Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions

Hua Fan
Alibaba Group
Hangzhou, China
guanming.fh@alibaba-inc.com

Wojciech Golab[*]
University of Waterloo
Waterloo, Canada
wgolab@uwaterloo.ca

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. 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.

PVLDB Reference Format:
Hua Fan and Wojciech Golab. Ocean Vista: Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions. *PVLDB*, 12(11): 1471-1484, 2019.
DOI: https://doi.org/10.14778/3342263.3342627

[*]Author supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342627

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, 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 [19]) 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., fix keyspace and fix 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 concurrency control mechanisms (e.g., OCC, 2PL) processes distributed transactions *synchronously*. They resolve transaction ordering 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, such as TAPIR [35] and MDCC [18], combine concurrency control and replication into a single protocol for efficiency. TAPIR allows WQRO for inconsistent reads, but uses an "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 watermark 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 asynchronously 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 based on synchronized clocks, and OV processes the transactions asynchronously according this order. To optimize parallelism, ACC transforms a read-write transaction to three operations: write-only operations to write placeholders (i.e., functors [12]) to the MV storage, read-only operations, and asynchronous writes of a specific version. The visibility watermark (***Vwatermark***) is a special version number, below which all transactions must have completed their write-only operations (S-phase). 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.

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 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 [16, 24, 29]. Additionally, OV supports transactions without pre-declared write-sets using a known technique, the reconnaissance query (see Section 3.4). 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 [35]. 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.

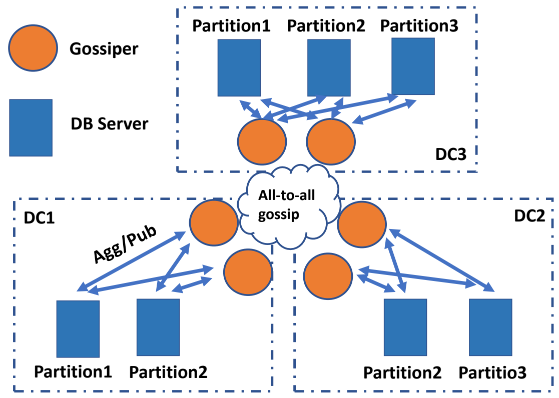

Figure 1: Architecture: each DC has at least one gossiper, which aggregates and publishes watermarks within its datacenter and exchanges the watermarks with other gossipers.

2. ARCHITECTURE

Figure 1 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 gossiper 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 gossiper generates and publishes the watermarks within its DC.

DB Servers. OV-DB partitions the database into shards. Each shard is managed by a group of DB server replicas. 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 [17].

(3) Transaction execution. Transaction logic is executed via

stored procedures in DB servers, when the transaction version number *ts* is below the *Vwatermark*.

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 3.3).

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

Given the above properties, 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*.

We assume the clocks of servers synchronized, e.g., using Network Time Protocol (NTP). Skewed clocks may only affect the performance of OV, and never compromise the consistency guarantee, in contrast to the TrueTime service in Spanner [6]. 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 sketch out a proof that OV provides strict serializability.

3.1 Asynchronous Concurrency Control

We first present the asynchronous concurrency control (ACC) technique for read-write transactions with a known write-set; other transaction types are detailed in Section 3.4. OV processes transactions in an *asynchronous* way: it transforms the processing of a read-write transaction to a sequence of write-only operations, read-only operations, and asynchronous writes. ACC decouples transaction ordering, accomplished by recording the keys in the write-set for a global version, from executing the transaction logic, which entails read-only operations and asynchronous writes only.

