

# Gossip-Based Visibility Control for High Performance Geo-Distributed Transactions

Hua Fan · Wojciech Golab

Received: date / Accepted: date

**Abstract** Providing ACID transactions under conflicts across globally distributed data is the Everest of transaction processing protocols. Transaction processing in this scenario is particularly costly due to the high latency of cross-continent network links, which inflates concurrency control and data replication overheads. To mitigate the problem, we introduce Ocean Vista – a novel distributed protocol that guarantees *strict serializability*. We observe that concurrency control and replication address different aspects of resolving the visibility of transactions, and we address both concerns using a multi-version protocol that tracks visibility using version watermarks and arrives at correct visibility decisions using efficient gossip. Gossiping the watermarks enables asynchronous transaction processing and acknowledging transaction visibility in batches in the *concurrency control* and *replication* protocols, which improves efficiency under high cross-datacenter network delays. In particular, Ocean Vista can process *conflicting* transactions in parallel, and supports efficient *write-quorum / read-one* access using one round trip in the common case. We demonstrate experimentally in a multi-data-center cloud environment that our design outperforms a leading distributed transaction processing engine (TAPIR) more than 10-fold in terms of peak throughput, albeit at the cost of additional latency for gossip and a more restricted transaction model. The latency penalty is generally bounded by one wide area network (WAN) round trip time (RTT), and in the best case

(i.e., under light load) our system nearly breaks even with TAPIR by committing transactions in around one WAN RTT.

**Keywords** Distributed transactions · Concurrency control · Replication · Geo-distributed data

## 1 Introduction

Cloud providers make it easier to deploy applications and data across geographically distributed datacenters for fault-tolerance, elastic scalability, service localization, and cost efficiency. With such infrastructures, the medium and small enterprises can also build globally distributed storage systems to serve customers around the world. Distributed transactions in geographically distributed database systems, while being convenient to applications thanks to ACID semantics, are notorious for their high overhead, especially for *high contention* workloads and *globally distributed* data.

The overhead of geo-distributed transactions arises not only from the transaction commitment and concurrency control protocols that coordinate different shards for atomicity and isolation, but also from the replication protocol (e.g., Paxos [21]) that coordinates the states of different replicas within a shard. Increased network latency in geo-distributed transactions leads to much higher contention than in local processing. For example, a distributed database benchmark workload (e.g., *fixed* keyspace and *fixed* target throughput) that creates low contention for a database distributed over many servers in a single region may exhibit high contention when the data set is distributed globally.

Conventional serializable concurrency control mechanisms (e.g., OCC, 2PL) processes distributed transactions *synchronously*. They resolve transaction ordering

---

Hua Fan(✉)  
Alibaba Group, Hangzhou, China  
E-mail: guanming.fh@alibaba-inc.com

Wojciech Golab  
University of Waterloo, Waterloo, Canada  
E-mail: wgolab@uwaterloo.ca

in the course of executing the transaction logic, and so they update the data objects only within the period of time when the transaction can exclusively access the objects for the latest version. Such exclusive access is assured using pessimistic locking or optimistic backward validation, which commonly entails at least one wide area network (WAN) round trip time (RTT) for any read-write transaction. We refer to this period of exclusive access as the *serialization window* of a transaction. Long serialization windows hinder parallel execution of conflicting transactions, which makes the design of high performance concurrency control protocols challenging.

The design of replication protocols for systems connected by a WAN is particularly difficult, because the network joining geographically diverse datacenters is both slow (delays in the hundreds of milliseconds) and unpredictable. As a result, the Write-All (**WA**) approach [4] suffers from stragglers delaying writes. On the other hand, the Write-Quorum (**WQ**) approach commonly reads from a dedicated leader or from a quorum, which leads to other problems. The leader potentially becomes the performance bottleneck, and reading quorum (**RQ**) requires much more work than reading one (**RO**) in common read-dominated workloads.

Some previous works on distributed transactions, such as TAPIR [38] and MDCC [20], combine concurrency control and replication into a single protocol for efficiency. TAPIR allows WQRO for inconsistent reads, but uses “application level validation” against a quorum of replicas for both concurrency control and replication. As a result, the validated read is basically equivalent to RQ, though its overhead is partly hidden in the concurrency control protocol in the special case when there are no conflicts. Furthermore, conflicts divert execution away from the fast path in the TAPIR and MDCC protocols, and the alternative slow path incurs additional rounds of messages.

This paper proposes a novel protocol, called *Ocean Vista (OV)*, for high performance geo-distributed transaction processing with strictly serializable (linearizable) isolation. OV combines concurrency control, transaction commitment, and replication in a single protocol whose function can be viewed as *visibility control*. Our insight is that the core mission of a consistent distributed transaction protocol is maintaining the visibility of transactions with respect to other transactions. Based on multi-versioning (MV), OV tracks visibility using version watermarks and arrives at correct visibility decisions using efficient gossip. The visibility watermark (**Vwatermark**) is a visibility boundary: transactions with versions **below the watermark** are visible, otherwise they are **above the watermark** and invisible. OV generates and gossips the watermarks asyn-

chronously and in a distributed manner *without* any centralized leader node. Specifically, each server has its own view of the watermarks, which are updated via gossip. Visibility control using watermarks enables a novel asynchronous concurrency control (**ACC**) scheme that can process conflicting transactions in parallel. ACC decouples transaction ordering from execution of transaction logic: transactions are totally ordered by global versions generated using synchronized clocks, and OV executes the transactions asynchronously according to this serializable order.

The *replica watermark* (**Rwatermark**), below which all versions of transactions have been fully replicated on all corresponding replicas, also enables efficient quorum replication. In OV, a write uses WQ (then asynchronously WA) to automatically avoid stragglers and failed nodes. Writes can succeed in one round trip in the fast path *regardless of conflicts*, and require two round trips in the slow path when too many failed nodes are present. OV can provide consistent reads using RO (i.e., read from any replica) in the common case for a transaction below the *Rwatermark*. Reading versions that have been made visible but are not fully replicated is the only case that requires RQ.

The OV protocol assumes a common case of distributed transactions, which are processed by stored procedures and have a known write-set of keys. This transaction model is used in several previous works [18, 27, 33]. Additionally, OV supports transactions without pre-declared write-sets using a known technique, the reconnaissance query (see Section 3.5). **Because OV uses this limited transaction model, OV does not target interactive transactions, i.e., ones that require user input in the middle of their execution.** We implement the OV protocol in a transactional storage system called OV-DB, and evaluate it for globally distributed transactions in AWS EC2. We compare OV-DB to the geo-distributed transaction protocol TAPIR [38]. The experimental results show that OV-DB outperforms TAPIR in terms of throughput for medium to high concurrency/contention cases, while incurring higher latency for the gossip of watermarks. In particular, OV-DB achieves one order of magnitude higher peak throughput even under a low pairwise conflict rate, and the gossip protocol generally costs one additional WAN RTT of latency. The experimental results also show that OV can achieve a best-case latency of one WAN RTT, which is equivalent to TAPIR.

This manuscript is an invited, extended version of the original Ocean Vista paper [14]. It features a number of additions and enhancements: we have added a conceptual overview of visibility control using watermarks (Section 2.1), more complete protocol pseudocode

for transaction version generation and management (Section 3), as well as a more detailed proof of correctness of OV (Section 3.6) and analysis of the protocol's resilience to clock skew. We also present a complete protocol for removing a coordinator in the view change (Section 5.2), and the full protocol for adding coordinators and gossipers to the cluster (Section 5.3). Next, we describe three extensions of the OV protocol: (a) a gossip-based garbage collection mechanism for obsolete versions (Section 6.1); (b) a protocol to generate consistent snapshots from OV, and a description of snapshot recovery (Section 6.2); (c) a full description of fine-grained replication watermarks for efficient reads (Section 6.3). Finally, we have added an evaluation of the latency percentiles for both low and high pairwise conflict rate scenarios (Section 7.2.5).

The remainder of the paper is organized as follows. Section 2 presents an overview of visibility control using version watermarks, and the architecture of OV. Section 3 details the asynchronous concurrency control mechanism of OV. The replication protocol is presented in Section 4. In Section 5, we discuss the fault tolerance mechanisms. Section 6 discusses three extensions of OV. Evaluation is presented in Section 7. Related work is covered in Section 8, and Section 9 concludes the paper.

## 2 Overview

Visibility control using version watermarks facilitates concurrency control, transaction commitment and replication by increasing parallelism and batching acknowledgments. This section first presents the mechanism of using *Vwatermark* in transaction processing. Then, we describe the properties provided by OV protocol that enable correct visibility control. Lastly, we present the architecture of OV-DB.

### 2.1 Inspiration: Visibility Watermark

The design of OV is inspired by two scenarios in which transactions can be processed without any concurrency control, despite the presence of conflicts:

- Blind write. When no reads are present, all transactions are write-only and each one writes a distinct version to the MV storage.
- Snapshot read. When no writes are present, all transactions are read-only and access a historical snapshot version.

Informally speaking, OV tries to unleash the inherent parallelism of the above scenarios: all the versions above

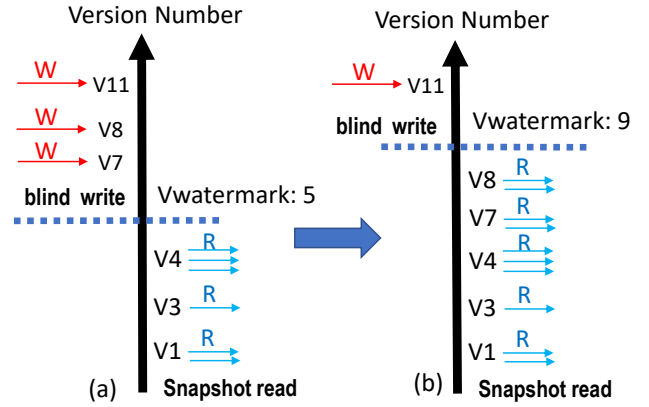


**Fig. 1** An example of *Vwatermark* growing from 5 to 9: only transactions with versions below *Vwatermark* are visible to other transactions.

*Vwatermark* can execute blind writes because they are invisible (i.e., cannot be read); on the other hand, all the versions below the watermark are visible and permit snapshot reads (i.e., no insertions are permitted below *Vwatermark*).

We first examine the case of read-only transactions and write-only transactions, then address the case of read-write transactions. Figure 1 (a) shows an example of the state of all transactions running in OV, when *Vwatermark* is 5. Each transaction is assigned a unique version number prior to its execution. Above *Vwatermark*, versions 7, 8 and 11 can be written in parallel regardless of conflicts on any shards or replicas, because they write on distinct versions against the MV storage. There are no more writes below version 5 (*Vwatermark*), thus versions 1, 3 and 4 can be read concurrently. In Figure 1 (b), *Vwatermark* increases to 9, which means that versions 7 and 8 have been written completely and are ready for reading.

For a general read-write transaction, OV transforms and separates the transaction logic into three operations: write-only operations to record placeholders (i.e., functors [13]) in the MV storage, read-only operations, and asynchronous writes of a specific version. Thus, we can define the *Vwatermark* as a special version number, below which all transactions must have completed their write-only operations (Store-phase, detailed later). We will explain transaction processing in more detail in Section 3.

The visibility control and MV in OV allow most of the above operations to run in parallel even if they come from conflicting transactions. In addition, the gossip of watermarks enables batch acknowledgment of transaction visibility, leading to efficient atomic multi-key writes for the write-only operations, because all versions below *Vwatermark* become visible for all keys.

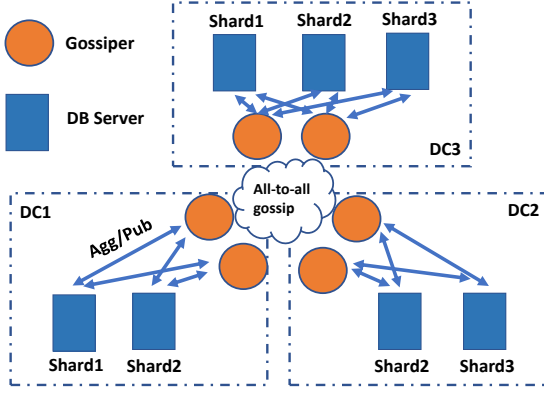


**Fig. 2** Architecture: each DC has at least one gossip, which aggregates and publishes watermarks within its datacenter and exchanges the watermarks with other gossipers.

## 2.2 Principles

The OV protocol enforces the following principles:

**P1:** Transactions take effect in order of their versions.

**P2:** Each coordinator assigns a *monotonically increasing* version number  $ts$  to each transaction, and tracks the lowest assigned version number that should not be visible (a.k.a. *Svw*, see Section 3.4).

