Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases

Tamer Eldeeb and Xincheng Xie, *Columbia University;* Philip A. Bernstein, *Microsoft Research;* Asaf Cidon and Junfeng Yang, *Columbia University*

https://www.usenix.org/conference/osdi23/presentation/eldeeb

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases

Tamer Eldeeb
Columbia University

Xincheng Xie
Columbia University

Philip A. Bernstein
Microsoft Research

Asaf Cidon
Columbia University

Junfeng Yang
Columbia University

Abstract

Distributed on-disk database systems could either use an expensive commit protocol like two-phase commit (2PC) to guarantee atomicity, and suffer from *slow* distributed transactions, or forgo 2PC, which lead to weaker semantics, limitations to the programming model, or constrained scalability, making the system less *general*. We argue this compromise is no longer necessary within modern datacenters. Low latency 2PC (∼150 µs on Azure for 2PC over Paxos) can be achieved using low-latency storage for the relatively small transaction logs, fast RPCs, and careful protocol design. With fast 2PC, the data contention bottleneck for many transactions shifts from 2PC to reading the data itself from the relatively slow storage while holding transaction locks.

We present Chardonnay, a scalable, on-disk, multi-versioned transactional key-value store optimized for single datacenter deployments with fast 2PC. Chardonnay has a *general* interface supporting point reads, scans, and writes within multi-step strictly serializable ACID transactions. The key mechanism underlying Chardonnay's design is strongly consistent snapshot reads on commodity hardware, using a novel lock-free read protocol. Chardonnay uses this protocol to cheaply determine the read-write sets of queries, enabling Chardonnay to transparently prefetch data needed for a transaction prior to the execution of the transaction and the acquisition of locks. This enables Chardonnay to achieve *fast* transactions by minimizing contention, and avoids aborts due to deadlocks by ordering lock requests.

1 Introduction

The holy grail of distributed databases is to provide an abstraction of a single-server database that can run SQL ACID transactions at high performance while maintaining high availability. Recent work [27, 33, 47, 52, 57, 69, 81, 82, 84] shows that ACID distributed transactions with strong isolation and consistency semantics can be made efficient and scalable within in-memory database systems. However, keeping all data in memory can be prohibitively expensive, especially for large applications, as DRAM's cost per GB is over 10–50× more expensive than regular (e.g., TLC or QLC) NAND SSD [34].

Therefore, due to their significantly lower cost, many applications use distributed databases [11, 12, 29, 74], which store their data on disk-based storage engines such as RocksDB [8, 32, 58] or LevelDB [7]. The classic architecture for such systems [65], popularized by System R* [59], is to shard the data horizontally across a collection of shared-nothing machines, and use a distributed commit protocol such as two-phase commit (2PC) [48] to ensure atomicity of distributed ACID transactions. Unfortunately, distributed transactions in these systems have significant performance limitations [17, 30, 42, 52, 57, 76, 81].

Due to these challenges, many scale-out on-disk systems avoid providing any multi-key ACID transaction support at all [26, 31], or limit it to local transactions accessing keys within a single machine or partition [23, 61]. Other systems offer support for distributed transactions, but forgo 2PC and sacrifice *generality* in one or more ways, e.g., by offering weaker semantics [16, 54, 78, 79], restricting the programming model [76], or employing an architecture that limits system scalability [13, 49, 87]. Nevertheless, due to strong developer demand [14], many popular SQL DBMSes now support general distributed ACID transactions [11, 29, 74], despite being a lot slower than local transactions. Table 1 shows the trade-offs made by various popular on-disk systems.

We argue that this compromise between performance and generality is no longer necessary within the modern datacenter. The high performance penalty of 2PC historically has been due to the high latency of RPCs and flushing log entries to disk. Fortunately, neither is the case any more. Modern datacenter networks are fast [22], and systems such as eRPC [46] have demonstrated that

System	Serializabile	Linearizable	General API	Distributed TX	High Contention
Spanner [29]	✓	✓	✓	Slow	X
Calvin [76]	✓	✓	X	Fast	✓
FoundationDB [87]	✓	✓	✓	Fast	X
Hyder [21]	✓	X	✓	N/A	X
Aurora (Multi-Master) [78]	X	X	✓	N/A	Partitionable Workloads
Chardonnay	✓	✓	✓	Fast	✓

Table 1: Comparison of representative on-disk distributed database systems.

