**SSP**	70.3	901.8	25.2	2112.6
	GeoTP	425.5	198.8	116.6	632.3

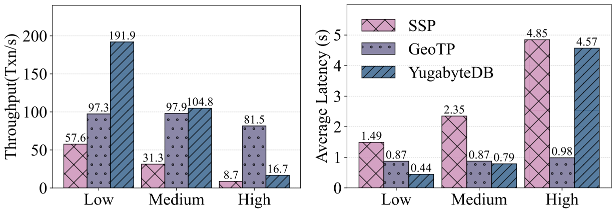
GeoTP outperforms SSP in all scenarios, while the effectiveness of each optimization varies across different contentions. In low-contention workloads, the performance gains from O1 $\sim$ O3 are not particularly advantageous compared to O1 alone. In this case, the execution latency of a transaction primarily consists of network latency. Meanwhile, the abort ratio and p99 latency are low for all approaches. In medium-contention workloads, both GeoTP (O1) and SSP decline. In this case, transactions exhibit more data dependencies, and a transaction's execution latency comprises both network latency and local execution latency. O2 reduces the contention span and improves the concurrency. SSP's abort rate rises due to prolonged blocking time, resulting in lock wait timeouts. In high-contention workloads, the performance of all methods declines significantly due to critical lock waits, highlighting the insufficiency to consider only network latency when scheduling transactions. However, the degradation of GeoTP (O1$\sim$O3) is minimal, with its p99 latency remaining the lowest, due to O3's ability to partially incorporate execution latency into scheduling while restricting access to hot records. The abort rate for GeoTP (O1$\sim$O3) is lower than both SSP and GeoTP (O1) but slightly higher than GeoTP (O1$\sim$O2) because O3 mitigates lock contention by selectively blocking or aborting transactions.

F. Impact of Heterogeneous Databases

We now evaluate the performance of GeoTP when deployed on heterogeneous data sources, i.e., either MySQL or PostgreSQL. We denote those data nodes as $N_1 \sim N_4$, respectively. We consider three scenarios: (**S1**) MySQL is deployed on nodes $N_1 \sim N_4$; (**S2**) PostgreSQL is deployed on nodes N_1 & N_3, and MySQL is deployed on nodes N_2 & N_4; (**S3**) PostgreSQL is deployed on nodes $N_1 \sim N_4$. As observed in Table I, GeoTP outperforms baselines in all deployments. The throughput improvement ranges from 3.6x to 7.5x, with the average latency reduction varying between 62% and 87.8%. MySQL and PostgreSQL suffer from the long lock contention span in geo-distributed scenarios, and GeoTP can improve the performance in these deployments.

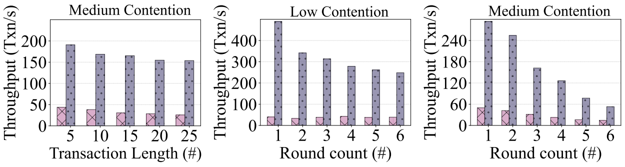
G. Comparison with YugabyteDB

We compare the performance of GeoTP with YugabyteDB, an advanced distributed database. To ensure a fair comparison, we deploy YugabyteDB across 4 data nodes and partition the



(a) Throughput (b) Average latency

Fig. 13: Comparison with YugabyteDB over YCSB



(a) Length (b) Round (c) Round

Fig. 14: Impact of transaction length over YCSB

data. We measure the performance over YCSB with varying contention levels. As shown in Figure 13, GeoTP achieves a 4.88x improvement in the high-contention workload due to the proposed latency-aware scheduling and heuristic optimization. In medium-contention workloads, GeoTP is on par with YugabyteDB. However, YugabyteDB outperforms GeoTP in low contention, due to its ability to perform data updates asynchronously for single-row/single-shard transactions after commitment. While GeoTP does not modify the underlying data source, thus typically lacking this optimization. In low-contention workloads, where transaction contentions no longer dominate performance, the performance gap due to these fundamental codebase differences of GeoTP and YugabyteDB becomes more apparent. However, GeoTP can benefit from the asynchronous update if supported by the underlying data source, making this technique orthogonal to our proposal.