Write-only operations. 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 [12], 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-partition transactions in which the value read from one partition may impact the value written in another partition. 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

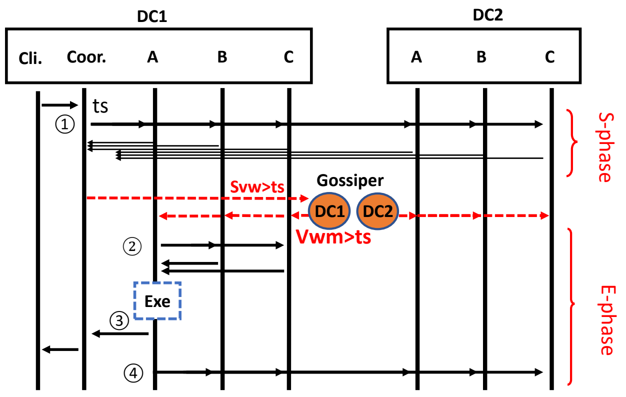

Figure 2: A transaction life cycle example using ACC. The transaction logic is executed by the first scheduled functor in DC1 partition A (dashed box). Red dashed lines denote the watermark messages.

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($k$, $ts$, $T$)                  ①                ⎪
5      if any call to Write fails then                           ⎬ S-phase
6          AbortTxn($ts$, $T$)                                   ⎪
7          return                                                ⎭
8      TidMgr.stored($ts$)
9      asyncRun
10         when $all\ replica\ succ.\ write\text{-}only$ then
11             TidMgr.fullyReplicated($ts$)
12     wait for execution results from $\forall T$ of line 4       ⎫ E-phase
13     return                                                    ⎭
14 Procedure Execute(transaction $T$, version $ts$)
15     parallel-for $\forall k \in T.readSet$ do
16         $input[k]$ = Read($k, ts-1$)          ②
17     $output$ = T.execute($input$)
18     reply with $output$ to coordinator        ③
19     parallel-for $\forall k \in T.writeSet, \forall replica$ has $k$ do
20         replica.Put($ts, k, output[k]$)       ④
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()
```

Algorithm 2: Procedures on a gossiper of DC_k.

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

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 2 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 pseudocode. 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 *TidMgr* (TxnIdManager) 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.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. 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 *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. 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* 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	✓

Discussion: 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 "serialize 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 summaries 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 for 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-dependent") *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 1 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) this 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.*

Discussion: latency. TAPIR uses a combined protocol to streamline coordination across a WAN. As a result, TAPIR at best incurs a latency of one WAN RTT (ignoring local message RTT), when there is a DC-local replica for reading *and* neither reads nor writes have conflicts. The latency increases further due to retries from conflicts as contention increases. Besides, the latency does not include the asynchronous COMMIT message that makes the writes visible. The OV protocol can also achieve best-case latency at *one WAN RTT.* From Figure 2, 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 gossiper 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 2, let's assume the clocks are synchronized and the WAN RTT is T_{wan}. At time t, DC2 has infrequent coordination, and $Gossiper_{dc2}$ sends to $Gossiper_{dc1}$ a *Dvw* (see Section 3.3) 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 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 $Gossiper_{dc2}$.

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 *Vwatermark* and version insertion only targets versions above *Vwatermark*. 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.

3.3 Gossip of Visibility Watermarks

Naively, to satisfy the property **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 gossiper-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 TidMgr.createTS() 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 gossiper generates the *Dvw* by maintaining a snapshot of the latest *Svw* of all DB servers within the DC. A gossiper continuously polls the DB servers for *Svw*, meanwhile, publishing the latest *Vwatermark* to every DB server in the DC. Each DB sever maintains a non-deceasing *Vwatermark*, and it always adopts the highest watermark if it receives different watermarks published by different gossipers.

Similarly, each gossiper 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.

3.4 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. Dependent transactions [29], 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* [29], 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 be 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.

3.5 Strict Serializability

In this subsection, we present a sketch on the proof that OV provides strict serializability for all committed transactions. Informally speaking, a schedule of such transactions is serializable when it is equivalent to *some* serial schedule. Strict serializability [5, 26] further requires that the serialization order be compatible with the "real-time" precedence order over non-overlapping transactions. The latter property is referred to as *linearizability* in the context of concurrent data structures [14]. Informally, a schedule of committed transactions is strictly serializable if each transaction appears to take effect at some time point between its invocation and response. We will refer to this point in time in our analysis as the *serialization point* (**SP**) of a transaction.

