Chablis: Fast and General Transactions in Geo-Distributed Systems

Tamer Eldeeb
Columbia University
tamer.eldeeb@columbia.edu

Philip A. Bernstein
Microsoft Research
phil.bernstein@microsoft.com

Asaf Cidon
Columbia University
asaf.cidon@columbia.edu

Junfeng Yang
Columbia University
junfeng@cs.columbia.edu

ABSTRACT

Geo-distributed DBMSes often do not support strictly-serializable transactions at all, or limit transactions to only a single region. Those that offer this support force the programmer to either give up low-latency regional writes or use a restricted programming model. We show that these compromises are no longer necessary in modern datacenters for workloads with regional locality, thanks to advances in low-latency datacenter networks and DBMS designs. We present Chablis, a scalable, geo-distributed, multi-versioned transactional key-value store. Chablis provides a *general* interface supporting range and point reads, as well as writes within multi-step strictly-serializable ACID transactions. It offers *fast* (i.e., low latency) read-write transactions accessing data homed in a single region, and global strictly serializable lock-free snapshot reads.

1 INTRODUCTION

Many applications need geo-distributed databases in order to achieve high availability in the face of regional failure, and to serve low-latency reads to clients that are themselves distributed geographically across multiple regions [26, 28]. Many workloads have locality in their access patterns [26, 28], where each data item has a natural home region and is almost always accessed only within that region, so it is crucial that local access to the data in its home region is fast. Nonetheless, it is sometimes necessary to read data from multiple regions in a single logical operation, and programmers want strong isolation and consistency for these transactions. Such multi-region queries take a long time to execute, partly due to the long latencies involved in network round-trips, but often also because of the analytical nature of read-only queries. Therefore, it is ideal to run these queries in a lock-free snapshot manner to avoid blocking other transactions.

Strict serializability (also known as external consistency [8]) is considered the gold standard of database isolation and consistency semantics [10]. It includes the combination of the following two properties [27]: (a) **serializability**: every execution is equivalent to some serial ordering of committed transactions, and (b) **linearizability**: if a transaction A commits before a transaction B starts, then A should precede B in the equivalent serial ordering.

Unfortunately, offering strong semantics like strict serializability in a geo-distributed setting has many well-known challenges [9, 13, 14, 20, 26], leading many popular systems to either disallow multi-region transactions completely [6], or offer weaker semantics instead [3, 19, 20, 22]. However, these weaker semantics expose anomalies to programmers making it harder to develop applications [4, 8, 26], and could lead to security vulnerabilities [29]. As a result, developer demand for strict serializability increased; e.g., Google's Spanner [8] is widely used both within Google and as a cloud offering, and has inspired many open source products [1, 28].

Existing geo-distributed systems that offer strong semantics [8, 16, 23–26, 30–32] compromise either in speed or generality. Many systems [16, 24, 31] incur at least one cross-region round-trip for **every** write, which slows all writes in the system. Spanner [8] guarantees correctness of readers by introducing a delay for writers during the commit protocol until the clock uncertainty interval has passed, which again slows down all writes in the system. Additionally, it uses specialized hardware [1] to achieve clock synchronization guarantees that are necessary for its strict serializability support. On the other end, Slog [26] and Detock [25] offer low-latency for local writes as well as high throughput and the ability to handle high contention even for cross-region writes. However, they rely on deterministic execution which requires restricting the programming model and makes them unable to support a general SQL interface.

We take a different approach, inspired by our recent work on Chardonnay [10]. Chardonnay shows that single-datacenter transactions can be fast and general, thanks to advances in low latency networking and storage. It uses epoch-based versioning to support strictly-serializable lock-free snapshot reads in a single datacenter. Low-latency, high-throughput datacenter RPCs (e.g., eRPC [15]) allow all committing transactions in Chardonnay to cheaply read a counter, called the *epoch*, that serves as a global serialization point.

Our main contribution in this paper is showing how to extend this technique to incorporate a global epoch that advances more slowly, but without impacting single-region transactions. The key idea is to decouple maintaining and advancing the epoch from publishing and reading the epoch. Thus, instead of a single service that replicates the epoch counter and serves reads to clients, our design has two. One is a regional publisher that exists in all regions

This paper appeared at CIDR 2024. This is an updated version clarifying an issue in Section 5.3.