RPCs can run at single-digit µs latency within the datacenter even without using RDMA. Additionally, storage devices based on low-latency SLC NAND [10] or 3DX-point [1] also provide single-digit µs latencies [6, 9, 10], making them ideal for persisting database logs.[1] Furthermore, many recent frameworks [45, 66, 83, 85, 86] fully or partially bypass the Linux I/O software stack, further boosting I/O performance.

This leads us to revisit the assumption that 2PC is the primary bottleneck inherent in scale-out on-disk database system designs. However, using a fast 2PC protocol reveals new bottlenecks. As we show in §4, even *eliminating the entire latency of the commit protocol is not sufficient* to achieve good performance for high-contention workloads, because transactions frequently hold locks while fetching cold items from storage. Therefore, the data contention bottleneck shifts to reading the data from disk, since reading data from a typical SSD can be orders of magnitude slower than the network.

We present Chardonnay, a distributed multi-version transactional key-value store that is deliberately tailored for this new era of fast 2PC. Chardonnay is designed for single-datacenter deployments, since cross-datacenter 2PC latency would be high. It supports point and range reads, as well as writes, within classical multi-step strictly serializable ACID transactions, making it suitable as the storage engine for a SQL database (e.g., similar to CockroachDB [74]). Chardonnay uses the classic shared-nothing architecture[2] and uses strict two-phase locking (2PL) [37] to guarantee strict serializability [43] for read-write transactions, as well as 2PC to ensure atomicity for distributed transactions.

The core insight of Chardonnay is that fast RPCs enable strictly serializable lock-free snapshot queries within the datacenter in a *general* fashion, i.e., without using specialized clocks, limiting scalability, or weakening the performance and semantics of read-write transactions. Low-latency, high-throughput RPCs are key to allow all committing transactions in Chardonnay to cheaply read a counter, called the *epoch*, that serves as a global serialization point. The system increments the epoch periodically, independent of transactions, so unlike designs with a centralized sequencer [18, 87], maintaining the epoch can be distributed and highly scalable. The main challenge is that unlike systems with a single global log or coordinator, Chardonnay uses one log per partition, so it cannot enforce global epoch ordering of commits. Instead, we co-design the snapshot read and commit protocols to guarantee their equivalence to epoch ordering (§6). The idea is rather simple: Snapshot queries may block waiting for write locks to be released (once) for correctness, but they do not acquire any locks, so they do not contend with the read-write transactions.

Beyond the direct benefit of efficient, lock-free read-only queries, this enables two important benefits, as Chardonnay leverages this snapshot read protocol to optimize the execution of read-write transactions. First, Chardonnay runs the user's transaction in a *dry run* mode using the snapshot protocol to (approximately) compute and prefetch the transaction's read set, which in the vast majority of cases allows Chardonnay to shift the work of reading cold data from storage outside of the contention period of the transaction. Second, since read and write sets can be efficiently computed using the snapshot protocol, Chardonnay also uses them to plan the locking scheduling in a manner that avoids deadlock aborts.

At the systems and design level, our main contribution is Chardonnay, the first (to our knowledge) on-disk system that achieves high performance for both low and high-contention workloads, without sacrificing strong semantics, restricting the programming model, or limiting scalability. The novel mechanisms introduced in Chardonnay are:

1. **Novel lock-free snaphsot read protocol**: Chardonnay uses fast RPCs to guarantee strict serializability without relying on specialized hardware, synchronized clocks, making assumptions about clock skew, or limiting scalability.
2. **Automatic prefetching**: Chardonnay leverages the snapshot protocol to do a "dry run" of the query, which loads and pins all the keys accessed by the

[1] It is of course possible to store the entire database on such devices, but they cost significantly more than commodity SSDs.

[2] Which, we posit, has aged like fine wine.

transaction to main memory. This allows Chardonnay to avoid waiting for data read from slow storage while holding locks. Unlike similar schemes introduced by prior work [3, 75, 76], Chardonnay's prefetching mechanism works for scans, and neither requires changes to the user code, nor incurs significant additional latency or contention.

3. **Lightweight deadlock avoidance**: By computing read and write sets in advance, Chardonnay avoids deadlocks by determining the lock acquisition order.