Serializability. Transactions executed by OV appear to take effect in order of their version numbers by principle **P1**. A transaction always reads keys for the version immediately before its own version (Algo. 1 line 16), and its updates are also visible only to transactions with higher versions.

Strict ordering. We construct the SP for each transaction as follows: for a transaction T_i with version number ts_i, its SP (denoted sp_i) is the time point when the first replica server within the system receives a *Vwatermark* $> ts_i$. In other words, sp_i is the earliest point in time when T_i is visible to any server. We assume that at one time point only one transaction can be made visible, thus the SPs are never equal for two transactions. It also follows easily that sp_i is between the transaction T_i invocation and response, because sp_i must be within the E-phase of T_i. Furthermore, the execution of T_i in OV is as if the transaction takes effect at sp_i because of the following:

1. Updates of T_i are *not* visible *before* sp_i. This is because prior to sp_i, no server has *Vwatermark* greater than ts_i, otherwise it conflicts with the definition of sp_i.

2. Updates of T_i are visible to other transactions *after* sp_i. According to Algo. 1, after sp_i, only a higher versioned transaction T_j (i.e., having $ts_j > ts_i$) may read the updates of T_i in the E-phase of T_j when *Vwatermark* $> ts_j$. Thus, the same *Vwatermark* $> ts_i$, which implies T_i is visible.

Serializable order matches strict order. We sketch out a proof that for any two committed transactions T_j and T_i in a schedule of OV, if their version numbers satisfy $ts_j > ts_i$ (serializable order), then $sp_j > sp_i$ (strict order). This means that transactions are ordered the same way by their version number as by their SPs. This point is proved by contradiction. At time point sp_j, a replica R must have *Vwatermark* $> ts_j$ (by definition of sp_j), and we assume for the sake of argument that $sp_i > sp_j$, as otherwise $sp_j > sp_i$ and the strict order matches the serializable order. That means T_i is not visible on replica R, and so *Vwatermark* $\leqslant ts_i$. Thus, $ts_i \geqslant$ *Vwatermark* $> ts_j$, which contradicts the assumption that $ts_j > ts_i$.

In summary, OV provides strict serializability.

3.6 On Clock Skew

The coordinator uses the timestamp from synchronized (e.g., by NTP) local clocks to generate global 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.

To enforce the monotonicity property of watermarks regardless of clock skew, we apply the following rules in the protocol. *First*, the coordinator records the highest assigned timestamp *lastTS*, and retries whenever a generated timestamp is lower than the *lastTS*. *Second*, during initialization, the coordinator will not accept client requests until the new coordinator is accepted (described later) by all gossipers of the DC. *Third*, a gossiper will accept a new coordinator only if the *Svw* from the coordinator is higher than the *Dvw*, in order to enforce the monotonicity property for *Dvw*.

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 [18, 24, 35], 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[1] 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 [20] and the consensus operation in TAPIR [35], `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 [20], the fast path needs a super quorum ($\lceil \frac{3}{2}f \rceil + 1$) of replicas to be available;[2] 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.

[1]Super quorums are defined as in FastPaxos [20].
[2]An alternative expression is using a super quorum size $2f+1$ out of $3f+1$ replicas.

`AbortTxn` succeeds when the decision is replicated in a fault-tolerant manner on each shard. Each shard succeeds as long as 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:

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 `Abort(`*key, ts*`)`, and replies *success* to the coordinator.

`AbortTxn` also needs to run the asynchronous procedure in Algorithm 1 line 9-11, which is used to indicate all replicas have succeeded `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 3.3 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 skip the detailed protocol for the gossip of *Rwatermarks* as it is similar to that of *Vwatermark*.

We can extend the protocol of Rwatermark further to *per shard Rwatermark* or *per host Rwatermark*. In other words, the gossipers and DB servers maintain a separate collection of *Srw*, *Drw* and *Rwatermark* for each shard or host.

4.3 The Read-Only Operation

The read-only operation `Read(`*key, ts*`)` ($ts < Vwatermark$) retrieves the latest version no greater than *ts* for *key*. When $ts < Rwatermark$, the `Read` can call `Get(`*key, ts*`)` directly on **any replica**. Thus, this sub-section focuses on reading versions between *Rwatermark* and *Vwatermark*.

The `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 `Read(`*key, ts*`)` operation sends requests to all replicas but waits for the fastest super quorum to respond (with retry on timeout), and find the highest version $ver \leqslant ts$ 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 `Read(`*key, ts*`)` is presented in Algorithm 3.

Algorithm 3: 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 ∀R∈ Replicas do
5             R.Get(k, target)
6         if MATCH CONDITION found then
7             return MATCHED VERSION
8         target = max{versions received at line 5} − 1
```