[1]Systems like CockroachDB [28] offer weaker consistency to avoid this dependency.

and allows transactions to read the epoch using fast datacenter RPCs so their commit protocol latency is not affected. The second one is global; it maintains and advances the epoch then updates all the regional publishers, which is a slower process that does not impact committing transactions at all. This introduces challenges in ensuring epoch synchronization across all regions, which are addressed in the snapshot read protocol (§5).

To show the practicality of the technique, we implemented it in Chablis, a geo-distributed multi-version transactional key-value store. Chablis is *fast*: it preserves low-latency and high throughput for transactions that access data in a single region, and supports global multi-region strictly serializable snapshot reads that are lock-free. It also offers quite a *general* interface, supporting point and range reads, as well as writes, within classical multi-step strictly serializable ACID transactions.

To the best of our knowledge, Chablis is the first geo-distributed transactional system that is both *fast* and *general*: it offers geo-distributed strict serializiability with fast regional transactions while presenting an unrestricted programming model and requiring no special platform support, such as specialized clock hardware or assumptions about maximum clock skew. Our results show comparable strong snapshot read latency to Cloud Spanner [2], while having an order of magnitude faster regional writes.

2 BACKGROUND ON CHARDONNAY

We now provide a brief overview of Chardonnay, with the minimal amount of detail to explain how we built Chablis on top of it. For a more detailed description please refer to the Chardonnay paper [10].

2.1 Components

Chardonnay has three major components, all of which are deployed *in the same datacenter*:

(1) **Local Epoch Service**. Responsible for maintaining and updating a single, monotonically-increasing counter called the **epoch**. The epoch service exposes only one RPC to its clients, which returns the latest epoch. Reading the epoch serves as a serialization point for all committing transactions. The epoch is used to assign transaction timestamps at commit time. The epoch is only read, not incremented, by each transaction.
(2) **KV Service**. The core service that stores the users' key-value data. It uses a replicated shared-nothing range-sharded architecture similar to other modern System R*-style systems [1, 8, 28].
(3) **Transaction State Store**. Responsible for authoritatively storing the transaction coordinator state in a replicated, highly-available manner so that client failures do not cause transaction blocking.

Figure 1 illustrates how the components interact during the lifetime of a transaction. The basic flow of a read-write transaction is almost the same as in a classic shared-nothing System R*-style system, except we add step 3b to read the epoch in parallel to the Prepare phase. Chardonnay uses Two-Phase Locking (2PL) [11] for concurrency control, and Two-Phase Commit (2PC) [17] to ensure atomicity of distributed transactions. Chablis uses the same protocols.

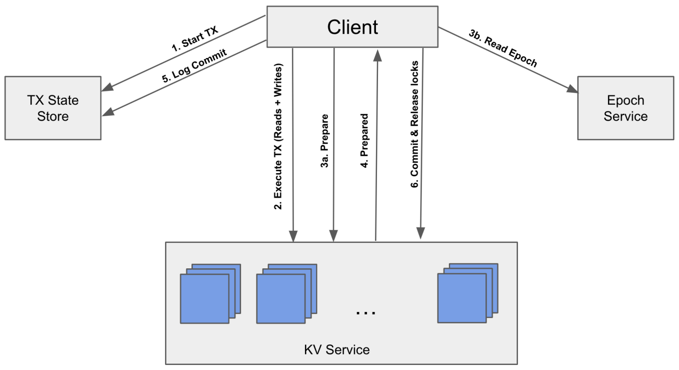

Figure 1: Transaction Lifetime in Chardonnay.

2.2 Local Epoch Service

The epoch service is a Multi-Paxos replicated state machine maintaining a single counter, the *epoch*. One replica is designated leader. It increments the epoch at a fixed configurable time interval (e.g., 5 ms) by appending an entry to the Paxos log so it is durably replicated. It exposes one RPC, *read-epoch*, which returns the value of the epoch. The system maintains the invariant:
Monotonic Epoch Invariant: If a *read-epoch* call returns a value *e*, then all subsequent read-epoch calls must return a value greater than or equal to *e*.

We cannot rely on simply reading the value from the leader replica, since a leader might lose its status without realizing it for a while. Instead, we consider the epoch updated when it is applied to the state of a majority of replicas, not just when it is appended to the log. The client sends read RPCs to all replicas and considers the current epoch value to be the one returned by a majority of the replicas. If no value has a majority, the client retries the read.

