Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp

Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, Binyu Zang
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

This paper presents DST, a *decentralized scalar timestamp* scheme to scale distributed transactions using multi-version concurrency control (MVCC). DST is efficient in storage and network by being a scalar timestamp but requiring no centralized timestamp service for coordination, which may become a scalability bottleneck. The key observation is that concurrency control (CC) protocols like OCC and 2PL already imply a serializable order among concurrent read-write transactions through conflicting database tuples. To this end, DST piggybacks on CC protocols to maintain the timestamp ordering with low cost and no new scalability bottleneck for read-write transactions. DST further provides snapshot reads with bounded staleness by using a hybrid scalar timestamp (physical clock and logical counter).

To demonstrate the generality of DST, we provide a general guideline for the integration of DST and further show the effectiveness by using three representative transactional systems (i.e., DrTM+R, MySQL cluster, and ROCOCO) with different CC protocols. Experimental results show that DST can achieve more than 95% of optimal performance (using Read Committed) without compromising correctness. With DST, DrTM+R achieves up to 1.8X higher peak throughput for `TPC-E` and outperforms other timestamp schemes by 6.3X for `TPC-C`. DST also leads up to 1.9X and 2.1X speedup on `TPC-C` for MySQL cluster and ROCOCO, respectively.

1 Introduction

Many large-scale applications like Web services, stock exchange, and e-commerce require accessing scalable sharded data stores in a consistent way. Among such accesses, a large fraction requires consistently scanning data over many shards despite concurrent updates on the fly. For example, an examination of `TPC-E` [57], a sophisticated online transaction processing benchmark that models stock exchange, uncovers that 79% of transactions are read-only ones at run time. It was also reported that 99.8% of accesses to Facebook's distributed data store TAO are reads [16], which need strong consistency along with transactional writes [7].

However, it is costly to provide transactional isolation to read-only transactions [39] because a user read request may result in thousands of sub-queries [7]. Pessimistically executing a read-only transaction may cause unnecessary blocking to itself and concurrent read-write transactions, while optimistically executing it is likely to cause excessive aborts. For instance, as shown in Fig. 1(a), there is a notable performance gap between using optimistic concurrency control (OCC) [29] and read committed (RC) protocol for `TPC-E`.[1]

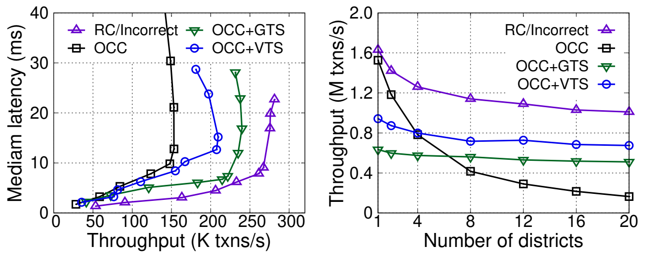

Fig. 1: *Performance of (a)* `TPC-E` *and (b)* `TPC-C` *on a local 16-node cluster using different CC protocols and TS schemes (see §5 for details). GTS and VTS stand for using OCC protocol, while the read-only transactions read the snapshots delimitated by GTS and VTS. RC/Incorrect stands for using RC protocol, which can provide optimal performance, but at the expense of correctness. One machine is dedicated for timestamp oracle, even OCC and RC have no need.*

A common approach is to leverage multi-version concurrency control (MVCC) [13, 65] for transactional systems, which has been widely adopted by nearly every commercial database like PostgreSQL [3], Oracle [4], MySQL/InnoDB [2], Hekaton [21], and SAP HANA [50]. MVCC simultaneously maintains multiple database snapshots by using timestamps to delimitate them. Thus, readers may read tuples from a stale snapshot while writers can write the tuples concurrently. It essentially unleashes the parallelism between concurrent readers and writers.

While MVCC extracts more concurrency for transactions (especially for read-only transactions), it does not necessarily approach optimal performance and/or scalability improvement (see Fig. 1), due to the overhead of maintaining timestamp ordering at scale (§2.3). More specifically, a centralized sequencer (timestamp oracle) is usually used to provide snapshot timestamp to transactions, which reflects a total order among transactions (i.e., global timestamp (GTS)). However, such a mechanism not only adds more communications but also causes overly-constrained concurrency control for read-write transactions, leading to performance degradation and scalability bottlenecks [14]. Vector timestamp (VTS), which leverages a clock per worker or machine, only mitigates the scalability bottleneck of centralized timestamp schemes but causes more network traffic, which grows lin-

[1] We evaluate different timestamp schemes on DrTM+R [18]. RC cannot provide correct results as it completely disregards the conflicts between read-write and read-only transactions. Detailed setup can be found in §5.

early with the increase of workers or machines in the system. Although recent work ameliorates the performance of centralized and/or vectorized timestamp schemes (e.g., batching requests [46, 27], timestamp compression [70], and dedicated fetch thread [70]), the fundamental performance and scalability bottlenecks remain.

In this paper, we propose a new timestamp scheme, namely, *decentralized scalar timestamp* (DST), which enables MVCC without a centralized sequencer or vector timestamps. DST is motivated by a key observation: *transaction ordering provided by existing CC protocols already implies serializable ordering among transactions, which can be reused to maintain timestamp ordering in a lightweight and scalable way.* This is because any pair of conflicting transactions must have conflicting accesses to a particular tuple. Thus, the *later* transaction should see the timestamp of the *former* transaction from the conflicting tuple and have a *larger* timestamp.

DST piggybacks on CC protocols to derive a scalable timestamp, in contrast to providing a separate timestamp scheme. Specifically, DST starts with a scalar timestamp for each transaction from a local clock and dynamically refines the tentative timestamp through transaction execution with the largest one from tuples in the read/write set. Upon commit, a transaction will also install the refined timestamp to the read/write set so that any transactions serialized after this transaction will have a larger timestamp.

One key challenge is how to derive a *consistent* yet *fresh* snapshot. DST leverages a decentralized design for read-only transactions, which introduces a *hybrid* scalar timestamp to provide snapshot reads with *bounded staleness*. Specifically, the read-only transaction can read fresh tuples whose timestamp is within two times the maximum physical clock drift under loosely synchronized clocks.[2] The fresh and consistent snapshot is obtained by attempting to read tuples using the latest hybrid timestamp while detecting and reordering any concurrent conflicting read-write transactions. In the hybrid timestamp, the physical part (a loosely synchronized clock) ensures the read-only transaction can read a fresh snapshot, and the logical part (a monotonically increasing counter) avoids possible overflow of the physical part.

To demonstrate the effectiveness and generality of DST, we have implemented DST on three representative transactional systems with different CC protocols, namely DrTM+R [18] (OCC), MySQL cluster [1] (2PL), and ROCOCO [43]. We also implemented two centralized timestamp schemes (GTS and VTS) on DrTM+R by following the state-of-the-art [46, 70]. The experimental results on three clusters show that DST can achieve more than 95% of optimal performance (using RC protocol) without compromising correctness. With DST, DrTM+R achieves up to 1.8X and 6.1X performance improvements for `TPC-E` and `TPC-C`.

[2]The clock drift (aka clock skew) can be obtained using a network time protocol like the precision time protocol (PTP), which only affects the freshness of reads in DST rather than correctness.

A comparison with other timestamp schemes shows DST is up to 1.7X and 6.3X faster than best of them for `TPC-E` and `TPC-C`, respectively. Further, DST also leads up to 1.9X and 2.1X speedup on `TPC-C` for MySQL cluster and ROCOCO.

DST shares some similarities with decentralized timestamps proposed in prior work [68, 37], which optimize a specific CC protocol for multi-core databases. For instance, TicToc [68] uses a data-driven timestamp scheme to reduce transaction aborts for OCC. Differently, DST is a general timestamp scheme for various CC protocols (§4.2) and can piggyback on each one efficiently in a distributed setting.

In summary, the contributions of this paper are:

- A *decentralized scalar* timestamp scheme called DST for MVCC that enables efficient read-only transactions with little impact on read-write transactions (§3.1 and §3.2), as well as an intuitive proof of correctness (§3.3).
- A *consistent* yet *fresh* snapshot-read approach based on a hybrid timestamp that provides bounded staleness (§3.4).
- To demonstrate the generality, DST is integrated into three representative transactional systems with different CC protocols, including OCC, 2PL, and ROCOCO (§4).
- A set of evaluations on three clusters with both microbenchmarks and applications (e.g., `TPC-E`, `TPC-C`, and `SmallBank`) confirms the performance gains of DST (§5).

The source code of three transactional systems with DST, including all benchmarks and experimental results, are available at `https://github.com/SJTU-IPADS/dst`.

2 Background and Motivation

2.1 Target Systems

DST is designed for general distributed transactions over database data partitioned to multiple storage nodes. The client's transaction request is handled by a coordinator, which interacts with storage nodes for executing the transaction. During the transaction's execution, the coordinator may send read/write requests to read/write data from the storage nodes; or send transactional requests (e.g., lock or unlock) according to the database's concurrency control protocol. It batches requests (e.g., write and unlock) to avoid extra network roundtrip.

Our goal is to support serializable read-only transaction that never aborts, and does not interfere with read-write transaction. Further, it is desirable to execute reads in the read-only transaction in one-roundtrip, i.e., the coordinator can retrieve a consistent view of the data from the storage nodes in one request.