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, gossiper, and datacenter-wide failures. 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 [25] or Zookeeper [15]. 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 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 gossiper-coordinator communication, as they already publish and aggregate watermarks frequently. Furthermore, the gossiper 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 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 gossiper suspects a failed coordinator and decides to replace it, it sends a message to the backup coordinator. The gossiper then uses 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.

5.3 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.

5.4 Gossiper Failure and Datacenter Failure

The failure of a single gossiper will not impact the progress of watermarks because each DC has multiple gossipers 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 gossipers of a DC are unavailable due to DC-wide disaster. This subsection presents how our gossip protocol can recover from losing all the gossipers 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 5.3). *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. EVALUATION

This section presents the experimental evaluation of OV-DB for geo-distributed transactions and compares against TAPIR [35]. TAPIR combines concurrency control and replication in one coordination protocol, and outperforms several conventional designs in terms of both throughput and latency [35]. 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.

6.1 Experimental Setup

Environment. All the experiments are run in AWS EC2 using c4.4xlarge instances. Each instance has 8 cores (hyper-threading disabled). We use three EC2 regions: US East

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

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

(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 [35] 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 [9] open-sourced by Facebook. In OV-DB, gossipers are co-located on the server hosts without using more machines than TAPIR, and send gossip messages to 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 sever 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 [30] 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 safety 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 we leave as future work.

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 of random read/write operations (50% read) 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 [7], which extends the key-value store benchmark YCSB with transactional support, and the Twitter-like workload Retwis [21]. Both YCSB+T and Retwis are used for evaluating TAPIR [35]. 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 [35].

Table 3: The transaction type distribution of Retwis, reproduced from TAPIR paper [35].

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%

All experiments run with 1M 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 transacions in previous works [12, 27, 29] 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 random 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.

6.2 YCSB+T Experiments

6.2.1 *Low Pairwise Conflict Rate*

Figure 3 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.

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 10000 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

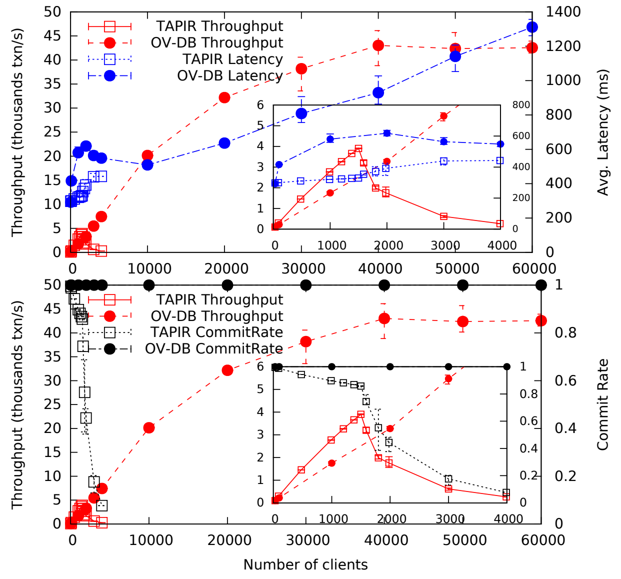

Figure 3: YCSB+T results for Zipf coefficient 0.5 workload and various numbers of clients, with close-up on the right.