Chardonnay uses the high-performance eRPC library [15] for all communications. A single core can support tens of thousands of clients and serve millions of eRPC calls per second [15]. The client library batches multiple read-epoch calls from multiple concurrent transactions into a single RPC. Since each RPC does very little work (reads a word from main memory that is usually cached), so we expect that the epoch service is never a bottleneck in practice. The Chardonnay paper [10] also describes how to scale it up if desired.

2.3 KV Service

The key universe is partitioned into disjoint contiguous subsets called **ranges**. Each range is assigned to a number of range servers (e.g., three) and is comprised of a database and a Write-Ahead Log (WAL), implemented via Paxos. One of the range replicas is designated as a *leader*, which holds a leader lease. It maintains a lock table to implement 2PL, using existing range locking techniques [18, 21]. All reads and writes go through the leader.

2.3.1 *Leader Selection and Disjointedness.*

Each range should have a designated leader replica that holds the leader lease. The leader selection is piggybacked on the Paxos log implementation, i.e., a replica attempting to acquire the leader lease does so by appending a lease acquisition entry to the Paxos log. This log entry includes,

```
class IChardonnay {
public:
    Transaction* start();
    std::string get(Transaction *tx, const std::string &key);
    std::vector<std::string> scan(Transaction *tx,
                        const std::string &lowerBoundInc,
                        const std::string &upperBoundExcl);
    std::string put(Transaction *tx,
                       const std::string &key,
                       const std::string &val);
    void del(Transaction *tx, const std::string &key);
    void abort(Transaction *tx);
    bool commit(Transaction *tx);
}
```

Figure 2: Simplified Chablis Client API

among other information, the identity of the replica that is the lease holder, an ***epoch interval*** entitling the replica to leadership status as long as the epoch (maintained by the epoch service) falls within this interval. When a leader is renewing its lease or a new leader is taking over, they read the epoch from the local epoch service and set the upper interval ahead of the current value (by 100 in our prototype); it is important that the upper end is not too far ahead of the epoch, because this would effectively prevent other replicas from taking over if the leader goes down, until the true epoch catches up.

To prevent two replicas from acquiring leases with overlapping epoch intervals, a lease acquisition entry by a replica includes a copy of the lease believed to be the most recent. Other replicas will reject a replica's attempt to get the lease if they are aware of a more recent lease having been granted. This guarantees that at any point in time there is at most one leader for any range, and that only one range leader can successfully prepare transactions for an epoch. We call this the ***Leader Disjointedness*** invariant. In §4.2 we explain how we use it to validate transaction locks, and later in §5 we describe its role in the correctness of our lock-free snapshot reads in both Chardonnay and Chablis.

3 CHALLENGE

Chardonnay makes each transaction read the (monotonically increasing) epoch after it finished execution, during running the commit protocol. This is a global synchronization point and establishes a global ordering of transactions equivalent to the epoch ordering, i.e. a transaction that reads an epoch value e is ordered before a transaction that reads an epoch value e+1. Chardonnay leverages this property to support lock-free strongly consistent snapshot reads. Given that, one might thank that we can just distribute Chardonnay's KV service geographically across multiple regions, assign each data to a home region, and satisfy the requirement of fast regional writes along with globally strictly serializable lock-free snapshot reads. Unfortunately, this does not work as desired because of the local epoch service. Each committing transaction in Chardonnay reads the local epoch from a majority of the epoch service replicas. Thus, in a geo-distributed setup, at least some of the regions will have to incur the cross-region latency during 2PC in order to read the epoch.

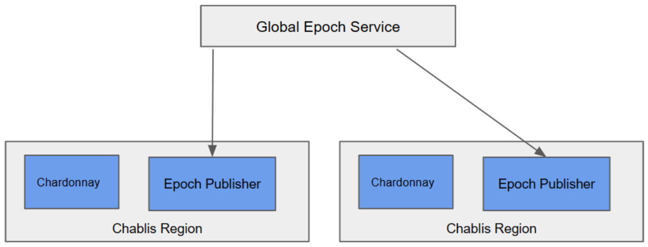

Figure 3: Two region Chablis deployment.

