


# Efficiently Supporting Multi-Level Serializability in Decentralized Database Systems

Zhanhao Zhao , Hongyao Zhao, Qiyu Zhuang, Wei Lu , Haixiang Li, Meihui Zhang , *Senior Member, IEEE*, Anqun Pan , and Xiaoyong Du , *Member, IEEE*

**Abstract**—In decentralized database systems, it is reported that serializability could still produce unexpected transaction orderings, leading to the stale read anomaly. To eliminate this anomaly, strict serializability imposes an additional ordering constraint, called the real-time order, which is required to be preserved among serializable transactions. Yet, preserving the real-time order in strict serializability often causes the performance to drop significantly. Because a weaker data consistency often yields better performance, in this paper, we model serializability from different consistency perspectives to properly leverage the performance and consistency. To do this, we first define a group of orderings, based on which we formulate multi-level serializability by preserving a certain set of ordering constraints among transactions. We then propose a bidirectional timestamp adjustment algorithm (abbreviated as BDTA) to support multi-level serializability with various optimizations. Our special design makes ordering constraints among transactions be preserved simply by adjusting timestamp intervals. Finally, we conduct extensive experiments to show the necessity of introducing multi-level serializability and confirm that BDTA achieves up to  $1.19 \times$  better performance than the state-of-the-art concurrency control algorithms.

**Index Terms**—Database, transactions, serializability, concurrency control.

## I. INTRODUCTION

DECENTRALIZED database systems [37] like Google Spanner [14], CockroachDB [44], and TiDB [25] have become increasingly popular to support large-scale web applications. In these systems, each coordinator individually coordinates transactions, each of which reads/writes data from a snapshot using a given timestamp. Due to inconsistent local

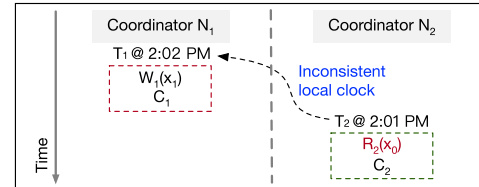


Fig. 1. The execution is serializable. However, due to two coordinators' inconsistent local clocks, a stale read anomaly  $R_2(x_0)$  occurs in  $T_2$ : reading stale data  $x_0$ , highlighted in red. Symbol  $W_i(x_i)$  represents a write by  $T_i$  on version  $x_i$  of data item  $x$ ,  $R_i(x_j)$  signifies a read by  $T_i$  on version  $x_j$  written by transaction  $T_j$ , and  $C_i$  denotes the commit of  $T_i$ .

clocks across coordinators, recent studies [16], [40] show serializability still produces unexpected transaction orderings that make transactions read stale data.

*Example 1:* Consider a user who deposits money via ATM by submitting transaction  $T_1$ . To confirm the deposit, the user subsequently checks the account balance  $x$  via an online banking service by  $T_2$ . Because  $T_2$  starts after  $T_1$  is accomplished, the user expects to observe the balance  $x$  written by  $T_1$ . However, as shown in Fig. 1,  $T_1$ 's write cannot be "seen" by  $T_2$ , leading to a stale read  $R_2(x_0)$  of  $T_2$ . The reason is that different coordinators execute  $T_1$  and  $T_2$  with inconsistent local clocks, i.e., the snapshot (2:01 PM) of  $T_2$  is earlier than the commit timestamp (2:02 PM) of  $T_1$ .  $\square$

The *real-time order* is first introduced in the linearizability consistency level [7], meaning that if one operation  $op_1$  starts after another operation  $op_2$  is accomplished, then  $op_1$ 's read must observe  $op_2$ ' write. Strict serializability [10], [24], [40] imposes the real-time order on serializability by extending the operation granularity to transaction granularity. Hence, it eliminates the stale read anomaly in serializable transactions, making  $T_2$  observe  $T_1$ ' write, i.e., read  $x_1$ , in Example 1. Thus far, preserving the real-time order can only be implemented by either (1) timestamp oracle [38] or (2) TrueTime [14]. When using timestamp oracle, each transaction is assigned with globally ordered timestamps, and hence every two transactions are comparable. For this method, however, obtaining the timestamps suffers from high network latency overhead and could become a bottleneck [8], [50]. TrueTime requires customized hardware like atomic clocks to avoid using the timestamp oracle but incurs expensive blocking overhead (i.e., commit-wait) to preserve the real-time order. For example, in the commit-wait scheme, transactions would wait for 4 ms to commit, leading to significant performance degradation.

Manuscript received 7 March 2022; revised 2 May 2023; accepted 16 May 2023. Date of publication 19 May 2023; date of current version 8 November 2023. This work was supported in part by the National Natural Science Foundation of China under Grants 61972403 and 61732014, and in part by Tencent Rhino-Bird Joint Research Program. Recommended for acceptance by Y. Tong. (Corresponding authors: Wei Lu; Xiaoyong Du.)

Zhanhao Zhao, Hongyao Zhao, Qiyu Zhuang, Wei Lu, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education China, School of Information, Renmin University of China, Beijing 100872, China (e-mail: zhanhaozhao@ruc.edu.cn; hongyaozhao@ruc.edu.cn; qyzhuang@ruc.edu.cn; lu-wei@ruc.edu.cn; duyong@ruc.edu.cn).

Haixiang Li and Anqun Pan are with the Tencent Inc., Shenzhen, Guangdong Province 518054, China (e-mail: bluesiali@tencent.com; aaronpan@tencent.com).

Meihui Zhang is with the Beijing Institute of Technology, Beijing 100081, China (e-mail: meihui\_zhang@bit.edu.cn).

Digital Object Identifier 10.1109/TKDE.2023.3277969

Often, weaker consistency yields better performance. Although strict serializability is taken as the strongest consistency level, it is not often supported in decentralized databases because of its poor performance. In practice, for better performance, a few newly-found consistency levels, with degrees weaker than strict serializability but stronger than serializability, are proposed. For example, strong session serializability [16] integrates session consistency with serializability, while Lynx [56] imposes read-your-writes consistency on serializability. However, these consistency levels are reported on a case-by-case basis, and hence, they cannot make the trade-off between consistency and performance to meet users' requirements. In this paper, we study the consistency over serializability and above, and achieve this trade-off by providing a systematic definition of consistency levels over serializability. We exclude weaker consistency levels, like snapshot isolation [33], [42], from our scope, because users often assume serializability is implicitly guaranteed [6], [20].

Inspired by different consistency models (popularized by linearizability, sequential consistency [28], causal consistency [27], etc.), we first define a group of orderings among transactions, including the write-read order, program order, causal-related order, real-time order, and write-legal order. We then model multi-level serializability to systematically formulate consistency: (1) serializability, (2) sequential serializability, and (3) strict serializability, by preserving these orders. For instance, sequential serializability is formulated by preserving the causal-related order and write-legal order. Besides providing systematic modeling, our ordering-based formulation of multi-level serializability is self-explanatory. Informally, given a transaction's write, serializability does not ensure it is always "seen" by late transactions; sequential serializability guarantees that it is always "seen" by some late transactions (e.g., subsequent transactions in the same session, ensured by the causal-related order), and strict serializability guarantees that it is always "seen" by all late transactions (ensured by the real-time order).

We then propose a novel concurrency control algorithm called *bidirectional timestamp adjustment* (abbreviated as BDTA). Inspired by dynamic timestamp adjustment (DTA) [31], [54], [55], BDTA introduces a timestamp interval  $[LB, UB]$  for each transaction  $T$ . For every two transactions  $T_i, T_j$  with an ordering constraint (e.g.,  $T_i \rightarrow T_j$ ), we guarantee that their timestamp intervals are disjoint, i.e.,  $T_i.UB < T_j.LB$ . Any transaction violating the required ordering constraint cannot produce a legal timestamp interval and aborts. We regulate the timestamp allocation scheme in BDTA to support multi-level serializability. In particular, we use the timestamp oracle to preserve the real-time order, and the hybrid logical clock [17] to preserve the causal-related order. Compared with existing DTA schemes, the differences of BDTA are two-fold: on one hand, BDTA adjusts timestamp intervals by preserving ordering constraints required in multi-level serializability; on the other hand, BDTA optimizes the size of the timestamp interval for each adjustment, leading to a lower transaction abort rate.

In summary, we make the following contributions:

- We systematically formulate multi-level serializability from different consistency perspectives. We define a group

of orderings among transactions and use these orders to achieve a unified formulation.

- We design a concurrency control algorithm, called BDTA, to support multi-level serializability. We propose a heuristic method to adaptively determine the size of the timestamp interval for each adjustment, which helps reduce the abort rate. Our special design makes read-only transactions always commit.
- We conduct extensive experiments to show the necessity of multi-level serializability. Additionally, we integrate BDTA and state-of-the-art concurrency control algorithms into Deneva [22], and results show BDTA achieve up to  $1.19 \times$  performance gain. We also integrate BDTA into Greenplum [21] and release our implementation publicly.

## II. BACKGROUND

In this section, we briefly introduce the system architecture of decentralized databases and discuss the state-of-the-art timestamp allocation schemes.

### A. Decentralized Database Systems

Decentralized database systems are particularly designed to support scalable transaction processing. Typically, the system architecture of transaction processing can be decomposed into two layers: the coordination layer and the storage layer. The first layer contains multiple coordinators, in which each process coordinates incoming transactions and returns results to users. The second layer consists of participant servers, each responsible for storing and manipulating data items. Data items are spread across all participant servers and are partitioned by a specific strategy like hash partitioning. Each transaction is coordinated by a single process in the coordinator. The process decomposes a transaction into one or multiple local transactions, which are then distributed to the corresponding participant server(s) that is/are responsible for managing the data items to be read/written. These systems always maintain multiple versions of each data item and adopt multi-version concurrency control (MVCC) to enable a transaction to read appropriate versions based on its snapshot.

Most decentralized database systems achieve high availability and fault tolerance using data replication, implemented using consensus protocols like Paxos [29] or Raft [36]. In this case, each partition has multiple replicas, which construct a Paxos/Raft group with one replica chosen as the leader replica. Because data synchronization among the replicas based on Paxos/Raft is orthogonal to this paper, to simplify the discussion, we assume coordinators always send local transactions to the leader replica of the corresponding partition with the required data items.

### B. Timestamp Allocation Schemes

In MVCC-based decentralized databases, each transaction should acquire a unique timestamp and use such a timestamp to determine the corresponding consistent snapshot. Some systems [38] use the timestamp oracle to allocate globally ordered

TABLE I  
SYMBOLS AND THEIR MEANINGS

| Symbol                | Definition   |
|-----------------------|--|
| $P_i$                 | $i$ -th process in the coordinator   |
| $H_i$                 | $i$ -th transaction history  |
| $S_i$                 | $i$ -th transactionally sequential history   |
| $T_i \rightarrow T_j$ | a partial order between $T_i$ and $T_j$  |
| $T_i$                 | $i$ -th transaction is a sequence of operations, which are either read $R_i(x_j)$ , write $W_i(x_i)$ , commit $C_i$ or abort $A_i$ |
| $T_i^s$               | the local transaction of $T_i$ in the participant $s$  |
| $T_i.LB/UB$           | the lower bound/upper bound of $T_i$ 's timestamp interval   |
| $T_i.sl$              | the spinlock of $T_i$  |
| $T_i.rs/ws$           | the read set/write set of $T_i$  |
| $T_i.ss$              | the snapshot of $T_i$  |
| $T_i.c$               | the commit timestamp of $T_i$  |
| $x$                   | a data item, associated with four fields:  |
| $x.pk$                | the primary key of $x$   |
| $x.RTS$               | the maximum $T_i.c$ of the transactions that have read $x$   |
| $x.WT$                | the transaction that is currently modifying $x$  |
| $x.RIDs$              | a list recording non-committed transactions that read $x$  |
| $x_i$                 | $i$ -th version of the data item $x$   |
| $x_i.cts$             | $T_i.c$ of the transaction that writes $x_i$   |
| $\mu_{i,j}$           | the timestamp interval space for the adjustment between $T_i$ and $T_j$  |

timestamps. Under such a scheme, each transaction would communicate with the centralized timestamp oracle through the network, which is costly and potentially becomes a performance bottleneck. Recently proposed systems [44] rely on the hybrid logical clock (HLC) [17] scheme to achieve consistent snapshot reads. Unlike centralized timestamp oracle, HLC allows each process to allocate timestamps individually, i.e., acquire timestamps in a decentralized manner. Each timestamp allocated by HLC consists of two parts: (1) physical clock  $pts$  [32], which maintains the local timestamp of that process, and (2) Lamport clock  $lts$  [27], which traces orders among operations through different processes. These systems assign an HLC timestamp to each transaction and update HLC upon an operation, i.e., each operation in a transaction will trigger the update of HLC in that process. Consequently, for the applications where each transaction has a large number of reads/writes, frequently updating HLC may hurt the performance. To alleviate this problem, we propose a strategy to update HLC once per transaction instead of once per operation (with more details in Section IV-C).