**P3:** The *Vwatermark* is no larger than the minimum *Svw* of all DB servers.

Given the above principles, all transactions with versions below *Vwatermark* can be made visible safely, and the transaction order is fixed because no transaction with a lower version may be created. Thus, OV is able to efficiently control the visibility of many transactions using the *Vwatermark*.

## 2.3 Architecture

Figure 2 shows the architecture of OV-DB, in which each datacenter (DC) includes at least one *gossiper*. Multiple gossipers within the same DC are used only for fault tolerance, as they work independently and redundantly.

**Gossipers.** A gossip aggregates visibility information within the DC in the form of watermarks, and exchanges the watermarks with other gossipers. With a view of the global visibility status, a gossip generates and publishes the watermarks within its DC.

**DB Servers.** OV-DB partitions the database into shards. For each shard, there is a set of replicas that reside on different DB servers. For simplicity of presentation, in this paper, each DB server only holds one shard replica. DB servers are in charge of three functionalities.

(1) *Transaction coordinator.* A server receives a transaction request from clients, and assigns it a **globally**

**unique** version  $ts$  based on its local clock (e.g., combination of timestamp, server ID, and monotonically increasing counter). The server replicates the transaction with  $ts$  to the participant servers, and acknowledges to the client when the first server replies with the transaction decision and return values.

(2) *Multi-version (MV) storage.* Each server maintains the data of a shard in multi-version storage. Each shard is replicated and accessed using consistent hashing [19].

(3) *Transaction execution.* Transaction logic is executed via stored procedures in DB servers, when the transaction version number  $ts$  is below the *Vwatermark*.

We assume the clocks of servers are synchronized, for example using the Network Time Protocol (NTP). Skewed clocks may only affect the performance of OV, and never compromise the consistency guarantee, in contrast to how the TrueTime service is used in Spanner [7]. The network may lose, reorder or delay any messages, but a message must be delivered eventually to a non-failed machine if the sender re-tries repeatedly. The OV protocol does not require full replication of the entire database within any datacenter, but this paper assumes full replication for simplicity of analysis.

## 3 Concurrency Control and Transaction Commitment

Transaction visibility control in OV enables processing transactions asynchronously and in batches. ACC can process conflicting transactions in parallel, thus enabling greater parallelism than in conventional systems, which resolve conflicting transactions by aborting or blocking. The transaction commitment is integrated into the visibility control of the OV protocol, and so two-phase commitment (2PC) is not needed. OV can batch the visibility information of many transactions into one watermark, enabling higher efficiency than protocols that control transaction visibility individually (e.g., 2PC). On closer inspection of the protocol, we observe that the best latency for a read-write transaction in OV is the same as in TAPIR: one WAN RTT. We also present a proof that OV provides strict serializability.

### 3.1 Asynchronous Concurrency Control

We first describe the asynchronous concurrency control (ACC) technique for read-write transactions with a known write-set; other transaction types are detailed in Section 3.5. OV processes transactions in an *asynchronous* way: it transforms the processing of a read-write transaction to a sequence of write-only opera-

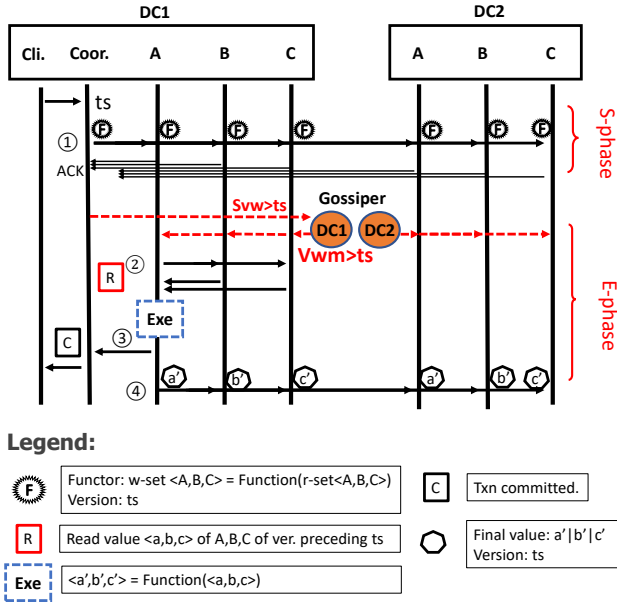


**Fig. 3** A transaction life cycle example using ACC. The transaction logic is executed by the first scheduled functor in DC1 shard A (dashed box). Red dashed lines denote the watermark messages.

#### Algorithm 1: ACC procedures on DB server.

```

Data: vwm: Vwatermark of this server.
1 Procedure Coordinate(transaction T)
2   ts = TidMgr.createTS()
3   parallel-for  $\forall k \in T.writeSet$  do
4     Write(k, ts, T)
5   if any call to Write fails then
6     AbortTxn(ts, T)
7     return
8   TidMgr.stored(ts)
9   asyncRun
10    when all replica succ. write-only then
11      TidMgr.fullyReplicated(ts)
12    wait for execution results from  $\forall T$  of line 4
13    return
14 Procedure Execute(transaction T, version ts)
15   parallel-for  $\forall k \in T.readSet$  do
16     input[k] = Read(k, ts - 1)
17     output = T.execute(input)
18     reply with output to coordinator
19   parallel-for  $\forall k \in T.writeSet, \forall replica$  that has key k do
20     replica.Put(k, ts, output[k])
21 Procedure Publish(Vwatermark vm)
22   vwm = max(vwm, vm)
23   asyncRun
24     parallel-for stored txn  $\forall \langle T, ts \rangle, ts < vwm$  do
25       Execute(T, ts)
26   return TidMgr.Svw()

```

tions, read-only operations, and asynchronous writes. ACC decouples transaction ordering, accomplished by recording the keys in the write-set for a global version,

#### Algorithm 2: Procedures on a gossip of $DC_k$ .

```

Data: vwm: Vwatermark of this gossip.
Sw[Si]: Svw of server Si  $\in \mathcal{S}$  (DC-wide server set).
Dw[Di]: Dvw of DC Di  $\in \mathcal{D}$  (set of all DCs).
1 Procedure Aggregate()
2   parallel-for  $\forall s \in \mathcal{S}$  do
3     Sw[s] = max(Sw[s], s.Publish(vwm))
4   Dw[DCk] = mins  $\in \mathcal{S}$  Sw[s]
5 Procedure XCHG(DC Di, Dvw wm)
6   Dw[Di] = max(Dw[Di], wm)
7   return Dw[DCk]
8 Procedure Gossip()
9   parallel-for  $\forall d \in \mathcal{D}$  do
10    Dw[d] = max(Dw[d], d.XCHG(DCk, Dw[DCk]))
11   vwm = mind  $\in \mathcal{D}$  Dw[d]

```

from executing the transaction logic, which entails read-only operations and asynchronous writes only.

**Write-only operations.** Because the transaction IDs are used as the tuple value versions in storage in ACC, we use the term *transaction versions* for both in this section. ACC records the keys in the write-set for transaction version *ts* using a placeholder for the value of the version in the Store-phase (S-phase). The placeholder is a functor [13], which has all the information required to process the whole transaction (e.g., includes the whole read-set, write-set, parameters). Such functors support general multi-shard transactions (a.k.a. multi-partition transactions) in which the value read from one shard may impact the value written in another shard. Transactions in S-phase are not yet visible and they write different versions, which simplifies concurrency control for atomic multi-key writes. Using MV, each transaction writes a unique version, and is made visible by gossiping *Vwatermark*, which has a similar effect to a batch of second round messages in 2PC.

**Read-only operations and asynchronous writes.** ACC begins executing a transaction only when its version is below *Vwatermark*. This is accomplished by executing any of the recorded functors in the Execution-phase (E-phase), after the transaction order for versions below *ts* is fixed. To execute a functor with version *ts*, read-only operations read the keys in the read-set for the version immediately before *ts*; then compute the transaction decision, return values (to the client) and compute final values for keys in the write-set; lastly, the asynchronous writes replace all functors stored in the S-phase by their final values. If the version read is a functor (i.e., not a final value), the DB servers will recursively compute that functor first, and then resolve the read-only operation using the computed final value.

Figure 3 illustrates the message flow of transaction processing in ACC, and Algorithm 1 presents the procedures. We use the circled numbers (e.g., ①) to link some of the crucial messages presented in both the graph and the pseudo-code. First, the transaction is submitted by a client to any DB server (at the local DC, if available), which will act as the transaction coordinator. The coordinator uses a transaction ID manager (*TidMgr*) to assign the transaction a globally unique version number  $ts$  based on its local clock.

① the coordinator **Writes** (see Section 4.1) the transaction to all participant shards of the transaction (shards A, B and C in this example). The receiving replicas save the entire transaction as the placeholder (functor) for the values of version  $ts$ , which will be replaced by the final values of the write set in the E-phase. The functor includes the whole read-set (A, B, and C), the whole write-set (A, B and C), and a function that generates the values for the write-set based on the reads from the read-set. DB servers that store the  $ts$  use MV (see Section 3.2), and S-phase never enforces ordering among transactions, thus the S-phase *never aborts* a transaction due to conflicts. When  $ts$  is stored on all shards, the coordinator marks the transaction stored in its *TidMgr*, otherwise it aborts the transaction (**AbortTxn** is detailed in Section 4.1). Algorithm 1 lines 9-11 constitute an asynchronous procedure for the replication protocol, which is non-blocking (see Section 4.2).

Each DB server continuously receives the latest *Vwatermark* from gossipers, and all transactions with version lower than the *Vwatermark* are ready for execution. Transactions are ordered by their version number. Execution of a functor begins with ② reading the keys in the read-set for the version immediately before its version number  $ts$ , then executing the transaction based on the values read. The read may contact the closest replica exclusively when  $ts$  is below the *Rwatermark* (see Section 4.3). ③ the execution output is sent to the coordinator, then ④ the values are asynchronously written to all replicas in the write set. In this example, all replicas of the tuple A, B and C replace functors of the version  $ts$  by their final values. OV assumes the transaction logic in the stored procedure is deterministic: functor execution outputs the same result with the same values read. Functors for version  $ts$  can be executed independently and in parallel. The first execution replaces each functor of version  $ts$  in all replicas by its final value to avoid redundant execution of the replaced functor. Determinism ensures that concurrent functor executions of the same transaction yield the same output, and so the coordinator can acknowledge completion to the client as soon as it receives the first response of functor executions.

**Table 1** Conflict matrix for ACC. Table entries indicate whether steps of two transactions can run in parallel when operating on common keys.

|             | Write-Only | Read-Only | Async Write |
|-------------|------------|-----------|-------------|
| Write-Only  | ✓          | ✓         | ✓           |
| Read-Only   | ✓          | ✓         | partially   |
| Async Write | ✓          | partially | ✓           |

**Parallelism.** Whenever a read-write or write-write conflict occurs between concurrent transactions, conventional synchronous concurrency controls (e.g., 2PL, OCC) cannot execute them concurrently. This leads to exclusive “serialization windows” of *at least 1 WAN RTT* for accessing conflicting data. In contrast, ACC makes it possible to execute most operations of conflicting transactions *in parallel*. Table 1 summarizes the behavior for combinations of ACC steps in two concurrent transactions, indicating whether two steps can run in parallel when operating on common keys. ACC steps of writes (i.e., write-only operations and async-write operations) from two transactions can run in parallel, even if they have overlapping write-sets, because they write different versions. Similarly, concurrent write-only and read-only operations must access different versions: one above the *Vwatermark* and the other below.

The only case when two steps of ACC cannot fully run in parallel is when a read-only operation has conflicts (“read-dependency”) *for a specific version* with another transaction’s asynchronous write. However, this case likely has smaller “contention footprint” than a read-write conflict in synchronous concurrency controls, which lasts at least one WAN RTT. This is because (1) the read operation can succeed once *the first replica* has been replaced by the final value via the asynchronous write, e.g., read from *the local DC* replica; and (2) the conflicting version does not hinder execution of other versions. For example, ACC can concurrently resolve multiple such read-dependent conflicts on the same key for *different versions*.

**Latency.** The OV protocol can *also* achieve best-case latency of *one WAN RTT* (ignoring local message RTT within a DC). From Figure 3, we can see that the S-phase can take only one WAN, and if the replicas are available in the DC then all E-phase messages may be DC-local except for the asynchronous writes, which are not included in latency.