The key contribution of Chablis is addressing this issue, by introducing a new (global) epoch service, which will be described in the next section.

4 CHABLIS OVERVIEW

Chablis has three components, illustrated in Figure 3:

(1) **Client Library**. Applications link this library to access Chablis. It is the 2PC coordinator, and provides APIs (Figure 2) for executing transactions.
(2) **Regional Chardonnay deployments**. One Chardonnay cluster in each region where the system operates.
(3) **Global Epoch Service**. A globally-replicated service that exists in all regions (§4.1). This is similar to the epoch service in each regional Chardonnay, except it maintains one global epoch across all regions.

Each record (i.e., key-value store) is homed to a single region (i.e., Chardonnay cluster), and the client library knows how to determine the home region for each key. The Chardonnay clusters largely operate independently from one another. All writes for a record are sent to its home region. For simplicity, we also send all reads for a key to its home region. A straightforward future extension would be to set up non-voting Paxos replicas [28] in other regions that can serve older snapshots.

4.1 Global Epoch Service

The global epoch service is similar to the local epoch services that exist in each Chardonnay region. It also maintains a single counter (the global epoch), and exposes only one RPC to its clients, *read-global-epoch*. The main differences are that the global epoch typically advances at a slower cadence than the local epochs, and it is not read directly by clients during 2PC. Instead, Chablis introduces intermediary *epoch publishers* between the epoch service and its clients. One epoch publisher exists in every region. Each publisher maintains a single counter (the epoch) and is Paxos replicated for high availability, much like the epoch service itself. However, the publishers do not advance the counter themselves. Instead, when the epoch is advanced by the global epoch service, it issues RPCs to each publisher to advance their epoch value. The epoch service does not advance the epoch again before it updates *all* the publishers (each of which is replicated for high availability). Thus, the state of each publisher replica can only be in one of the following two states: either (a) equal to the global epoch, or (b) one behind the global epoch, if the global epoch service is still in the middle of the process of updating the publishers.

Chablis clients read the global epoch from their region's local publisher, and use the same procedure to read the epoch from that publisher exactly as it would from a local epoch service. This design requires a slightly weaker epoch invariant, since a client might read a global epoch value that is one less than the true epoch. Hence, the system maintains the invariant:
Global Epoch Invariant: If a *read-global-epoch* call returns a value *e*, then all subsequent read-global-epoch calls must return a value greater than or equal to *e*-1.

In §5, we explain how Chablis clients can achieve linearizability of snapshot reads given this weaker invariant.

4.2 Single-Region Read-Write Transactions

Figure 4 illustrates the basic flow of a single read-write transaction that only accesses data within a single region (and therefore a single Chardonnay cluster). The flow is almost the same as in a classic shared-nothing system, except we add steps 3b and 3c to read the local and global epochs in parallel to the Prepare phase. Note that despite having to read the global epoch, the client only needs to read from its local publisher in exactly the same way as it reads the local epoch, so this does not increase the latency of 2PC.

The Chablis client provides an interface for users to access the database, and also acts as the 2PC coordinator in Chablis. After the transaction finishes execution, the client reads the local and global epochs in parallel to issuing Prepare RPCs to participant range leaders. Each leader that accepts the Prepare request responds with a *Prepared* message that includes the local epoch interval of its lease. The client then checks that the local epoch it read falls within the lease's epoch interval of every participant, and if not, aborts the transaction. This is necessary to maintain the leader disjointedness invariant. If all the participants prepare successfully and the lease validations pass, the client then calls the transaction state store to record the transaction's commit durably. The Commit record includes the participant ranges and the value of the epoch. Finally, the client calls the participant range leaders to notify them of the commit so they can record it locally and release all the locks. Transactions in Chablis must wait until the transaction Commit is recorded before releasing any locks (including read locks), for the correctness of snapshot reads (§5).

4.3 Multi-Region Read-Write Transactions

Chablis supports read-write transactions that accesses data in multiple regions, albeit with increased latency and contention due to wide-area round-trips. Such a transaction executes in the same manner as a single-region transaction, except that the client needs to involve KV-service ranges from the different regions in 2PC, and needs to read the local epoch for each involved region in step 3b. As we mentioned earlier, Chablis targets workloads with locality in data access, so we assume that multi-region read-write transactions are used sparingly, perhaps for some small tables that are updated infrequently but are read frequently in all regions.