H. Impact of Transaction Length

We now study the impact of the transaction length and interactive rounds. We evaluate GeoTP and baseline in medium contention workloads with 20% distributed transaction. Figure 14 shows results for two settings. First, we examine the performance of fixed one-interaction round transactions while adjusting transaction length. As seen in Figure 14a, throughput for both GeoTP and SSP decreases by 19.1% and 41.3% as the length increases from 5 to 25. It remains relatively stable compared to the number of interaction rounds. Next, we vary the number of interaction rounds and evaluate GeoTP in both low- and medium-contention workloads. As shown in Figure 14b and 14c, with 6 interaction rounds, GeoTP outperforms SSP by 1.5x in low-contention and 3.4x in medium-contention environments. This indicates that network round trip is the primary bottleneck. As the number of rounds increases, the advantages of the decentralized prepare mechanism decrease, while latency-aware scheduling and high-contention optimizations continue to provide performance gains.

VIII. Related Work

Database middleware techniques. Substantial efforts have been dedicated to enhancing transactional capability across heterogeneous databases. For example, Skeena [42] efficiently integrates different engines within the same database system, and each engine operates autonomously. It identifies disparities in engine processing capabilities and employs a snapshot map in shared memory to ensure isolation. In contrast, some middlewares [5], [43], [44] implement transaction management and concurrency control over the abstractions of underlying engines, making it extendable to more kinds of engines, including NoSQLs. Other solutions focus on managing connections between databases and clients [45]–[48]. These solutions enable the routing of statements to one or multiple database servers, thereby offering high performance. GeoTP is designed for database middleware with geo-distributed data sources. These techniques are orthogonal to our contributions.

Other distributed transaction processing techniques. Except for the studies [23]–[25] in Section IV-A, there are some works explore the scheduling and locking techniques in geo-distributed scenarios. Some approaches focus on reducing network round trips in conventional networks [13]–[15], [49]. For instance, Multi-level 2PC [50] reduces costly WAN communication by organizing participants hierarchically, though it incurs higher LAN coordination overhead. Another line of research aims to reduce lock contention by enforcing partial or full determinism in concurrency control. Calvin [51] and Detock [52] use a global agreement scheme to sequence lock requests deterministically. Deterministic techniques require a priori knowledge of read-set and write-set. Moreover, methods mention above involve significant modifications to database system or kernel-level transaction protocol, which limits their applicability in database middleware. In contrast, GeoTP is a lightweight approch that reduce WAN communication cost and lock contention by accounting for differential network latency—an aspect often overlooked by above approaches.

IX. Conclusion

In this paper, we present GeoTP, a latency-aware geo-distributed transaction processing in database middlewares without modifying the database kernels. The core idea of GeoTP is to minimize latency and reduce the lock contention span of distributed transactions. To achieve this, we introduce a decentralized prepare mechanism, which eliminates one WAN round trip for each distributed transaction. Furthermore, we present a latency-aware scheduling approach that postpones the lock acquisition time for some subtransactions. Lastly, we enhance latency-aware scheduling with heuristic optimizations for high-contention workloads. Extensive experiments on YCSB and TPC-C show that GeoTP outperforms baselines. Compared to other distributed databases, GeoTP performs comparably in medium-contention workloads and excels in high-contention workloads.

ACKNOWLEDGE

This work was supported by the National Natural Science Foundation of China (Grant Nos. 62472429 and 6241101193). Wei Lu is the corresponding author.

References

[1] P. Bharadwaj, S. Kumar, A. Raj, and M. Hudnurkar, *Role of Database Management in E-Commerce Firms*, ch. Chapter 15, pp. 297–313.

[2] "Online banking," https://en.wikipedia.org/wiki/Online_banking, 2023.