### III. MULTI-LEVEL SERIALIZABILITY MODELING

In this section, we define ordering among transactions, and systematically formulate multi-level serializability from different consistency perspectives. Table I summarizes the notations used throughout the paper.

Each transaction, denoted by  $T$ , is a sequence of operations, that are either read  $R_i(x_j)$ , write  $W_i(x_i)$ , commit  $C_i$  or abort  $A_i$ . Without loss of generality, we assume each transaction is separately coordinated by a single process in the coordinator. Each operation, denoted by  $op$ , consists of an invocation event  $Inv(op, P)$  and a response event  $Res(op, P)$ , where  $P$  represents a process in the coordinator. For simplicity, we omit process  $P$  or data item  $x_i$  when the context is clear, e.g.,  $Inv(R_2(x_0), P_2)$  can be simplified by  $Inv(R_2(x_0))$ .

Following widely adopted terminology defined in [1], [24], we denote an execution of a set of transactions as a history. A history,

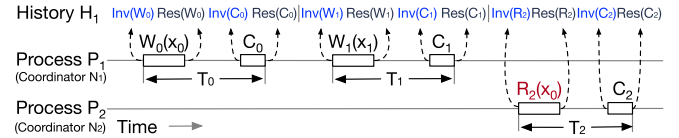


Fig. 2. A history  $H_1$  corresponds to Fig. 1.

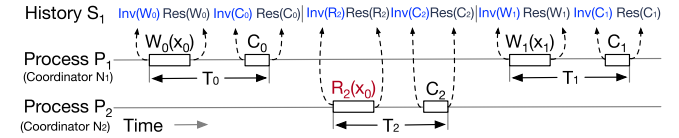


Fig. 3. An equivalent history  $S_1$  to  $H_1$ .

denoted by  $H$ , is a finite sequence of events in transactions. For example, we present history  $H_1$  in Fig. 2 that corresponds to the execution of transactions in Fig. 1. We project a history to the data item level, transaction level, and process level:

- A *data item projection*,  $H|x$ , of a history  $H$  is the subsequence of all invocation and response events in  $H$  of operations executed on data item  $x$ .
- A *transaction projection*,  $H|T$ , of a history  $H$  is the subsequence of all events in  $H$  whose operations are from  $T$ .
- A *process projection*,  $H|P$ , of a history  $H$  is the subsequence of all events in  $H$  coordinating by  $P$ .

**Definition 1 (Equivalent Histories):** Two histories  $H$  and  $H'$  are **equivalent** if  $\forall P, H|P = H'|P$ .  $\square$

For example, history  $S_1$  shown in Fig. 3 is equivalent to  $H_1$  by swapping the order  $T_1$  and  $T_2$  located in  $P_1$  and  $P_2$  respectively, from the global clock perspective. A transaction  $T_i$  is said to be *well-formed* in  $H$  if its transaction projection  $H|T_i$  satisfies the following conditions: (1) the first event is an invocation; (2) each invocation, except the last, is immediately followed by the response of the same operation; (3) each response, except the last, is immediately followed by an invocation; (4) no events follow the response of  $C_i$  or  $A_i$ . In this paper, we assume that in a history, transactions are well-formed and finally commit. Besides, we assume each process  $P$  coordinates transaction sequentially, i.e.,  $P$  only starts the first event of one transaction after receiving the response of commit or abort of another transaction.

#### A. Ordering Definitions

Given a history  $H$ , and two operations  $op_1, op_2$  of  $H$ , we define four partial orders between  $op_1$  and  $op_2$ :

**Definition 2 (Program Order,  $\prec_H^{pr}$ ):**  $op_1 \prec_H^{pr} op_2$  if they are in the same  $P$  and  $Res(op_1, P)$  precedes  $Inv(op_2, P)$ .  $\square$

**Definition 3 (Write-Read Order,  $\prec_H^{wr}$ ):**  $op_1 \prec_H^{wr} op_2$  if  $op_2$  reads a version written by  $op_1$ .  $\square$

**Definition 4 (Causal-related Order,  $\prec_H^{cr}$ ):**  $op_1 \prec_H^{cr} op_2$  if (a)  $op_1 \prec_H^{pr} op_2$  or (b)  $op_1 \prec_H^{wr} op_2$ , or they are related by a transitive closure leveraging (a) and/or (b).  $\square$

**Definition 5 (Real-time Order,  $\prec_H^{rt}$ ):**  $op_1 \prec_H^{rt} op_2$  if  $Res(op_1, P_i)$  precedes  $Inv(op_2, P_j)$ , where  $op_1, op_2$  are from  $P_i, P_j$ , respectively.  $\square$

Partial orders given in Definitions 2, 3, 4, and 5 are defined in operation granularity. They are widely used in the concurrent system to model linearizability, sequential consistency, causal consistency, etc. In our case, we make an extension from operation granularity to transaction granularity.

Given two transactions  $T_i$  and  $T_j$  in  $H$ , we define four partial orders between  $T_i$  and  $T_j$ :

- $T_i \prec_H^{pr} T_j$ , if  $T_i$  and  $T_j$  are in the same process  $P$  and the last event of  $T_i$  precedes the first event of  $T_j$ .
- $T_i \prec_H^{wr} T_j$ , if there exists an operation  $op_1$  in  $T_i$  and another operation  $op_2$  in  $T_j$  such that  $op_1 \prec_H^{wr} op_2$ .
- $T_i \prec_H^{cr} T_j$ , if (a)  $T_i \prec_H^{pr} T_j$  or (b)  $T_i \prec_H^{wr} T_j$ , or they are related by a transitive closure leveraging (a) and/or (b).
- $T_i \prec_H^{rt} T_j$ , if  $T_i$ 's last event precedes the first event of  $T_j$ .

**Definition 6 ((Transactionally) Sequential History):** A history  $S$  is (transactionally) sequential if, for any two transactions  $T_i$  and  $T_j$  in  $S$ , either the last event of  $S|T_i$  precedes the first event of  $S|T_j$  or the last event of  $S|T_j$  precedes the first event of  $S|T_i$ . We denote the order between  $T_i$  and  $T_j$  by  $T_i \rightarrow T_j$  if  $T_i$  precedes  $T_j$  in  $S$ .  $\square$

For brevity, we refer to transactionally sequential history as sequential history unless otherwise specified.

**Definition 7 (Write-legal Order):** A sequential history preserves the write-legal order if for  $\forall x, Inv(R_j(x_i), P_k)$  in  $S|x$  immediately comes after  $Res(W_i(x_i), P_l)$  by removing events of other read operations in  $S|x$ , i.e., roughly, reads of  $x_i$  immediately come after the write of  $x_i$  in  $S|x$ .  $\square$

For example,  $S_1$  shown in Fig. 3 is a sequential history, with  $T_0 \prec_H^{pr} T_1$ ,  $T_0 \prec_H^{wr} T_2$ . Besides,  $S_1$  preserves the write-legal order because  $Inv(R_2(x_0))$  immediately comes after  $Res(W_0(x_0))$  in  $S_1|x$ , indicating the order of  $S_1$  is  $T_0 \rightarrow T_2 \rightarrow T_1$ . However,  $H_1$  does not preserve the write-legal order since  $Inv(R_2(x_0))$  does not immediately come after  $Res(W_0(x_0))$  in  $H_1|x$ .

## B. Multi-Level Serializability

By selectively imposing orders on sequential history, we model multi-level serializability below. Given two histories  $H$  and  $S$ ,  $\prec_H^{cr} \subseteq \prec_S^{cr}$  means that  $\forall T_i, T_j$ , if  $T_i \prec_H^{cr} T_j$ , then  $T_i \prec_S^{cr} T_j$  (this is also applicable to other orders).

**Definition 8 (Multi-level Serializability):**

- **Serializability:** A history  $H$  ensures serializability if there is a sequential history  $S_H$ , which preserves the write-legal order, with  $\forall T, S_H|T = H|T$  and  $\prec_H^{wr} \subseteq \prec_{S_H}^{wr}$ .
- **Sequential Serializability:**  $H$  guarantees sequential serializability if  $H$  is equivalent to a sequential history  $S_H$ , which preserves the write-legal order, with  $\prec_H^{cr} \subseteq \prec_{S_H}^{cr}$ .
- **Strict Serializability:**  $H$  ensures strict serializability if  $H$  is equivalent to a sequential history  $S_H$ , which preserves the write-legal order, with  $\prec_H^{cr} \subseteq \prec_{S_H}^{cr}$  and  $\prec_H^{rt} \subseteq \prec_{S_H}^{rt}$ .  $\square$

Informally, the execution of  $H$  is said to satisfy (1) serializability when there exists a sequential history  $S_H$  preserving the write-read order and write-legal order, (2) sequential serializability when an equivalent sequential history  $S_H$  to  $H$  preserves the causal-related order and write-legal order, (3) strict serializability when an equivalent sequential history  $S_H$

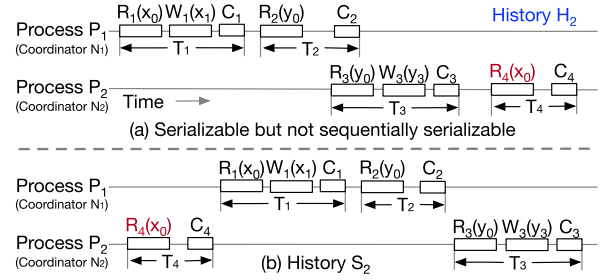


Fig. 4. A history  $H_2$  and a sequential history  $S_2$ .

to  $H$  preserves the real-time order, causal-related order, and write-legal order. Take  $H_1$ , shown in Fig. 2, for example.  $H_1$  satisfies sequential serializability because  $H_1$  is equivalent to  $S_1$  shown in Fig. 3, which is a sequential history preserving the write-legal order, with  $\prec_{H_1}^{cr} \subseteq \prec_{S_1}^{cr}$ . Assume  $T_2$  reads  $x_1$  instead of  $x_0$  in  $H_1$ , i.e.,  $R_2(x_1)$ ,  $H_1$  would satisfy strict serializability. On the contrary, consider history  $H_2$  shown in Fig. 4(a).  $H_2$  does not preserve the write-legal order. We can find a sequential history  $S_2$  (shown in Fig. 4(b)) to  $H_2$ , which preserve the write legal order, with  $\forall T \in H_2, H_2|T = S_2|T$  and  $\prec_{H_2}^{wr} \subseteq \prec_{S_2}^{wr}$ . Thus,  $H_2$  is serializable but not sequentially serializable because  $T_3 \prec_{H_2}^{pr} T_4$  is not preserved in  $S_2$ .

In essence, we model multi-level serializability by combining serializability with the consistency model, including linearizability and sequential consistency. In the following, we theoretically show that the combination of serializability with causal consistency and consistency below it can be reduced to sequential serializability.

**Theorem 1:** Imposing causal consistency and consistency below it on serializability can be reduced to sequential serializability.

*Proof:* As mentioned in [34], causal consistency and weaker consistency levels, like read-your-write consistency, cannot preserve a total order of operations, leading to different processes observing conflict orders, e.g.,  $T_i \rightarrow T_j$  observed from the process  $P_1$  and  $T_j \rightarrow T_i$  observed from  $P_2$ . Since serializability imposes a total order of transactions, imposing causal consistency or weaker consistency levels on serializability is reduced to sequential serializability.  $\square$

## IV. CONCURRENCY CONTROL ALGORITHM

In this section, we give an overview of BDTA to support multi-level serializability and elaborate on how BDTA works correctly in decentralized MVCC-based databases.