Transactions reading data from multiple regions but only writing data in a single region have the option of using snapshot isolation

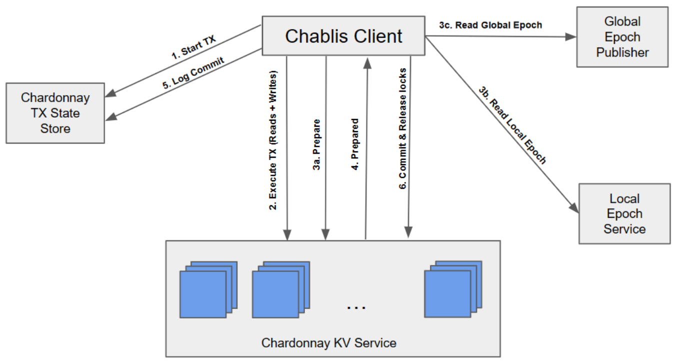

Figure 4: Single-Region Transaction Lifetime in Chablis.

instead, executing their reads using the snapshot read algorithm described in §5.

5 SNAPSHOT READ

This section describes Chablis's multi-versioning and snapshot read protocols. Queries have to be declared as read-only from the start; a transaction that starts normally without this declaration but only performs reads is treated as a read-write transaction by the system.

5.1 Global Record Versioning

Each user record has a key *k* and one or more versions stored in the Chardonnay database. The key for each version is the pair ⟨k, VID⟩, where VID (version ID) is determined as follows. Its prefix is the value of the global epoch of the transaction (see next paragraph on how that is determined). A counter (starting from 1) is appended to the epoch to distinguish writes by different transactions in the same epoch. A transaction chooses a single suffix that makes its VID greater than that of the existing VIDs in its write set. Deletes need to have versions as well, so they appear as tombstones.

In contrast to versioning based on local epoch in Chardonnay, global epoch versioning cannot just rely on the value read in parallel to the Prepare phase (i.e., step 3c in Figure 4), since that value could be one less than the true epoch as we discussed in §4.1. Instead, each range leader also keeps track in its memory of the highest value of the global epoch it has seen, and returns it to the client with its Prepare reply. The client sets the transaction's epoch to be the maximum of the value from step 3c and the values returned by the range leaders. Leaders do not need to persist the highest global epoch value they have seen; before a new leader takes over it just waits for the global epoch to advance once and resets to that value. Clients also keep track of the highest epoch they have seen, to ensure the epoch remains strictly monotonically increasing within a single session.

5.2 Multi-Region Read Algorithm

Global Epoch Ordering Property: The transaction ordering enforced by Chablis's strict 2PL is equivalent to one in which for all pairs of committed transactions, T_1 with a global epoch e_1, and T_2 with a global epoch e_2, if $e_1 < e_2$, then T_1 precedes T_2.

Proof Sketch: We show that if $e_1 < e_2$, then T_1 cannot have a dependency or anti-dependency on T_2. Given that, we can show that the transaction dependency DAG has no edges that go from a transaction with a higher to a lower epoch.

We proceed by contradiction by assuming this is false. This implies that there is (transitively) a read-write or write-write conflict between T_1 and T_2, and T_2 was ordered first. Therefore, T_2 released a lock L and sometime later T_1 acquired L. Transactions in Chablis do not release any locks until they commit. At the time when T_2 released L, the range leader that granted L must have recorded the value of the global epoch to e_2, as we discussed in §5.1. Thus, the value of the epoch at the time of T_1 commit has to be greater than or equal to e_2. But $e_1 < e_2$, a contradiction. □

The global epoch ordering property ensures that global epoch boundaries are consistent points in the serial order and appropriate for serializable snapshot reads. That is, a transaction can get a consistent snapshot as of the beginning of the current epoch e_c by ensuring it observes the effects of all committed transactions that have a lower epoch. Suppose all the transactions with an epoch $e < e_c$ have committed. Reading a user key k as of the start of epoch e_c translates to reading the value of key $\langle$k, VID$\rangle$ such that VID is the largest value less than $\langle e_c, 0\rangle$ in the database. Hence, the snapshot read algorithm would simply work by reading the epoch e_c, then reading the appropriate key versions.

```
Procedure StartTX():
    $e_c \leftarrow read\_global\_epoch()$;
    wait for global epoch to advance once;
    return $e_c$;
// ReadKey Executes on the leader of the range
// containing k
Procedure ReadKey($k$, $e_c$, $l_R$):
    if $l_R$ is nil then
        $l_R \leftarrow read\_local\_epoch()$;
    end
    if $l_R$ above leader lease epoch interval then
        abort;
    end
    wait for write lock on $k$ to be released;
    $v \leftarrow read\_version(k, e_c)$;
    return $v$;
```