[3] "Where does tiktok store u.s. user data?" https://usds.tiktok.com/where-does-tiktok-store-u-s-user-data/, 2023.

[4] "Shardingsphere," https://github.com/apache/shardingsphere, 2023.

[5] H. Yamada, T. Suzuki, Y. Ito, and J. Nemoto, "Scalardb: Universal transaction manager for polystores," *Proc. VLDB Endow.*, vol. 16, no. 12, pp. 3768–3780, 2023.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, p. 8, 2013.

[7] "Yugabytedb," https://github.com/yugabyte/yugabyte-db, 2023.

[8] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "Cockroachdb: The resilient geo-distributed SQL database," in *SIGMOD Conference*. ACM, 2020, pp. 1493–1509.

[9] "Powered by shardingsphere," https://shardingsphere.apache.org/community/en/powered-by/, 2023.

[10] "How neo4j reduced churn by moving renewals out of salesforce," https://retool.com/customers/neo4j, 2023.

[11] "mysql-server," https://github.com/mysql/mysql-server.git, 2023.

[12] "Postgresql: The world's most advanced open source relational database," https://www.postgresql.org/, 2023.

[13] S. Maiyya, F. Nawab, D. Agrawal, and A. E. Abbadi, "Unifying consensus and atomic commitment for effective cloud data management," *Proc. VLDB Endow.*, vol. 12, no. 5, pp. 611–623, 2019.

[14] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *SOSP*. ACM, 2015, pp. 263–278.

[15] Q. Zhang, J. Li, H. Zhao, Q. Xu, W. Lu, J. Xiao, F. Han, C. Yang, and X. Du, "Efficient distributed transaction processing in heterogeneous networks," *Proc. VLDB Endow.*, vol. 16, no. 6, pp. 1372–1385, 2023.

[16] P. Antonopoulos, A. Budovski, C. Diaconu, A. H. Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, V. Purohit, H. Qu, C. S. Ravella, K. Reisteter, S. Shrotri, D. Tang, and V. Wakade, "Socrates: The new SQL server in the cloud," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 1743–1756.

[17] C. Barthels, I. Müller, K. Taranov, G. Alonso, and T. Hoefler, "Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores," *Proc. VLDB Endow.*, vol. 12, no. 13, pp. 2325–2338, 2019.

[18] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," *Proc. VLDB Endow.*, vol. 10, no. 7, pp. 781–792, 2017.

[19] J. W. Stamos and F. Cristian, "A low-cost atomic commit protocol," in *SRDS*. IEEE Computer Society, 1990, pp. 66–75.

[20] X. Yan, L. Yang, H. Zhang, X. C. Lin, B. Wong, K. Salem, and T. Brecht, "Carousel: Low-latency transaction processing for globally-distributed data," in *SIGMOD Conference*. ACM, 2018, pp. 231–243.

[21] L. Yang, X. Yan, and B. Wong, "Natto: Providing distributed transaction prioritization for high-contention workloads," in *SIGMOD Conference*. ACM, 2022, pp. 715–729.

[22] S. Mu, L. Nelson, W. Lloyd, and J. Li, "Consolidating concurrency control and consensus for commits under conflicts," in *OSDI*. USENIX Association, 2016, pp. 517–532.

[23] C. Yan and A. Cheung, "Leveraging lock contention to improve OLTP application performance," *Proc. VLDB Endow.*, vol. 9, no. 5, pp. 444–455, 2016.

[24] E. Zamanian, J. Shun, C. Binnig, and T. Kraska, "Chiller: Contention-centric transaction execution and data partitioning for modern networks," in *SIGMOD Conference*. ACM, 2020, pp. 511–526.

[25] X. Chen, H. Song, J. Jiang, C. Ruan, C. Li, S. Wang, G. Zhang, R. Cheng, and H. Cui, "Achieving low tail-latency and high scalability for serializable transactions in edge computing," in *EuroSys*. ACM, 2021, pp. 210–227.