### A. An Overview of BDTA

The basic idea of BDTA is to preserve required orders defined in multi-level serializability during the transaction execution. To start, we give an example to show how BDTA preserves required orders under serializability.

**Example 2:** Consider history  $H_3$  shown on the top part of Fig. 5. For reference, we list all orders required in serializability. We present how to construct  $S_3$  during the execution, where  $S_3$

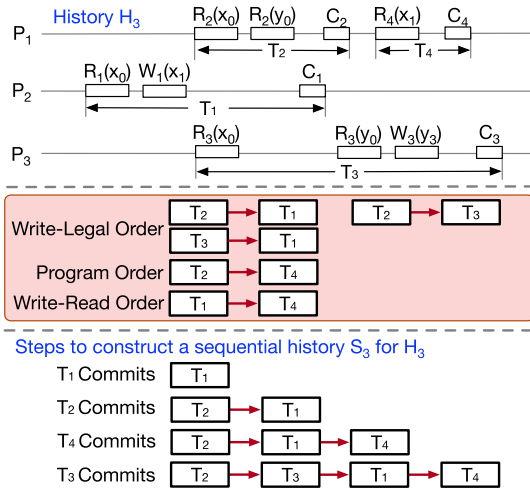


Fig. 5. An example of dynamic ordering using BDTA.

is a sequential history that preserves the write-legal order of  $H_3$ , with  $\forall T, S_3|T = H_3|T$  and  $\prec_{H_3}^{wr} \subseteq \prec_{S_3}^{wr}$ . From the global clock perspective,  $T_1$  first commits, and we set  $T_1$  as the first transaction in  $S_3$ .  $T_2$  then commits. Theoretically,  $T_2$  can be ordered before  $T_1$  or after  $T_1$  in  $S_3$ . BDTA orders  $T_2$  before  $T_1$  by detecting the write-legal order  $T_2 \rightarrow T_1$ . Next,  $T_4$  commits and is ordered after  $T_1$  in  $S_3$  because of the write-read order  $T_1 \rightarrow T_4$ . Finally,  $T_3$  commits. Due to the write-legal orders  $T_3 \rightarrow T_1$  and  $T_2 \rightarrow T_3$ ,  $T_3$  can only be ordered between  $T_2$  and  $T_1$  in  $S_3$ . In conclusion,  $S_3$  is shown at the bottom part of Fig. 5, with the order as  $T_2 \rightarrow T_3 \rightarrow T_1 \rightarrow T_4$ .  $\square$

Different from other concurrency control algorithms, like T/O, or 2PL, that order transactions statically (e.g., T/O orders transactions based on their start timestamps), BDTA orders transactions dynamically, and hence possibly leads to a lower transaction abort rate. As we can see from Example 2, by using BDTA,  $T_2$  is ordered before  $T_1$  and can commit successfully, while by using T/O,  $T_2$  is ordered after  $T_1$  and should abort. BDTA relies on timestamp intervals to order transactions dynamically. Inspired by the DTA scheme [9], [30], we introduce a timestamp interval  $[LB, UB]$  for each transaction  $T$ . For every two transactions  $T_i, T_j$ , during the entire execution, we guarantee that:

$$T_i.UB < T_j.LB \quad \text{if } T_i \rightarrow T_j \quad (1)$$

(1) ensures that, for any two transactions  $T_i$  and  $T_j$ , if there exists a partial order  $T_i \rightarrow T_j$ , we have  $T_i.UB < T_j.LB$ , i.e., ordering  $T_i$  before  $T_j$  in the sequential history. Any transaction violating the required ordering constraint cannot produce a legal timestamp interval and aborts. We then give an example to show how BDTA preserves required orders by adjusting the timestamp intervals of transactions.

*Example 3:* Reconsider  $H_3$  in Example 2. We present how the timestamp intervals are adjusted in Fig. 6. From the global clock perspective,  $T_1$  first starts, followed by  $T_2, T_3$ , and  $T_4$ . First, when  $T_3$  starts, the timestamp intervals of  $T_1, T_2$ , and

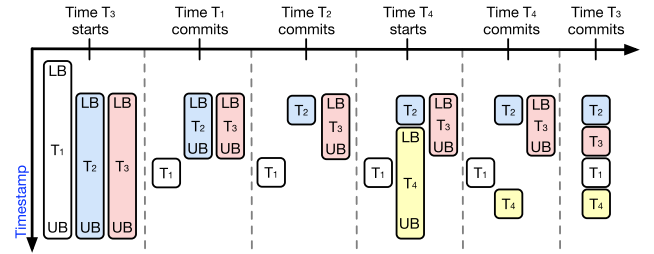


Fig. 6. An example of bidirectional timestamp adjustment.

$T_3$  are shown in the first column. Next, when  $T_1$  commits, we can detect the write-legal orders ( $T_2 \rightarrow T_1$  and  $T_3 \rightarrow T_1$ ) and bidirectionally adjust the timestamp intervals of  $T_2$  and  $T_3$  with  $T_1$ , making  $T_2.UB < T_1.LB$  and  $T_3.UB < T_1.LB$  (second column). Then, when  $T_2$  commits, because we cannot detect the write-legal order  $T_2 \rightarrow T_3$  and the program order  $T_2 \rightarrow T_4$ , we do not perform the adjustment of  $T_2$  with other transactions but simply set  $T_2.UB$  to  $T_2.LB$  (third column). Afterward, when  $T_4$  starts, by detecting program order  $T_2 \rightarrow T_4$ , we adjust  $T_4.LB$  to guarantee  $T_4.LB > T_2.UB$  (fourth column). When  $T_4$  commits, we adjust  $T_4.LB$  to guarantee  $T_4.LB > T_1.UB$  because of the write-read order  $T_1 \rightarrow T_4$  (fifth column). Finally, when  $T_3$  commits, we detect write-legal order  $T_2 \rightarrow T_3$  and adjust  $T_3.LB$  to guarantee  $T_3.LB > T_2.UB$  (sixth column).  $\square$

As illustrated in Example 3, we preserve the partial orders required in sequential history  $S_3$  by adjusting timestamp intervals of transactions. We must emphasize that BDTA is different from existing DTA algorithms, like Sundial [55], MaaT [31], TCM [30]. First and foremost, BDTA adjusts timestamp intervals to preserve required orders in multi-level serializability. Second, BDTA optimizes the size of the timestamp interval for each adjustment, leading to a lower transaction abort rate. Take history  $H_3$  as an example. As discussed, the order  $T_2 \rightarrow T_3 \rightarrow T_1$  needs to be preserved. However, existing DTA algorithms cannot preserve such an order, causing  $T_2$  or  $T_3$  to abort. Specifically, Sundial and MaaT do not have the capability to adjust timestamp intervals bidirectionally, meaning that if there exists an order  $T_i \rightarrow T_j$ , when  $T_i$  or  $T_j$  commits, the timestamp interval of  $T_i$  or  $T_j$  is adjusted individually. For this reason, when  $T_1$  commits, they set  $T_1.UB = T_1.LB$  without adjusting  $T_2.LB$  and  $T_2.UB$ . In Fig. 6,  $T_2$  starts after  $T_1$ , indicating  $T_2.UB \geq T_2.LB > T_1.LB$ . When  $T_2$  attempts to commit, due to the write-legal order  $T_2 \rightarrow T_1$ , there does not exist a legal timestamp interval of  $T_2$  to guarantee  $T_2.UB < T_1.LB$ , causing  $T_2$  to abort. For TCM, although it performs the bidirectional adjustment, its adjustment cannot leave enough legal interval space for transactions to commit. For example, to preserve  $T_2 \rightarrow T_1$ , TCM sets  $T_1.LB = T_2.LB + 1$  and  $T_2.UB = T_2.LB$  to make  $T_2.UB < T_1.LB$ . However, by doing this, there does not exist any interval space between  $T_2.UB$  and  $T_1.LB$ , and any transaction ordered between  $T_2$  and  $T_1$ , like  $T_3$  will abort. BDTA solves the interval space problem of TCM by introducing an adaptive

timestamp interval selection method, which is discussed in Section V-A.

According to Definition 8, if a transaction violates the ordering constraints, the corresponding history must contain a cycle of partial orders. BDTA ensures such a transaction cannot produce a legal timestamp interval:

**Theorem 2:** Given a set of transactions in the history that form a cycle of partial orders, there must exist at least one transaction  $T$  with  $T.LB > T.UB$ .  $\square$

*Proof:* If a history  $H$  contains a cycle of partial orders, there must exist the order  $T_i \rightarrow T_j \rightarrow \dots \rightarrow T_i$ . Suppose each transaction  $T$  in the history  $H$  satisfies  $T.LB \leq T.UB$ . According to (1), we ensure  $T_i.UB < T_j.LB \leq T_j.UB < T_i.LB$ , which indicates  $T_i.UB < T_i.LB$ . Therefore,  $T_i$  cannot obtain a legal timestamp interval, and the history  $H$  containing a cycle of partial orders must include a transaction like  $T_i$  with  $T_i.LB > T_i.UB$ .  $\square$

Theorem 2 guarantees that, during the entire execution, if a given set of transactions forms a cycle of partial orders, we can abort the transactions with illegal timestamp intervals to destroy the cycle and ensure correctness.

## B. BDTA in Action

We elaborate on how BDTA works in decentralized MVCC-based databases, especially when and how timestamp intervals of transactions are initialized and adjusted. BDTA follows the optimistic way to do concurrency control. For this purpose, the process  $P_k$  coordinates the entire lifecycle of a transaction  $T_i$  from the initialization, through the local execution, to the validation and commit.

- **Initialization:** The process  $P_k$  creates an execution context for  $T_i$ , including transaction snapshot  $T_i.ss$ , timestamp interval  $[LB, UB]$ , and commit timestamp  $T_i.c$ , etc. Because we target MVCC-based databases, we process read requests based on snapshot isolation, i.e.,  $T_i$  does reads and writes based on the snapshot  $T_i.ss$ . For this reason,  $T_i.ss$  is initialized below:

$$T_i.ss = \begin{cases} \text{GET\_LC()} & \text{serializability,} \\ \text{GET\_HLC()} & \text{sequential serializability,} \\ \text{TS\_ORACLE()} & \text{strict serializability} \end{cases} \quad (2)$$

Function GET\\_LC() returns the local timestamp of the process  $P_k$  and is used for serializability. Function GET\\_HLC(), shown in Algorithm 1, returns an HLC timestamp, which is used for sequential serializability. Note that HLC timestamps of two transactions with the causal-related order are pairwise comparable. Function TIME\\_ORACLE() returns the current global timestamp allocated from a centralized timestamp oracle [38] and is used for strict serializability.  $[LB, UB]$  of  $T_i$  is initialized:

$$T_i.LB = T_i.ss, T_i.UB = +\infty \quad (3)$$

$[LB, UB]$  of  $T_i$  will be dynamically adjusted during the local execution and validation. Given a distributed transaction  $T_i$ , we denote the set of participant servers that are involved in the execution of  $T_i$  as  $S(T_i)$ , and denote

the local transaction executed in the participant server  $s \in S(T_i)$  as  $T_i^s$ . We do the initialization for  $T_i^s$  below.

$$T_i^s.ss = T_i.ss, T_i^s.LB = T_i.LB, T_i^s.UB = T_i.UB \quad (4)$$

- **Local execution:** Local transaction  $T_i^s$  of  $T_i$  is executed in participant server  $s$  respectively. We now present how to adjust  $[LB, UB]$  of  $T_i^s$  to preserve the write-legal order during the local execution. Upon a read  $R_i(x_m)$  by  $T_i^s$ , if BDTA detects that a new version  $x_{m+1}$  is generated by another committed transaction  $T_{m+1}$ , the timestamp interval of  $T_i^s$  is adjusted below:

$$T_i^s.UB = x_{m+1}.cts - 1 \quad (5)$$

where  $x_{m+1}.cts$  is the commit timestamp of  $T_{m+1}$ , denoted as  $T_{m+1}.c$  (version  $x_{m+1}$  is written by  $T_{m+1}$  that takes the same subscripts). Note that if the write  $W_{m+1}(x_{m+1})$  happens after  $R_i(x_m)$  from the global clock perspective, the write-legal order is guaranteed during the local validation of  $T_{m+1}^s$ . Besides, we preserve the write-read order based on snapshot isolation. Because for each data item  $x_m$ , only  $T_i^s$  with  $T_i^s.ss \geq x_m.cts$  can “see”  $x_m$ , which guarantees the write-read order  $T_m \rightarrow T_i$ . We will elaborate on the local execution in Section IV-D.