Gossip messages among gossipers are across DCs. However, a gossip  $G$  need not wait for a message from remote gossipers if  $G$  has already received a higher watermark than  $ts$ . In the example of Figure 3, let’s assume the clocks are synchronized and the WAN RTT is  $T_{wan}$ . At time  $t$ , DC2 has infrequent coordination, and *Gossiper<sub>dc2</sub>* sends to *Gossiper<sub>dc1</sub>* a *Dvw* (see Sec-

tion 3.4) equal to  $t$ ;  $Gossiper_{dc1}$  receives the gossip message at time  $t + T_{wan}/2$ . Also at time  $t$ ,  $Gossiper_{dc1}$  assigns  $t$  to a transaction  $txn$ , which completes the S-phase at time  $t + T_{wan}$ . Thus at the time  $t + T_{wan}$ ,  $Gossiper_{dc1}$  has already received the  $Dvw$  of DC2, which is greater than  $t$ ; then,  $Gossiper_{dc1}$  can immediately make the decision that version  $t$  is visible without any round-trip message exchange with  $Gossiper_{dc2}$ .

TAPIR also incurs a latency of one WAN RTT in the best case. This occurs when the READs go to the DC-local replica and the “application level validation” successfully exchanges one round of PREPARE messages with all replicas in all shards (see Figure 6 in [38] for details). However, the latency increases further due to retries from conflicts as contention increases. Besides, the latency also does not include the asynchronous COMMIT message that makes the writes visible.

### 3.2 Multi-version Storage

A DB server stores the database data using MV. The APIs are similar to other MV systems: **Put**(*key*, *version*, *value*) stores the *value* to the *version*; **Get**(*key*, *version*) returns the value and version number of the latest version not greater than *version*; **Abort**(*key*, *version*) removes the *version* and promises never to accept the same version again.

To be used in ACC, our MV storage offers two additional features. *First*, the value to **Put** could be either a functor or a final value; the stored final values are immutable, while the stored functor will be replaced by the final value later. The **Get** operation always returns the final value, because when the version accessed is a functor other than a final value, the DB server will resolve the functor first before returning the value. *Second*, ACC **Get** only targets versions below  $V_{watermark}$  and version insertion only targets versions above  $V_{watermark}$ . Our MV storage sorts data by version for fast version retrieval in **Get**, and for garbage collecting obsolete versions that are older than a threshold of time. The MV storage does not depend on a specific data structure, and data structures used in other MVCC systems (e.g., Bw-Tree [24]) can be incorporated into ACC as long as their implementation provides the additional features mentioned in this section.

### 3.3 Transaction Version Management

In contrast to previous works [1, 15, 33] that use a centralized sequencer to assign transaction version numbers, OV uses a decentralized method to avoid a single point of failure and performance bottleneck. The

---

#### Algorithm 3: TidMgr procedures on a DB server.

---

**Data:** *lastTS*: last assigned timestamp by the TidMgr of this server.

*TSset*: versions that have not completed S-phase.  
*rTSset*: versions that have not been stored on all replicas.

```

1 Procedure nextTS()
2    $ts = \text{uniqueTS}()$ 
3   while  $ts \leq \text{lastTS}$  do
4      $ts = \text{uniqueTS}()$ 
5    $\text{lastTS} = ts$ 
6   return  $ts$ 
7 Procedure createTS()
8    $ts = \text{nextTS}()$ 
9    $\text{TSset.insert}(ts)$ 
10   $\text{rTSset.insert}(ts)$ 
11  return  $ts$ 
12 Procedure stored( $ts$ )
13    $\text{TSset.erase}(ts)$ 
14 Procedure fullyReplicated( $ts$ )
15    $\text{rTSset.erase}(ts)$ 
16 Procedure Svw()
17   if  $\text{TSset}$  is not empty then
18     return  $\text{TSset.min}()$ 
19   else
20     return nextTS()
21 Procedure Srw()
22   if  $\text{rTSset}$  is not empty then
23     return  $\text{rTSset.min}()$ 
24   else
25     return nextTS()

```

---

transaction version is assigned by *each server* based on its local clock, which is synchronized using ordinary NTP. Each transaction coordinator has a transaction ID manager (*TidMgr*) for version number assignment and bookkeeping.

Algorithm 3 shows the pseudocode of the transaction ID manager. The procedure **nextTS** creates a unique version number using *uniqueTS()*, which combines a timestamp obtained from the local clock with a globally unique identifier, and the returned version number must be greater than the last generated version number. *TidMgr* records the highest assigned timestamp *lastTS* in a stable storage, and retries whenever a generated timestamp is lower than the *lastTS*. The *lastTS* is a server-local variable; generating and checking this variable requires no cross-server coordination. For instance, the per-server throughput of transaction generations in Section 7 (around 5k txn/s on average) is far below the performance bottleneck of processing server-local variable operations by a single thread. Coordinator executes **createTS** to assign a version number to a transaction. When the transaction completes the S-phase, the coordinator calls the **stored** procedure, and updates the status of the version number in

TidMgr. Similarly, when the transaction has been fully replicated to all corresponding replicas, the coordinator calls the procedure `fullyReplicated` for the transaction's version. Servers use the procedures `Svw` and `Srw` to access the server watermark for visibility and replication, which are detailed in Section 3.4 and Section 4.2.

### 3.4 Gossip of Visibility Watermarks

Naively, to satisfy the principle **P3**, the *Vwatermark* can be obtained by contacting all DB servers for the minimum version in the S-phase. However, Algorithm 2 presents a more efficient and distributed *Vwatermark* generation protocol using server-gossiper DC-wide message exchanges and gossip-gossiper cross-DC message exchanges.

**Server visibility watermark (Svw)** is the minimum version number in the S-phase on a *DB server*. Each DB server maintains its *Svw* by tracking all versions in the S-phase in a set in the TidMgr, referred to as *TSset*. When `TidMgr.createTS()` generates a monotonically increasing version number *ts* (Algo. 1 line 2), TidMgr adds *ts* to the *TSset*. When transaction *ts* completes its S-phase by calling `TidMgr.stored(ts)` (Algo. 1 line 8), TidMgr removes *ts* from the *TSset*. If the *TSset* is not empty, the minimum version within the set is the *Svw*, otherwise the TidMgr calls `nextTS` to generate a *ts* as the *Svw* because any version number generated in the future must be higher.

**DC visibility watermark (Dvw)** is the version number equal to the minimum *Svw* in *this DC*, thus any version below *Dvw* has completed the S-phase in the DC. Each gossip generates the *Dvw* by maintaining a snapshot of the latest *Svw* of all DB servers within the DC. A gossip continuously polls the DB servers for *Svw*, meanwhile, publishing the latest *Vwatermark* to every DB server in the DC. Each DB server maintains a non-decreasing *Vwatermark*, and it always adopts the highest watermark if it receives different watermarks published by different gossipers.

Similarly, each gossip also maintains a snapshot of the latest *Dvw* of each DC by exchanging its *Dvw* with other gossipers periodically. The gossipers always keep the highest *Dvw* for each DC, because the *Svw* and *Dvw* must be non-decreasing. The **Vwatermark** is simply generated as the minimum *Dvw* snapshot across all DCs.

### 3.5 Other Transactions

Read-only transactions and write-only transactions can skip some of the ACC steps required for more general read-write transactions, which boosts their performance. We also explain in this subsection how to execute dependent transactions, which do not have pre-known write-set.

**Read-only transactions.** ACC can achieve strictly serializable read-only transactions in *one* round of messages *regardless of contention*. The overhead is similar to a snapshot read. The protocol for a read-only transaction is similar to that of a read-write transaction, except that the S-phase only stores a functor on the *coordinator DB server*. Hence, the entire S-phase and the functor-to-coordinator acknowledgment are all local operations on the coordinator server. The E-phase only needs to execute the read-only operations for the *ts* assigned to the transaction. In particular, when *ts* < *Rwatermark*, the read-only operations can be applied to the closest replica (see Section 4.3).

**Write-only transactions.** The protocol for write-only transactions only needs the S-phase, which writes the final values directly to the MV storage. The coordinator can acknowledge to the client as long as the *Vwatermark* is greater than the transaction version.

**Dependent transactions.** The dependent transactions [33], which need to read the values of the database to determine the full read-set and write-set, are not supported in the basic protocol. However, a known technique, *reconnaissance query* [33], can support this category of transactions. A reconnaissance query is a read-only transaction that is used to find the tentative read-set and write-set for the dependent transaction. The dependent transaction is executed based on the tentative sets, but the E-phase first verifies whether the tentative read-set and write-set generated by the reconnaissance query are still valid *by* re-reading the values. If the verification fails, the transaction execution output is “abort the transaction,” otherwise the transaction is executed in the same way as in the basic protocol.

### 3.6 Strict Serializability

In this subsection, we prove that OV provides strict serializability for all committed transactions. Specifically, the transactions are *strict one-copy serializable (strict 1SR)*. Informally speaking, a schedule of such transactions is serializable when it is equivalent to *some* serial schedule. A schedule of transactions on multi-version (MV) storage is one-copy serializable (1SR) when the equivalent serial schedule is as if accessing a single version storage [4]. Strict serializability [5, 29] further re-

quires that the serialization order be compatible with the “real-time” precedence order over non-overlapping transactions. For example, when a transaction  $T_a$  finishes before the transaction  $T_b$  starts ( $T_a$  precedes  $T_b$ ), strict serializability requires that  $T_a$  takes effect before  $T_b$ . The latter property is referred to as *linearizability* in the context of concurrent data structures [16].

**Serializability.** The formal proof technique prescribed by [4] is to start with a multi-version schedule  $S$  of committed transactions, which indicates the exact version of a data item created by each write or read by each read operation, and construct a corresponding multi-version serialization graph (MVSG) for  $S$ , where the vertices represent transactions and the edges represent precedence constraints. The schedule  $S$  is one-copy serializable (1SR) if and only if its MVSG is acyclic. To that end, we will show that edges in the MVSG are always directed from a transaction with a lower timestamp to a transaction with a higher timestamp, which not only implies that the MVSG is acyclic but also that transactions appear to take effect in order of their version numbers (P1).

*Claim* Let  $S$  be any multi-version schedule of committed transactions in OV, and let  $G$  be its MVSG. Then  $G$  is acyclic, and if an edge exists from  $T_i$  with timestamp  $ts_i$  to  $T_j$  with timestamp  $ts_j$ , then  $ts_i < ts_j$ .

*Proof* Suppose that an edge  $(T_i, T_j)$  exists in  $G$ . By definition [4], an MVSG graph contains two categories of edges: conflict edges, and version order edges. Suppose first that  $(T_i, T_j)$  is a conflict edge. Since  $S$  is a multi-version schedule, the only possible conflict is a reads-from dependency whereby  $T_i$  writes a version read by  $T_j$ . In this case, OV ensures that  $ts_i < ts_j$  because a transaction can only read versions less than or equal to its own timestamp, and since distinct transactions have distinct timestamps. Thus, a conflict edge always points in the direction of increasing transaction timestamp, as required.

Next, suppose that  $(T_i, T_j)$  is a version edge. By definition [4], such an edge occurs in two cases.

Case 1:  $T_i$  and  $T_j$  both write the same key and  $T_i$ 's version precedes  $T_j$ 's in the version order for that key. In that case  $ts_i < ts_j$  holds, as required, since version order is determined by transaction timestamp order.

Case 2:  $T_i$  reads a version of some key that is written by another transaction  $T_k$  such that  $k \notin \{i, j\}$ , and this version precedes the one written by  $T_j$  to the same key. Let  $ts_k$  be the timestamp of  $T_k$ . If  $ts_i < ts_j$  then the claim holds, otherwise suppose for contradiction that  $ts_j < ts_i$ . Since  $T_k$ 's version precedes  $T_j$ 's, our assumption implies that  $ts_k < ts_j < ts_i$ . However, this contradicts  $T_i$  reading  $T_k$ 's version and not  $T_j$ 's version, since

$ts_j < ts_i$  implies that  $T_j$  is visible to  $T_i$  (see principle P1). To see the latter point, note that a server begins the E-phase of  $T_j$  only once  $ts_j$  is below that server's view of  $Vwatermark$ , which is a lower bound on the visibility watermark  $Svw$  at any server. As a result, all versions less than  $ts_j$  are also visible, including both  $ts_k$  and  $ts_j$ .  $\square$

**Strict ordering.** We also show that the order of the transaction versions is strict with respect to the “real-time” precedence order.

*Claim* Let  $S$  be any schedule of committed transactions in OV, and consider any two distinct transactions  $T_i$  and  $T_j$  with version numbers  $ts_i$  and  $ts_j$ , respectively. If  $T_i$  completes before  $T_j$  starts in  $S$ , then  $ts_i < ts_j$ .

*Proof* First, at the time point  $time_1$  when  $T_i$  completes, the  $Vwatermark$  on the server coordinating  $T_i$  must be greater than  $ts_i$ , because the execution phase of  $T_i$  is started only when  $ts_i$  is below  $Vwatermark$ . At the same time, that  $Vwatermark$  value must be less than or equal to the  $lastTS$  on the server that assigns the version number  $ts_j$ , say its value is  $lastTS_1$ . This is because  $Vwatermark$  at one server is less than or equal to  $Svw$  on any server, and  $Svw$  at any server is less than or equal to  $lastTS$  at the same server. Thus,  $ts_i < lastTS_1$ .