Collectively, these techniques enable Chardonnay to have excellent performance under high contention. Indeed, as we show in §9, Chardonnay's throughput under extremely high contention is only 15% lower than under extremely low contention. In contrast, the throughput of a baseline System R*-style system (even utilizing fast 2PC) drops by over 85%. The dry run phase adds overhead which is largely wasteful for low contention workloads, but we consider this a worthwhile trade-off, and we allow disabling dry runs on a per transaction basis.

A general takeaway is that within on-disk systems, the availability of fast datacenter RPCs makes distributed and multi-core system designs look increasingly similar. Some of our ideas (epoch-based versioning) are inspired by multi-core database systems [77]. This unlocks the potential for adopting additional insights from multi-core single-node systems in a distributed setting. The flow of ideas can also go in the other direction: while distributed transactions were our primary motivation when designing Chardonnay, the challenge of high contention is not unique to distributed transactions, and in fact many single-node database systems run with low isolation precisely to mitigate this issue [15]. Our results show that Chardonnay's techniques can be useful for them too.

2 Background

This section discusses transaction semantics and commit protocol performance in distributed database systems.

2.1 Strict Serializability

Strict Serializability [43] (also known as External Consistency [29]) is considered the gold standard of distributed transaction semantics. It is the combination of the following two properties [69]:

- **Serializability**: every execution is equivalent to some serial ordering of committed transactions.
- **Linearizability**: if transaction A commits before transaction B starts, then A should precede B in the equivalent serial ordering.

2.2 2PC Recap

Two-phase commit (2PC) is a classic commit protocol with many variants [48]. The basic flow works as follows: after a transaction finishes execution on multiple *participant* servers or shards, a *coordinator* starts the first phase by issuing *Prepare* RPCs to all participant. Each participant can vote yes or no in response to the RPC, where a yes vote is a promise by the participant that it will not unilaterally abort the transaction and will be able to (eventually) commit the transaction when asked. Before voting yes to a Prepare RPC, the participant typically persists all of the transactions writes to a durable log so it can recover from any failures. If any participant votes no (or never responds due to failures or timeouts), the coordinator aborts the transaction. Otherwise, it logs the decision to commit to durable storage and then runs the second phase of the protocol by issuing *Commit* RPCs to the participants so they can apply the transaction and release locks. A well known problem of 2PC is that it is *blocking* [20, 70], wherein the failure of the coordinator at inopportune moments prevents the participants from making progress. This can be addressed by replicating the coordinator state for availability [17, 29, 40].

2.3 The Penalty of 2PC

2PC traditionally incurs a significant performance overhead for two main reasons. First, it requires at least two network round trips and two synchronous log writes to persistent storage per transaction [41, 57], which incurs network and storage I/O overhead, as well as CPU usage by the TCP/IP stack [81]. For example, typical 2PC commit latency within a single datacenter in systems like Spanner is in the double digit milliseconds [29], which puts a hard upper bound of less than 100 TPS on transactions that update a write-hot record. Second, the coordination necessary to guarantee isolation can significantly decrease concurrency, leading to performance degradation, as well as high abort rates [15]. This increased contention due to 2PC is particularly harmful for short transactions common in OLTP workloads, due to the high latency of the commit protocol relative to the time it takes to execute the transaction logic [76]. The impact of contention is evident in locking-based concurrency control schemes such as 2PL, but optimistic concurrency control (OCC) schemes are also not immune, and can in fact perform worse under high contention [41, 51, 82].

3 Requirements

We now define Chardonnay's stated objective, *fast* and *general* transactions for on-disk databases, in more detail. Fast encompasses the following requirements: First,

latency for short OLTP transactions should be low (hundreds of µs) regardless of whether it is single partition or cross-partition; hence the performance penalty of distributed transactions should be relatively small. Second, the system needs to support long-running read-only queries efficiently, without impacting OLTP read-write transactions. Finally, the system should be able to maintain high throughput for both low and high contention workloads. General means providing a general, unrestricted programming model and API (e.g. capable of supporting a full SQL layer) and the highest level of semantics (i.e. strict serializability) without imposing overall scalability limits or using specialized hardware.