2.2 MVCC and Timestamps

A common approach to support serializable read-only transaction without interfering with read-write transaction is through multi-version concurrency control (MVCC). There are two major design considerations for an efficient MVCC system compared with single-version mechanisms [65]. The first is *how to cheaply allocate a globally-ordered version*

```
2PL with global timestamp (GTS)

At Oracle:                               ▸ timestamp server
+ GlobalTS                     ▸ monotonic global timestamp
+ StableGTS                     ▸ snapshot global timestamp
+ Queue                          ▸ pending global timestamp

  INSTALL(gts):
+1  add gts to Queue

  STABILIZE( ):                        ▸ run asynchronously
+1  for each gts in Queue do
+2    if gts is ready then
+3      dequeue gts and gts → StableGTS

At Workeri:                  ▸ i denotes the worker number
  WRITE(tx, id, data)
 1  acquire lock
 2  add ⟨id, data⟩ to tx.wset

  READ(tx, id)
 1  acquire lock and get latest ⟨data⟩
 2  add ⟨id, data⟩ to tx.rset
 3  return ⟨data⟩

  COMMIT(tx)
+1  tx.TS ← Oracle.GlobalTS         ▸ network round trip
 2  for each w in tx.wset do
:3    update ⟨w.data, tx.TS⟩ and release lock
 3  for each r in tx.rset do
 4    release lock
+5  Oracle.INSTALL(tx.TS)           ▸ network round trip

  ROTX(tx)                                  ▸ snapshot read
+1  tx.TS ← Oracle.StableGTS
+2  for each r in tx.rset do
+3    get ⟨r.data⟩ up to tx.TS
```

Fig. 2: *Using GTS (i.e., blue code lines) to enable consistent snapshots for read-only transactions with 2PL.* `+N` *and* `:N` *denote new and modified lines of code respectively.*

for updating tuples transactionally. Deciding the version installed with tuples should have minimal impacts on read-write transactions. The second is *how to efficiently allocate a freshly-stable version for reading tuples consistently*. Read-only transactions should have access to consistent snapshots with low latency and high freshness. MVCC schemes typically adopt the concept of *timestamps* for tuple versions. However, it is non-trivial to design a general timestamp scheme that supports *efficient* snapshot reads while incurring *minimal overhead* for broad CC protocols.

To motivate the design of DST, we start by briefly reviewing how existing timestamp schemes are applied to two-phase locking (2PL) for MVCC and snapshot reads [45, 65].

Global timestamp (GTS). This approach leverages a timestamp service, namely *timestamp oracle*, to manage globally ordered timestamps [46, 15, 21]. It provides two functions for MVCC systems, as shown in Fig. 2. First, the read-write transaction contacts the oracle for a commit timestamp (GlobalTS) at the commit phase. Upon a successful commit, this transaction creates a new version denoted by the commit timestamp for each tuple in the write set (*line:3* of COMMIT) and sends back the committed timestamp to the oracle (*line:5*). Second, the read-only transaction contacts the oracle for a read timestamp (StableGTS) and retrieves tuples in the read set with versions no larger than the read timestamp (*line:2-3* of ROTX).

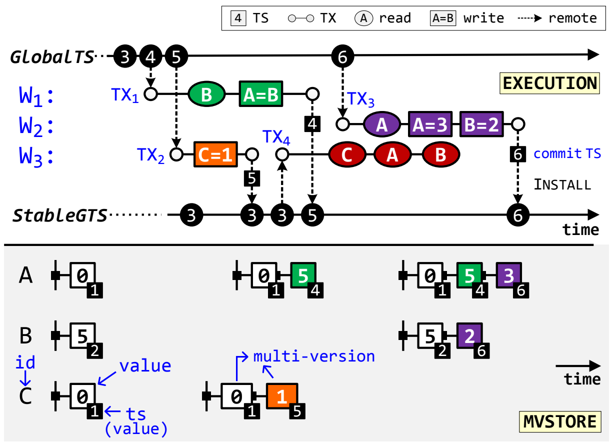

Fig. 3: *A sample case of using GTS, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).*

Given the specification of extensions to 2PL with GTS in Fig. 2, we analyze the transaction behavior in the case shown in Fig. 3 to explain the design of GTS. There are four transactions (TX_1–TX_4), which operate on three tuples (A, B, and C). Note that non-conflicting transactions TX_1 (green) and TX_2 (orange) are both forced to acquire GlobalTS according to the specification. This operation is necessary to maintain the global timestamp ordering, yet results in overly-constrained concurrency control and an extra network round trip compared to the vanilla 2PL.

The necessity of the oracle to maintain StableGTS can be revealed with the conflict between the timestamp order and the commit order concerning TX_1 and TX_2. In this case, TX_2 acquires a larger GlobalTS but commits before TX_1. When read-only transaction TX_4 (red) starts, it cannot simply use the latest committed timestamp (GlobalTS=5) for snapshot reads. The snapshot would be inconsistent if the read-only transaction observes TX_2 before TX_1 commits. Thus, transactions must install commit timestamps so that the oracle can determine the read timestamp (StableGTS=3) for TX_4.

Vector timestamp (VTS). To reduce the overhead of acquiring GlobalTS in the critical path of read-write transactions, VTS replaces the global timestamp counter with a vector of local timestamps. The vector contains a slot for each worker, which records the per-worker timestamp. In each worker, a local counter (LocalTS) is used to assign the commit timestamp for transactions, hence reducing one network round trip compared to GTS. However, the oracle is retained in VTS to maintain the StableVTS with similar reasons as GTS. Fig. 4 shows the specification of extensions to 2PL with VTS.

Fig. 5 presents a concrete case of using VTS. Each worker maintains its local counter (W_1:3, W_2:2, and W_3:5). The version of a tuple is represented as $\langle i : ts \rangle$, where i is the worker ID, and ts is the commit timestamp of the transaction that writes the tuple. The initial StableVTS is $(3, 2, 5)$, which means that tuples with versions less than $\langle 1 : 3 \rangle$, $\langle 2 : 2 \rangle$, and

```
2PL with vector timestamp (VTS)

At Oracle:                              ▸ timestamp server
+ StableVTS                   ▸ snapshot global timestamp
+ Queues                       ▸ pending global timestamp

  INSTALL(⟨i:ts⟩, deps)
+1  add ⟨⟨i:ts⟩, deps⟩ to a Queues[i]

  STABILIZE( )                     ▸ run asynchronously
+1  for each queue in Queues do
+2    for each ⟨⟨i:ts⟩, deps⟩ in queue do
+3      if deps is ready then      ▸ stability protocol
+4        dequeue ⟨⟨i:ts⟩, deps⟩
+5        ⟨i:ts⟩ → StableVTS

At Worker_i:                    ▸ i denotes the worker number
+ LocalTS                        ▸ monotonic local timestamp

  WRITE(tx, id, data)
:1  acquire lock and get latest ⟨i:ts⟩
 2  add ⟨id, data⟩ to tx.wset
+3  add ⟨i:ts⟩ to tx.deps

  READ(tx, id)
:1  acquire lock and get latest ⟨data, ⟨i:ts⟩⟩
 2  add ⟨id, data⟩ to tx.rset
+3  add ⟨i:ts⟩ to tx.deps
 4  return ⟨data⟩

  COMMIT(tx)
+1  tx.TS ← LocalTS
 2  for each w in tx.wset do
:3    update ⟨w.data, ⟨i:tx.TS⟩⟩ and release lock
 4  for each r in tx.rset do
 5    release lock
+6  Oracle.INSTALL(⟨i:tx.TS⟩, tx.deps)  ▸ NT round trip

  ROTX(tx)                              ▸ snapshot read
+1  tx.TS ← Oracle.StableVTS
+2  for each r in tx.rset do
+3    get ⟨r.data⟩ up to tx.TS
```

Fig. 4: *Using VTS (i.e., blue code lines) to enable consistent snapshots for read-only transactions with 2PL. +*N *and* :N *denote new and modified lines of code respectively.*

⟨3 : 5⟩ can be consistently read by read-only transactions.

Maintaining the stable timestamp (StableVTS) becomes more complex in VTS because the per-worker timestamps are not directly comparable [70, 8]. To convey the ordering of transactions to the oracle for deciding StableVTS, workers collect observed timestamps of accessed tuples from other workers (e.g., ⟨1 : 2⟩ of C for TX_2). Note that when read-write transactions (TX_1, TX_2, and TX_3) commit, they must send all observed timestamps (*deps*) to the oracle (INSTALL in Fig. 4). Moreover, read-only transactions (TX_4) must request the whole vector timestamp (StableVTS) from the oracle to start a snapshot read.

2.3 Analysis of Network Overhead

We present an in-depth analysis of centralized timestamp schemes[3] and attribute performance overhead and scalability bottleneck to three main aspects:

[3]For brevity, we avoid prior sophisticated optimizations (incl. batching requests [46, 27], timestamp compression and dedicated fetch thread [70]) for timestamps in here, but enable all of them in the evaluation (§5).