Second, at the time point  $time_2$  when  $T_j$  starts,  $ts_j$  must be greater than the  $lastTS$ , say its value is  $lastTS_2$ . Because  $T_i$  completes before  $T_j$  starts ( $time_1$  is before  $time_2$ ), and each server only creates *monotonically increasing* version numbers,  $lastTS_1 \leq lastTS_2$ . Thus, it follows that  $lastTS_1 < ts_j$ . Combining this with the previous observation that  $ts_i < lastTS_1$ , we obtain  $ts_i < ts_j$ .  $\square$

In summary, OV provides strict one-copy serializability.

### 3.7 On Clock Skew

The coordinator uses the timestamp from synchronized (e.g., by NTP) local clocks to generate *globally* unique transaction versions. Skewed clocks do not impact the correctness of ACC, because our protocol has no requirement regarding the difference between the timestamp and “real-time,” as long as the  $Svw$  and  $Dvw$  are monotonically increasing.

We provide an informal explanation of the reason why clock skew never violates correctness. In a nutshell, the rule of monotonically increasing transaction version generation (based on timestamps) prohibits the creation of an arbitrary historical version (e.g., below  $Vwatermark$ ) due to clock skew. Consider the case when

a transaction  $T_i$  is *started* after another transaction  $T_j$ , and yet  $T_i$ 's timestamp  $ts_i$  is less than  $T_j$ 's timestamp  $ts_j$ . ACC ensures that both  $ts_i$  and  $ts_j$  are greater than the  $Vwatermark$  at time  $t$  when  $ts_i$  is created (by `TidMgr.createTS()` in Algorithm 3 at line 7). This is because transaction versions are created in a monotonically increasing manner following rule P2 in Section 2, and the versions must be greater than  $Svw$  on the server, and  $Vwatermark$  must be no greater than any  $Svw$ . In particular, `TidMgr` may need to retry generating a timestamp to guarantee monotonicity (Algorithm 3 at line 3). Thus, at the time  $t$ , both  $T_i$  and  $T_j$  have started, but have not begun the E-phase yet (i.e., both  $T_i$  and  $T_j$  are not visible yet). That means the executions of  $T_i$  and  $T_j$  overlap in time, and our system ensures that  $T_i$  takes effect before  $T_j$  (P1). In other words,  $T_i$  is only started later than  $T_j$  and the two transactions run concurrently, but  $T_i$  takes effect before  $T_j$ . In our protocol, it is impossible for  $T_i$  to start after  $T_j$  has finished executing.

Skewed clocks may, however, impact the performance of the system. Because the transactions are ordered by their version number, a transaction number assigned by a “fast clock” may cause transaction processing to be delayed unfairly, leading to a latency penalty. This side-effect can be mitigated using the following strategies. *First*, the servers within the same DC use the same local time source connected by a LAN, thus the clock skew is small within a single DC. *Second*, the coordinator will not accept new client requests when it has an abnormal clock. Abnormality can be detected by several methods, such as detecting a large time offset from the time source or from peers (i.e., other DB servers or gossipers) within the DC, or detecting a large time gap between its local clock and the  $Vwatermark$ .

## 4 Replication Protocol

OV combines concurrency control and replication in a single gossip-based visibility control protocol. The idea of combining protocols is borrowed from previous works [20,27,38], though our gossip-based approach with visibility information represented compactly using watermarks enables more efficient replication for geo-distributed data. In our replication scheme, writes succeed in one RTT *regardless of conflicts* as long as a super quorum<sup>1</sup> of replicas is available, or in two RTTs once a majority quorum is available, whichever comes first. As for read-only operations, in ACC these are always tagged with target versions. With the gossip of watermarks, reads can always retrieve data from any

replica when the target version is below the  $Rwatermark$ . This achieves **WQRO** (write-quorum read-one) for common requests. When the target version is between  $Rwatermark$  and  $Vwatermark$ , the reader needs to contact additional replicas to find a quorum with matching versions.

### 4.1 Write-Only Operations

Transactions in the S-phase perform write-only operations exclusively: **Write** and **AbortTxn**. Inspired by FastPaxos [22] and the consensus operation in TAPIR [38], **Write** reaches consensus on whether a specific version is stored. This requires one RTT in the fast path, and two RTTs in the slow path. Unlike FastPaxos and TAPIR, which suffer a performance penalty via the slow path when a conflict is detected, **Write** in OV is *nearly conflict-free* because it merely inserts a globally unique version using MV (recall no reading in S-phase). Our slow path is needed *only for failures and stragglers*. To tolerate  $f$  replica failures using a total of  $2f + 1$  replicas, similarly to FastPaxos [22], the fast path needs a super quorum ( $\lceil \frac{3}{2}f \rceil + 1$ ) of replicas to be available;<sup>2</sup> the slow path needs only a majority quorum ( $f + 1$ ) of replicas to be available. For easy comparison, we describe the protocol using the same terminology as TAPIR:

1. The coordinator sends  $\langle key, ts, value \rangle$  to all replicas within the shard, and each replica stores the record as TENTATIVE. The replica responds upon success.

2. **Fast path.** If a super quorum of replicas replies (within a timeout), the coordinator takes the fast path and **acknowledges** the *success* to the caller, then sends  $\langle FINALIZE, ts \rangle$  asynchronously to all replicas.

3. Otherwise, once the coordinator receives a majority quorum response, it takes the slow path. The coordinator sends  $\langle FINALIZE, ts \rangle$  to all replicas.

4. On receiving FINALIZE, each replica marks the version recorded as FINALIZED, and responds CONFIRM to the coordinator.

5. **Slow path.** The coordinator **acknowledges** the *success* to the caller when it receives  $f + 1$  CONFIRM responses. Otherwise, it returns *fail* if a majority of replicas are not available.

**AbortTxn** succeeds when the decision is replicated in a fault-tolerant manner on each shard. Each shard succeeds as long as a majority quorum of replicas respond because, unlike the TENTATIVE status in **Write**, the abort decision may never be revoked. The protocol for **AbortTxn** is as follows:

<sup>1</sup> Super quorums are defined as in FastPaxos [22].

<sup>2</sup> An alternative expression is using a super quorum size  $2f + 1$  out of  $3f + 1$  replicas.

1. The coordinator sends  $\langle \text{ABORT}, key, ts \rangle$  to all replicas of all participant shards; it acknowledges to the caller once a majority quorum of replicas respond from each shard.

2. Upon receiving the request, each replica calls  $\text{Abort}(key, ts)$ , and replies *success* to the coordinator.

$\text{AbortTxn}$  also needs to call  $\text{TidMgr.stored}(ts)$  when the abort decision is acknowledged by shards, and needs to run the asynchronous procedure in Algorithm 1 line 9-11, which is used to indicate **that** all replicas have **successfully completed Abort**.

#### 4.2 Replica Watermarks: Rwatermark

The *replica watermark* (**Rwatermark**) is a special version number, transactions below which must have replicated their write-only operations on all participant replicas. Read-only operations for versions below *Rwatermark* can access the value directly from any replica (e.g., in the local DC). The gossip protocol for *Rwatermark* can be simply extended from the protocol of *Vwatermark* generation in Section 3.4 by adding extra fields: server replica watermark (*Srw*), DC replica watermark (*Drw*) and *Rwatermark*.

The gossiped information is leveraged as follows. The *Vwatermark* is used for concurrency control and commitment to decide if a transaction version is visible to other transactions, while *Rwatermark* is designed for reading any replica (e.g., the closest replica) in the replication protocol. Specifically, gossipers generate the *Drw* using the minimum of the DC-wide *Srw*, and generate the *Rwatermark* using the minimum of all DCs' *Drw*. On each DB server, the *TidMgr* maintains the *Srw* by tracking all version numbers for which the write-only operations are not yet fully replicated. Furthermore, the DB servers automatically set all versions below *Rwatermark* as FINALIZED. Note that Algorithm 1 line 9 starts an asynchronous procedure that notifies the *TidMgr* regarding the full replication of *ts*. The procedure is executed when all responses of write-only operations (**Write**, **AbortTxn**) are received, which may be during or even after the E-phase of the transaction. We **omit** the detailed protocol for the gossip of *Rwatermarks* as it is similar to that of *Vwatermark*.

#### 4.3 The Read-Only Operation

The read-only operation  $\text{Read}(key, ts)$  ( $ts < Vwatermark$ ) retrieves the latest version no greater than *ts* for *key*. When  $ts < Rwatermark$ , the **Read** can call  $\text{Get}(key, ts)$  directly on **any replica**. Thus, this sub-section

---

#### Algorithm 4: Read-only operation for versions between *Rwatermark* and *Vwatermark*.

---

```

1 Procedure Read(key k, version ts)
2   target = ts
3   while true do
4     parallel-for  $\forall R \in \text{Replicas}$  do
5       R.Get(k, target)
6     if MATCH CONDITION found then
7       return MATCHED VERSION
8     target =  $\max\{\text{versions received at line 5}\} - 1$ 

```

---

focuses on reading versions between *Rwatermark* and *Vwatermark*.

The  $\text{Get}(key, ts)$  operation presented in Section 3.2 not only returns the *version* and *value*, but also the replication tag TENTATIVE or FINALIZED (see Section 4.1). Assuming  $2f + 1$  replicas in a shard, a transaction with version *ts* that is **below Vwatermark** must be in one of the two following cases, otherwise it should not be visible: (1) **Write** succeeded for *ts*, thus there are at least  $f + 1$  FINALIZED replicas **or**  $\lceil \frac{3}{2}f \rceil + 1$  TENTATIVE-or-FINALIZED replicas; or (2) **AbortTxn** succeeded for *ts*, thus there are at most *f* TENTATIVE replicas **and** there is no FINALIZED replica. The TENTATIVE-or-FINALIZED case may happen when the **Write** succeeds in the fast path, and some of the TENTATIVE replicas adopt the FINALIZED tag in the asynchronous confirmation.

Thus, the  $\text{Read}(key, ts)$  operation sends requests to all replicas but waits for the fastest super quorum to respond (with retry on timeout), and **finds** the highest version *ver*  $\leq ts$  for which at least one replica is FINALIZED or at least  $f + 1$  replicas are TENTATIVE. We call this the MATCH CONDITION for **Read**, and the highest version is called the MATCHED VERSION. An aborted version cannot satisfy the MATCH CONDITION, and the highest committed *ver* must satisfy the MATCH CONDITION within a super quorum of replicas. The protocol for  $\text{Read}(key, ts)$  is presented in Algorithm 4.

### 5 Fault Tolerance

Replication protects the data stored in OV from permanent loss. However, failures of the coordinators and gossipers may stall the dissemination of watermarks in the gossip-based visibility control protocol, leading to a loss of availability. This section presents the protocols for recovering from server, gossip, and datacenter-wide failures. **Our solutions do not exclude alternative approaches to designing fault-tolerant protocols for OV, such as incorporating a fault-tolerant group membership protocol [6].** These protocols assume the database state is recoverable, meaning that to tolerate *f* failure

among  $2f + 1$  replicas, at least a super quorum of replicas is available for each shard. Under this assumption, our recovery protocol can leverage the **Read** operation from Section 4.3 to decide if a version should be recovered or discarded.

### 5.1 Membership Representation

To track the shards and server locations, OV uses a coordination service such as Raft [28] or Zookeeper [17]. Each shard maintains its own membership state, and views over this state are distinguished using monotonically increasing numbers. Each server caches the latest view of all shards, and includes the view number of the target shard in any cross-server communication. A server updates its own view when it learns from the communication response that its view is stale. Similarly to TAPIR, when a server is in **VIEW-CHANGING** status, it will not process new requests except for recovery.

Inspired by Corfu [1], the view can have different memberships for different ranges of transaction versions. For example, a shard with membership  $\langle A, B, C \rangle$  serving version  $[0, \infty]$  may replace a failed server  $C$  by a new server  $D$  starting from version 1000. Thus, the new view of membership may be  $\langle A, B \rangle$  serving version  $[0, 1000]$  AND  $\langle A, B, D \rangle$  serving version  $[1001, \infty]$ , while  $D$  is recovering the versions below 1000.

### 5.2 Coordinator Failure

Visibility control relies on a collection of watermarks from coordinators. Thus, it is important to have quick coordinator failure detection and fast recovery. The failure detection of coordinators is integrated with the gossip-coordinator communication, as they already publish and aggregate watermarks frequently. Furthermore, the gossip and the coordinators are always in the same DC, where the network is much more predictable than the WAN. In addition, the coordinators try to avoid slow responses due to overload, using methods such as load shedding (e.g., reject new client requests when the load is high) and resource isolation (e.g., exclusive access to some CPU cores).