- **Validation:** After completing local execution, the process  $P_k$  coordinates all local transactions  $T_i^s$  following two-phase commit (2 PC). To begin with, in the first phase of 2 PC, called the prepare phase, each local transaction does the local validation and determines a proper  $[LB, UB]$  of  $T_i^s$ . To preserve the write-legal order, we examine the write set  $T_i^s.ws$  of  $T_i^s$  and adjust  $T_i^s.LB$ :

$$T_i^s.LB = \max\{T_i^s.LB, \max\{x.RTS + 1 | x \in T_i^s.ws\}\} \quad (6)$$

where  $x.RTS$  is the maximum commit timestamp of all committed transactions that ever read data item  $x$ . We then adjust the timestamp intervals of  $T_i^s$  and  $T_j^s$  bidirectionally, i.e., we identify every concurrent transaction  $T_j^s$  that reads data item  $x \in T_i^s.ws$ , and adjust  $T_i^s.LB$ :

$$T_i^s.LB = \max\{T_i^s.LB, \max\{T_j^s.LB + \mu_{j,i} | T_i^s.ws \cap T_j^s.rs \neq \emptyset\}\} \quad (7)$$

where  $\mu_{j,i}$  represents the interval space for the bidirectional adjustment. Finally, we adjust  $T_j^s.UB$  of every  $T_j^s$ :

$$T_j^s.UB = \min\{T_j^s.UB, T_i^s.LB - 1\} \quad (8)$$

Reconsider Example 3. As shown in Fig. 7, during the validation of  $T_1^s$ , by examining the write set of  $T_1^s$ , BDTA identifies transactions  $T_2^s$  and  $T_3^s$  that read the data item  $x$ , and hence bidirectionally adjusts  $T_1^s.LB$  and  $T_2^s.UB$  ( $T_1^s.LB$  and  $T_3^s.UB$ ) to preserve the write-legal order  $T_2 \rightarrow T_1$  ( $T_3 \rightarrow T_1$ ). To achieve this, we first set  $T_1^s.LB$  to  $\max\{T_2^s.LB + \mu_{2,1}, T_3^s.LB + \mu_{3,1}\}$  based on (7) and then set  $T_2^s.UB$  and  $T_3^s.UB$  to  $T_1^s.LB - 1$  based on (8).

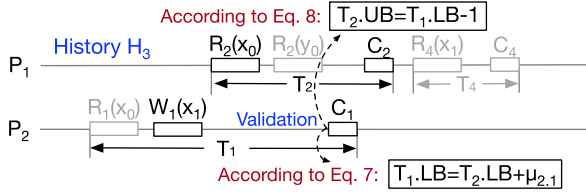


Fig. 7. Timestamp interval adjustment for Example 3.

- *Commit*: After all local transactions finish the validation phase,  $T_i$  comes to the second phase of 2 PC named as commit phase. In the commit phase, the process  $P_k$  collects  $[LB, UB]$  of  $T_i^s$  from each participant server  $s \in S(T_i)$  and updates  $[LB, UB]$  of  $T_i$  based on (9). If  $T_i.LB > T_i.UB$ ,  $P_k$  notifies each participant to abort the local transaction; otherwise,  $P_k$  notifies each participant server to commit the local transaction.

$$\begin{aligned} T_i.LB &= \max\{T_i^s.LB | s \in S(T_i)\} \\ T_i.UB &= \min\{T_i^s.UB | s \in S(T_i)\} \end{aligned} \quad (9)$$

Finally, the commit timestamp of  $T_i$  is set below.

$$T_i.c = T_i.LB \quad (10)$$

Using (10), we guarantee the order of transactions' commit timestamps follows that determined by BDTA. We will elaborate on the validation and commit in Section IV-E.

### C. Multi-Level Serializability Guarantee

BDTA preserves orders among serializable transactions by maintaining disjoint timestamp intervals.

*Theorem 3*: Given a history  $H$ , BDTA guarantees the execution of  $H$  satisfies serializability.  $\square$

*Proof*: Given any transaction  $T_i$  in  $H$ , for local transaction  $T_i^s$  in each participant server  $s \in S(T_i)$ , BDTA guarantees there exists a sequential history  $\bar{H}$  preserving the write-legal order ((5)–(8)), with  $\bar{H}|T_i^s = H|T_i^s$ . Besides, due to snapshot isolation, we ensure the write-read order in  $\bar{H}$ . Thus, according to Definition 8, BDTA achieves serializability in each participant server. Further, by imposing 2 PC on  $\forall s \in S(T_i)$  to make an agreement on  $[LB, UB]$  of  $T_i$  (9), plus (10), we guarantee that  $H$  satisfies serializability.  $\square$

Theorem 3 guarantees a serializable execution using BDTA. We now discuss how BDTA guarantees strict serializability and sequential serializability, respectively.

- *Strict serializability guarantee*: We additionally preserve the real-time order based on TIME\_ORACLE(). In fact, when  $T_i$  commits, if  $T_i.LB \leq \text{TIME\_ORACLE}()$ , the real-time order is preserved by (10), ensuring  $T_i.c \leq \text{TIME\_ORACLE}()$ . By so doing, a new transaction  $T_j$  starting after  $T_i$ 's commit will have  $T_j.ss \geq \text{TIME\_ORACLE}()$ , and hence,  $T_j$  can always “see”  $T_i$ 's writes. Otherwise, if  $T_i.LB > \text{TIME\_ORACLE}()$ , to preserve the real-time order,  $T_i$  waits until  $T_i.LB \leq \text{TIME\_ORACLE}()$  to commit. Thus, under strict serializability, we introduce an additional constraint for  $T_i$  to commit, as shown below.

$$T_i.LB \leq \text{TIME\_ORACLE()} \quad (11)$$

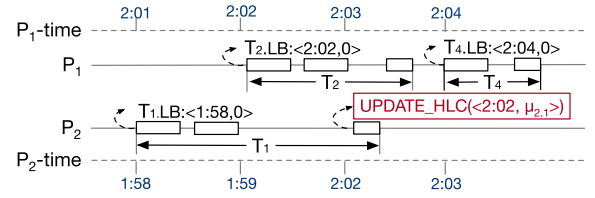


Fig. 8. An example of allocating HLC timestamps.

#### Algorithm 1: GET\_HLC() [17].

```

1: if  $pts_k \geq$  current process timestamp then  $lts_k++$ ;
2: else  $pts_k \leftarrow$  current process timestamp,  $lts_k \leftarrow 0$ ;
3: return  $\langle pts_k, lts_k \rangle$ ;

```

#### Algorithm 2: UPDATE\_HLC( $pts, lts$ ) [17].

```

1:  $temp \leftarrow pts_k$ ;
2:  $pts_k \leftarrow \max\{pts_k, pts, \text{current process timestamp}\}$ ;
3: if  $pts_k = temp$  and  $pts_k = pts$  then
 $lts_k \leftarrow \max\{lts_k, lts\} + 1$ ;
4: else if  $pts_k = temp$  then  $lts_k++$ ;
5: else if  $pts_k = pts$  then  $lts_k \leftarrow lts++$ ;
6: else  $lts_k \leftarrow 0$ ;

```

- *Sequential serializability guarantee*: We preserve the causal-related order based on HLC [17]. We implement HLC by allowing each process to allocate timestamps individually. We must emphasize that for each transaction, BDTA updates HLC only once. Algorithm 1 describes the HLC timestamp allocation, while Algorithm 2 presents the HLC update upon  $T_i$  commit using  $T_i.c.ppts$  and  $T_i.c.lts$ . Note that  $pts_k$  and  $lts_k$  denote the physical clock  $pts$  and Lamport clock  $lts$  of the process  $P_k$ , respectively. Reconsider Example 3. As shown in Fig. 8, upon the commit of  $T_1$ ,  $T_1.c$  is set to  $\langle 2:02, \mu_{2,1} \rangle$ , larger than  $T_2.LB$ ,  $\langle 2:02, 0 \rangle$ . The process  $P_2$  updates its HLC timestamp using  $T_1.c$ .  $T_2$  then commits with  $T_2.LB$ . All subsequent transactions in the process  $P_2$ , e.g.,  $T_4$ , take larger HLC timestamps than  $T_2.c$ . Our transaction-level HLC scheme preserves the causal-related order. For example, since  $T_4.ss$  is larger than  $T_2.c$  and  $T_1.c$ , all writes seen by  $T_2$  and  $T_1$  can also be seen by  $T_4$ , preserving the causal-related orders  $T_2 \rightarrow T_4$  and  $T_1 \rightarrow T_4$ .

- *Summarization*: To summarize BDTA, the adjustment is triggered by a transaction  $T_i$  upon either (1) CONDITION 1:  $T_i$  reads a version  $x_m$ , and a new version  $x_{m+1}$  is generated by a committed  $T_{m+1}$  before  $T_i$ 's read, or (2) CONDITION 2:  $T_i$  enters the validation phase. CONDITION 1, 2 are used to preserve the write-legal order, and CONDITION 1 plus snapshot isolation preserves the write-read order.

### D. Local Execution

We present how local execution in BDTA works. In our design, a data item  $x$  is associated with metadata that has four fields (shown in Fig. 9): (1)  $x.pk$  is the primary key of  $x$ ; (2)  $x.RTS$  is the maximum commit timestamp of all committed transactions that ever read  $x$ ; (3)  $x.WT$  is the transaction that is currently

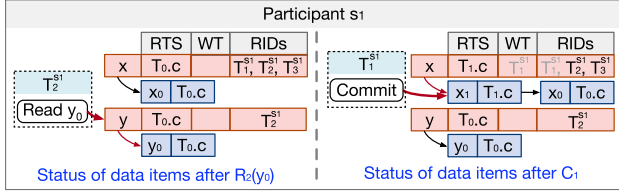


Fig. 9. Status of data items using BDTA.

**Algorithm 3:** Execution of Local Transaction  $T_i^s$ .

```

1 Function Read ( $T_i^s, key$ ):
2   if  $\langle key, x_i \rangle \in T_i^s.ws$  then return  $x_i$ ;
3    $T_i^s.rs \leftarrow T_i^s.rs \cup \{key\}$ ;
4    $x \leftarrow$  location the data item according to  $key$ ;
5    $x.RIDs \leftarrow x.RIDs \cup \{T_i^s\}$ ;
6    $x_m, x_{m+1} \leftarrow$  SnapshotRead ( $key, T_i^s.ss$ );
7   if  $x_{m+1}$  then
8     # lock() acquires the spinlock  $T_i^s.sl$ 
9      $T_i^s.UB \leftarrow \min\{T_i^s.UB, x_{m+1}.cts - 1\}$ ;
10    # unlock() releases the spinlock  $T_i^s.sl$ 
11  return  $x_m$ ;
12 Function Write ( $T_i^s, x.pk, x_i$ ):
13  if  $\langle x.pk, x_i' \rangle \notin T_i^s.ws$  then
14     $T_i^s.ws \leftarrow T_i^s.ws \cup \{\langle x.pk, x_i \rangle\}$ ;
15  else replace  $\langle x.pk, x_i' \rangle$  with  $\langle x.pk, x_i \rangle$  in  $T_i^s.ws$ ;

```

modifying  $x$ . It is set during the validation, and acts like a “soft-lock” to prevent write-write conflicts, i.e., two transactions are disallowed to modify  $x$  simultaneously; (4)  $x.RIDs$  is a lock-free list, recording every non-committed transaction that reads  $x$ . Like many optimistic algorithms, we maintain the read set  $T_i^s.rs$  and write set  $T_i^s.ws$  of  $T_i^s$ . To enable exclusive access to  $[LB, UB]$  of  $T_i^s$ , we provide a spinlock  $T_i^s.sl$ , and any access to  $[LB, UB]$  of  $T_i^s$  must hold the lock.