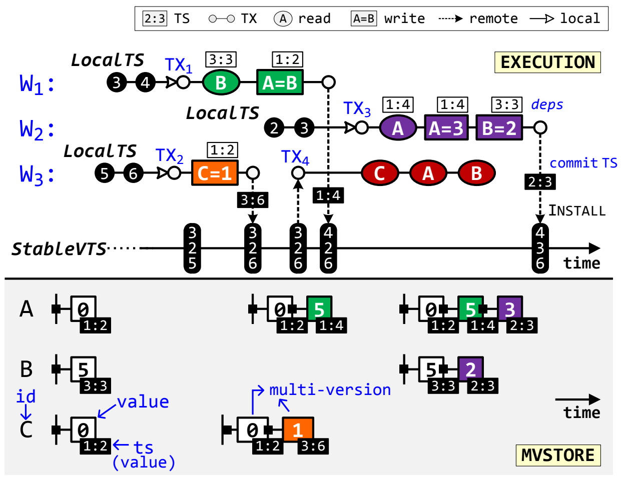

Fig. 5: *A sample case of using VTS, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).*

Non-scalable timestamp oracle. Prior work [46, 61, 11, 59, 67, 37] has shown that a centralized timestamp oracle will become the scalability bottleneck of MVCC systems. The throughput of schemes using a shared counter with atomic operations (GTS) [31, 21, 25] is limited to less than 10 M ops/s (`GlobalCNT` in Fig. 6(a)). The throughput will further decrease due to maintaining the stable timestamp for read-only transactions (`+StableTS`). VTS mitigates the scalability issue by using a local counter for read-write transactions. Besides, prior work [70] avoids the mechanism for the stable timestamp (reaching close to 40 M ops/s) at the expense of increasing transaction aborts. However, the network will first become the bottleneck for both GTS and VTS (`Network`). Consequently, the throughput of timestamp oracle (`TSOracle`) can only reach 1.26 M and 2.39 M ops/s for GTS and VTS respectively, which may be enough for `TPC-E` (281 K txns/s) but far not enough for `TPC-C` (1.64 M txns/s) and `SmallBank` (80 M txns/s) even only scaling out to 16 machines.

Using fast networks can boost the throughput of timestamp oracle, while the performance of transactional systems will also increase much [23, 64, 26, 70], and CPU may first become the bottleneck [59]. Moreover, batching requests [46, 27] or dedicated fetch thread [70] can alleviate the timestamp-related load on the network[4], while these techniques also amplify the staleness of the data retrieved by read-only transactions, and increase the abort rate and the end-to-end latency of read-write transactions (see §5.1).

Costly timestamp allocation. A centralized timestamp scheme will inevitably cause extra network communication overhead for each read-write transaction. GTS demands two network round trips, one for obtaining the commit timestamp and one for installing it. VTS uses per-worker local counters to assign the commit timestamp, but still demands one network round trip to install the timestamp. Given that most transactions operate on tuples in local partitions [56, 54, 55], especially for read-write transactions, additional network round trips will notably lengthen the critical section of trans-

[4]We enabled these optimizations for GTS and VTS in our evaluation (§5).

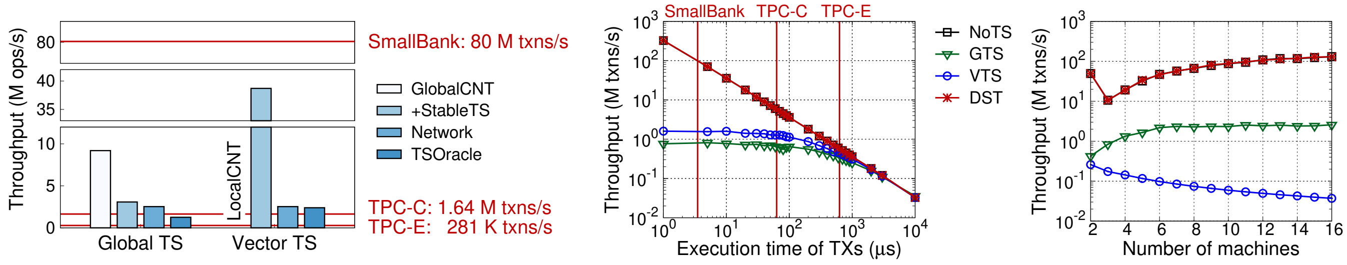

Fig. 6: *(a) Analysis of peak performance and bottleneck of timestamp oracle for GTS and VTS using a 24core machine with 10GbE. (b) The performance of `read-write` transactions with the increase of execution time for different timestamp schemes. The weighted average median latency of read-write transactions in `TPC-E`, `TPC-C` and `SmallBank` are labeled by red lines. (c) The performance of `read-only` transactions with the increase of machines for different timestamp schemes. All experiments are conducted on a local 16-node cluster with 10GbE network (§5). One machine is dedicated for timestamp oracle, even NoTS has no need. Each machine spawns 24 server workers.*

actions and further increase the chance of conflicts, causing extra transaction aborts or blocking time. Thus, it is non-trivial to hide the network round trips without sacrificing the latency of transactions (e.g., batching requests [46, 27]).

The overhead of timestamp allocation highly depends on the execution time of transactions. Hence, we implement a microbenchmark only consisting of read-write transactions, which do not access any tuples and just spin in a loop for a given time. As shown in Fig. 6(b), the overhead of VTS is moderate (from 10% to 30%) compared to not using timestamp schemes (NoTS), when the execution time is close to that of read-write transactions in `TPC-E` (from 1,400μs to 470μs). The throughput will significantly drop more than 80% when transactions execute in about 50μs, which is similar to that of read-write transactions in `TPC-C`. Further, GTS can only achieve half of VTS throughput, since it demands one more round trip to obtain the commit timestamp.

Large traffic size. VTS mitigates the timestamp overhead by using per-worker local counters as the commit timestamp for read-write transactions. However, a critical downside is that a whole vector of per-worker timestamps must be obtained as the read timestamp first, and then be transferred to every tuple for performing consistent snapshot reads. In contrast to the scalar timestamp (e.g., GTS), this overhead grows linearly with the increase of workers or machines in the system. For most transactional workloads [54, 55, 56, 57], the size of the vector timestamp can become orders of magnitude larger than the tuple size, even in a moderate-sized cluster. Using the per-machine counter in VTS (i.e., all workers on one machine share one timestamp slot) can reduce traffic size [8]. However, these workers have to share a local counter by using *atomic* operations (e.g., CAS), which will incur additional overhead on read-write transactions [70].

To demonstrate the impact of traffic size, we implement a microbenchmark only consisting of read-only transactions, which read ten 8-byte tuples with 90% of which being local accesses. In Fig. 6(c), the performance collapse of VTS is due to the increase of timestamp vector obtained from the oracle and transferred to remote tuples. Note that GTS is still one order of magnitude slower due to extra one round-trip to fetch the read timestamp (even scalar), compared to NoTS.

3 Decentralized Scalar Timestamp (DST)

Managing globally ordered timestamps in a centralized service inevitably results in the problem of maintaining the consistency between timestamp ordering and transaction ordering. More importantly, without a holistic decentralized design, the timestamp scheme cannot achieve good scalability. This observation can be backed by the aforementioned performance bottlenecks due to acquiring commit/read timestamps and installing committed timestamps. Such operations add significant overhead to the execution of CC protocols.

To fundamentally overcome the above drawbacks of traditional timestamp schemes, we propose DST, a *decentralized scalar timestamp* that facilitates the multi-version concurrency control (MVCC) implementation for broad CC protocols with efficient snapshot read support and minimal overhead. The intuition behind our design is that *the timestamp scheme can piggyback on concurrency control protocols to maintain the timestamp ordering with low cost and no new scalability bottleneck to read-write transactions.*

In this section, we first use two-phase locking (2PL) as an example to explain the basic protocol of DST for read-write and read-only transactions (§3.1 and §3.2). We then prove the serializability of read-only transactions with DST (§3.3) and introduce a hybrid scalar timestamp to provide snapshot reads with bounded staleness (§3.4). Finally, we discuss the impact of DST on the fault-tolerance scheme (§3.5).

3.1 Timestamps in Read-write Transaction

DST is a fully decentralized timestamp without a centralized sequencer (timestamp oracle) to provide total order timestamps for read-write transactions and stable timestamps for read-only transactions. Therefore, DST must ensure that the derived timestamps for read-write transactions always match the transaction ordering.

The CC protocol is used to ensure the serializable transaction ordering and provide the following three properties, where Transaction A (TX_A) commits before Transaction B (TX_B), and both of them access a conflicting tuple O.

PROPERTY 1: Write-Write. TX_B's write ($W_B(O)$) should overwrite TX_A's write ($W_A(O)$) or generate a newer version.

```
Read-write Transaction: 2PL with DST
At Worker_i:                     ▸ i denotes the worker number
+  LocalTS                       ▸ monotonic local timestamp

   START(x)
+1   x.TS ← LocalTS

   WRITE(x, id, data)
:1   acquire lock and get ts
 2   add ⟨id, data⟩ to x.wset
+3   x.TS ← max(x.TS, ts+1)

   READ(x, id)
:1   acquire lock and get latest ⟨data, ts⟩
 2   add ⟨id, data⟩ to x.rset
+3   x.TS ← max(x.TS, ts+1)
 4   return data

   COMMIT(x)
 1   for each w in x.wset do
:2     update ⟨w.data, x.TS⟩ and release lock
 3   for each r in x.rset do
:4     update ⟨x.TS⟩ and release lock
+5   LocalTS ← max(LocalTS, x.TS)
```

Fig. 7: *Specification of* `read-write` *transaction for 2PL with DST. +N and* `:N` *denote new and modified lines of code respectively.*

PROPERTY 2: Write-Read. TX_B's read ($R_B(O)$) should retrieve TX_A's write ($W_A(O)$).