Algorithm 1: Multi-Region Snapshot Read Procedure

The main challenge is ensuring that the snapshot is complete, i.e., no more transactions will be committing with an epoch below e_c. By waiting for the global epoch at its epoch publisher to advance once[2] (i.e., become $e_c + 1$) before starting the read, we can guarantee that any transaction that has not started to prepare will have an epoch of at least e_c, by the global epoch invariant.

The problem is prepared (or preparing) transactions that are not yet known to have committed after establishing the snapshot's global epoch e_c. Fortunately, any such transaction that could possibly commit writes must *already* be holding write locks at their respective current range leaders. More formally, any transaction T with a global epoch below e_c that could commit a write to a key k that is homed in region R must be holding a write-lock on k at the regional Chardonnay range leader in R (proof in the Chardonnay paper [10]). Suppose the current local epoch in R is l_R. It must be that locks are held on the leader whose lease's epoch range contains l_R (and by the leader disjointedness invariant, there can be at most one such replica), and any subsequent leader replica. Thus, the read algorithm reads the current local epoch l_R (once per transaction) for all regions R it accesses after reading the global epoch value e_c. It ensures that l_R is below the upper end of the leader's epoch interval, and *waits* for the current holders of write locks (if any) on its read set to release these locks before executing the reads. The read is not attempting to *acquire* locks, so it does not contend with read-write transactions. Algorithm 1 shows the read procedure.

5.3 Multi-Region Snapshot Read Consistency

Algorithm 1 as described so far guarantees serializability, but the snapshot could be stale because a transaction T would not observe the effects of transactions in epoch e_c that committed before T started. If desired, ensuring strong consistency (i.e., observing the effects of all transactions that committed before T started) is easy at the cost of some latency: After T starts, it reads the current value of the global epoch e_c directly from the global epoch service instead of the local publisher. Then, it waits for the epoch at the local publisher of each region it is reading from to become e_c+1, and then executes the read as of the start of e_c+1. Note that the step of waiting for the epoch to advance overlaps the RPCs to the remote regions. With this modification, the snapshot reads provide *single-key linearizability* [28], which is weaker than strict serializability. This is because, given two transactions T_1 and T_2 running on different clients and with non-overlapping key sets, it is possible that T_1 commits before T_2 starts but T_1 gets an epoch e_c+1 while T_2 gets an epoch e_c, due to the weakened global epoch invariant. In such a case, the snapshot would include T_2 but not T_1, violating linearizability. This is unlikely to be an issue in practice, but we discuss how to handle this in §5.3.1.

If the system is serving a large number of multi-region strongly consistent snapshot reads, it might be desirable to avoid frequent cross-region RPCs. This is possible at the cost of additional latency: after T starts, it can read the epoch from its local publisher, then wait for the epoch to advance twice, and then use the new epoch in Algorithm 1.

5.3.1 *Linearizable Snapshots.*

When running a snapshot as of the start of epoch e_c+1, linearizability can be violated only if the snapshot's read set includes some members whose snapshot version is exactly at the epoch e_c, and other members that were updated in epoch e_c+1. Since reads execute at the range leaders, we always know the latest version of a record (or range) at the time of the read. To ensure linearizability, the algorithm performs the following additional check: If the read set includes some members whose snapshot version is e_c, and other members whose latest version is equal to or greater than e_c+1, it aborts and retries by advancing its epoch.

This should work well under low contention, but there is no termination guarantee if the readset keeps getting updated frequently

[2]Alternatively, the read could be executed at e_c-1 instead of waiting.

by newer transactions. One option is falling back to executing as a locking transaction. But that would require locking the readset while executing high latency multi-region reads under high contention, which contradicts Chablis' goals.