Algorithm 3 shows the pseudo-code of Read() and Write() functions. Read() takes a local transaction  $T_i^s$  and a search key  $key$  as the input (line 1). We directly return  $x_i$  if it is already in  $T_i^s.ws$  (line 2). Otherwise, we add  $key$  to  $T_i^s.rs$  (line 3), find  $x$ , and update  $x.RIDs$  by adding  $T_i^s$  (lines 4–5). We next invoke SnapshotRead() to read a proper version  $x_m$  and its next version  $x_{m+1}$  (if any) (line 6). In SnapshotRead(), if we detect (1) some  $T_j^s$  in the validation phase is writing  $x$  (i.e.,  $x.WT = T_j^s$ ), and (2)  $x_m$  is the latest version while  $x_j$  is not visible yet (the order between  $T_i^s$  and  $T_j^s$  remains undetermined), we do not read  $x$  until  $T_j^s$  commits to guarantee correctness. If  $x_{m+1}$  exists, we then adjust  $T_i^s.UB$  based on (5) by holding the lock  $T_i^s.sl$  (lines 7–10). Finally,  $x_m$  is returned (line 11). Function Write() takes a local transaction  $T_i^s$ ,  $x$ 's primary key  $x.pk$ , and a new version  $x_i$  to be written as the input. If a version  $x_i'$  with the same  $x.pk$  is in  $T_i^s.ws$ ,  $x_i'$  is updated by  $x_i$ ; otherwise, a pair  $\langle x.pk, x_i \rangle$  is added into  $T_i^s.ws$ .

*Example 4:* In reference to Figs. 5 and 6, let us reconsider the history  $H_3$ .  $T_1$  first starts and executes  $R_1(x_0)$ , during which we store  $x.pk$  into  $T_1.rs$ , insert  $T_1$  into  $x.RIDs$ , and read the proper version  $x_0$ .  $T_1$  then executes  $W_1(x_1)$  to store

$\langle x.pk, x_1' \rangle$  into  $T_1.ws$ . After that,  $T_2$  and  $T_3$  perform  $R_2(x_0)$ ,  $R_3(x_0)$  and  $R_2(y_0)$  using the same logic as  $R_1(x_0)$ , described in Algorithm 3. Now we have  $x.RIDs = \{T_1^s, T_2^s, T_3^s\}$  and  $y.RIDs = T_2^s$ , as shown in the left part of Fig. 9.

**E. Validation and Commit**

We introduce the validation of a local transaction  $T_i^s$  in Algorithm 4.  $\forall x_i \in T_i^s.ws$ , we set a soft-lock on  $x$  by  $T_i^s$  using compare-and-swap (lines 2–4). If a write-write conflict on  $x$  is detected, we abort  $T_i^s$ . Next, by invoking the function BiAdjust(), we bidirectionally adjust the timestamp intervals of  $T_i^s$  with transactions in  $x.RIDs$  based on (6)–(8) (line 5). We then adjust  $T_i^s.LB$  to preserve the write-read order (line 7). We abort  $T_i^s$  if its timestamp interval is illegal (line 9); otherwise, the validation of  $T_i^s$  is passed (line 10).

We present the commit of a local transaction  $T_i^s$  in Algorithm 5. We encapsulate the local validation in the prepare phase of 2 PC, and once the validation phase of  $T_i^s$  completes successfully, the process  $P_k$  coordinates  $T_i^s$  to commit by writing data items to the database, updating  $x.RTS$ , and releasing the “soft lock” (setting  $x.WT$  to 0, lines 2–5). Besides, for each  $key \in T_i^s.rs$ , we update  $x.RTS$  by the commit timestamp of  $T_i^s$  using atomic read-modify-write (RMW), and remove  $T_i^s$  from the read list  $x.RIDs$  (lines 6–8). If  $T_i^s$ 's validation fails, the process  $P_k$  coordinates  $T_i^s$  to abort by resetting  $x.WT$  and removing  $T_i^s$  from  $x.RIDs$ .

*Handling Contentions on list  $x.RIDs$ :* We use a lock-free list [23] to implement  $x.RIDs$  (line 12, Algorithm 4) for better performance. It is unnecessary to acquire locks on  $x.RIDs$ , and the reasons are two-fold. First, setting  $x.WT$  to  $T_i^s$  during the validation phase of the transaction  $T_i^s$  (line 3, Algorithm 4) blocks other transactions to read the current version of  $x$  (line 6, Algorithm 3). Second, for transactions that read previous versions of  $x$  but are not in  $x.RIDs$ , their timestamp intervals are adjusted in line 9, Algorithm 3.

*Example 5:* Let us continue Example 4 to validate whether  $T_1$  can commit. We set  $x.WT$  to  $T_1^s$ , adjust  $T_1^s.LB$  to ensure  $T_1^s.LB > x.RTS$ , and bidirectionally adjust the timestamp intervals of  $T_1^s$  and transactions in  $x.RIDs$  ( $T_2^s$  and  $T_3^s$ ). Because of the order  $T_2 \rightarrow T_1$  and  $T_3 \rightarrow T_1$ , we adjust  $T_1^s.LB$  to  $\max(T_2^s.LB + \mu_{2,1}, T_3^s.LB + \mu_{3,1}, T_1^s.LB)$ . We also set  $T_2^s.UB$  and  $T_3^s.UB$  to  $T_1^s.LB - 1$ . We calculate  $T_1.LB$  based on  $T_1^s.LB$  as discussed in (9), and then commit  $T_1$  with  $T_1.c = T_1.LB$ . Besides, we update  $x.RTS$  to  $T_1.c$ , reset  $x.WT$  to 0, remove  $T_1^s$  from  $x.RIDs$ , and write version  $x_1$  to the data item  $x$ , according to Algorithm 5. The status of data items is shown in the right part of Fig. 9. Next,  $T_2$  commits with  $T_2.c = T_2.LB$ , removes  $T_2^s$  from  $x.RIDs$  and  $y.RIDs$ , and updates  $y.RTS$  to  $T_2.c$ . After that,  $T_4$  then reads  $x_1$  and commits, and hence,  $T_4.c$  is larger than  $T_1.c$ . Finally,  $T_3$  executes  $R_3(y_0)$  and  $W_3(y_3)$ , and start to commit. During the validation,  $T_3$  acquires  $y.WT$ , adjusts  $T_3.LB$  to  $y.RTS + 1$  ( $T_2.c + 1$ ), and examines whether  $T_3.UB > T_3.LB$ . Because we introduce the adaptive timestamp interval selection method,  $\mu_{2,1}$  and  $\mu_{3,1}$  are larger enough to obtain a legal timestamp interval for  $T_3$ , and therefore,  $T_3$  can



**Algorithm 4:** Validation of Local Transaction  $T_i^s$ .

```

1 Function Validation ( $T_i^s$ ):
2   for  $\langle x.pk, x_i \rangle \in T_i^s.ws$  do
3     if  $x.WT \neq T_i^s$  and  $\neg CAS(x.WT, 0, T_i^s)$  then
4       return false;
5     BiAdjust ( $T_i^s, x$ );
6     # lock() acquires the spinlock  $T_i^s.sl$ 
7      $T_i^s.LB \leftarrow \max\{T_i^s.LB, x.RTS + 1\}$ ;
8     # unlock() releases the spinlock  $T_i^s.sl$ 
9     if  $T_i^s.LB > T_i^s.UB$  then return false;
10  return true;
11 Function BiAdjust ( $T_i^s, x$ ):
12  for  $T_j^s \in x.RIDs$  do
13    # lock() acquires the spinlocks  $T_i^s.sl, T_j^s.sl$  by the
14    # order  $T_i \rightarrow T_j$ 
15    if  $T_j^s$  has been local validated then
16      wait until Commit( $T_j^s, T_j.c$ ) finishes or
17      timeouts;
18      continue;
19    if  $T_i^s.LB \leq T_j^s.LB$  then
20       $T_i^s.LB \leftarrow T_j^s.LB + \mu_{j,i}$ ;
21     $T_j^s.UB \leftarrow \min\{T_j^s.UB, T_i^s.LB - 1\}$ ;
22    # unlock() releases the spinlocks  $T_i^s.sl, T_j^s.sl$  by
23    # the order  $T_i \rightarrow T_j$ 

```

**Algorithm 5:** Commit of Local Transaction  $T_i^s$ .

```

1 Function Commit ( $T_i^s, T_i.c$ ):
2   for  $\langle x.pk, x_i \rangle \in T_i^s.ws$  do
3     make  $x_i$  visible in the database;
4      $x.RTS \leftarrow \max\{x.RTS, T_i.c\}$ ;
5     # atomic RMW  $x.WT \leftarrow 0$ ;
6   for  $key \in T_i^s.rs$  do
7      $x.RTS \leftarrow \max\{x.RTS, T_i.c\}$ ;
8     # atomic RMW  $x.RIDs \leftarrow x.RIDs \setminus \{T_i^s\}$ ;
9 Function Abort ( $T_i^s$ ):
10  for  $\langle x.pk, x_i \rangle \in T_i^s.ws$  do CAS ( $x.WT, T_i^s, 0$ );
11  for  $key \in T_i^s.rs$  do  $x.RIDs \leftarrow x.RIDs \setminus \{T_i^s\}$ ;

```

commit successfully. We follow Algorithm 5 to commit  $T_3$  and write version  $y_3$  into  $y$ .

**F. Preventing Phantom Reads**

We present how BDTA handles the phantom read, which occurs when one transaction issues a predicate-based read twice but obtains two different read sets. Given two concurrent transactions  $T_1, T_2$ ,  $T_1$ 's snapshot  $T_1.ss$  might be larger than  $T_2$ 's commit timestamp due to inconsistent local clocks. When  $T_1$  triggers a predicate-based read through the index, it may observe  $T_2$ 's write after  $T_2$  commits, leading to the phantom read. We handle the phantom read by encapsulating BDTA into the index scheme. Without loss of generality, we assume predicate-based reads can be regarded as traversing the  $B^+$ -tree index. We treat each leaf node of the index as a data item, and we associate each leaf node (denoted as  $ln$ ) of the index with metadata  $ln.RTS, ln.WT$ , and  $ln.RIDs$  similar to the data item (Fig. 9). Because a predicate-based read needs to access leaf nodes of

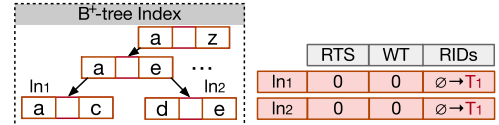


Fig. 10. Status of a  $B^+$ -tree in BDTA.

the index, and a write needs to update a leaf node, we then preserve the write-read order and write-legal order over the leaf nodes. Consider two concurrent transactions  $T_1$  and  $T_2$ .  $T_1$  has a predicate-based read to search keys in the range  $[a, d]$ . As shown in Fig. 10,  $T_1$  needs to access leaf nodes  $ln_1$  and  $ln_2$ , and  $T_1$  is added to  $ln_1.RIDs$  and  $ln_2.RIDs$ . Afterward, suppose that  $T_2$  writes the index key  $b$  to the leaf node  $ln_1$  and commits. BDTA preserves the write-legal order  $T_1 \rightarrow T_2$  by ensuring  $T_1.UB < T_2.LB$ . Since  $T_1.ss < T_2.c$  is preserved,  $T_1$  cannot observe the index key  $b$ . In this way, the phantom read is eliminated by BDTA.

**V. OPTIMIZATIONS**

In this section, we introduce a heuristic method to adaptively determine the size of the timestamp interval for each adjustment and explain how BDTA makes read-only transactions always commit.