PROPERTY 3: Read-Write. TX_A's read ($R_A(O)$) should not retrieve TX_B's write ($W_B(O)$).

To match the transaction ordering, DST should ensure TX_B's commit timestamp (TS_B) is larger than TX_A's commit timestamp (TS_A) under the above case. The general idea is to piggyback over the CC protocol to derive a commit timestamp from conflicting tuples. Fig. 7 presents how DST is integrated with two-phase locking (2PL), and Fig. 8 also illustrates the execution of sample transactions with DST. DST leverages conflicting tuples and above three properties to transmit commit timestamps between dependent transactions. The additional codes for DST in WRITE, READ, and COMMIT (see Fig. 7) are commented on corresponding operations in the following explanations.

Write-Write property. Transaction TX_A installs value (V_A) with commit timestamp (TS_A) into the tuple O.

$$\langle V_A, TS_A\rangle \rightarrow O$$
$$TS_A \rightarrow O.ts \qquad \triangleright\ \textit{line:2}\ \text{of COMMIT}$$

Transaction TX_B reads the timestamp of tuple O (O.ts) and installs new value (V_B) with a *larger* commit timestamp (TS_B) into the tuple O.

$$O.ts \rightarrow ts \qquad \triangleright\ \textit{line:1}\ \text{of WRITE}$$
$$max(ts{+}1, TS_B) \rightarrow TS_B \qquad \triangleright\ \textit{line:3}\ \text{of WRITE}$$
$$\langle V_B, TS_B\rangle \rightarrow O$$
$$TS_B \rightarrow O.ts \qquad \triangleright\ \textit{line:2}\ \text{of COMMIT}$$

In Fig. 8, TX_1 (green) commits before TX_3 (purple), and both of them write tuple A. Therefore, the commit timestamp of TX_3 should be larger than that of TX_1. Using DST, TX_1 installs its value (5) with its commit timestamp (TS_1=4) into tuple A. After that, TX_3 should derive a larger timestamp (TS_3=5) from the timestamp of tuple A (A.ts=4) and use it to install new value (3) into tuple A. Note that the write operations will update both the tuple's timestamp and the value's timestamp (as a tuple may have multiple values with different versions).

Write-Read property. Transaction TX_A installs value V_A with its commit timestamp TS_A into tuple O.

$$\langle V_A, TS_A\rangle \rightarrow O$$
$$TS_A \rightarrow O.ts \qquad \triangleright\ \textit{line:2}\ \text{of COMMIT}$$

Transaction TX_B reads value V_A of tuple O with timestamp O.ts and installs a larger commit timestamp TS_B.

$$O \rightarrow \langle V_A, ts\rangle \qquad \triangleright\ \textit{line:1}\ \text{of READ}$$
$$max(ts{+}1, TS_B) \rightarrow TS_B \qquad \triangleright\ \textit{line:3}\ \text{of READ}$$
$$TS_B \rightarrow O.ts \qquad \triangleright\ \textit{line:4}\ \text{of COMMIT}$$

In Fig. 8, TX_1 (green) commits before TX_3 (purple), and TX_1 writes tuple A before TX_3 reads it. Therefore, the commit timestamp of TX_3 should be larger than that of TX_1. Using DST, TX_1 installs its value 5 with its commit timestamp (TS_1=4) into tuple A. After that, TX_3 reads the timestamp of tuple A (A.ts=4) and derives a larger timestamp (TS_3=5).

Read-Write property. Transaction TX_A installs commit timestamp TS_A into tuple O since it has read the value of tuple O.

$$TS_A \rightarrow O.ts \qquad \triangleright\ \textit{line:4}\ \text{of COMMIT}$$

TX_B reads timestamp of tuple O (O.ts) and installs new value V_B with a larger timestamp TS_B into tuple O (O.ts).

$$O.ts \rightarrow ts \qquad \triangleright\ \textit{line:1}\ \text{of WRITE}$$
$$max(ts{+}1, TS_B) \rightarrow TS_B \qquad \triangleright\ \textit{line:3}\ \text{of WRITE}$$
$$\langle V_B, TS_B\rangle \rightarrow O$$
$$TS_B \rightarrow O.ts \qquad \triangleright\ \textit{line:2}\ \text{of COMMIT}$$

In Fig. 8, TX_1 (green) commits before TX_3 (purple), and TX_1 reads tuple B before TX_3 writes it. Therefore, the commit timestamp of TX_3 should be larger than that of TX_1. Using DST, TX_1 reads an old value (5) of tuple B and installs its commit timestamp (TS_1=4) into tuple B. After that, TX_3 will derive a larger timestamp (TS_3=5) and use it to install new value (2) into tuple B.

3.2 Timestamps in Read-only Transaction

DST ensures that the order of derived commit timestamps for read-write transactions always matches the transaction ordering. Therefore, read-only transactions can directly pick any timestamp (TS_{RO}) to read a consistent snapshot by comparing its read timestamp with the timestamps of tuples.

Since the (snapshot) read-only transaction does not follow the CC protocol (e.g., lock/unlock tuples before/after reading values), the read-only transaction may read a part of updates of a concurrent read-write transaction. For example, in Fig. 8, the read-only transaction TX_4 (red) and the read-write transaction TX_3 (purple) are concurrently executed. If TX_3 commits between the read operations to tuple A and tuple B in TX_4, and then TX_4 will read an old version of tuple A (5) and a new version of tuple B (2).

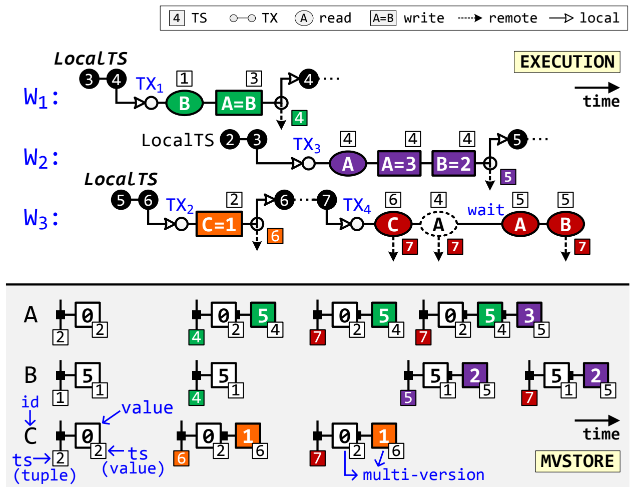

Fig. 8: *A sample case of using DST, where four transactions (TX_1-TX_4) operate on three tuples (A, B, and C).*

To ensure the serializability of read-only transactions, DST asks the read-only transaction to claim its operations actively before reading the tuple. It first installs its read timestamp (TS_{RO}) into the tuple and waits until the conflicting read-write transaction commits (e.g., the tuple is not locked), if the timestamp of the tuple is not larger than the read timestamp (DEP_READ in Fig. 9). Note that the read-only transaction will only wait for at most one conflicting read-write transaction because if the concurrent read-write transaction starts after the claim, it will definitely see the read timestamp through accessing the tuple and derive a larger commit timestamp. Consequently, the read-only transaction will skip all of the updates from this transaction. If the concurrent read-write transaction starts before the claim, it will hold the lock of the tuple. The read-only transaction will wait until the read-write transaction commits. No matter the commit timestamp is larger or smaller than the read timestamp, a read-only transaction can always read a consistent snapshot by ignoring or reading all of the updates from conflicting transactions. Note that CC protocols ensure the atomicity of read-write transaction's updates.

As shown in Fig. 8, the read-only transaction TX_4 will install its read timestamp (TS_4=7) into tuples with smaller tuple timestamps (*line:1* of DEP_READ in Fig. 9). For unlocked tuple C, TX_4 will directly read the value up to the timestamp (1). For locked tuple A and B, TX_4 will wait until the concurrent read-write transaction TX_3 commits. In this example, since TX_3 does not see the read timestamp of TX_4, the commit timestamp of TX_3 is still smaller than the read timestamp of TX_4 (5 vs. 7). Hence, TX_4 can read all updates from TX_3 (A=3 and B=2).