Each coordinator has a backup coordinator within the local DC, which will serve as the substitute for completing the in-flight transactions when the coordinator fails. Note that only the execution of the write-only operations (**Write** and **AbortTxn**) may impact the visibility control to the gossipers. Thus, the coordinator will synchronously replicate its write-only operation log (before execution) and the operation decisions (after execution) to the backup coordinator. This ensures that

the backup coordinator has all the in-flight transactions. When the gossipers suspect a failed coordinator and decides to remove the coordinator from the current view, the view change leader will send the new view, which marks both the failed coordinator and the backup coordinator, to the two coordinators and all the gossipers within the DC. The gossipers then use the state of the backup coordinator to generate watermarks. During fail-over, the backup coordinator stops accepting replication requests from the original coordinator, which disables the original coordinator in case it has not really failed. The backup coordinator simply re-executes all the in-flight write-only operations, and any transactions that have already been aborted must fail during re-execution as a majority of replicas have promised not to accept the aborted version (see Section 3.2). In the event that both the coordinator and its backup fail, none of the committed transactions are lost because each such transaction must have been replicated cross-datacenter; any unresolved functors will be replaced by final values the next time they are accessed.

The full protocol of removing a coordinator in the view change is as follows:

1. The new view is created, which marks the coordinator  $F$  as failed, and the backup coordinator  $B$  as its backup. The new view is sent to all the members within the DC.
2. Upon receiving the new view,  $B$  stops accepting replication requests from the original coordinator  $F$  and begins to re-execute all the in-flight write-only operations.  $B$  should also respond to requests for  $Svw$  from gossipers as  $F$  fails, based on the lowest versions of the remaining in-flight transactions.
3. Having received the new view, the gossipers within the DC begin to use the maximum  $Svw$  received from  $B$  and (previously) from  $F$  for  $Dvw$  calculation. However, the gossipers that have not yet received the new view will continue to contact  $F$  for  $Svw$ . In that case, these gossipers may receive a low  $Svw$ , which does not include the progress made by  $B$ 's re-execution of in-flight write-only operations. However, that only hinders generation of higher  $Vwatermark$  for those gossipers, and it never breaks the correctness of OV.
4. Once  $B$  has completed all the in-flight write-only operations, both  $B$  and  $F$  can be removed from the membership safely.

### 5.3 Add Coordinator and Gossiper

OV needs to add coordinators to the cluster membership when failed coordinators are recovered or new

nodes join the cluster. To enforce the monotonicity property of watermarks regardless of clock skew, a newly added coordinator must not create a version number below the  $Dvw$  and  $Vwatermark$ . In order to enforce the monotonicity property for  $Dvw$ , a gossipster will accept a new coordinator only if the  $Svw$  from that coordinator is no less than any  $Dvw$  generated in that DC. Thus, during initialization the coordinator will not accept client requests until the new coordinator is accepted (described later) by all gossipsters of the DC. The full protocol of adding a new coordinator is as follows:

1. The new view, including the new coordinator, is sent to all members within the DC.
2. The coordinator is initialized with status INIT, in which the coordinator's behavior is different from NORMAL status in several aspects: the one with INIT status (1) will reject any client requests; (2)  $lastTS$  is initialized with 0; (3) procedure  $Svw$  in Algorithm 3 line 16 will not refresh  $lastTS$  when the  $TSset$  is empty.
3. Each gossipster contacts the new coordinator with its  $Dvw$ ,  $t$  for instance, and the coordinator replies with  $t$  as the  $Svw$  to the gossipster, which promises to never create a version number lower than  $t$ . In particular, if  $t$  is greater than the  $lastTS$  of that coordinator, the coordinator assigns  $t$  to  $lastTS$ .
4. When the coordinator has collected  $Dvw$  from all gossipsters within the DC, it transitions to NORMAL status and is ready for client requests.

Adding new gossipsters to the cluster should also prevent a breach of the monotonicity property in watermark generation. The procedures in Algorithm 2 work in the situation of a new gossipster, as long as the gossipster has the state of each  $Svw$  within the DC, even if the initial value for watermarks of remote  $Dvw$  or  $Vwatermark$  is zero. This means that in the bootstrapping stage of the gossipster, it will create a valid  $Dvw$  and possible lower  $Vwatermark$  (due to zero remote  $Dvw$ ), which will be ignored automatically because the coordinators always use the greatest  $Vwatermark$  received.

#### 5.4 Replica Node Failure

A replica node failure does not hinder the functionality of a shard as long as a quorum remains available, because the replication protocol already provides fault tolerance. Thus, the replica node failure detection and recovery can tolerate longer communication delays. The protocol for membership reconfiguration that replaces a failed replica node by a new node is described next.

We assume that a leader within a shard is elected using Raft or Zookeeper to execute the reconfiguration

procedure. *First*, the leader sends a VIEW-CHANGING message to all members, and the live members reply with their highest recorded transaction versions. *Second*, the leader chooses the starting version  $newTS$  for the new membership, which is the highest recorded transaction version returned by the live members. The new membership is represented as two ranges: versions  $[0, newTS]$  are served by the live members without changing the quorum size, and versions  $(newTS, \infty)$  are served by the live members plus the new node. The leader increments the view tag of the membership, and sends out the new membership. *Third*, the new node begins to serve versions above  $newTS$  immediately, and recovers the versions below  $newTS$  when  $newTS$  is below  $Vwatermark$ . The missing range of versions is recovered by pulling from the live members in bulk; similarly to Read in Section 4.3, versions that satisfy the MATCH CONDITION should be kept as FINALIZED, otherwise they should be discarded. *Last*, when the new node has caught up to  $newTS$ , the leader updates the membership again to merge the two version ranges ( $\leq newTS$  and  $> newTS$ ) into one.

#### 5.5 Gossiper Failure and Datacenter Failure

The failure of a single gossipster will not impact the progress of watermarks because each DC has multiple gossipsters that work independently and redundantly. However, missing the updates from one DC *entirely* will prevent the watermark from advancing, because the DC-wide watermarks will not be refreshed for the failed DC. This may happen when all gossipsters of a DC are unavailable due to DC-wide disaster. This subsection presents how our gossip protocol can recover from losing all the gossipsters of a single DC.

The protocol assumes that servers cache a view of the membership of each DC. *First*, the new membership, with the failed DC removed, is sent to the failure recovery leader of each shard (see para. 2 of Section 5.4). *Second*, each leader forwards the new membership to the shard replicas, and the replicas reply with the highest recorded transaction version and promise never to accept transactions coordinated from the failed DC. *Third*, the recovery process aggregates the highest recorded version number  $maxRecordedTS$  across all the leaders of the shards. Note that the failed DC only impacts the visibility control for versions between  $Vwatermark$  and  $maxRecordedTS$ , because versions below  $Vwatermark$  have already been confirmed visible and all versions above  $maxRecordedTS$  are not coordinated from the failed DC. *Then*, the recovery process pulls the status of transactions between  $Vwatermark$  and  $maxRecordedTS$  and computes the transaction decisions

following the **Coordinate** procedure in Algorithm 1. *Last*, after the status of the versions below *maxRecordedTS* has been resolved, the recovery process sends the new DC membership and *maxRecordedTS* as the new *Vwatermark* to all remaining gossipers.

## 6 Extensions

This section presents three extensions to OV-DB.

### 6.1 Garbage Collection Watermark

Garbage Collection (GC) removes obsolete versions that are never needed for concurrency control, i.e., the version may never be read by the execution phase of any functor. We assume that the historical versions, except for **the ones** being used for concurrency control and snapshotting, are allowed to be removed. This subsection presents a GC mechanism that can remove such versions safely by gossiping GC watermarks.

For key  $k$ , a historical version  $v_i$  will never be read by a functor computing with version  $v_f$  if (1)  $v_i < v_f$ , and (2) there exists a version  $v_j$  for key  $k$  satisfying  $v_i < v_j < v_f$ . In other words, to read key  $k$  for computing version  $v_f$ , the system prefers version  $v_j$  over  $v_i$ . Recall that computing a functor of version  $v_f$  requires the version that immediately precedes  $v_f$ . If the version  $v_i$  may never be read by *any uncomputed functors*, it can be removed safely. Similarly to generating *Vwatermark*, we use gossip of watermarks to track the lowest version of an uncomputed functor.

Similarly to *Svw*, each DB server maintains a watermark called Server GC Watermark (*Sgw*), below which all functors in that server have been computed and are stored in the form of final values. This can be achieved by bookkeeping all the versions of uncomputed functors: adding a version when an uncomputed functor is stored in this server; removing a version when the functor is replaced by the final value. Each gossipier maintains DC GC Watermark (*Dgw*) and *GCwatermark*, similarly to *Dvw* and *Vwatermark*. *Dgw* and *GCwatermark* denote versions below which there exist no uncomputed functors within the datacenter and cluster, respectively. The protocol to generate *GCwatermark* is similar to that of *Vwatermark*, as detailed in Section 3.4.

Upon receiving the *GCwatermark* from gossipers, each DB server can delete any version below *GCwatermark* for a given key as long as the version is not the latest version preceding *GCwatermark* for that key.

### 6.2 Consistent Snapshot

A snapshot saves the values of all keys for a consistent state of the database, which can be used for data recovery. Using MV storage, OV-DB can dump a consistent snapshot of the data preceding a given version without blocking any running transactions. This can be achieved simply by a read-only transaction for all keys with a specific version. However, the snapshot procedure must guarantee that (1) the snapshot version is below the *Vwatermark* and (2) it should prevent the snapshotted version from being deleted by garbage collection. When recovering the OV-DB instance from the dumped snapshot, the system not only needs to recover the values of the keys, but also restore valid watermarks (e.g., *Svw*, *Srw*, *Dvw*, *Drw*, *Vwatermark* etc.) that ensure the correctness properties in Section 2.2.

The procedure of taking a snapshot is as follows. We assume the system uses the GC mechanism described in Section 6.1.

1. A server (coordinator for the snapshotting) receives a snapshot request for version  $v_s$ . If  $v_s$  is an unspecified version (e.g., *latest*),  $v_s$  is assigned by the current *Vwatermark*. If  $v_s$  is lower than the *GCwatermark*, the snapshot request fails. The server ensures that *Sgw* does not exceed  $v_s$  until the snapshot is complete, to prevent garbage collection of any versions needed by the snapshot.
2. The read-only requests for all keys are sent to all shards, and the coordinator stores all the results (key, value, version tuples) to persistent storage as a snapshot file.
3. The coordinator stores snapshot version  $v_s$  to the snapshot file, which will be used for recovery. Finally, *Sgw* is allowed to continue increasing.

Recovery from a snapshot first loads the key values into the database, then uses the snapshot version  $v_s$  as the initial value for all the watermarks (*Svw*, *Dvw*, *Vwatermark* etc.). This guarantees that any new version in the recovered system must be greater than  $v_s$ .

### 6.3 Fine-Grained Replication Watermark

The *Rwatermark* is used for efficient reading, because below it, all versions have been fully replicated on all replicas. Thus, reads for a version below *Rwatermark* can be served by any replica. However, the growth of *Rwatermark* may be curtailed by stragglers within a single replication group or even a single slow server.

This subsection proposes a fine-grained replication watermark optimization whereby the replication watermark is maintained for each shard or for each server,

below which all versions have been fully replicated on replicas within this shard, or below which the server has stored all versions for its shard. We illustrate the benefit of fine-grained replication watermarks using an example. Consider a single-key write operation that succeeds using a quorum of replicas for version  $v$ , and where an unresponded straggler server  $s$  can hinder the growth of  $Rwatermark$ . However, using a per-shard replication watermark, the straggler  $s$  only affects the replication watermark for that single shard, and so most reads may still complete using read-one. Moreover, using per-server replication watermarks can further improve the likelihood of using read-one. A read operation for version  $v$  of that key, can be served by any single replica within the replication group without sacrificing consistency, as long as the server has replication watermark greater than  $v$ .

We only present the protocol for per-server replication watermarks in this subsection, because it provides extended ability to avoid quorum read for efficient reading. The mechanism for per-shard watermarks is similar.

1. In the RPC layer, each server maintains a list of in-flight write-only requests for each remote server. Thus, the server can build a map including each remote server and the lowest unacknowledged write version for that server (comparable to the  $Srw$  in Section 4.2).
2. The gossipier retrieves the above maps from all servers in the local DC. Then, it merges the maps into one, choosing the minimum unacknowledged version for each remote server. That map includes all servers and the lowest version of any in-flight writes (originating from this DC) for that server (comparable to the  $Drw$  in Section 4.2).
3. By gossiping the map, the gossipers can generate the replication watermark for each server (comparable to the  $Rwatermark$  in Section 4.2).