4 Measuring Contention Footprint

Data contention is a major issue for traditional on-disk shared-nothing distributed database designs. Most real-world workloads have low contention most of the time, but occasionally a small number of extremely hot data items appear, significantly degrading overall throughput [39, 76]. Other workloads are characterised by high skew such that a small portion of the database receives a majority of the load. For example, half of the NYSE trades happen on 1% of the symbols, and breaking news can cause a sharp spike in trades on a small group of symbols [68]. Indeed, data contention is a bottleneck that hinders truly scalable transaction processing, even in RDMA-enabled in-memory distributed database systems [82], and on multi-core single-node systems [62].

Following the terminology of Calvin [76], we define a transaction's ***contention footprint*** as the total duration from the instant the transaction acquires its first lock until it releases its last lock. In this section we use YCSB [28] to study the contention footprint of simple, single operation transactions in System R*-style systems. To this end, we built two simple baseline systems based on the System R* architecture on top of RocksDB, using its transaction and 2PC support in our experiments:

- **Baseline-Slow**. The client invokes database functions using (slow) gRPC [5]. Both the write-ahead log (WAL) and the database are placed on a directly attached SSD.
- **Baseline-Fast**. Uses (fast) eRPC (with FlatBuffers [4] for serialization format) instead of gRPC, and the WAL is put on an emulated fast NVMe device.

Our baseline implementations ignore crucial practical considerations (such as replicating coordinator state for high availability to deal with the well-known 2PC blocking problem), and transactions more complicated than a single read or write. Therefore, our results underestimate the contention footprint. Nevertheless, they are instructive. All our experiments run on Microsoft Azure VMs. The entire key universe is assigned to a single shard. We run YCSB-A with 50% point reads and 50% point writes with uniform random distribution. All experiments use one client with 5 threads, which runs on a dedicated VM in the same Azure region as the server. To control the amount of DRAM used by the system, we disable the OS page cache and vary the size of the block cache, which is RocksDB's read cache. We run a full 2PC at the end of each transaction, including in the case of reads, to measure transaction overhead, even though technically 2PC is not needed since there is only one shard. Read transactions release locks during the Prepare phase, so the Commit phase does not contribute to their contention footprint. For durability, Calls to Prepare and Commit always wait for the write to be flushed to storage.

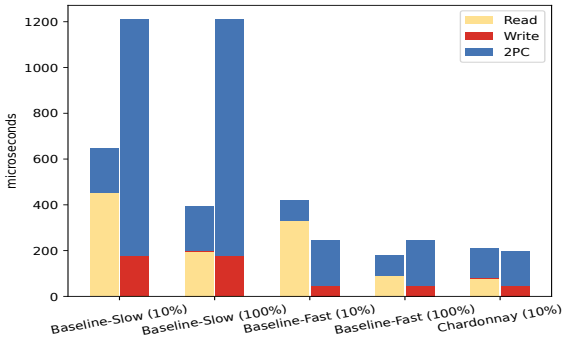


Figure 1: Contention footprint of YCSB read (left bar) and write (right bar) transactions. % represent the proportion of the data in DRAM. Chardonnay achieves a similar contention footprint to fully in-memory ("Baseline-Fast 100%") with only 10% of its data in DRAM.

We show how the average latency of read and write operations each contribute to the contention footprint in Figure 1. On Baseline-Slow, the bulk of the contention footprint comes from running 2PC. On Baseline-Fast, the latency of 2PC is significantly lower due to the fast RPC library and fast log storage. The yellow bars show that the contention footprint of read transactions is much higher when only 10% of the dataset is in main-memory, since the majority of reads have to fetch data from SSD storage. Write transactions (red bars) are not much affected by the available DRAM, since writes are buffered in-memory (at the server) until the Prepare phase where they get written to the WAL.

We deduce two takeaways from this simple experiment. First, with a modern RPC library, fast intra-datacenter network, and small amount of fast NVMe storage, distributed databases can significantly reduce 2PC latency. Second, once the latency of 2PC is reduced, the data contention bottleneck becomes reading the data needed by the transaction from the relatively slow SSD.

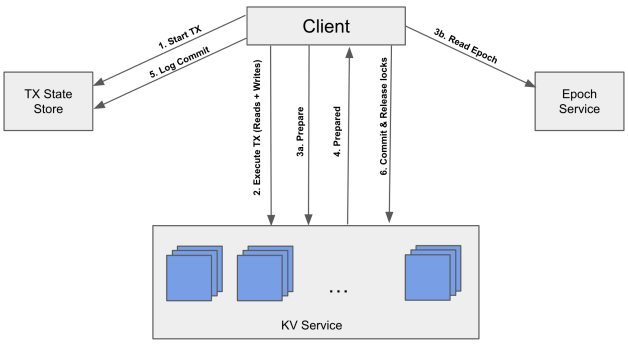


Figure 2: Transaction Lifetime in Chardonnay.