3.3 Proof of Correctness

THEOREM (SERIALIZABILITY). *DST implements serializable read-only transactions, which always read a consistent snapshot generated by serializable read-write transactions.*

PROOF SKETCH. The intuition of the proof is that if a read-only transaction can be serialized with read-write transactions, then it reads a consistent snapshot. We provide a proof sketch by contradiction based on this intuition: i.e., if a read-only transaction cannot be serialized with read-write transactions, then it leads to a contradiction. Before giving the proof, we need to prove following two lemmas first:

```
Read-only Transaction: 2PL with DST
At Worker_i:                       ▸ i denotes the worker number
+  LocalTS                         ▸ monotonic local timestamp

   ROTX(x)                         ▸ snapshot read
+1    x.TS ← LocalTS
+2    for each r in x.rset do
+3       DEP_READ(x, r)

   DEP_READ(x, r)
+1    if r.ts <= x.TS then
+2       r.ts ← x.TS                    ▸ atomic (CAS)
+3       wait until r not Locked        ▸ if conflict
+4    get ⟨r.data⟩ up to x.TS
```

Fig. 9: *Specification of* `read-only` *transaction for 2PL with DST.*

LEMMA 1. *Given two dependent read-write transactions TX_1 and TX_2, if TX_2 depends on TX_1, then TX_2's timestamp (TS_2) is larger than TX_1's timestamp (TS_1).*

PROOF. If TX_2 directly depends on TX_1[5], this lemma follows directly from the algorithm (see §3.1) that TX_2 always calculates TS_2 based TS_1. If TX_2 transitively depends on TX_1, in a proof by contradiction we assume TS_1 is not smaller than TS_2, then in the partial dependent graph denoted by $TX_1 \rightarrow ... \rightarrow TX_i \rightarrow TX_j ... \rightarrow TX_2$[6], there exists TX_i and TX_j that TX_j directly depends on TX_i, but its timestamp is not larger than TX_i's, which is a contradiction with the first case. ☐

LEMMA 2. *Given a read-only transaction TX_{RO} and a read-write transaction TX_{RW}, TX_{RO} observes TX_{RW}'s update on tuple O*[7]*, if and only if TX_{RO}'s timestamp (TS_{RO}) is not smaller than TX_{RW}'s timestamp (TS_{RW}).*

PROOF. First, if TX_{RO} observes TX_{RW}'s update on O, then TS_{RO} is not smaller than TS_{RW}. Because TX_{RW} updates O with TS_{RW} and content atomically (e.g., 2PL), TX_{RO} waits for TX_{RW}'s commit. Second, if TS_{RO} is not smaller than TS_{RW}, then TX_{RO} eventually observes TX_{RW}'s update on O. Assume TX_{RO} does not observe TX_{RW}'s update, then TX_{RO} reads O before TX_{RW} commits its update. One situation is TX_{RO} reads O before TX_{RW}'s request arrives, it leads a contradiction that TX_{RO} update O's timestamp to be TS_{RO} before the read. Another situation is TX_{RO} reads O after TX_{RW} calculates TS_{RW}, but before committing its update. This leads to the contradiction that TX_{RO} always waits for the concurrent TX_{RW} to commit (e.g., 2PL). ☐

PROOF OF THE THEOREM. TX_1 updates A, TX_2 updates B, and TX_2 depends on TX_1. Assume read-only transaction TX_{RO} only observes TX_2's update on B, but does not observe TX_1's

[5]TX_2 is conflicting with TX_1, and TX_2 accesses the conflicting tuples immediately after TX_1.
[6]The symbol $\rightarrow$ indicates the happen-before relation.
[7]It means TX_{RO}'s read on O happens after TX_{RW}'s update.

update on A (i.e., inconsistent reads).[8] From LEMMA 2, we have $\mathsf{TS}_{\mathsf{RO}}$ is not smaller than TS_2, while TS_1 is larger than $\mathsf{TS}_{\mathsf{RO}}$. Therefore, we have TS_1 is larger than TS_2, which is contradictory to LEMMA 1. □

3.4 Hybrid Timestamp and Bounded Staleness

Hybrid timestamp. The commit timestamp of a read-write transaction is derived from the timestamps of tuples in its read/write set, and the read timestamp of a read-only transaction can be any timestamp in the past, at present, or even in the future. Therefore, the local timestamp (LocalTS) is not essential for the correctness of DST. However, the read-only transaction may suffer from either staleness or performance issues if using an improper read timestamp. If the read timestamp is too small (past), the read-only transaction may read an excessively stale snapshot. If the read timestamp is too large (future), the read-only transaction will frequently install its read timestamp into tuples and wait until conflicting read-write transactions commit (DEP_READ in Fig. 9).

DST adopts a combination of *physical* clock and *logic* counter as a hybrid timestamp. The 64-bit timestamp consists of the 48-bit physical part (high-order bits) and the 16-bit logic part (low-order bits). DST uses a loosely synchronized clock as the physical part and uses a monotonically increasing counter as the logical part. At the beginning of the transaction, it will acquire a local hybrid timestamp composed of the current physical clock and zero-initialized logic counter (START in Fig. 7 and *line:1* of ROTX in Fig. 9). The logical part of the hybrid timestamp is used to avoid possible overflow of the physical part since the timestamp will be incremented when calculating the maximum timestamp (e.g., *line:3* of WRITE in Fig. 7). On the other hand, the physical part of the hybrid timestamp is used to ensure the read-only transaction can read a fresh snapshot.

Bounded staleness. Based on the hybrid timestamp, DST can provide snapshot reads with bounded staleness.

THEOREM (BOUNDED STALENESS). *The updates of read-write transactions can be observed in at most Δ, where Δ is the maximal duration any machine needs to make its local clock increased by $2 \times \varepsilon$, and ε is the maximal clock drift between any two machines in the cluster.*

PROOF SKETCH. First, we prove the following two lemmas:

LEMMA 1. *Given a read-write transaction TX_{RW}, its commit timestamp (TS_{RW}) is not larger than $t_m + \varepsilon$, where t_m is the local machine time on TX_{RW} commits.*

PROOF. If $\mathsf{TS}_{\mathsf{RW}}$ is larger than $t_m + \varepsilon$, then there is a TX_{i} which accesses a tuple before $\mathsf{TX}_{\mathsf{RW}}$, and TS_{i} is larger than $t_m + \varepsilon$. As the timestamp is calculated from its local machine time or the tuples it accessed, we can inductively find a transaction TX_{j} whose timestamp is larger than $t_m + \varepsilon$, and it is calculated from its local machine time. It is a contradiction to the maximal clock drift between any two nodes is ε. □

[8]The proof is also correct for TX_1 and TX_2 are the same transaction.

LEMMA 2. *For any read-only transaction TX_{RO} starts after TX_{RW} commits, its read timestamp TS_{RO} is larger than $t_m - \varepsilon$.*

PROOF. This follows that $\mathsf{TX}_{\mathsf{RO}}$ calculates its timestamp based on local machine time and the clock drift between any two nodes cannot be larger than ε. □

PROOF OF THE THEOREM. With LEMMA 1 and 2, we can have a fact that, if $\mathsf{TX}_{\mathsf{RO}}$ starts after $\mathsf{TX}_{\mathsf{RW}}$, then $\mathsf{TS}_{\mathsf{RO}}$ cannot be smaller than $\mathsf{TS}_{\mathsf{RW}} - 2 \times \varepsilon$. Since any machine is able to increase its local machine time by $2 \times \varepsilon$ in Δ, we can conclude that the updates of $\mathsf{TX}_{\mathsf{RW}}$ will be visible in the duration of Δ. □

3.5 Failure and Recovery

The CC protocol should provide a proper fault-tolerance scheme to recover the transactional system from various failures. For example, the primary-backup replication [30] is widely used to provide high availability in prior work [23, 18, 26]. The fault-tolerance schemes can usually work with various timestamp schemes by replicating tuples together with the commit timestamps of read-write transactions. However, the fully decentralized design of DST has two sides. The advantage of this approach is to avoid handling the failure of centralized timestamp oracle, which may cause a stop-the-world recovery [70]. The disadvantage is the potential cost to maintain the consistency of decentralized timestamps before and after some failure occurs.

An obvious, but costly solution is to replicate the read timestamps of read-only transactions together with tuples, as the commit timestamps of read-write transactions. Because the missing read timestamp may cause a new conflicting read-write transaction to use a smaller commit timestamp to write tuples; the read-only transaction may read some tuples with an old version and other tuples with a new version before and after the failure occurs, respectively.

To avoid replicating or persisting read timestamps, DST provides two alternative solutions that can be selected according to the behavior of workloads or the CC protocol associated. More specifically, after recovery, DST can selectively abort and re-execute either the remaining read-only transactions that read tuples on crashed machines or the remaining read-write transactions that write tuples on crashed machines. Consequently, there is no additional overhead and modification associated with the normal execution of transactions, regardless of which approach is selected.

4 Generality of DST

DST is a general timestamp scheme to enable efficient read-only transactions with little impact on read-write transactions. Hence, it is easy to integrate DST with various CC protocols, and DST can also cooperate with many optimizations [37, 43] on CC protocols. In this section, we first lay out a general guideline for piggybacking DST on various CC protocols, and then demonstrate the efficacy of this guideline by applying it to three representative transactional systems

(DrTM+R, MySQL cluster, and ROCOCO) with different CC protocols (OCC, 2PL, and ROCOCO).

4.1 A Guideline for Integrating DST

Read-write transaction. DST should allocate a commit timestamp for the read-write transaction that is larger than any dependent transactions' timestamp. Thus, two following tasks (`RW1` and `RW2`) should piggyback on CC protocols.

1. *select a commit timestamp larger than both the current local timestamp and the timestamps of tuples in the read/write set.* (`RW1`)
2. *install the commit timestamp to tuples in the read/write set before the transaction commits.* (`RW2`)

Read-only transaction. DST should guarantee the read-only transaction can read the value of tuples up to the read timestamp. Thus, two following tasks (`RO1` and `RO2`) should piggyback on CC protocols.

1. *select an appropriate read timestamp according to the current local timestamp.* (`RO1`)
2. *ensure the tuple has an equal or larger timestamp before reading its value up to the read timestamp.* (`RO2`)

4.2 Case Study

The description below focuses on the general comments about integrating DST; we omit a few details and corner cases due to space limitations.