**A. Adaptive Timestamp Interval Selection**

As discussed in Section IV-A, selecting a good timestamp interval size is essential to reduce the transaction abort rate. Reconsider Example 3. If no interval space exists between  $T_2.LB$  and  $T_1.LB$ , transactions ordered between  $T_2$  and  $T_1$  (e.g.,  $T_3$ ) would abort. Given any two transactions  $T_i$  and  $T_j$  with order constraint  $T_i \rightarrow T_j$ , we use  $\mu_{i,j}$  to denote the interval space  $[T_i.LB, T_j.LB]$  between  $T_i$  and  $T_j$ , i.e.,  $\mu_{i,j} = T_j.LB - T_i.LB$ . Theoretically, a proper  $\mu_{i,j}$  should meet the following two requirements. First, we require  $\mu_{i,j} > \mathcal{N}_{i,j}$ , where  $\mathcal{N}_{i,j}$  is the number of transactions ordered between  $T_i$  and  $T_j$ . By so doing, transactions ordered between  $T_i$  and  $T_j$  are more likely to find a legal timestamp interval and commit. Second, we need  $\mu_{i,j} \leq T_k.UB - T_i.UB$ , where  $T_k$  is the transaction with the smallest  $UB$  among transactions ordered after  $T_j$ . This property ensures transactions ordered after  $T_j$  will not be influenced by  $\mu_{i,j}$ . Otherwise, if  $\mu_{i,j} > T_k.UB - T_i.UB$ ,  $T_j.LB = T_i.UB + \mu_{i,j}$  can be larger than  $T_k.UB$ , causing transactions ordered after  $T_j$  (e.g.,  $T_k$ ) to abort.

Yet, computation of the best  $\mu_{i,j}$  for any two transactions  $T_i$  and  $T_j$  with constraint  $T_i \rightarrow T_j$  is infeasible because we cannot obtain the precise value of  $\mathcal{N}_{i,j}$  and detect  $T_k$  in advance. For this reason, we instead propose a heuristic method to adaptively estimate  $\mu_{i,j}$  based on the contention level of the data items  $T_i$  and  $T_j$  accessed. For each data item  $x$ , we collect the number of calls in bidirectional adjustment ( $x.cno$ ) to represent the contention level on  $x$ , denoted as  $\mathcal{L}(x)$ . Recall that during the validation phase of  $T_j$ , for each  $x \in T_j^s.ws$ , we use  $\mu_{i,j}$  to bidirectionally adjust the timestamp intervals of  $T_j^s$  and any

**Algorithm 6:** Adjust Timestamp Interval  $\mu_k$ .

---

```

1 Function AdaptiveAdjust ( $\mu_k, \mathcal{T}_{min}$ ):
2   Initialize  $\mathcal{F}^* \leftarrow \mathcal{F}(\mu_k), \mathcal{T}$ ;
3   while  $\mathcal{T} > \mathcal{T}_{min}$  do
4     Generate a random timestamp interval  $\hat{\mu}$ ;
5      $\Delta t \leftarrow \mathcal{F}(\hat{\mu}) - \mathcal{F}^*$ ;
6     if  $\Delta t < 0$  or ( $\Delta t \geq 0$  and  $e^{-\Delta t/c\mathcal{T}} > seed$ ) then
7        $\mathcal{F}^* \leftarrow \mathcal{F}(\hat{\mu})$ ;
8        $\mu_k \leftarrow \hat{\mu}$ ;
9      $\mathcal{T} \leftarrow \lambda \cdot \mathcal{T}$ ;
10  return  $\mu_k$ ;

```

---

other  $T_i^s$  in  $x.RIDs$  (line 18, Algorithm 4). Consequently, a higher contention level  $\mathcal{L}(x)$  indicates more transactions are likely to be ordered between  $T_i$  and  $T_j$ , which requires  $\mu_{i,j}$  to be positively correlated with  $\mathcal{L}(x)$ , i.e., the higher  $\mathcal{L}(x)$  is, the larger  $\mu_{i,j}$  should be assigned. We classify  $\mathcal{L}(x)$  into three contention levels: low, medium, and high contention, by simply comparing  $x.cno$  with two pre-defined thresholds  $\tau_1$  and  $\tau_2$ . Besides, we assign  $\mu_l, \mu_m$ , and  $\mu_h$  for each contention level to represent the optimal timestamp interval space of that contention level, as shown in (12).

$$\mu_{i,j} = \begin{cases} \mu_l & x.cno \leq \tau_1 & \text{low contention,} \\ \mu_m & \tau_1 < x.cno \leq \tau_2 & \text{medium contention,} \\ \mu_h & \tau_2 \leq x.cno & \text{high contention} \end{cases} \quad (12)$$

We adaptively refine these interval spaces during the execution based on Algorithm 6. Initially,  $\mu_l, \mu_m, \mu_h$  are set to 1. Then, we create an individual thread and periodically refine them using Algorithm 6, which is constructed based on the well-known simulated annealing (SA) algorithm. We take the timestamp interval to be adjusted  $\mu_k, \mu_k \in \{\mu_l, \mu_m, \mu_h\}$ , and the temperature threshold  $\mathcal{T}_{min}$  as the input. We denote  $\mathcal{F}(\mu_k)$  as the abort rate after applying  $\mu_k$ . After initialization (line 2), we iteratively select a random  $\hat{\mu}$  (line 4) and examine whether adopting  $\hat{\mu}$  can reduce the abort rate. If the abort rate drops, we update  $\mu_k = \hat{\mu}$ ; otherwise, we accept  $\hat{\mu}$  with a certain probability (lines 6–8). The probability follows the Boltzmann distribution by examining  $e^{-\Delta t/c\mathcal{T}}$  and a random value  $seed \in (0, 1)$ , where  $c$  is Boltzmann constant (line 6). For each iteration, temperature  $\mathcal{T}$  is decreased to  $\lambda \cdot \mathcal{T}$ , where  $\lambda$  is a hyper-parameter and set to 0.6 by default (line 9). We terminate the iteration and output  $\mu_k$  if  $\mathcal{T}$  is decreased to the temperature threshold  $\mathcal{T}_{min}$  (line 10).

### B. Non-Validation for Read-Only Transactions

We observe in BDTA, the timestamp interval of every read-only transaction  $T_i$  is always legal, i.e.,  $T_i.LB \leq T_i.UB$  is guaranteed. The reason is that according to (5)–(8),  $T_i.LB = T_i.ss$  and  $T_i.ss < T_i.UB$  are always true. Thus, we skip the validation in this case and replace the costly 2 PC with one phase commit for read-only transactions.

## VI. IMPLEMENTATION

In this section, we present our prototype system by integrating BDTA into Greenplum. Greenplum [21] is a distributed database

system technically built on top of PostgreSQL. It has a single coordinator (master) and several participant servers (segments). Each master/participant server runs a PostgreSQL instance. To integrate BDTA into Greenplum, we make the following extensions, and our implementation is publicly available via <https://github.com/dbir/BDTA>.

- *Storage Engine:* We re-construct the storage layer from the traditional heap store to the key-value store using RocksDB. We then implement data partitioning based on the hash strategy.
- *Multi-coordinator architecture:* We extend Greenplum to support multi-coordinator architecture. In this extension, each coordinator runs a PostgreSQL instance, in which each process coordinates transactions individually.
- *Timestamp Allocation:* We implement timestamp oracle and HLC as discussed to assign timestamps. To ensure high available timestamp allocation, timestamp oracle is implemented as a raft-based service. In our implementation, timestamp oracle serves around ten million timestamps per second. The performance may be influenced by high network latency over a WAN network.
- *Concurrency Control:* We integrate BDTA into Greenplum to support multi-level serializability. First, we replace the globally shared snapshot to avoid costly deadlock detection by our timestamp allocation schemes. We then encapsulate the validation phase and commit phase into 2 PC. To accommodate BDTA in Greenplum, we maintain read/write sets of transactions in segments to reduce the communication overhead. Besides, for simplicity, we store data items and their metadata for concurrency control separately. The metadata is stored in memory and indexed with a red-black tree. We further use a separate thread to execute Algorithm 6 to periodically update the optimal interval space for different contention levels.

## VII. EVALUATION

Our experimental evaluation is conducted from two perspectives. First, we integrate BDTA and the state-of-the-art concurrency control algorithms into a distributed transaction testbed, called Deneva [22]. We compare them in the same context and report our findings. Second, we conduct experiments on Greenplum integrated with BDTA to verify the necessity of introducing multi-level serializability.

### A. Workloads and Experiment Setup

We use the following workloads to conduct the experiments:

*YCSB* [13] is a synthetic benchmark modeling large-scale Internet applications. It uses a relation with 10 attributes, in which one is taken as the primary key. Each record in this relation occupies 1 KB. The dataset is horizontally partitioned, and each partition is assigned to a participant server. Following Deneva [22], we set each partition to have 16 million records, indicating the data size of each participant server is 16 GB. By aiming to simulate different contention levels, we follow Zipfian distribution to control the access on the same records using a skew factor, denoted as  $\theta$ . When  $\theta = 0$ , we access each

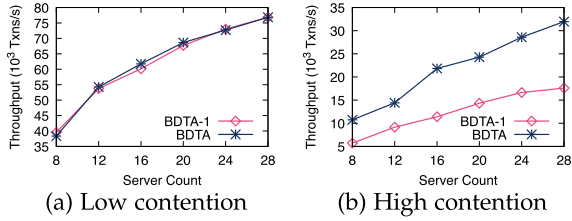


Fig. 11. Effect of adaptive timestamp interval selection.

record in equal probability. Besides, we vary the *write ratio* to control the ratio of reads and writes by taking operations from transactions as a whole, i.e., *write ratio* = 50% means there are totally 50% writes and 50% reads in transactions. By default, we set *write ratio* = 50% and  $\theta = 0.6$ .

TPC-C [46] is a popular OLTP benchmark simulating a warehouse order processing application. It contains 9 relations. Each warehouse contains 100 MB data size, and by default, we set 32 warehouses per participant server. TPC-C simulates 5 types of transactions, in which NewOrder, Payment, and Delivery are read-write transactions, and Stock-Level and Order-Status are read-only transactions (a transaction with both reads and writes is referred to as a read-write transaction). Following Deneva [22], we do not include “think time” and user data errors that cause 1% of NewOrder transactions to abort, aiming to test the peak performance. Unless otherwise specified, we use the default transaction mix of standard TPC-C in our experiments.

We conduct experiments except Section VII-E using an in-house cluster with 28 virtual nodes, each of which has 4 cores/8 threads and 32 GB memory. Unless otherwise specified, we run the protocols on 16 nodes, each containing 1 coordinator and 1 participant server. The RTT of the network is around 0.3 ms. For each experiment, we first run 30 s for warm-up and then collect results of the following 60 s. We evaluate the performance in terms of: (1) throughput, which is the number of committed transactions; and (2) abort rate, which is the percentage of aborted transactions against all finished transactions.

### B. Effect of Adaptive Timestamp Interval Selection

We first study the effectiveness of the adaptive timestamp interval selection method by comparing BDTA and BDTA without adaptive timestamp interval (denoted as BDTA-1) under sequential serializability. We set  $\mu$  adaptively in BDTA, and fix  $\mu = 1$  in BDTA-1. We run them under the low contention workload ( $\theta = 0.25$ ) and high contention workload ( $\theta = 0.75$ ), respectively. We vary the cluster from 8 to 28 nodes, and plot the results in Fig. 11. Fig. 11(a) shows that under the low contention workload, BDTA achieves comparable performance with BDTA-1, showing the additional cost for running the auto-tuning algorithm is negligible. We further study the benefit of adaptive timestamp interval selection over the high contention workload, and plot the results in Fig. 11(b). We can observe that BDTA achieves higher throughput and better scalability than BDTA-1 by up to 91.42%. The performance of BDTA gains from the adaptive timestamp interval selection, which helps most

TABLE II  
COMPARISON OF THE ABORT RATE BETWEEN BDTA AND BDTA-1

|        | Low Contention | High Contention |
|--------|----------------|-----------------|
| BDTA   | 0.80%          | 55.37%          |
| BDTA-1 | 0.88%          | 77.33%          |

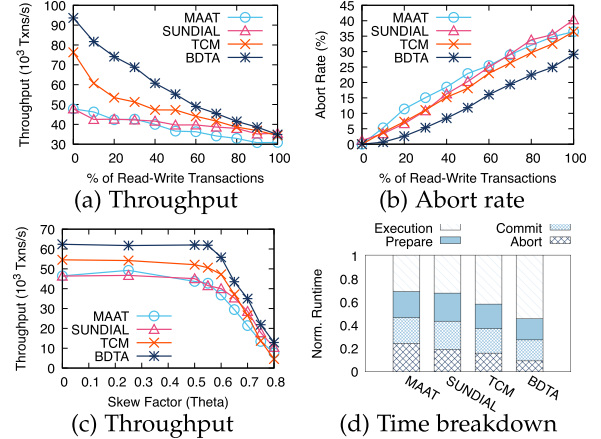


Fig. 12. Serializability with varying  $\theta$  and *write ratio*.

transactions get a proper timestamp interval, thereby reducing the abort rate. As shown in Table II, although more transactions need to abort under the high contention workload, the abort rate of BDTA is lower than that of BDTA-1 by a factor of 21.96%.