To conclude, we compare the per-server protocol with the system-wide one ( $Rwatermark$ ) in terms of reading efficiency and protocol overhead. The per-server protocol is more efficient for reading because it exploits opportunities to avoid sending messages to an entire quorum of replicas in the read path. On the other hand, it uses larger messages. In the per-server protocol, the message size for gossip may be linear in the number of servers, while the system-wide protocol has a fixed gossip message size in each round (only one version number for the watermark). [For example, in an OV cluster comprising 100 servers, the per-server protocol uses messages of a few dozen KB in size, containing at minimum an IPv4 address and timestamp for each server.](#)

**Table 2** Ping RTT between used EC2 regions (ms).

|             | N. Virginia | Frankfurt | Soul |
|-------------|-------------|-----------|------|
| N. Virginia | < 1         | 91        | 188  |
| Frankfurt   | -           | < 1       | 253  |
| Soul        | -           | -         | < 1  |

[The system-wide protocol, in contrast, requires only a few bytes per message in this scenario.](#)

## 7 Evaluation

This section presents the experimental evaluation of OV-DB for geo-distributed transactions and compares against TAPIR [38]. TAPIR combines concurrency control and replication in one coordination protocol, and outperforms several conventional designs in terms of both throughput and latency [38]. OV is also a combined protocol, and its best-case latency is equivalent to TAPIR – one WAN RTT – when the system is lightly loaded. All the experiments presented in this section distribute data globally across US, Asia and EU regions using AWS EC2 virtual machines, where the wide-area latency inherently leads to higher contention. The experimental results show that OV-DB outperforms TAPIR for throughput under medium to high contention, at the cost of higher latency for the watermark gossips, which usually takes one WAN RTT in common cases. Under a distributed load with Zipf coefficient 0.5, OV-DB achieves more than 10-fold higher peak throughput than TAPIR, reaching 43k read-write transactions per second.

### 7.1 Experimental Setup

**Environment.** All the experiments are run in AWS EC2 using c4.4xlarge instances. Each instance has eight cores (hyper-threading disabled). We use three EC2 regions: US East (N. Virginia), EU (Frankfurt) and Asia (Soul), with ping latencies as shown in Table 2. All the virtual machines enable the NTP service for clock synchronization. We adopt the TAPIR [38] configuration where each shard of data is handled by a separate replication group. The database is partitioned into three shards and each shard has three replicas placed across the three geographic regions. In this setting, at least two out of three replicas form a majority quorum and all three replicas form a super quorum.

**Experimental Systems.** We implemented OV-DB using the C++ RPC framework fbthrift [10] open-sourced by Facebook. In OV-DB deployment, gossipers are co-located on the server hosts without using more machines than TAPIR, and send gossip messages to

**Table 3** The transaction type distribution of Retwis, reproduced from TAPIR paper [38].

| Transaction Type | # gets     | # puts | Percentage |
|------------------|------------|--------|------------|
| Add Users        | 1          | 3      | 5%         |
| Follow           | 2          | 2      | 15%        |
| Post Tweet       | 3          | 5      | 30%        |
| Get Timeline     | rand(1,10) | 0      | 50%        |

remote gossipers at 25ms intervals. We deploy one gossiper per DC. In each EC2 region, we use ten separate client machines, which is sufficient to generate our experimental workloads. The clients in OV-DB can send a request to any server in the DC, and that server is the transaction coordinator. In contrast, the transaction coordinator in TAPIR is on the client side, which offloads work from the TAPIR server.

To measure the performance of TAPIR, we use its open-source implementation [34] created by the authors of TAPIR. We include the modification of the implementation to allow TAPIR to issue multiple independent read requests from the same transaction in parallel, and to run multiple server threads concurrently on each server host. Both systems use in-memory storage to store the data, and neither implementation includes failure recovery. OV-DB simply assumes that obsolete versions can be garbage collected safely when they are 10 seconds older than the *Vwatermark*, which is sufficient for our settings. However, a more general solution can be developed by tracking the minimum uncomputed functor watermark, which is presented in Section 6.1.

We estimate the overhead of multi-version storage as less than 1GB per host in our experiments given a 10s retention period for old versions. Furthermore, we ran a micro-benchmark against the MV storage of random read/write operations (50% read) for 60 seconds (the first and last 10 seconds excluded) to measure the impact of MV on performance, and observed that the storage layers of TAPIR and OV-DB can perform 368k ops/s and 424k ops/s, respectively, in a single thread. These results show that MV storage is far from becoming the system bottleneck with respect to aggregate throughput.

We use closed-loop synchronous clients, each of which issues one transaction at a time, back-to-back. The clients do *not* retry any aborted transaction, which benefits TAPIR’s throughput and latency, especially for high contention cases where conflicts lead to frequent aborts. We present the end-to-end latency measured from the clients and the aggregated throughput across all servers. All experiments run 60 seconds, and the results exclude the first and last 10 seconds.

**Workloads.** We evaluate the systems using two workloads: YCSB+T [8], which extends the key-value store benchmark YCSB with transactional support, and

the Twitter-like workload Retwis [23]. Both YCSB+T and Retwis are used for evaluating TAPIR [38]. We take the same configuration as the published paper. Specially, a transaction in YCSB+T comprises four read-modify-write operations on distinct keys. Retwis simulates four kinds of transactions (add user, post tweet, get timeline, follow user) that apply gets and puts over the key-value store. Table 3 reproduces the transaction profile of Retwis from TAPIR [38].

All experiments in this section (including both Retwis and YCSB+T) run with 1 million keys, where both keys and values are 64 bytes long. We use two different key distributions for tuning various contention scenarios. The first distribution is Zipfian with the coefficient varying from 0.5 to 0.95 (extremely skewed distribution). The second distribution is *contention index* (CI), used in benchmarking distributed transactions in previous works [13, 31, 33] as the tuning knob for the contention among two concurrent transactions. In the CI distribution, the key set is divided into “hot keys” and “cold keys”. Each transaction randomly accesses exactly one hot key and three distinct cold keys. The CI is the fraction of hot keys over the size of the entire key space. For example, if there are 1000 hot keys among 1 million keys, the CI is 0.001. In that case, for systems that cannot commit conflicting transactions in parallel (e.g., TAPIR), at most 1000 concurrent transactions can be committed successfully in parallel.

## 7.2 YCSB+T Experiments

### 7.2.1 Low Pairwise Conflict Rate

Figure 4 presents the throughput, latency and commit rate results for various numbers of clients using the Zipf coefficient 0.5 distribution. A close-up view is included in the bottom corner of each graph for client number less than 4000, and uses the same axis units as the standard view.

**Throughput.** In the terms of peak throughput, OV-DB outperforms TAPIR more than 10-fold, achieving around 43k txn/s. TAPIR achieves higher throughput when the number of clients is less than around 1600, because it maintains a high commit rate and lower latency. As concurrency increases with the number of clients, contention grows and the commit rate of TAPIR drops sharply beyond 1500 clients. TAPIR reaches its peak throughput when the increased parallelism from adding clients is counteracted by a dropping commit rate due to rising contention. In comparison, OV-DB keeps the commit rate steady at 100% as it never aborts a transaction due to conflicts, albeit with

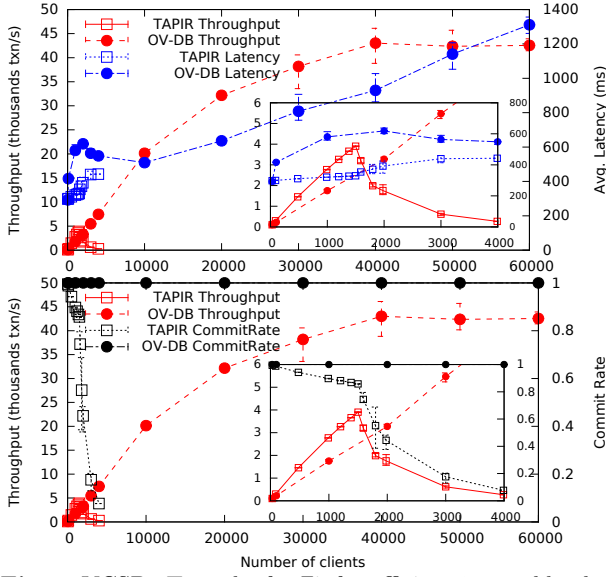


**Fig. 4** YCSB+T results for Zipf coefficient 0.5 workload and various numbers of clients, with close-up on the right.

a more limited transaction model (pre-defined write-set) than that of TAPIR.

**Latency.** In the figure, the average latency of TAPIR is around 300–400ms in most cases, because there is no retry for the aborted transactions in the experiments. When there are fewer than 4000 clients, OV-DB’s average latency is generally higher than TAPIR by roughly one WAN RTT. This indicates that, on average, each transaction in OV-DB waits roughly one WAN RTT to receive a gossiped watermark that makes the transaction visible. This latency overhead is the main cost of OV as compared with other distributed transaction protocols. We observe that the average latency of OV-DB declines slightly for 2000–10000 clients, where requests fill up the processing pipelines in the servers and schedulers more easily find a transaction that is ready for processing. With more than 10000 clients, OV-DB’s latency increases linearly as the servers’ queue length grows. Note that in the latency result, OV-DB and TAPIR both achieve the best latency of around 300ms (roughly one WAN RTT) when the client number is close to 0, as explained in Section 3.1.

### 7.2.2 High Pairwise Conflict Rate

The performance impact of conflicts varies with the concurrency control mechanism. We are therefore interested in examining the performance of the two systems in high contention workloads. We use various numbers of clients with the CI distribution, using a CI of 0.001 (1000 hot keys).

**Throughput.** In Figure 5, OV-DB achieves 100% commit rate in all experiments, while TAPIR exhibits

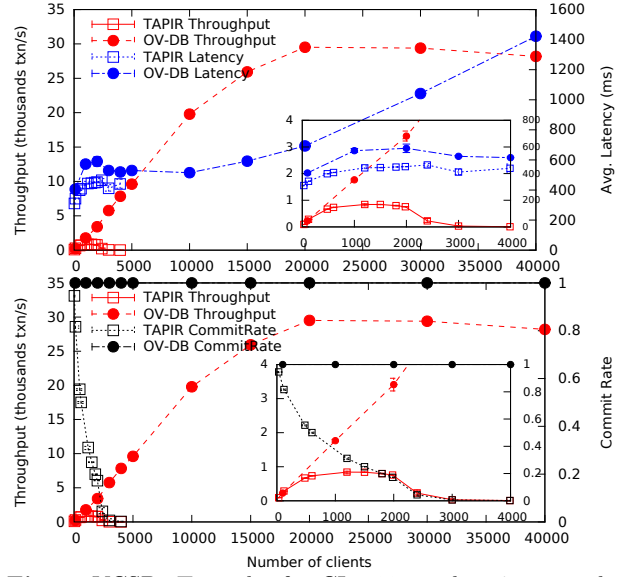


**Fig. 5** YCSB+T results for CI 0.001 and various numbers of clients, with close-up on the right.

a much lower commit rate as the number of clients increases. As a result, OV-DB outperforms TAPIR for throughput under various numbers of clients, albeit at the cost of higher latency. The peak throughput of OV-DB is 30x higher than of TAPIR. The spread is higher than observed in Figure 4, indicating the performance of OV-DB is less sensitive to contention than TAPIR, which is unable to commit conflicting transactions in parallel. ACC allows many operations of conflicting transactions to run concurrently in OV-DB (recall Table 1), but the read-only operations still need to wait for the first available asynchronous write value if there is a read-dependency. Thus, we observe that the throughput does not increase when there are more than 20000 clients in OV-DB, as the additional request concurrency tends to lengthen the E-phase and increase latency.

**Latency.** We observe similar trends to that of Zipf coefficient 0.5 distribution in latency. However, comparing with the latency results in Figure 4, TAPIR has slightly increased latency under the same number of clients, while OV-DB keeps nearly the same latency in the close-up figure.