Note that if there are no transactions in an epoch e_c, a linearizable snapshot read as of the beginning of epoch e_c+1 is guaranteed to succeed. Hence, the system can be configured to avoid committing transactions in epochs that are a multiple of a configurable value k, by waiting for the epoch to advance to the next epoch before releasing locks. The value of k controls the tradeoff between how often read-write transactions need to wait during commit vs. the upper bound on how many times a snapshot read needs to retry. This is analogous to Spanner's commit algorithm that waits out the TrueTime uncertainty interval, except it is amortized across many epochs instead of done for every read-write transaction.

5.4 Single-Region Snapshots

Chablis can be configured such that single-region snapshot reads exclusively use versions based on the local epoch and utilize Chardonnay's snapshot reads which are described in Section 6 of [10]. This significantly reduces the latency of single-region snapshot reads, at the expense of storing each version twice. In such a configuration, a snapshot read is assumed to be single-region by default (unless declared otherwise), until it tries to access data in a remote region in which case it will be restarted as a multi-region query.

5.5 Handling Regional Failure

Chablis is a CP system in the CAP theorem [12] sense. Modern datacenter networks make arbitrary partitions exceedingly rare [5] so in practice the system also achieves high availability. However, dealing with an entire region going down is one of the main reasons why users use geo-distribution. The global epoch service requires updating the epoch in all regional publishers before advancing again. If a region is down, this would block updating the global epoch. While read-write transactions and regional snapshot reads can proceed, the global snapshot would get very stale. The local publishers will stop serving the stale epoch if they do not get an update from the global epoch service, so transactions will stop committing in the failed region (in case machines are active but cut from the rest of the world). In such a situation, an operator can configure the global epoch service to exclude the failed region from its set of publishers so that the rest of the system can continue operating until the region recovers, in which case in can be added again to the set of publishers.

6 EVALUATION

We evaluate Chablis' ability to preserve low-latency for single-region transactions, while performing global strongly-consistent snapshot reads.

6.1 Setup

We run our experiments on Microsoft Azure. Our experimental setup has two Chardonnay deployments, one in the *us-east1* region and one in the *us-west1* region. Each of these regions has a global epoch service publisher. The global epoch service itself is deployed in the *us-central1* region so that its maximum latency to both publishers is minimized.

Within each region, the Chardonnay KV service shard leaders use Standard_L8s_v2 Azure VMs, which provide 8 vCPU cores and 64GB of memory and support accelerated networking necessary for eRPC. We advance the local epochs every 10ms. There are two shards in each region. The entire database fits in the DRAM cache. We also disable Chardonnay's dry runs as they are not relevant for our experiment.

The global epoch does not advance on a fixed timer interval. Instead, the epoch service continuously advances the epoch as fast as it can by issuing parallel calls to the epoch publishers in both regions, waiting until they complete, then advancing again in a loop. Hence, the global epoch interval is determined by the network latency to the furthest region.

	Read	Write
P50 (μs)	214	199

Table 1: YCSB Regional Latency Results.

6.2 Experiment

We run YCSB-A [7] with 50% point reads and 50% point writes with uniform random distribution. Each Chardonnay region has one client with 5 threads, which runs on a dedicated VM. Each client issues YCSB transactions only to the shards co-located with it; hence the YCSB transactions are all single-region. The YCSB reads are not declared as snapshot queries so they execute using locks, not using the snapshot algorithm. Additionally, the client in the *us-west1* region periodically executes a multi-region strongly consistent snapshot read query that reads one key in each region. We chose *us-west1* because it has a higher latency to the *us-central1* region, so it presents the worst case. We run the experiment for 10 minutes.

6.2.1 Results. First, we measure the global epoch update intervals. The median is ∼47ms, and the P99 is ∼76ms. The mean update duration for the epoch is under 55ms. Second, we measure the latency of multi-region snapshots. On average the query has to wait ∼82ms in the initial stage reading the global epoch and waiting for the epoch to advance, and in total takes an average of ∼107ms. Finally, we measure the latency of the YCSB transactions. The results are shown in Table 1