5 Architecture

Chardonnay has four main components:

1. **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 global serialization point for all committing transactions. The epoch is used to assign transaction timestamps at commit time and is essential for our lock-free strongly consistent snapshot reads (§6). The epoch is only read, not incremented, by each transaction.
2. **KV Service**. The core service that stores the user key-value data. It uses a replicated shared-nothing range-sharded architecture similar to other modern System R*-style systems [11, 29, 74].
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. We chose to store the transaction state separately from the user's key-value data to enable 2PC latency optimizations, which we describe in appendix A.1.
4. **Client Library**. Applications link this library to access Chardonnay. It is the 2PC coordinator, and provides APIs (Figure 3) for executing transactions.

Figure 2 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.

5.1 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., 10 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. It is possible to run the client RPCs through the Paxos state machine. However, since each committing transaction reads the epoch, this would be too costly. 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.

There is a trade-off in choosing the epoch advancing interval. It needs to be long enough compared to typical transaction duration that the value is usually read from the CPU caches of replicas, and without requiring retries due to no value having a majority. On the other hand, if it is too long, it adds to linearizable snapshot read-only transaction latency, as we explain in §6. We find that advancing the epoch once every 10 milliseconds works well in our experiments.

A single core can support tens of thousands of clients and serve up to millions of eRPC calls per second [46]. Furthermore, 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), we expect this design to be sufficient for all practical purposes. Nonetheless, in the interest of generality we show how to scale-out the epoch service in appendix A.3.

5.2 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 WAL that is implemented via Paxos. The WAL is placed on a fast NVMe device for low latency, while the database is stored on commodity SSD storage. One of the range replicas is designated as a *leader*, which holds a leader lease. It maintains a lock table to implement two-phase locking, using existing range locking techniques [50, 55]. All reads and writes go through the leader.

To simplify the description in this paper we will assume the ranges and replica-to-server assignments are static, although in practice ranges need to be moved, split and merged to balance load effectively. This can be accomplished using well-known techniques [23, 26, 29, 74], which we leave for future work.

5.2.1 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, 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, and ***leader sequence number***, which is incremented whenever a new replica becomes the leader (but not when an existing leader renews its lease). The leader returns the sequence number to the client on every request, so the client can detect leadership changes and abort the transaction if the transaction observes two different leaders for the same range. When a leader is renewing its lease or a new leader is taking over, they read the epoch from the 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 §5.4 we explain how we use it to validate transaction locks, and later in §6 we describe its role in the correctness of our lock-free snapshot reads.

5.3 Transaction State Store

The transaction state store is responsible for storing the state of active transactions in the system in a fault-tolerant, replicated manner, to mitigate 2PC blocking.

Each transaction can be in one of the following states: *Started*, *Committed*, *Aborted*, and *Done*. Note that being *Prepared* is not of concern here. We use the well-known presumed abort optimization [59], meaning that the service replies *Aborted* to a participant's inquiry about the state of a transaction unknown to the service. Being in *Done* state means that all transaction participant *ranges* have learned about the commit outcome of the transaction so that the service can safely forget about it.