### 7.2.3 Non-Aborting Workload

This subsection studies a special case of workloads where no pair of concurrent transactions ever conflicts (i.e., commit rate 100% is guaranteed). The two systems are subjected to two *different* workloads, each of which is considered an “ideal” workload for the system’s throughput. For TAPIR, based on the CI 0.001 distribution, we fixed the number of clients to 1000 and each client

**Table 4** Peak throughput (unit: txn/s) comparison of OV-DB and TAPIR under no aborting workload (CI-fix) and high pairwise conflict rate workload (CI 0.001 from Figure 5).

|         | OV-DB | TAPIR | Speedup |
|---------|-------|-------|---------|
| CI-fix  | 39247 | 2781  | 14x     |
| CI      | 29580 | 465   | 64x     |
| Speedup | 1.3x  | 6x    |         |

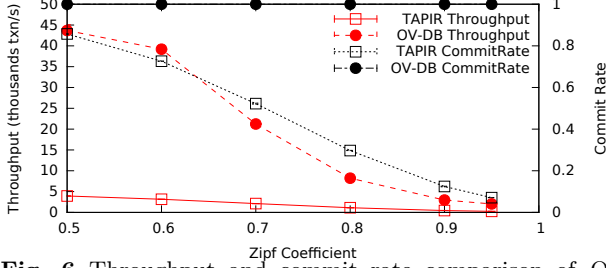


**Fig. 6** Throughput and commit rate comparison of OV-DB and TAPIR for Zipfian workload.

accesses a distinct fixed hot key and a disjoint range of cold keys. Thus, concurrent transactions never have conflicts because their key sets have an empty intersection. We refer to this method of generating keys as *CI-fix* in the remainder of this subsection. We use this workload to simulate the best throughput TAPIR may achieve for a CI of 0.001. At any time, the system is running approximately 1000 concurrent transactions, and any additional client/transaction would introduce conflicts and cause aborting. For OV-DB, we also use CI-fix, but we do not limit the number of clients to 1000 because conflicting transactions in OV-DB never abort. Although the numbers of clients used for TAPIR and OV-DB are not identical, we believe that 1000 is roughly the optimal number for TAPIR as adding more clients only hurts throughput.

In the results summarized in Table 4, the throughput of TAPIR under the non-aborting workload CI-fix is close to the throughput limits observed in previous experiments with 1000 concurrent clients. Thus, TAPIR for the CI-fix workload exhibits a 6x throughput speedup as compared with standard CI workload. The speedup is closer to 1x for OV-DB, though our system still outperforms TAPIR for throughput under CI-fix by 14x, despite the lack of aborting. Interestingly, the outcome is loosely comparable to running both systems with a workload where 14 conflicting transactions are issued in parallel, even though neither system aborts any transactions in the CI-fix workload.

#### 7.2.4 Various Zipf Coefficients

To evaluate further the relative merits of OV-DB and TAPIR, we measure the throughput of the two systems under various Zipf coefficients. We use 1500 clients for

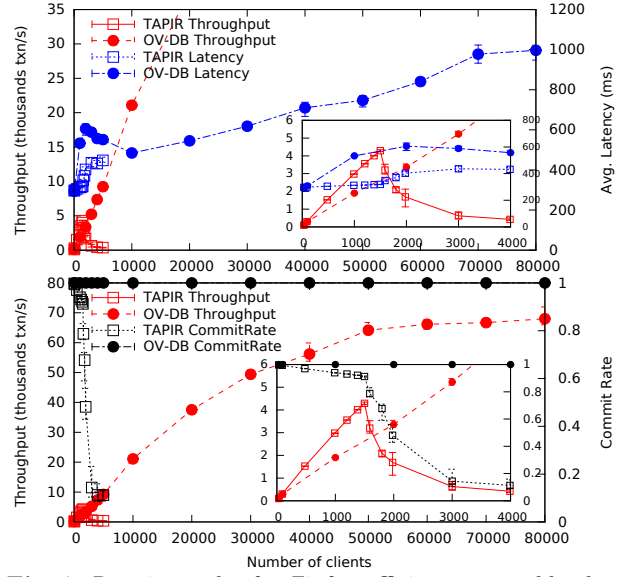


**Fig. 7** Retwis results for Zipf coefficient 0.5 workload and various numbers of clients, with close-up on the right.

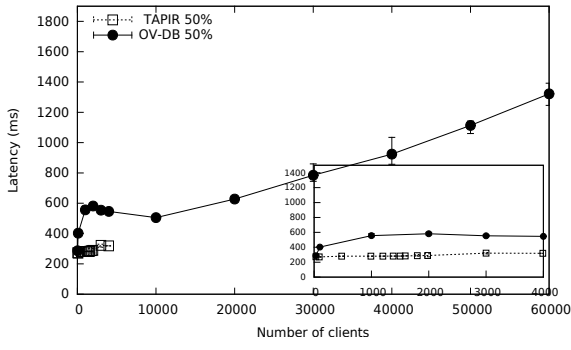
TAPIR, where it achieves the highest throughput in the previous experiments, and 40000 clients for OV-DB. With a fixed number of clients, Zipf coefficients 0.5 and 0.95 represent a low and high pairwise conflict rate, respectively.

In Figure 6, under various values of the Zipf coefficient, throughput drops for both systems as conflicts increase, but OV-DB outperforms TAPIR in throughput by 6-11x. OV-DB maintains the commit rate at 100%, while the commit rate of TAPIR drops as conflicts increase.

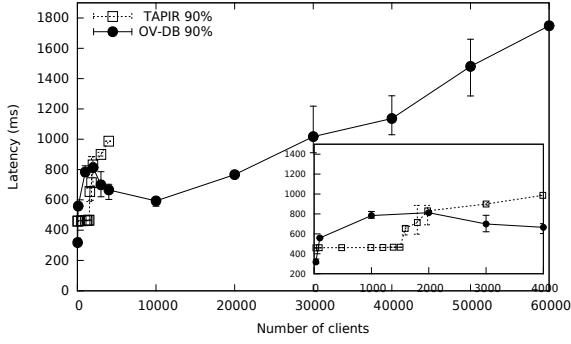
#### 7.2.5 Latency Percentiles Results

To further compare the latency character of the two systems, we also measured the latency percentiles in the experiments for both low pairwise conflict rate (Section 7.2.1) and high pairwise conflict rate (Section 7.2.2) scenarios.

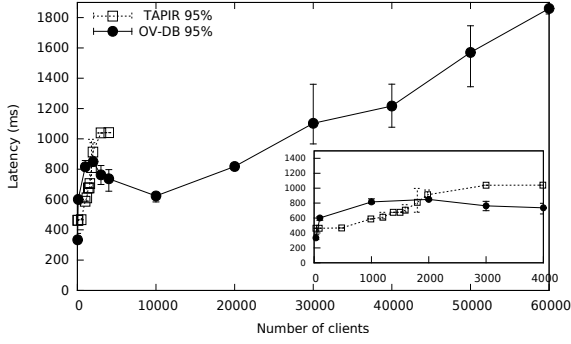
Figure 8 presents the 50%, 90%, 95% and 99% latency percentiles of the experiments of Figure 4. In the result, we observed that 50%-ile results are similar to that of average latency. However, when the number of clients exceeds roughly 2000, the 90%, 95% and 99%-ile latencies of TAPIR are higher than those of OV-DB when the contention is increased. For the latency margins between OV and TAPIR when the number of clients is less than 2000, we observe higher margins in higher percentile latency experiments. For example, TAPIR has maximum around 400ms lower 90%-ile latency than OV, but has only maximum around 200ms lower 95%-ile latency than OV. It indicates that growing contention as the number of clients increases has



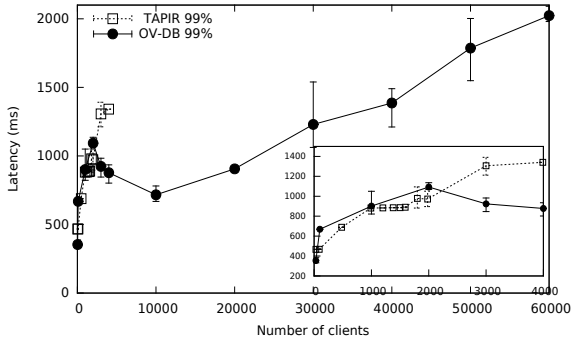
(a) 50% latency percentile results.



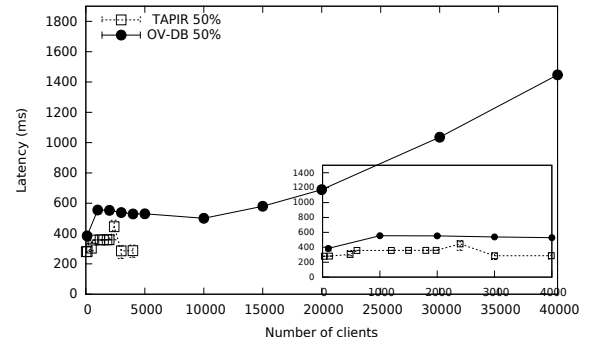
(b) 90% latency percentile results.



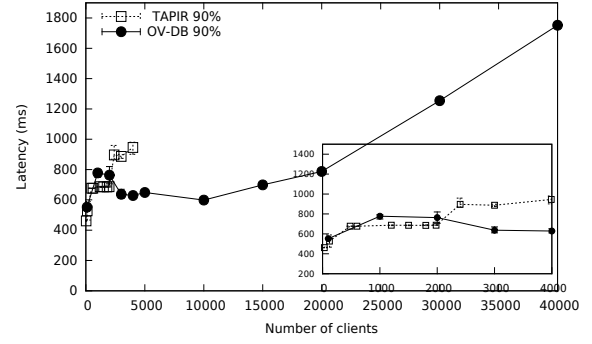
(c) 95% latency percentile results.



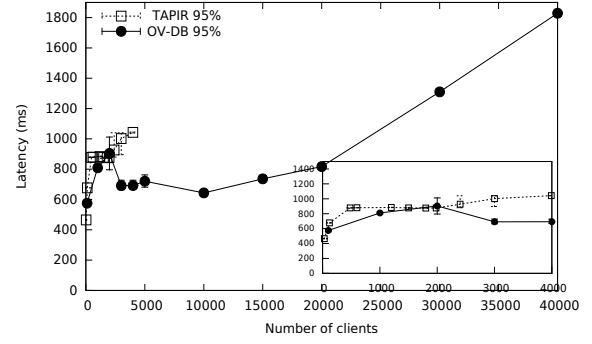
(d) 99% latency percentile results.

**Fig. 8** YCSB+T latency percentile results for Zipf coefficient 0.5 workload and various numbers of clients, with close-up on the right.


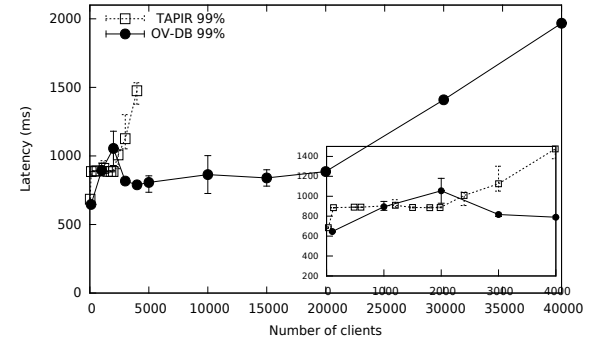
(a) 50% latency percentile results.



(b) 90% latency percentile results.



(c) 95% latency percentile results.



(d) 99% latency percentile results.

**Fig. 9** YCSB+T latency percentile results for CI 0.001 and various numbers of clients, with close-up on the right.

more impact for TAPIR than OV in terms of upper percentile latency. That trend is more pronounced when the number of clients becomes higher. Specifically, even with 40000 clients, OV has 99%-ile latency less than 1386ms, while TAPIR reaches 1341ms with only 4000 clients.

Figure 9 presents the 50%, 90%, 95% and 99% latency percentiles of the experiments of Figure 5. Generally, the result is similar to that of Figure 8, except that 95% to 99%-ile latency results of TAPIR are worse in high pairwise conflict rate scenarios than in low pairwise conflict rate scenarios. The latency margin difference between OV and TAPIR is smaller as compared with results in Figure 8 when the client number is less than 2000. OV even has superior 95% and 99%-ile latency when the number of clients is less than 1000.

### 7.3 Retwis Experiments

#### 7.3.1 Various numbers of clients

We present the throughput, latency and commit rate in Figure 7 for Retwis experiments under various numbers of clients for the Zipf coefficient 0.5 distribution. OV-DB achieves a peak throughput of 68k txn/s, which outperforms TAPIR roughly 16-fold. The performance in terms of throughput and latency is better for both systems in this experiment than in the YCSB+T experiments because there are 50% read-only transactions in the Retwis workload profile. On the other hand, we observe similar trends in both the Retwis and YCSB+T results: when the number of clients is less than 1600, TAPIR throughput is higher than OV-DB, but OV-DB outperforms in terms of throughput when more clients are present. OV-DB achieves much higher peak throughput at the cost of higher latency.