[26] R. Li, L. Zhang, J. Pan, J. Liu, P. Wang, N. Sun, S. Wang, C. Chen, F. Gu, and S. Guo, "Apache shardingsphere: A holistic and pluggable platform for data sharding," in *ICDE*. IEEE, 2022, pp. 2468–2480.

[27] "Citus," https://docs.citusdata.com/en/v12.0/, 2023.

[28] "Mysql cluster," http://www.mysql.com/products/cluster, 2023.

[29] "Mysql :: Mysql 8.0 reference manual :: 10.9.3 optimizer hints," https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html, 2024.

[30] "Postgresql: Documentation: 17: Chapter 14. performance tips," https://www.postgresql.org/docs/current/performance-tips.html, 2024.

[31] "Supplementary material of geotp," https://github.com/dbiir/GeoTP.git, 2024.

[32] R. Guerraoui and A. Schiper, "The decentralized non-blocking atomic commitment protocol," in *SPDP*. IEEE, 1995, pp. 2–9.

[33] A. Dey, A. D. Fekete, R. Nambiar, and U. Röhm, "YCSB+T: benchmarking web-scale transactional databases," in *ICDE Workshops*. IEEE Computer Society, 2014, pp. 223–230.

[34] Z. Zhao, H. Zhao, Q. Zhuang, W. Lu, H. Li, M. Zhang, A. Pan, and X. Du, "Efficiently supporting multi-level serializability in decentralized database systems," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12 618–12 633, 2023.

[35] "Tpc-c," http://www.tpc.org/tpcc/, 2023.

[36] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 209–220, 2014.

[37] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 553–564, 2017.

[38] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proc. VLDB Endow.*, vol. 7, no. 4, pp. 277–288, 2013.

[39] "tc(8) - linux manual page," https://man7.org/linux/man-pages/man8/tc.8.html, 2024.

[40] "Elastic compute service (ecs): Elastic secure cloud servers - alibaba cloud," https://www.alibabacloud.com/en/product/ecs, 2023.

[41] J. Kurose and K. Ross, "Computer networks: A top down approach featuring the internet," *Peorsoim Addison Wesley*, 2010.

[42] J. Zhang, K. Huang, T. Wang, and K. Lv, "Skeena: Efficient and consistent cross-engine transactions," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 34–48.

[43] A. Dey, A. D. Fekete, and U. Röhm, "Scalable distributed transactions across heterogeneous stores," in *ICDE*. IEEE Computer Society, 2015, pp. 125–136.

[44] P. Kraft, Q. Li, X. Zhou, P. Bailis, M. Stonebraker, X. Yu, and M. Zaharia, "Epoxy: ACID transactions across diverse data stores," *Proc. VLDB Endow.*, vol. 16, no. 11, pp. 2742–2754, 2023.

[45] "Pgbouncer - lightweight connection pooler for postgresql," https://www.pgbouncer.org, 2023.

[46] "Proxysql - a high performance open source mysql proxy," https://proxysql.com, 2023.

[47] "Maxscale," https://mariadb.com/kb/en/maxscale/, 2023.

[48] M. Butrovich, K. Ramanathan, J. Rollinson, W. S. Lim, W. Zhang, J. Sherry, and A. Pavlo, "Tigger: A database proxy that bounces with user-bypass," *Proc. VLDB Endow.*, vol. 16, no. 11, pp. 3335–3348, 2023.

[49] J. A. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination," in *USENIX ATC*. USENIX Association, 2012, pp. 223–235.

[50] C. Mohan, B. G. Lindsay, and R. Obermarck, "Transaction management in the r* distributed database management system," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 378–396, 1986.

[51] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *SIGMOD Conference*. ACM, 2012, pp. 1–12.

[52] C. D. T. Nguyen, J. K. Miller, and D. J. Abadi, "Detock: High performance multi-region transactions at scale," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 148:1–148:27, 2023.