DrTM+R. Optimistic concurrency control (OCC) is widely adopted by modern transactional systems [21, 10, 59, 62, 22, 23, 18, 26, 63]. The read-only transaction in OCC will take two or more rounds of reads for consistent results without MVCC and timestamp schemes, due to conflicting read-write transactions. We use DrTM+R [18] to demonstrate how DST piggybacks on OCC.[9]

For the read-write transaction, we can obtain the timestamp of tuples in the read and write set when validating and locking them respectively and then derive a larger commit timestamp (`RW1`). Before committing, we should install the commit timestamp to the tuples in the read and write set (`RW2`). Note that there is no need to lock tuples in the read set since dummy timestamps from aborted transactions are benign. For the read-only transaction, all CC protocols share (almost) the same implementation (see Fig. 9). The only difference is how to wait for conflicting transactions (*line:3* of DEP_READ). For OCC, the conflicting read-write transaction will lock the tuple when installing its timestamp for updates. Therefore, similar to 2PL, the read-only transaction will confirm that the tuple is not locked before reading the value up to its read timestamp.

MySQL cluster. Two-phase locking (2PL) is another classic CC protocol used by many transactional systems [1, 19, 36]. The read-only transaction in 2PL will be blocked without MVCC and timestamp schemes, due to conflicting read-write transactions. We use MySQL cluster [1] (v7.6.8) to show the integration of 2PL and DST, mainly following the specification in Fig. 7 and 9.[10] To support the read-write lock in MySQL cluster, the transaction only needs to install timestamp into tuples in the read set atomically (i.e., compare-and-swap) and avoids overwriting a larger timestamp. Further, we leverage the lock queue mechanism in MySQL cluster to wait for conflicting transactions (*line:3* of DEP_READ in Fig. 9), which avoids spinning on the tuple.

Table 1: *Measurement clusters.*

Name	#	Hardware
AWS	32	r4.2xlarge (8x vCPU, 61GB DRAM, up to 10GbE)
VAL	16	2x Intel Xeon E5-2650 v4 (12 cores), 128GB DRAM, 1x Intel I350 10GbE
VLR	16	2x Intel Xeon E5-2650 v4 (12 cores), 128GB DRAM, 2x Mellanox ConnectX-4 100Gbps InfiniBand RNICs

ROCOCO. ROCOCO [43] is a research CC protocol that outperforms traditional protocols under high contention workloads by reordering conflicting read-write transactions instead of aborting them. The read-write transaction is chopped into pieces by an offline checker and uses a two-phase mechanism. The *start* phase explores a dependency graph, and then the *commit* phase executes conflicting transactions with a serializable order according to the dependency graph. The read-only transaction in ROCOCO is blocked until the completion of conflicting transactions and uses multiple rounds for reading consistent results.

To extend ROCOCO[11] with DST, the general idea is to use the dependency graph to collect timestamps of dependent tuples and derive a larger commit timestamp for the read-write transaction in the start phase (`RW1`). Then the commit timestamp can be installed to tuples in the commit phase (`RW2`). For the read-only transaction, the blocking mechanism in ROCOCO is reused to wait for conflicting transactions (*line:3* of DEP_READ in Fig. 9).

5 Evaluation

We have integrated DST with three representative transactional systems, namely DrTM+R, MySQL cluster, and ROCOCO, with different CC protocols, and also implemented two centralized timestamp schemes (GTS and VTS) by following the state-of-the-art [46, 70][12] with many carefully tuned optimizations (e.g., batching requests [46, 27], cooperative multitasking [26], timestamp compression and dedicated fetch thread [70]). These optimizations have significant performance improvements on GTS and VTS. For example, cooperative multitasking improves the peak per-machine throughput of GTS on DrTM+R by 3.04X, and timestamp compression improves VTS by 2.7X on a 16-node cluster.

Testbed and setup. To study the impact of hardware plat-

[9]We use the version of DrTM+R [63] without HTM, which also enables coroutine and supports various networks (e.g., TCP/IP and RDMA).

[10]Although MySQL cluster uses read committed (RC) protocol by default, it also provides serializability by using per-row 2PL.

[11]Source code: `https://github.com/shuaimu/rococo`.

[12]Different than Percolator [46], we use the stabilization process to avoid holding locks when acquiring write timestamp, since it will significantly increase transaction abort rate.

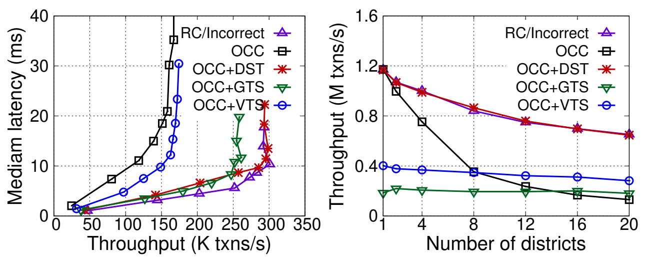

Fig. 10: *Performance of (a)* `TPC-E` *and (b)* `TPC-C` *on AWS.*

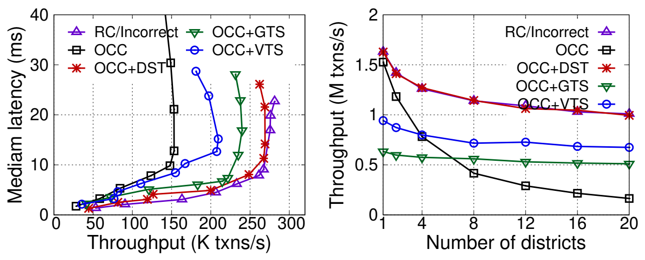

Fig. 11: *Performance of (a)* `TPC-E` *and (b)* `TPC-C` *on VAL.*

forms on DST, we use three clusters with different networks and CPU processing power (Table 1). Without explicit mention, one machine in each cluster is dedicated to the timestamp oracle, even only GTS and VTS need. Other machines serve as both database nodes and clients. We use these machines in a *symmetric* setting [23], namely each machine both executes transactions and store database data.

Benchmarks and performance overview. As the performance benefit of using MVCC and snapshot reads is sensitive to characteristics of read-only transactions in OLTP workloads, we chose three different benchmarks, namely `TPC-E`, `TPC-C`, and `SmallBank`, to show the benefits of DST comprehensively. `TPC-E` [57] presents the workload of a brokerage firm with a high proportion of read-only transactions (79% of the standard mix) and complicated operations (massive range queries and distributed accesses). DST is expected to improve the performance much compared to the vanilla CC protocols for this target workload, with *a relaxed consistency level from strict serializability to serializability*. `TPC-C` [56] simulates a warehouse-centric order processing application with a few read-only transactions (8% of the standard mix). DST is expected to show gradual improvement with the increase of execution time in read-only transactions (not affect proportion). We increase the number of districts (one district by default) accessed by the read-only `stock-level` transactions (4%). `SmallBank` [54] models a simple banking application where transactions perform very simple read and write operations (less than four) on user accounts. DST is expected not to incur perceptible overhead and show order-of-magnitude speedup compared to centralized timestamp schemes (GTS and VTS). In all benchmarks, DST should achieve close to optimal performance using RC (5%) but without compromising correctness, which can be backed by the experimental results of DST on motivating microbenchmarks (see Fig. 6).

5.1 DrTM+R

We deploy one server at each machine and co-locate clients to saturate the performance of servers as prior work [58, 60, 23, 64, 26]. Due to space limitations, we do not report the experimental results on `SmallBank`, which are as expected.

TPC-E. Fig. 10(a) shows the results of `TPC-E` on AWS. `TPC-E` has a high proportion of read-only transactions, and most of them are distributed. Compared to using snapshot reads (GTS, VTS, and DST), the vanilla OCC protocol provides *strict serializability* and requires an additional round to validate tuples in the read set. Thus, many read-only transactions will abort under heavy workloads. As a reference, RC can outperform OCC by 1.79X (yet with incorrect results), since it simply skips the validation phase. DST achieves almost the same performance as RC, as it also avoids the validation phase and never aborts read-only transactions. Differently, DST ensures the read-only transaction can read a consistent yet fresh snapshot. Moreover, compared to GTS and VTS with the same consistency level (*serializability*), DST can outperform the throughput of them by 1.16X and 1.72X, respectively. Because DST omits the communication to the timestamp oracle and avoids large traffic size due to using a fully decentralized design and scalar timestamps (see §2.3).

We further evaluate `TPC-E` on VAL. As shown in Fig. 11(a), DST can still achieve similar performance as RC and provides 1.13X and 1.29X speedup compared to GTS and VTS, respectively. VTS performs slightly better on VAL due to using a relatively smaller vector timestamp.

TPC-C. Fig. 10(b) and Fig. 11(b) show the peak throughput of `TPC-C` on AWS and VAL with the increase of districts accessed by the read-only `stock-level` transactions. Note that the default setting in `TPC-C` accesses one district (the first data point of every line). Besides, we retain all default settings, like the proportion of `stock-level` transactions (4%).

As shown in Fig. 10(b), when accessing one district, DST has a very close performance compared to RC. These results indicate that DST has little overhead to read-write transactions. In comparison to DST, GTS and VTS are 6.29X and 2.93X slower than RC, due to the significant cost for maintaining centralized and/or vectorized timestamps (see §2.3).