#### 7.3.2 Various Zipf Coefficients

Figure 10 shows the throughput and commit rate comparison between OV-DB and TAPIR for various Zipf coefficients. We see that OV-DB has more than one order of magnitude higher throughput than TAPIR under various Zipf coefficients. The commit rate of TAPIR drops from 90% to 13%, while OV-DB keeps the commit rate at 100% when the Zipf coefficient increases from 0.5 to 0.95. Comparing these variations for the Retwis workload with the YCSB+T results, we observe that the OV-DB throughput drops less sharply under the Retwis workload, where 50% of transactions are read-only. When conflicts are rare, TAPIR can process read-only transactions in one round trip if there is an up-to-date “validated version,” but needs additional

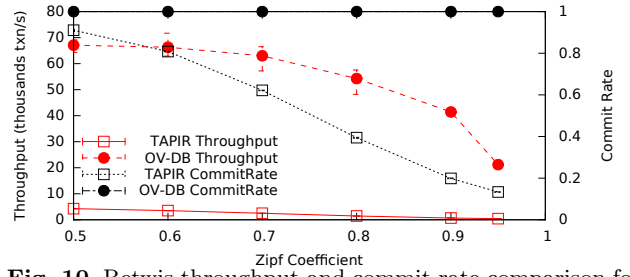


**Fig. 10** Retwis throughput and commit rate comparison for various Zipf coefficients.

round trips, or even suffers aborts, in cases where such a “validated version” is not found. In comparison, the read-only transaction in OV-DB always reads a specific version, and it never aborts due to conflicts.

### 8 Related Work

Distributed transactions are notoriously costly, especially for high-contention and geo-distributed workloads. Thus, many existing works focus on weak consistency [25, 26, 32] or avoid geo-distributed deployments [2, 3, 15, 35]. This section reviews techniques for high performance distributed transactions under strong consistency guarantees.

**Combining concurrency control and replication.** TAPIR [38] is the state-of-the-art high performance geo-distributed transaction protocol. TAPIR can save one round trip compared to a classic layered protocol if conflicts are not present, combining concurrency control and replication in the same round of coordination. However, conflicts from both concurrency control and replication are detected and resolved using validation and retry in OCC, and performance may suffer from contention. OV also uses a combined protocol to reduce coordination overhead, but uses visibility control via ACC to resolve conflicts. Compared with TAPIR, OV incurs additional overhead for watermark gossip, which adds latency, but each gossip message can provide visibility control for many transactions regardless of contention.

**Deterministic databases.** Calvin [33] is a deterministic database that resolves conflicts by executing transactions in a deterministic ordering. Calvin orders the input transactions using a centralized *sequencer* into a deterministic order, then replicates the ordered transactions via a consensus protocol such as Paxos [21] or Zookeeper [17]. Even though Calvin is designed for distributed transactions, it faces several practical difficulties for globally-distributed transactions: (1) the centralized sequencer could potentially be a performance bottleneck and single point of failure when considering

client requests and failure detection over a WAN; (2) the layered concurrency control and replication protocols need more round trips than a combined protocol (e.g., TAPIR, OV), which results in increased latency; (3) Calvin only replicates the transaction input to different replication groups, and there is no coordination within a replication group in execution. This means that transactions cannot read from remote DC’s replicas for fault tolerance when the local replica fails or is overloaded. In OV, the S-phase is non-deterministic, as any transaction can be aborted if any shard votes to do so, but the E-phase executes transactions in the global version order, which resembles deterministic ordering. Unlike Calvin, OV orders the transactions via version numbers assigned using synchronized clocks, and does not require additional centralized coordination. The systems BOHM [11] and PWV [12] also use deterministic execution and multi-version storage to increase transaction processing parallelism, but they are designed for a single multi-core machine.

**Re-ordering conflicting transactions.** Similarly to TAPIR, Janus [27] is a combined concurrency control and replication protocol, but for a more restrictive transaction model. Instead of aborting a transaction on conflicts, Janus transforms transaction execution to a deterministic order to resolve the conflict. As a result, Janus cannot run conflicting transactions in parallel. For example, in Janus, two conflicting write-only transactions need one RTT to detect the conflicts and another RTT to re-order, then the transactions are executed serially. In comparison, our ACC scheme allows both transactions to run in parallel with nearly no coordination in one WAN RTT. Furthermore, Janus’ fast path requires all servers in the replication group (Write-All), which is vulnerable to stragglers in a geo-distributed setting; OV only needs a super quorum of replicas (Write-Quorum). In terms of transaction model, Janus executes a “transaction piece” on each server, whose input and output are limited to the local server, whereas OV executes transactions in a more flexible manner using functors. The Janus paper presents an extension to handle the case of one piece’s input depending on the output of another piece, but it does not support the case when two pieces are mutually dependent on each other. For instance, a money transfer transaction that checks the status of both accounts (located on different servers) prior to the transfer cannot be handled.

**Low-latency distributed transactions.** The system Lynx [39] executes a limited category of transactions with serializability and low latency. Lynx assumes that transactions can be executed piece-wise as a chain (one piece per hop), and that only the first hop contains

user-initiated aborts. Lynx can acknowledge transaction commitment as soon as the first hop completes.

Carousel [37] is a combined protocol targeting low-latency geo-distributed transaction processing. It uses OCC, similarly to TAPIR, and therefore exhibits serialization windows of at least one WAN RTT. As a result, its throughput suffers under conflicts. Contrarian protocol [9] is designed for key-value stores supporting latency-optimal read-only transactions, which require two round trips of client-server messages. It only provides weak (i.e., causal) consistency. SLOG [30] combines determinism with assignment of a *home region* to each data granule to achieve low-latency for transactions that only write and (linearizably) read home regions. However, for a transaction that accesses data having replication masters spanning different DCs (i.e., multi-home transaction), SLOG also needs cross-DC coordination, in which case the overhead exceeds a deterministic database (e.g., Calvin), as reported in the evaluation of the SLOG paper [30]. Like other master-based replication protocols, SLOG also faces the risk of performance bottlenecks or stragglers from the master nodes.

Amazon Aurora [35, 36] uses a “storage and computing independent” architecture, where storage follows the “log as database” paradigm. The log sequence numbers (LSNs) can be used for batch acknowledgment [36], similarly to the watermarks in OV. Nevertheless, Aurora solves a simpler problem than OV in several aspects: Aurora targets a cluster in a single geographical region (e.g., us-east-1) with multiple availability zones, while OV targets geo-distributed data; Aurora assumes there is only a single writer (the DB instance) which assigns consecutive LSNs and is able to coordinate transaction concurrency control locally; Aurora uses traditional concurrency control (e.g., locking) and hence cannot execute conflicting transactions in parallel.

## 9 Conclusion

This paper proposes Ocean Vista, a high performance geo-distributed strictly serializable transaction processing protocol. OV combines concurrency control, transaction commitment, and replication in a single transaction visibility control protocol that uses version watermarks. Our Asynchronous Concurrency Control scheme is based on watermark gossip, and is able to process conflicting transactions in parallel. Furthermore, our replication scheme makes it possible to use an efficient Write-Quorum / Read-One access pattern in the common case. OV provides high throughput even under high contention for globally distributed transactions,

though pays a latency cost for gossiping the watermarks. The best-case latency to commit a transaction in OV is only one WAN RTT, which is equivalent to the best latency of TAPIR [38] – a state-of-the-art geo-distributed transaction protocol. Our experimental evaluation shows that our system, called OV-DB, outperforms TAPIR in terms of throughput in conflict-prone workloads. In particular, OV-DB achieves one order of magnitude higher peak throughput even under a workload with Zipf coefficient 0.5. When conflicts are rare, OV-DB generally pays a latency penalty of one WAN RTT.

**Acknowledgements** We are grateful to the anonymous reviewers from both PVLDB and VLDBJ, and to Ken Salem, for their helpful feedback on this research. Wojciech Golab is supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and by Ripple.

## References

- Balakrishnan, M., Malkhi, D., Davis, J.D., Prabhakaran, V., Wei, M., Wobber, T.: CORFU: A Distributed Shared Log. *ACM Trans. Comput. Syst.* **31**(4), 10:1–10:24 (2013)
- Bernstein, P., Reid, C., Das, S.: Hyder - a transactional record manager for shared flash. In: Proceedings of the 5th Biennial Conference on Innovative Data Systems Research, CIDR’11, pp. 9–20 (2011)
- Bernstein, P.A., Das, S., Ding, B., Pilman, M.: Optimizing optimistic concurrency control for tree-structured, log-structured databases. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD’15, pp. 1295–1309 (2015)
- Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1987)
- Bernstein, P.A., Shipman, D.W., Wong, W.S.: Formal aspects of serializability in database concurrency control. *IEEE Trans. Softw. Eng.* **5**(3), 203–216 (1979)
- Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: A comprehensive study. *ACM Comput. Surv.* **33**(4), 427–469 (2001)
- Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google’s globally distributed database. *ACM Trans. Comput. Syst.* **31**(3), 8:1–8:22 (2013)
- Dey, A., Fekete, A., Nambiar, R., Röhm, U.: YCSB+T: Benchmarking web-scale transactional databases. In: 2014 IEEE 30th International Conference on Data Engineering Workshops, pp. 223–230 (2014)
- Didona, D., Guerraoui, R., Wang, J., Zwaenepoel, W.: Causal consistency and latency optimality: Friend or Foe? *PVLDB* **11**(11), 1618–1632 (2018)
- Facebook: fbthrift. <https://github.com/facebook/fbthrift>
- Faleiro, J.M., Abadi, D.J.: Rethinking serializable multi-version concurrency control. *PVLDB* **8**(11), 1190–1201 (2015)
- Faleiro, J.M., Abadi, D.J., Hellerstein, J.M.: High performance transactions via early write visibility. *PVLDB* **10**(5), 613–624 (2017)
- Fan, H., Golab, W.: Scalable transaction processing using functors. In: Proceedings of the 38th IEEE International Conference on Distributed Computing Systems, ICDCS’18, pp. 1004–1016 (2018)
- Fan, H., Golab, W.: Ocean Vista: Gossip-based visibility control for speedy geo-distributed transactions. *PVLDB* **12**(11), 1471–1484 (2019)
- Goel, A.K., Pound, J., Auch, N., Bumbulis, P., MacLean, S., Färber, F., Gropengiesser, F., Mathis, C., Bodner, T., Lehner, W.: Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *PVLDB* **8**(12), 1716–1727 (2015)
- Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3), 463–492 (1990)
- Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: Wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC’10, pp. 11–11 (2010)
- Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E.P.C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., Abadi, D.J.: H-store: A high-performance, distributed main memory transaction processing system. *PVLDB* **1**(2), 1496–1499 (2008)
- Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing, STOC’97, pp. 654–663 (1997)
- Kraska, T., Pang, G., Franklin, M.J., Madden, S., Fekete, A.: MDCC: Multi-data center consistency. In: Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys’13, pp. 113–126 (2013)
- Lamport, L.: The part-time parliament. *ACM Trans. Comput. Syst.* **16**(2), 133–169 (1998)
- Lamport, L.: Fast Paxos. *Distributed Computing* **19**, 79–103 (2006)
- Leau, C.: Spring Data Redis - Retwis-J. <https://docs.spring.io/springdata/data-keyvalue/examples/retwisj/current/> (2013)
- Levandowski, J.J., Lomet, D.B., Sengupta, S.: The Bw-Tree: A B-tree for new hardware platforms. In: Proceedings of the 2013 IEEE International Conference on Data Engineering, ICDE’13, p. 302–313 (2013)
- Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI’13, pp. 313–328 (2013)
- Lockerman, J., Faleiro, J.M., Kim, J., Sankaran, S., Abadi, D.J., Aspnes, J., Sen, S., Balakrishnan, M.: The fuzzylog: A partially ordered shared log. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI’18, pp. 357–372 (2018)
- Mu, S., Nelson, L., Lloyd, W., Li, J.: Consolidating concurrency control and consensus for commits under conflicts. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI’16, pp. 517–532 (2016)

28. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'14, pp. 305–320 (2014)
29. Papadimitriou, C.H.: The serializability of concurrent database updates. *J. ACM* **26**(4), 631–653 (1979)
30. Ren, K., Li, D., Abadi, D.J.: SLOG: Serializable, low-latency, geo-replicated transactions. *PVLDB* **12**(11), 1747–1761 (2019)
31. Ren, K., Thomson, A., Abadi, D.J.: An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB* **7**(10), 821–832 (2014)
32. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP'11, pp. 385–400 (2011)
33. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: Fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD'12, pp. 1–12 (2012)
34. UWSysLab: TAPIR implementation. <https://github.com/UWSysLab/tapir> (2018)
35. Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., Bao, X.: Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD'17, pp. 1041–1052 (2017)
36. Verbitski, A., Gupta, A., Saha, D., Corey, J., Gupta, K., Brahmadesam, M., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., Bao, X.: Amazon Aurora: On avoiding distributed consensus for I/Os, commits, and membership changes. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD'18, pp. 789–796 (2018)
37. Yan, X., Yang, L., Zhang, H., Lin, X.C., Wong, B., Salem, K., Brecht, T.: Carousel: Low-latency transaction processing for globally-distributed data. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD'18, pp. 231–243 (2018)
38. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.* **35**(4), 12:1–12:37 (2018)
39. Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M.K., Li, J.: Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP'13, pp. 276–291 (2013)