The service is hash-partitioned by transaction id. Each partition is assigned to (typically) three servers. We do not need a per partition log to order transactions, since transactions are already ordered by 2PL. Instead, within a partition, each transaction state is represented as its own Multi-Paxos replicated log, which can have at most 3 entries. Position 0 always contains the Started entry, position 1 can either contain Committed or Aborted, and position 2 is to record Done state. This unusual design is key to a 2PC latency optimization that we describe in appendix A.1.

```
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 3: Simplified Chardonnay Client API

Recall that the client in Chardonnay acts as the 2PC coordinator. If the client crashes after starting the *Prepare* phase and before completing the transaction, the participant ranges need to determine whether to commit or abort. A KV Service range leader will attempt to put an Abort entry in the transaction state log (in position 1). If it succeeds, it can safely abort the transaction. The transaction state store is the source of truth regarding a transaction outcome. If the KV range leader successfully installs an abort decision for the transaction with the TX state store, a slow client cannot then succeed in committing it at a later point. Alternatively, after running the Paxos state machine, the KV range could learn that the client already put a Commit entry in that log position, in which case it can safely apply the transaction.

5.4 Client

The client provides an interface for users to access the database, and also acts as the 2PC coordinator in Chardonnay. After the transaction finishes execution, the client reads the epoch from the epoch service 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 epoch interval on its lease. The client then checks that the 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 Chardonnay must wait until the transaction Commit is recorded before releasing any locks, for the correctness of snapshot reads (§6). This implies that even read locks for successfully prepared transactions have to survive leader changes and thus must be logged in the WAL during the Prepare phase.

Many, if not most transactions only touch keys within a single range, so they do not need 2PC. First, the client reads the epoch. Then, it sends a Commit message to the leader, which checks that the epoch falls within the lease's epoch interval. If so, the leader appends to the WAL and if successful, returns success. If not, it aborts.

6 Snapshots

This section describes Chardonnay's multi-versioning and snapshot read protocols. Snapshot reads are essential to efficiently support read-only queries. They also underpin the techniques described in subsequent sections. 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, and does not utilize the lock-free snapshot read algorithm.

6.1 Versioning

Each user record has a key k and one or more versions stored in the database. The key for each version is the pair $\langle$k, VID$\rangle$, where VID (version ID) is determined as follows. Its prefix is the value of the epoch that the client reads in parallel to running the Prepare phase of 2PC. 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. For convenience, the system also stores an unversioned record with just the key k which holds the latest value and is updated in place.

6.2 Read Algorithm

Epoch Ordering Property: There exists an equivalent ordering to the transaction ordering enforced by Chardonnay's strict 2PL such that for all pairs of committed transactions, T_1 with an epoch e_1, and T_2 with an epoch e_2, if $e_1 < e_2$, then T_1 precedes T_2.

We present a proof sketch of this property in appendix A.2. The epoch ordering property ensures that epoch boundaries are consistent points in the serial order and appropriate for serializable snapshot reads, i.e., 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 $< \langle e_c, 0\rangle$ in the database. Hence, the snapshot read algorithm would simply work by reading the epoch e_c, then reading the appropriate key versions.

The main challenge is ensuring that the snapshot is complete, i.e., no more transactions will be committing with an epoch below e_c. Any transaction that has not started to prepare is guaranteed to have an epoch of at least e_c, by the monotonic epoch invariant.

The problem is prepared (or preparing) transactions that are not yet known to have committed. Fortunately, any such transaction that could possibly commit writes must *already* be holding write locks at the current range leader. More formally, the transaction must be holding write locks on any replica whose leader lease's epoch interval upper end is above e_c. To see why this holds, suppose a transaction T with an epoch $e_T < e_c$ has completed the Prepare phase but not the Commit phase. Recall from §5.4 that the client acting as T's coordinator receives the epoch range of the lease from the range leader it used to perform the Prepare, and checks whether e_T falls within that epoch range. If it did not, then the client aborts the transaction so it cannot possibly commit. Otherwise, recall that transactions do not release any locks until the commit phase, including across leader changes. Therefore, it must be that the locks are held on the leader whose lease's epoch range contains e_c (and by the leader disjointedness invariant, there can be at most one such replica), and any subsequent leader replica. A similar argument shows why the same holds for transactions that started but have not finished the Prepare phase. Hence, the read algorithm first reads the current epoch e_c (once per transaction), ensures it 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.

The algorithm as described so far does not guarantee linearizability, because a transaction T would not observe the effects of transactions in epoch e_c that committed before T started. If desired, ensuring linearizability is easy at the cost of some latency; after T starts, it waits for the epoch to advance once and then use the new epoch.

6.3 Garbage Collection