6.2.2 Comparison to Cloud Spanner. Cloud Spanner [2] is a managed service offered by Google Cloud Platform (GCP). GCP does not have the same regional layout as Azure, so we use the nam3 configuration as it is the closest to ours. It has replicas in us-east1, us-east4, and us-central1 regions, a read only replica in us-west1, and the primary replica in us-east4. We find that strong (i.e., linearizable) reads from the the central region have a median latency of ∼39ms, while strong reads from us-west1 have a median latency of ∼65ms. Thus, Cloud Spanner strong read latency is lower than, but comparable to, Chablis (which does not require special hardware synchronized clocks). On the other hand, all writes in Cloud Spanner, even for a single-region deployment, have latency of many milliseconds, which is an order of magnitude slower than Chablis (see Table 1). The comparison is not strictly apples-to-apples, since Cloud Spanner is not using fast RPCs and log storage unlike Chablis. However, we believe it would still be a lot slower in that case due to having to wait out clock uncertainty during commit.

7 CONCLUSIONS

We presented Chablis, a scale-out, geo-distributed, transactional, general purpose key-value store. Chablis takes advantage of fast datacenter RPCs and epoch-based versioning to support global strictly serializable lock-free snapshot reads without impacting single-region transactions, relying on specialized clock hardware or making assumptions about maximum clock skew. This makes Chablis the first geo-distributed transactional system that is both fast and general, showing that the compromise between speed and generality is not necessary in modern datacenters with fast RPCs.

8 ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their feedback and suggestions. This work was supported in part by NSF awards CNS 2106530, CNS 2104292 and CNS 2143868, and gifts from Google, DiDi, and Accenture.

REFERENCES

[1] [n.d.]. yugabyteDB. https://yugabyte.com/.

[2] 2023. Cloud Spanner | Google Cloud. https://cloud.google.com/spanner.

[3] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. HAT, Not CAP: Towards Highly Available Transactions. In *HotOS XIV*. https://www.usenix.org/conference/hotos13/session/bailis

[4] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*.

[5] Eric Brewer. 2017. *Spanner, TrueTime and the CAP Theorem.* Technical Report.

[6] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (aug 2008), 1277–1288. https://doi.org/10.14778/1454159.1454167

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB *(SoCC '10)*. 143–154. https://doi.org/10.1145/1807128.1807152

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (aug 2013), 22 pages. https://doi.org/10.1145/2491245

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store *(SOSP '07)*. 205–220. https://doi.org/10.1145/1294261.1294281

[10] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. 2023. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/osdi23/presentation/eldeeb

[11] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (nov 1976), 624–633. https://doi.org/10.1145/360363.360369

[12] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33, 2 (jun 2002), 51–59. https://doi.org/10.1145/564585.564601

[13] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The Dangers of Replication and a Solution *(SIGMOD '96)*. 173–182. https://doi.org/10.1145/233269.233330

[14] Pat Helland. 2007. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR*.

[15] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[16] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[17] Butler W. Lampson and David B. Lomet. 1993. A New Presumed Commit Optimization for Two Phase Commit. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 630–640.

[18] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. Multi-Version Range Concurrency Control in Deuteronomy. *Proc. VLDB Endow.* 8, 13 (sep 2015), 2146–2157. https://doi.org/10.14778/2831360.2831368

[19] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[20] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI 13*. USENIX Association, Lombard, IL, 313–328. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd

[21] David B. Lomet and Mohamed F. Mokbel. 2009. Locking Key Ranges with Unbundled Transaction Services. *Proc. VLDB Endow.* 2 (2009), 265–276.

[22] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades *(NSDI'17)*. USENIX Association, USA, 453–468.

[23] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts *(OSDI'16)*. USENIX Association, USA, 517–532.

[24] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1279–1294. https://doi.org/10.1145/2723372.2723729

[25] Cuong D. T. Nguyen, Johann K. Miller, and Daniel J. Abadi. 2023. Detock: High Performance Multi-Region Transactions at Scale. *Proc. ACM Manag. Data* 1, 2, Article 148 (jun 2023), 27 pages. https://doi.org/10.1145/3589293

[26] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647

[27] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 433–448. https://doi.org/10.1145/3299869.3300069

[28] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database *(SIGMOD '20)*. 1493–1509. https://doi.org/10.1145/3318464.3386134

[29] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications *(SIGMOD '17)*. 5–20. https://doi.org/10.1145/3035918.3064037

[30] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 231–243. https://doi.org/10.1145/3183713.3196912

[31] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404

[32] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems *(SOSP '13)*. 276–291. https://doi.org/10.1145/2517349.2522729