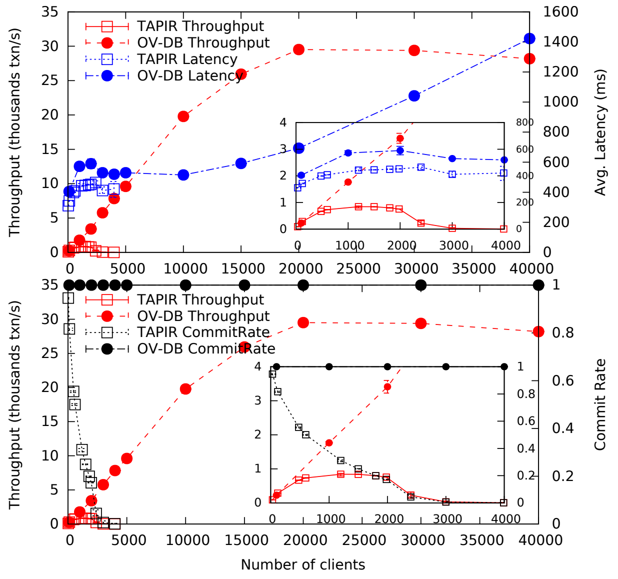

Figure 4: YCSB+T results for CI 0.001 and various numbers of clients, with close-up of the right.

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 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. We also measured the latency percentiles in the experiment, and observed that 50%-ile results are similar to average latency. However, when client number exceeds roughly 2000, the 90%, 95% and 99%-ile latencies of TAPIR are higher than those of OV-DB when the contention is increased. Specifically, even with 40000 clients, OV has 99%-ile latency less than 1386ms, while TAPIR reaches 1341ms with only 4000 clients. We observe similar latency trends in the other experiments, and so we continue to focus on average latency in the remainder of this section.

6.2.2 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 4, OV-DB achieves 100% commit rate in all experiments, while TAPIR exhibits a much lower commit rate as the number of clients increases. As a result, OV-DB outperforms TAPIR for throughput under various number 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 3, indicating the performance of OV-DB is less sensitive to contention than TAPIR, which is unable to commit conflicting transaction 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

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 4).

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

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 3, TAPIR has slightly increased latency under the same number of clients, while OV-DB keeps nearly the same latency in the close-up figure.

6.2.3 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 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

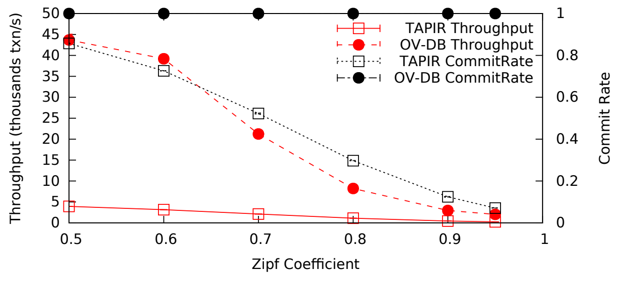

Figure 5: Throughput and commit rate comparison of OV-DB and TAPIR for Zipfian workload.

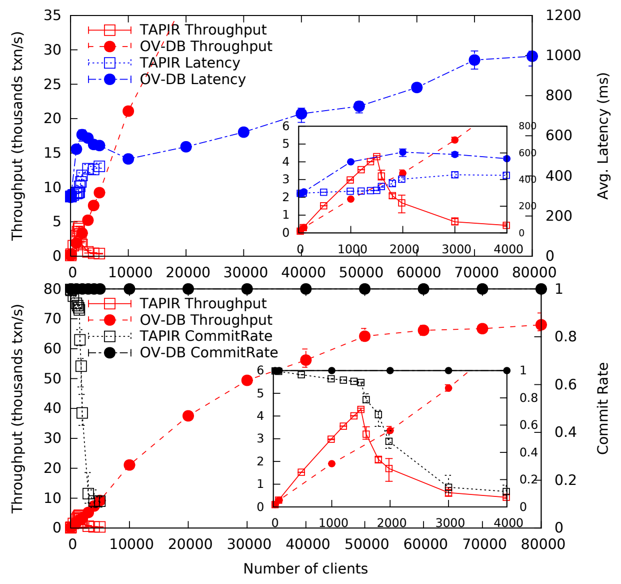

Figure 6: Retwis results for Zipf coefficient 0.5 workload and various numbers of clients, with close-up on the right.