In the following experiments, we adopt the adaptive timestamp interval selection in BDTA by default.

### C. Comparisons With Dynamic Ordering Algorithms

We compare BDTA with three recently proposed concurrency control algorithms using the DTA scheme, i.e., **MaaT** [31], **Sundial** [55], and **TCM** [30]. Because MaaT and Sundial only support serializability, we conduct our experiments under serializability for fair comparisons.

We make the comparison under different contentions by varying *write ratio*, and plot the results in Fig. 12(a). As we can see, for the read-only transactions (0% of read-write transactions), BDTA outperforms the others by a factor of 38.66%. This is because, for read-only transactions, BDTA eliminates the expensive 2 PC cost, and hence reduces the coordination overhead. Besides, when the percentage of read-write transactions increases, all algorithms suffer higher abort rates (shown in Fig. 12(b)), causing the performance to drop. Due to the adaptive timestamp interval selection, transactions are more likely to obtain proper timestamp intervals and commit, and therefore, BDTA performs the best with the lowest abort rate.

We evaluate the effect of  $\theta$ , and plot the results in Fig. 12(c). BDTA outperforms MaaT, Sundial, and TCM by up to 22.32% due to three reasons: First, MaaT and Sundial are single-version based, while BDTA is multi-version based, which allows reads do not block by the writes to increase concurrency. Second, MaaT and Sundial have to issue expensive 2 PC for read-only transactions to commit while BDTA does not. Third, MaaT,

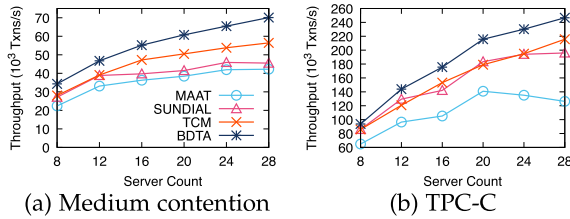


Fig. 13. Scalability under serializability.

Sundial, and TCM use a fixed space for timestamp adjustment, while BDTA leverages the adaptive timestamp interval selection, which further reduces abort rates and improves performance. The time breakdown in Fig. 12(d) with  $\theta = 0.6$  indicates that the time spent on aborting transactions in MaaT, Sundial, and TCM is much higher than that in BDTA.

We next investigate the scalability by varying the cluster from 8 to 28 nodes, and plot the results in Fig. 13.

1) *Scalability over YCSB*: We study the scalability over the medium contention workload ( $\theta = 0.6$ ), and plot the results in Fig. 13(a). We can observe that BDTA achieves up to 24.23% performance gain and the best scalability when the number of nodes varies.

2) *Scalability over TPC-C*: We further evaluate the performance under the TPC-C workload, and report the results in Fig. 13(b). In this experiment, we customize the TPC-C workload with 50% NewOrder transactions and 50% Payment transactions. BDTA still achieves up to 20.51% higher throughput over the next-best algorithm. Again, the scalability benefit of BDTA mainly comes from our special design that adjusts timestamp bidirectionally, which reduces the overhead of coordinators. As discussed in Section IV-E, each transaction in BDTA locally adjusts timestamp intervals in involved participant servers, and coordinators are just responsible for collecting all local timestamp intervals.

#### D. Comparisons With Static Ordering Algorithms

We compare BDTA with three static ordering concurrency control algorithms under sequential serializability: **2PL** [4], **MVCC** [5], and **Silo** [47]. For 2PL, we implement the No-Wait variant to prevent deadlock. We implement MVCC by ordering transactions based on their start timestamps. Silo is an OCC-based algorithm and uses the serialization point to order transactions, and we extend it into distributed setting according to Google F1 [41]. We make local timestamps of each process monotonically increase using HLC, which is capable of preserving the program order.

We first study the effect of contentions by varying the skew factor  $\theta$ , and plot the results in Fig. 14. As we can see, BDTA performs up to 56.58% better than the next-best algorithm. As shown in Fig. 14(a), when  $\theta < 0.6$ , Silo performs the worst because Silo introduces additional overhead in the validation phase, where a transaction reads data items in its read set again to examine whether they remain unchanged. When  $\theta$  reaches 0.6, the cost of aborting transactions increases and becomes the

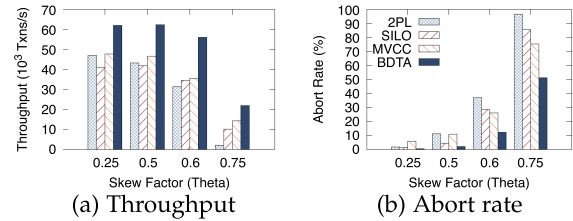
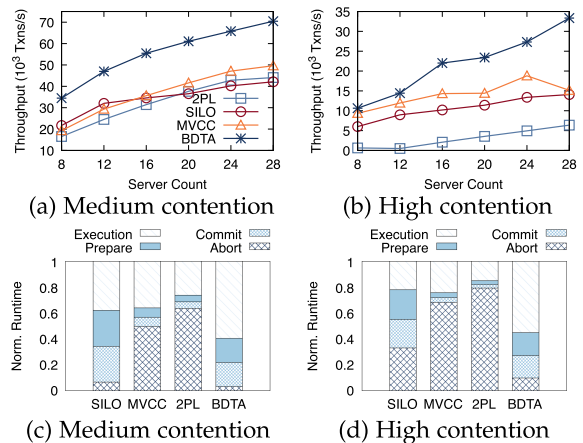
Fig. 14. Sequential serializability with varying  $\theta$ .

Fig. 15. Scalability under sequential serializability.

bottleneck for 2PL, MVCC, and Silo, which can be verified in Fig. 14(b). Because BDTA orders transactions dynamically, BDTA shows a better tolerance on contentions, leading to a higher throughput.

We then perform the scalability evaluation under sequential serializability. As observed in Fig. 15, BDTA achieves up to  $1.19 \times$  higher throughput than the second-best algorithm under the medium contention workload ( $\theta = 0.6$ ) and high contention workload ( $\theta = 0.75$ ). The performance of BDTA is mainly due to the bidirectional timestamp adjustment mechanism, ensuring the lowest abort rate, as verified in Fig. 15(c) and (d).

#### E. Comparisons of Multi-Level Serializability

In this section, we conduct experiments on an in-house cluster with 3 high-performance nodes running CentOS 7.4. Each node has two Intel(R) Xeon(R) Platinum 8276 CPUs (28 cores  $\times$  2 HT), 8  $\times$  128 GB DRAM, and 3 TB NVMe SSDs. We deploy Greenplum with 2 coordinators and 3 participant servers in this cluster. Each node hosts at most 1 coordinator and 1 participant server. By default, the Round-Trip Time (RTT), an indicator to measure the network latency, in the cluster is 0.03 ms. To better evaluate the performance of different serializability levels, we set RTT=1.5 ms to simulate the deployment over a WAN network (e.g., a cross datacenter cluster). Note that setting RTT=1.5 ms to simulate a cross datacenter deployment is reasonable. For example, the RTT from New York to Dallas is 40 ms [35]. Thus, we use RTT = 0.03 ms, and RTT=1.5 ms to simulate the network latency over a LAN network, and WAN network, respectively.

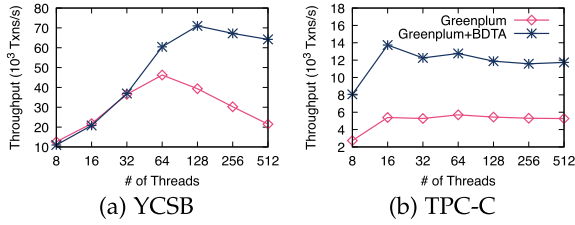


Fig. 16. Effect of integrating BDTA into Greenplum.

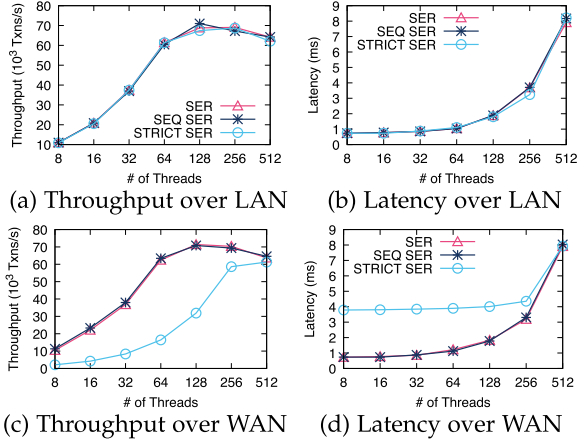


Fig. 17. Multi-level serializability on Greenplum+BDTA.

We first study the effectiveness of integrating BDTA into Greenplum (denoted as Greenplum+BDTA), and report the comparison between Greenplum+BDTA and Greenplum in Fig. 16. Note that Greenplum+BDTA is set under the sequential serializability level. Since Greenplum only supports the read committed and repeatable read isolation level, we set Greenplum under the read committed level to obtain the peak performance. As we can see, even if Greenplum is set under the read committed level, Greenplum+BDTA still outperforms Greenplum by a factor of up to  $2.01 \times$  and  $1.95 \times$  on YCSB and TPC-C workload, respectively. The reason is two-fold. On the one hand, Greenplum+BDTA can tolerate more transaction concurrency, which leads to better performance. As mentioned, BDTA is employed in Greenplum+BDTA which reduces the abort rate and improves performance. On the other hand, Greenplum coordinates transactions with a costly distributed deadlock detection component, which is eliminated in Greenplum+BDTA.

We then report the experimental results of executing Greenplum+BDTA under different serializability levels over LAN in Fig. 17(a) and (b). We observe that the performance under strict serializability (labeled as STRICT SER), sequential serializability (labeled as SEQ SER), and serializability (labeled as SER) almost coincide. This is because, in a low latency network environment, like LAN, the effect of requesting timestamps from timestamp oracle service on the overall performance is negligible. The main cost comes from doing concurrency control, which is roughly the same under strict serializability and sequential serializability.

We finally report the experimental results over a simulated WAN network in Fig. 17(c) and (d). We find that sequential serializability and serializability almost perform the same, and their throughput is up to  $4.53 \times$  higher than that of strict serializability. The reason is that, in a high latency network environment, like WAN, the cost of requesting timestamps is comparable to that of doing concurrency control, and probably becomes a dominant factor to the overall performance (could be verified in Fig. 17(d)). Besides, by varying the number of client connections from 8 to 128, sequential serializability and serializability take an increasingly significant benefit against strict serializability. Yet, by adding more client connections, the contentions among transactions become the bottleneck and cause the performance to drop.

### F. Summary

We summarize the major experimental findings below:

- We show the efficiency and effectiveness of the adaptive timestamp interval selection method, which reduces the abort rate by up to 21.96% and improves the throughput by up to 91.42% (Section VII-B).
- We confirm that BDTA outperforms the state-of-the-art concurrency control algorithms, including dynamic ordering and static ordering algorithms (Section VII-C and D).
- We recommend using strict serializability in the low latency network, e.g., LAN, and sequential serializability in the high latency network, e.g., WAN (Section VII-E).