OCC performs well on the original TPC-C due to the limited read-only transactions in the standard-mix (8%). On the other hand, when increasing the execution time of read-only `stock-level` transactions (by accessing more districts), the performance difference between RC and OCC is more evident because OCC has more overheads for validating the read-set of the `stock-level`. DST still performs close to RC and is 4.94X faster than vanilla OCC (accessing 20 districts) with a relaxed consistency level. Finally, DST still outperforms VTS and GTS by 2.29X and 3.56X when accessing 20 districts, respectively.

In Fig. 11(b), the performance of DST is also very close to RC for `TPC-C` on VAL. On the other hand, the overhead of GTS and VTS still incurs up to 2.57X (from 1.95X) and 1.73X (from 1.47X) slowdown, compared with DST. Differ-

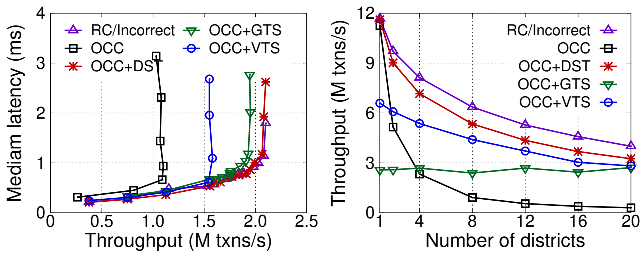

Fig. 12: *Performance of (a)* `TPC-E` *and (b)* `TPC-C` *on VLR.*

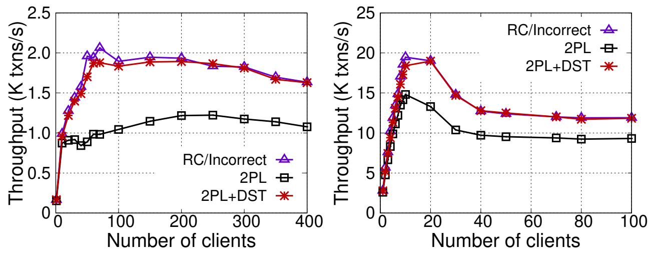

Fig. 13: *Performance of (a)* `TPC-C` *and (b)* `SmallBank` *for MySQL cluster with different CC protocols on VAL.*

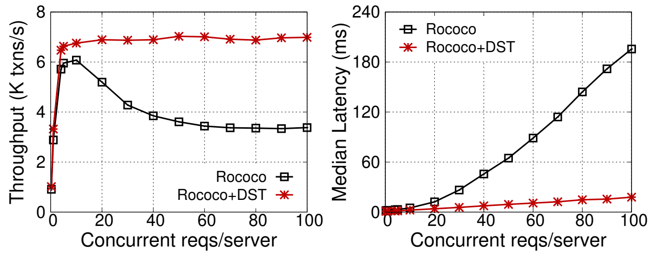

Fig. 14: *(a) Throughput of* `new-order` *transactions and (b) median latency of* `stock-level` *transactions in* `TPC-C` *mixed workload.*

ent than AWS, the lower latency of network round-trip on VAL (90μs) is beneficial for centralized timestamp schemes, but the effect is quite limited.

Using fast network (i.e., RDMA). Readers might be interested in how the performance of networks impacts the performance of timestamp schemes, especially using RDMA. DrTM+R naturally supports RDMA, and we adopt FaSST-RPC [26] to implement the timestamp oracle for GTS and VTS. By using 100Gbps RDMA, the CPU may become the bottleneck in the timestamp oracle for GTS, about 3.0 M ops/s (see §2.3). For VTS, the timestamp oracle will not limit the performance of `TPC-E` and `TPC-C` with only 16 machines, while the increase of transaction abort rate (due to optimizations [70]) and large traffic size still incur non-trivial costs, compared to the decentralized scalar timestamp (like DST).

As shown in Fig. 12, the fast network (RDMA) in VLR has a significant positive impact on all of the settings, as expected. For `TPC-E`, DST still outperforms GTS and VTS by 1.07X, and 1.32X, respectively. RDMA reduces the overhead of centralized timestamp allocation for GTS, while the impact of traffic size in VTS remains. For `TPC-C`, DST is still 4.49X (from 1.19X) and 1.76X (from 1.15X) faster than GTS and VTS.

5.2 MySQL cluster

We evaluate MySQL cluster with DST by using `TPC-C` and `SmallBank` on VAL. We increase the number of clients until the throughput is saturated. As shown in Fig. 13, with DST, MySQL cluster achieves up to 1.91X (from 1.09X) and 1.28X (from 1.07X) higher throughput for `TPC-C` and `SmallBank`, respectively. The main reason is due to enabling snapshot reads to avoid blocking for the read-only transactions. It also mitigates the contention in the read-write transactions. DST is more effective in `TPC-C` since it is more sensitive to blocking time from conflicting transactions due to relatively longer execution time compared to `SmallBank`. On the other hand, DST can provide comparable performance to RC but still guarantee serializability for correctness.

5.3 ROCOCO

We follow the methodology (benchmarks and settings) in prior work [43, 39] to evaluate ROCOCO on VAL.[13] Fig. 14 shows the performance of ROCOCO by increasing the number of concurrent requests per server. In Fig. 14(a), using DST on ROCOCO can improve the throughput of `new-order` transactions by 2.09X with 100 concurrent requests per server, due to reducing transaction aborts and skipping the validation process in read-only transactions. For example, less than 4% of `stock-level` transactions can be committed when there are more than 50 concurrent requests per server. Thus, the server CPU is wasted on retrying and validating read-only transactions. Further, as shown in Fig. 14(b), ROCOCO+DST has a much lower median latency of (read-only) `stock-level` transactions, thanks to reading a consistent snapshot by one round of execution without validation.

5.4 A Study of DST Cost

To study the overhead from blocking and additional timestamp updates in DST, we use two workloads that share most characteristics with `TPC-C`. We tuned the workload behavior to better reflect these overheads.

Blocking overhead. One read-only transaction accesses 10 tuples, while another write-only transaction continuously updates these tuples with locking. This is considered as the worst-case scenario for DST, since the read tuples are locked most of the time. Fig. 15(a) shows the impact on the median latency of read-only transactions when varying the staleness of read timestamps. When using the current time (staleness=0ms) as the read timestamp, 10% of the reads are blocked by concurrent writes, which incur 83% overhead of the median latency (1.72ms vs. 0.96ms). With the increase of staleness (smaller timestamp), fewer reads are blocked since the tuples have been updated with larger commit timestamps. The blocking overhead becomes trivial when staleness exceeds 100ms. Note that this is an extreme case for blocking: reads always touch the locked tuples. In reality, we only observe about 160 and 200 blocks per second at each machine

[13] We try our best to compare with ROCOCO-SNOW [39], which also optimizes the read-only transaction of ROCOCO. Unfortunately, it failed to run on our testbed.

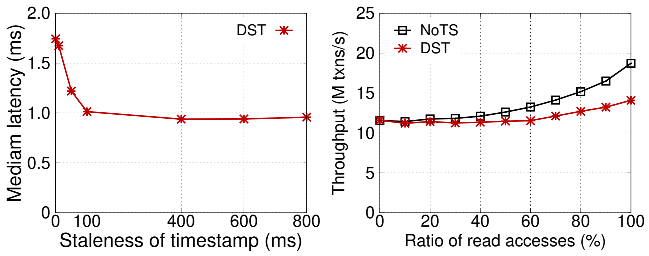

Fig. 15: *(a) The impact of latency for read-only transactions by using stale timestamp. (b) The overhead of DST with the ratio increase of read-only accesses in read-write transactions.*

under peak throughput for TPC-E and TPC-C, respectively.

Timestamp update overhead. Fig. 15(b) presents the overhead of DST to read-write transactions. Each transaction accesses 10 tuples, while some tuples are made read-only. We can see that when all tuples are updated, there is no overhead for DST, since the timestamp update will piggyback on the unlock operation. With the increase of read(-only) ratio, DST adds up to 25% overhead to the overall performance. Because DST will update the timestamp of tuples even just reading them, which requires additional synchronizations using atomic operations. Fortunately, most of the read and write sets are overlapping in OLTP workloads.

6 Discussion

Performance overhead. Compared to traditional centralized timestamp schemes, DST needs to update the timestamps of tuples in the read set for read-write transactions, which may incur additional costs. However, these operations can easily piggyback on original operations in CC protocols (see Fig. 7), like the locking and the validating in 2PL and OCC, respectively. Moreover, the read-only transaction may also update the timestamps of tuples, while it only happens as the read timestamp is larger (DEP_READ in Fig. 9). Thus, using a hybrid timestamp can effectively mitigate it. To study the potential performance overhead for DST, we designed two microbenchmarks to model the worst-case scenarios (see §5.4), and the experimental results show limited cost.

Range scans and phantom reads. DST relies on the CC protocol to detect conflicts, including range scans and phantom reads, and also needs to assign timestamps to certain "guard" (e.g., index structures) [32, 48]. For example, the next-key locking mechanism [42] is widely used by 2PL to support range scans. The CC protocol acquires such locks, and DST assigns timestamps to them. For OCC, DST assigns timestamps to the internal nodes in the index structure as the versions during the validation phase.