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.

6.2.4 *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 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.

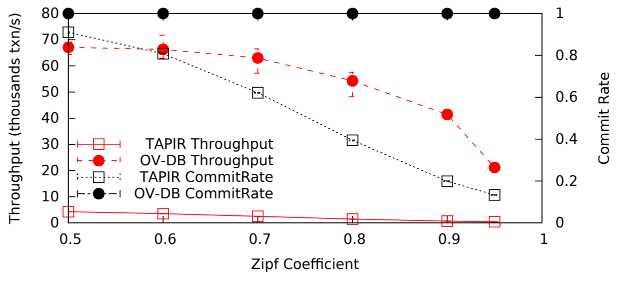

Figure 7: Retwis throughput and commit rate comparison for various Zipf coefficients.

In Figure 5, 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.

6.3 Retwis Experiments

6.3.1 *Various numbers of clients*

We present the throughput, latency and commit rate in Figure 6 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 the both 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.

6.3.2 *Various Zipf Coefficients*

Figure 7 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 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.

7. RELATED WORK

Distributed transactions are notoriously costly, especially for high-contention and geo-distributed workloads. Thus, many existing works focus on weak consistency [22, 23, 28] or avoid geo-distributed deployments [2, 3, 13, 31]. This section reviews techniques for high performance distributed transactions under strong consistency guarantees.

Combination of concurrency control and replication. TAPIR [35] 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 [29] 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 [19] or Zookeeper [15]. 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 rounds 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. BOHM [10] and PWV [11] 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 [24] 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. Lynx [36] 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 [34] 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 [8] 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. Amazon Aurora [32, 33] 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 [33], 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 (AZs), 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.

8. 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 [35] – 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.

9. ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers and to Ken Salem for their helpful feedback on this research.

10. REFERENCES

[1] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A Distributed Shared Log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, Dec. 2013.

[2] P. Bernstein, C. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, pages 9–20, January 2011.

[3] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. 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, pages 1295–1309, New York, NY, USA, 2015. ACM.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[5] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Trans. Softw. Eng.*, 5(3):203–216, May 1979.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.

[7] A. Dey, A. Fekete, R. Nambiar, and U. Rhm. YCSB+T: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230, March 2014.

[8] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel. Causal consistency and latency optimality: Friend or Foe? *PVLDB*, 11(11):1618–1632, 2018.

[9] Facebook. fbthrift. `https://github.com/facebook/fbthrift`.

[10] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[11] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5):613–624, 2017.

[12] H. Fan and W. Golab. Scalable transaction processing using functors. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1004–1016, July 2018.

[13] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *PVLDB*, 8(12):1716–1727, 2015.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[16] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[18] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[19] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[20] L. Lamport. Fast paxos. *Distributed Computing*, 19:79–103, October 2006.

[21] C. Leau. Spring data redis - retwis-j. `https://docs.spring.io/springdata/data-keyvalue/examples/retwisj/current/`, 2013.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[23] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan. The fuzzylog: A partially ordered shared log. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 357–372, Berkeley, CA, USA, 2018. USENIX Association.

[24] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 517–532, Berkeley, CA, USA, 2016. USENIX Association.

[25] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[26] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[27] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB*, 7(10):821–832, 2014.

[28] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages

385–400, New York, NY, USA, 2011. ACM.

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

[30] UWSysLab. TAPIR implementation. `https://github.com/UWSysLab/tapir`, 2018.

[31] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. 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, pages 1041–1052, New York, NY, USA, 2017. ACM.

[32] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. 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, pages 1041–1052, New York, NY, USA, 2017. ACM.

[33] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvilli, and X. Bao. 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, pages 789–796, New York, NY, USA, 2018. ACM.

[34] X. Yan, L. Yang, H. Zhang, X. C. Lin, B. Wong, K. Salem, and T. Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 231–243, New York, NY, USA, 2018. ACM.

[35] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.*, 35(4):12:1–12:37, Dec. 2018.

[36] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.