## VIII. RELATED WORK

Our study relates to formalizing consistency and isolation levels, as well as distributed concurrency control algorithms.

In ACID databases, isolation levels are typically defined by disallowing certain kinds of data anomalies. The ANSI/ISO SQL-92 specifies four data anomalies (e.g., dirty write/read) and defines four isolation levels accordingly [51]. By arguing that the definitions in SQL-92 lack mathematical formalization and could incur ambiguous interpretations, a few works make formal re-definitions of data anomalies [1], [3], [15], [19], [43]. Much effort has been devoted to the identification of new data anomalies, including skewed read/write [3], aborted read [53], intermediate read [53], etc. There are quite a few works to model data consistency from different perspectives, e.g., result visibility [43], state matrix [15], dependency graph [1], [2], and abstract execution [11]. Recently, there is an increasing interest in imposing consistency models [11], [24], [27], [28], [49] on isolation levels. Quite a few works [12], [18], [48] impose additional constraints like the causal-related order on snapshot isolation [3]. Salt [52] imposes the eventual consistency [11] on ACID transactions to provide BASE transactions. To be more related, strict serializability [40] imposes the real-time order on serializability. Strong session serializability [16] imposes the program order on serializability. Lynx [56] studies serializability with the read-your-writes order. These works impose partial orders on serializability in a case-by-case manner. We model multi-level serializability to provide a systematic analysis of consistency levels over serializability. It is worth mentioning that

strong session serializability [16] and Lynx [56] can be reduced to sequential serializability.

Distributed concurrency control algorithms are typically divided into two categories: (1) static ordering algorithms and (2) dynamic ordering algorithms. The first category determines the order of transactions statically. T/O [4] orders transactions based on their start timestamps. OCC [26] and its variants like Silo [47] determine the order based on either the validation point or the serialization point. 2PL [4], [19] orders transactions by the first granted lock on conflict data items. Calvin [45] uses a deterministic method to order transactions before execution. Imposing T/O over MVCC [39] can potentially support sequential serializability by the monotonic increasing local timestamp and strict serializability by the timestamp oracle [38]. Yet, the static ordering could cause a high abort rate due to their strict order requirements, which is verified in our experiments. On the contrary, the second category determines the order of transactions dynamically. Similar to BDTA, they determine the order by adjusting the timestamp intervals of transactions. Boksenbaum et al. are the first to use DTA in distributed concurrency control [9]. MaaT [31] and Sundial [55] are single-version based, and employ logical timestamps to do the adjustment. TCM [30] integrates DTA into the multi-version 2PL protocol, which is mainly designed for centralized databases. TCM requires all concurrent transactions to shrink their timestamp intervals upon a conflict, which could incur unnecessary adjustment overhead for aborted transactions. BDTA eliminates this overhead by only adjusting other transactions' timestamp intervals during the validation. BDTA is different from the other algorithms. First, BDTA adjusts timestamp intervals to preserve required orders in multi-level serializability. Second, BDTA adopts the adaptive timestamp interval selection, leading to a lower transaction abort rate.

## IX. CONCLUSION

In this paper, we study serializability from different consistency perspectives and formalize multi-level serializability. To support multi-level serializability, we propose a novel concurrency control algorithm called BDTA. BDTA can dynamically order serializable transactions and preserve partial orders among transactions required in the consistency models. We integrate BDTA into Greenplum, and release the implementation as open source. We conduct extensive experiments to show the necessity of introducing multi-level serializability and the performance gain of BDTA compared with state-of-the-art concurrency control algorithms.

## ACKNOWLEDGMENTS

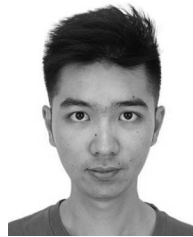
We would like to thank the anonymous reviewers for their constructive comments.

## REFERENCES

- [1] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Tech. Rep., 1999. [Online]. Available: <https://dl.acm.org/doi/10.5555/888672>

- [2] P. Bailis, A. Davidson, A. D. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," in *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, 2013.
- [3] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1995, pp. 1–10.
- [4] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185–221, 1981.
- [5] P. A. Bernstein and N. Goodman, "Multiversion concurrency control - theory and algorithms," *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 465–483, 1983.
- [6] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA, USA: Addison-Wesley, 1987.
- [7] P. A. Bernstein, D. W. Shipman, and W. S. Wong, "Formal aspects of serializability in database concurrency control," *IEEE Trans. Softw. Eng.*, vol. 5, no. 3, pp. 203–216, May 1979.
- [8] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," in *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 528–539, 2016.
- [9] C. Boksenbaum, M. Cart, J. Ferrié, and J. Pons, "Concurrent certifications by intervals of timestamps in distributed database systems," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 4, pp. 409–419, Apr. 1987.
- [10] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz, "Overview of multi-database transaction management," *VLDB J.*, vol. 1, no. 2, pp. 181–239, 1992.
- [11] S. Burckhardt, A. Gotsman, and H. Yang, "Understanding eventual consistency," Microsoft Research Tech. Rep. MSR-TR-2013-39, 2013.
- [12] A. Cerone, A. Gotsman, and H. Yang, "Transaction chopping for parallel snapshot isolation," in *Proc. Int. Symp. Distrib. Comput.*, Springer, 2015, pp. 388–404.
- [13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
- [14] J. C. Corbett, "Spanner: Google's globally-distributed database," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 251–264.
- [15] N. Crooks, Y. Pu, L. Alvisi, and A. Clement, "Seeing is believing: A client-centric specification of database isolation," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2017, pp. 73–82.
- [16] K. Daudjee and K. Salem, "Lazy database replication with ordering guarantees," in *Proc. IEEE Int. Conf. Data Eng.*, 2004, pp. 424–435.
- [17] M. Demirbas, M. Leone, B. Avva, D. Madeppa, and S. Kulkarni, "Logical physical clocks and consistent snapshots in globally distributed databases," 2014.
- [18] S. Elnikety, W. Zwaenepoel, and F. Pedone, "Database replication using generalized snapshot isolation," in *Proc. IEEE 24th Symp. Reliable Distrib. Syst.*, 2005, pp. 73–84.
- [19] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in *Proc. IFIP Work. Conf. Modelling Data Base Manage. Syst.*, 1976, pp. 365–394.
- [20] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 1993.
- [21] Greenplum. [Online]. Available: <https://greenplum.org/>
- [22] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," in *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 553–564, 2017.
- [23] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proc. Int. Symp. Distrib. Comput.*, Springer, 2001, pp. 300–314.
- [24] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Prog. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.
- [25] D. Huang et al., "TiDB: A raft-based HTAP database," in *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3072–3084, 2020.
- [26] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.
- [27] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [28] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.
- [29] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, pp. 51–58, Dec. 2001.

- [30] D. B. Lomet, A. D. Fekete, R. Wang, and P. Ward, "Multi-version concurrency via timestamp range conflict management," in *Proc. IEEE Int. Conf. Data Eng.*, 2012, pp. 714–725.
- [31] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. E. Abbadi, "MaaT: Effective and scalable coordination of distributed transactions in the cloud," in *Proc. VLDB Endow.*, vol. 7, no. 5, pp. 329–340, 2014.
- [32] D. L. Mills, "A brief history of NTP time: Memoirs of an internet timekeeper," *Comput. Commun. Rev.*, vol. 33, no. 2, pp. 9–21, 2003.
- [33] H. Moniz, J. Leitão, R. J. Dias, J. Gehrke, N. M. Pregoça, and R. Rodrigues, "Blotter: Low latency transactions for geo-replicated storage," in *Proc. Int. Conf. World Wide Web*, 2017, pp. 263–272.
- [34] D. Mosberger, "Memory consistency models," *Operating Syst. Rev.*, vol. 27, no. 1, pp. 18–26, 1993.
- [35] NetworkLatency, 2023. [Online]. Available: <https://wondernetwork.com/pings>
- [36] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.
- [37] A. Pavlo and M. Aslett, "What's really new with NewSQL?," *SIGMOD Rec.*, vol. 45, no. 2, pp. 45–55, 2016.
- [38] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 251–264.
- [39] D. P. Reed, "Naming and synchronization in a decentralized computer system," PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.
- [40] K. Ren, D. Li, and D. J. Abadi, "SLOG: Serializable, low-latency, geo-replicated transactions," in *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1747–1761, 2019.
- [41] J. Shute et al., "F1: A distributed SQL database that scales," in *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1068–1079, 2013.
- [42] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. ACM Symp. Operating Syst. Princ.*, 2011, pp. 385–400.
- [43] A. Szekeres and I. Zhang, "Making consistency more consistent: A unified model for coherence, consistency and isolation," in *Proc. 5th Workshop Princ. Pract. Consistency Distrib. Data*, 2018, pp. 7:1–7:8.
- [44] R. Taft et al., "CockroachDB: The resilient geo-distributed SQL database," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1493–1509.
- [45] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 1–12.
- [46] TPC-C, 1988. [Online]. Available: <http://www.tpc.org/tpcc/>
- [47] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. ACM Symp. Operating Syst. Princ.*, 2013, pp. 18–32.
- [48] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, and J. Mulrow, "Implementation of cluster-wide logical clock and causal consistency in mongoddb," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2019, pp. 636–650.
- [49] P. Viotti and M. Vukolic, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surveys*, vol. 49, no. 1, pp. 19:1–19:34, 2016.
- [50] X. Wei, R. Chen, H. Chen, Z. Wang, Z. Gong, and B. Zang, "Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 357–372.
- [51] A. X3.135–1992, *American National Standard for Information Systems - Database Language-SQL*. Washington, D.C., USA: American National Standards Institute, 1992.
- [52] C. Xie et al., "Salt: Combining ACID and BASE in a distributed database," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 495–509.
- [53] C. Xie, C. Su, C. Littlely, L. Alvisi, M. Kapritsos, and Y. Wang, "High-performance ACID via modular concurrency control," in *Proc. ACM Symp. Operating Syst. Princ.*, 2015, pp. 279–294.
- [54] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas, "TicToc: Time traveling optimistic concurrency control," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1629–1642.
- [55] X. Yu, Y. Xia, A. Pavlo, D. Sánchez, L. Rudolph, and S. Devadas, "Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system," in *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1289–1302, 2018.
- [56] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, "Transaction chains: Achieving serializability with low latency in geo-distributed storage systems," in *Proc. ACM Symp. Operating Syst. Princ.*, 2013, pp. 276–291.



**Zhanhao Zhao** is currently working toward the PhD degree with the School of Information, Renmin University of China. His research interests include distributed database systems and transaction processing.



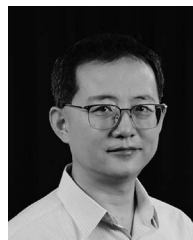
**Hongyao Zhao** is currently working toward the PhD degree with the School of Information, Renmin University of China. His research interests include distributed databases and transaction processing.



**Qiyu Zhuang** is currently working toward the PhD degree with the School of Information, Renmin University of China. His research interests include distributed database systems and transaction processing.



**Wei Lu** received the PhD degree in computer science from the Renmin University of China in 2011. He is currently a professor with the Renmin University of China. His research interests include query processing in the context of spatiotemporal, cloud database systems and applications.



**Haixiang Li** is currently a senior expert with Tencent. His research interests include transaction processing, query optimization, distributed consistency, high availability, database system architecture, cloud database and distributed database systems.



**Meihui Zhang** (Senior Member, IEEE) is currently a professor with the Beijing Institute of Technology, China. Her research interests include crowdsourcing-powered data analytics, massive data integration, and spatio-temporal databases.



**Anqun Pan** is a technical director of Tencent Billing Platform Department, and has more than 15 years of experience in the research and development of distributed computing and storage systems. He is currently responsible for the research and development of distributed database system (TDSQL).



**Xiaoyong Du** (Member, IEEE) received the PhD degree from the Nagoya Institute of Technology in 1997. He is a professor with the Renmin University of China. His research focuses on intelligent information retrieval, high-performance database, and unstructured data management.