The SNOW theorem. The SNOW Theorem [39] describes the fact that strict serializability (S), non-blocking read-only transactions (N), one-response from each tuple (O), and compatible with conflicting write transactions (W) cannot be satisfied at the same time. Yet, SNOW-optimal and latency-optimal read-only transactions can achieve three of the above properties (i.e., N+O+W) without strict serializability (S). DST also relaxes S to serializability for read-only transactions, and satisfies O and W apparently. DST can simply satisfy N by letting reads return a relatively stale data. However, it may be not reasonable; thus, DST chooses to provide bounded staleness with much fewer blocking operations (see §5.4).

Session strict serializability. DST only ensures *serializability* to read-only transactions rather than *strict serializability*, while it is equal to or better than most snapshot-based systems [39, 19, 6, 1]. Further, DST can provide session guarantees [53, 8] (i.e., read-my-write [52] and read-after-write [40] consistency), such that read-only transactions can always observe the latest updates of read-write transactions within the same session (e.g., issued from the same client or handled by the same server). DST returns the commit timestamp to the session manager (e.g., client or server) after the transaction commits. The session manager will always use the largest observed commit timestamp as the read timestamp for successive read-only transactions.

7 Related Work

Using timestamp for snapshot reads. A centralized timestamp is the most straightforward way to support MVCC for snapshot reads, which is widely adopted by centralized systems [21, 25, 31, 28, 44, 66, 33]. Many distributed systems also use timestamps to provide MVCC [46, 17, 51, 19, 6, 14, 70], while most of them only support weaker isolation guarantees (e.g., Snapshot Isolation) [46, 6, 14, 70]. For example, Percolator [46] uses a global timestamp oracle, and NAM-DB [70] uses vectorized centralized timestamps. Spanner [19] is based on a combination of 2PL and MVCC developed in previous decades [45]. Spanner relies on TrueTime API to provide scalable timestamps for strict serializable read-only transactions and snapshot reads, which requires specific hardware (GPS and atomic clocks) to ensure clocks with bounded uncertainty. Further, the read-write transactions still require blocking to ensure the match of timestamp and transaction ordering. DST chooses to support serializable read-only transactions with bounded staleness. It requires no external timestamp service and does not block read-write transactions. RAMP [9] introduces Read Atomic isolation and uses timestamps to identify and retry inconsistent reads. TxCache [47] provides a distributed transactional cache that always returns a consistent snapshot by lazily selecting the timestamps for transactions. Causalspartan [49] also uses Hybrid Logical Clocks to optimize timestamps in causal consistency systems.

DST naturally piggybacks timestamp allocation to existing CC protocols, which avoids additional communications for maintaining timestamps. Further, DST can work with a border range of CC protocols and is orthogonal to prior optimizations on CC protocols [37, 43].

Using timestamp for concurrency control. Many systems directly leverage a timestamp-based mechanism to commit transactions orderly [12, 5, 20, 34, 35, 71, 8]. CLOCC [5] combines optimistic timestamp ordering with loosely synchronized clocks, which avoids a centralized counter for checking serializability in the original OCC protocol [29]. Granola [20] uses the timestamp based on a distributed voting mechanism to order independent transactions deterministically and treats distributed transactions in locking mode. TAPIR [71] uses loosely synchronized clocks at the clients in OCC's validation for read-write transactions. The clock drift in these systems will increase false aborts and impact the execution of read-write transactions. On the contrary, the clock drift in DST only affects the freshness of snapshot reads.

Several variant timestamp schemes have been proposed to *mitigate the cost from frequent aborts due to the violation between timestamp and transaction ordering*. Lomet et al. [38] introduce timestamp ranges to reduce transaction conflicts, while the timestamp management is centralized. MaaT [41] uses dynamic timestamp ranges to avoid distributed locking for the atomic commitment in OCC. Further, some prior systems also use decentralized timestamp schemes, but most of them focus on *optimizing one particular CC protocol*. TicToc [68] introduces a data-driven timestamp scheme for multicore platforms, which allows each read-write transaction to compute a valid commit timestamp from tuples before it commits. However, the read-only transaction still needs additional validations and incurs more aborts due to conflicts. Clock-SI [24] also uses loosely synchronized clocks to create consistent snapshots with fewer network round trips, while snapshot reads must be delayed due to concurrent transactions and clock drift. Sundial [69] uses logical timestamps as leases to reduce aborts in distributed read-write transactions. Pelieus [52] derives a commit timestamp for the read-write transaction from all involved servers (not tuples), which is used in the validation phase with different rules to support different concurrency levels (e.g., SI and Serializability).

Differently, DST is a decentralized timestamp scheme for various CC protocols and can piggyback on them efficiently. Thus, DST will not interfere with the execution of read-write transactions and has no need of extra validations and aborts.

8 Conclusion

This paper presents DST, a decentralized scalar timestamp that can unify timestamp management with existing CC protocols. We have integrated DST with two classic protocols, namely 2PL and OCC, and a recent research proposal, Rococo. Our evaluation with three transactional systems and three benchmarks confirmed the benefit of DST.

9 Acknowledgment

We sincerely thank our shepherd Irene Zhang and the anonymous reviewers for their insightful comments and feedback. We also thank Hong Yang for porting DST to ROCOCO. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003), the National Natural Science Foundation of China (No. 61772335, 61925206, 61902242), the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100, 20ZR1428100), and research grants from Ant Group through Ant Research Program and Alibaba Group through Alibaba Innovative Research (AIR) Program. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] MySQL Cluster. http://www.mysql.com/products/cluster.

[2] MySQL/InnoDB. http://www.mysql.com.

[3] PostgreSQL. http://www.postgresql.org.

[4] Oracle Database Concepts: Data Concurrency and Consistency. https://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm, Janeury 2017.

[5] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 23–34, New York, NY, USA, 1995. ACM.

[6] M. K. Aguilera, J. B. Leners, R. Kotla, and M. Walfish. Yesquel: scalable sql storage for web applications. In *SOSP*. ACM, 2015.

[7] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[8] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414. IEEE, 2016.

[9] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems (TODS)*, 41(3):15, 2016.

[10] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th biennial Conference on Innovative Data Systems Research*, CIDR'11, pages 223–234, 2011.

[11] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[12] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*, pages 285–300. VLDB Endowment, 1980.

[13] P. A. Bernstein and N. Goodman. Multiversion concurrency control theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[14] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[15] C. Binnig, S. Hildenbrand, F. Färber, D. Kossmann, J. Lee, and N. May. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *The VLDB Journal*, 23(6):987–1011, 2014.

[16] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[17] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, Dec. 2009.

[18] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 26. ACM, 2016.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[20] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 223–235, Boston, MA, 2012. USENIX.

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

[22] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 262–275, New York, NY, USA, 2015. ACM.

[23] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[24] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS)*, pages 173–184, 2013.

[25] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, 2015.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201. USENIX Association, 2016.

[27] A. K. M. Kaminsky and D. G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[28] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687. ACM, 2016.

[29] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[30] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC'09, pages 312–313, New York, NY, USA, 2009. ACM.

[31] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[32] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[33] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *IEEE 30th International Conference on Data Engineering*, ICDE'14, pages 580–591. IEEE, 2014.

[34] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. 2015.

[35] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in deuteronomy. *Proceedings of the VLDB Endowment*, 8(13):2146–2157, 2015.

[36] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 104–120, New York, NY, USA, 2017. ACM.

[37] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35. ACM, 2017.

[38] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. In *IEEE 28th International Conference on Data Engineering*, ICDE, pages 714–725, 2012.

[39] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The snow theorem and latency-optimal read-only transactions. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, page 135, 2016.

[40] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[41] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, Jan. 2014.

[42] C. Mohan. Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 392–405, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[43] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 479–494, Berkeley, CA, USA, 2014. USENIX Association.

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

[45] A. Pavlo and M. Aslett. What's really new with newsql? *SIGMOD Rec.*, 45(2):45–55, Sept. 2016.

[46] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, volume 10, pages 1–15, 2010.

[47] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, volume 10, pages 1–15, 2010.

[48] M. Reimer. Solving the phantom problem by predicative optimistic concurrency control. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, pages 81–88, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[49] M. Roohitavaf, M. Demirbas, and S. Kulkarni. Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 184–193. IEEE, 2017.

[50] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 731–742. ACM, 2012.

[51] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[52] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, and M. K. Aguilera. Transactions with consistency choices on geo-replicated cloud storage. 2013.

[53] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems*, PDIS '94, pages 140–150, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[54] The H-Store Team. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[55] The H-Store Team. TATP Benchmark. https://github.com/apavlo/h-store/tree/master/src/benchmarks/edu/brown/benchmark/tm1/.

[56] The Transaction Processing Council. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/.

[57] The Transaction Processing Council. TPC-E Benchmark V1.14. http://www.tpc.org/tpce/.

[58] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 1–12. ACM, 2012.

[59] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[60] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 18–32. ACM, 2013.

[61] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 8. ACM, 2012.

[62] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 26:1–26:15. ACM, 2014.

[63] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 233–251, 2018.

[64] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[65] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, Mar. 2017.

[66] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, 2017.

[67] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[68] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, 2016. ACM.

[69] X. Yu, Y. Xia, A. Pavlo, D. Sanchez, L. Rudolph, and S. Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, June 2018.

[70] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, Feb. 2017.

[71] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 263–278, New York, NY, USA, 2015. ACM.