Chardonnay must periodically remove old record versions to avoid running out of space. Chardonnay uses the lower end of its range leader lease's epoch interval to determine which versions are no longer needed and can be garbage collected. There is a background job running on each range replica that removes versioned records (other than the newest version of a record) whose epoch is less than a delta from the lower end of the epoch interval. A snapshot read must validate that its epoch value lies within that delta from the lower end of the interval after executing its reads, to avoid reading an incomplete snapshot due to versions being deleted. In our experiments we configure the delta to be 6000, so that versions are kept at least for roughly one minute before they are GC'd. Additionally, since snapshot reads only happen at epoch boundaries, when a new version of a record is inserted, if it has the same epoch as the previous version then that previous version is immediately deleted. This optimization significantly reduces the number of versions maintained for records that are updated very frequently (i.e. highly-contended records).

7 Prefetching

Our experiments in §4 show that with fast 2PC, reading from slow storage becomes a primary cause of a transaction's contention footprint. Hot contended records will typically be cached in the database's memory. However, this does not fully address the issue because a transaction might access hot records along with other cold records that are not good candidates for caching. There are several well-known techniques to work around this problem [20]. For example, the programmer could manually prefetch records before executing the transaction. Another technique is to ensure hot records are the last to be accessed. This is beneficial because it minimizes the execution time during which access to the hot record causes a conflict. Unfortunately, these are are not always applicable, and they push a lot of complexity to programmers.

We could require the programmers to label their queries with the read set. Then the system can prefetch the records (i.e., key-value pairs) identified by those keys to memory before executing the transaction and pin them until the transaction finishes, so that no time is spent reading from slow storage while locks are held. However, this scheme restricts the programming model, and is incapable of supporting *dependent queries*, that is, ones whose read set cannot be determined prior to executing the query [76]. This contradicts Chardonnay's goal of a general programming model (e.g., supporting SQL).

Instead, Chardonnay *transparently* uses the client's code to first execute the query in a lock-free, *dry run* mode to load the read set to memory, then executes normally with 2PL.

It is of course possible for the read set to change by the time the actual transaction executes. One reason is that only the *values* of one or more records change due to a write by another transaction. Chardonnay handle this correctly and with no performance penalty, by reflecting the changes in its prefetching buffer (§7.3). The other possibility, in the case of dependent queries, is that the set of *keys* itself changes, so the transaction has to read some keys that had not been prefetched and pinned. This does not pose a correctness problem but may cause a transaction to read additional data from disk while it is holding the locks, and thereby increasing its contention footprint. Fortunately, prior work has shown this seldom happens in real-world workloads [76]; dependent queries are commonly ones that must read from a secondary index to identify their full read and write sets. Since secondary indices are fairly expensive to modify, they are seldom kept on fields whose values are updated very frequently. One example of such transactions is the "Payment" transaction of the TPC-C benchmark. Since the TPC-C benchmark workload never modifies the index on which Payment transactions' read and write sets may depend [76], the set of keys read by a Payment transaction never changes between the dry run and the execution.

One additional benefit of strongly consistent dry run queries is that if the application logic aborts the transaction on its own, there is no need to perform the actual execution. On the other hand, using dry run queries has two main disadvantages. First, it adds to the query latency, although this additional latency does not contribute to the contention footprint. Second, it requires executing the transaction logic twice before committing. While OLTP read-write transactions tend to be small, this could still be wasteful if the transaction is compute-intensive, particularly in low contention cases. The user can disable dry run queries by using a different API. In the future, we plan to explore automatically deciding when to do prefetching based on the characteristics of the workload.

7.1 API

The API shown in Figure 3 is more suited to user-interactive transactions (e.g., a user executing a multi-statement SQL transaction at a console, examining intermediate results before writing more queries). To use prefetching, a slightly different API is used to start the transaction where the caller passes a function that executes the transaction logic, i.e.,

```
<typename T>
T run(std::function< T( Transaction* ) > query)
```

Within the function, the code can freely call the read, scan, or write APIs using the transaction object that gets

passed. There are essentially no restrictions on the code inside the function, even though in practice it would have no side effects beyond the transaction's writes to the database itself. This does not add any unusual restrictions; any transaction might have to abort, and side effects outside of the transaction cannot be rolled back.

7.2 Semantics

The dry run query runs under snapshot isolation using our snapshot read mechanism that we described in §6, and can be configured to be strictly serializable if desired. Running under a lower isolation level such as *read committed* [19] could be problematic because it exposes the programmer's code to anomalies that would not happen in the serializable execution. This might cause the client's code to abort the transaction, prematurely ending prefetching, or worse, crash. Therefore, we do not use a lower isolation level because prefetching should be transparent to the user.

7.3 Design

Each range leader maintains a *prefetching buffer* to store a transaction's read set's records in main memory. The prefetching buffer tracks which records are in main memory and allows transactions to request pinning keys. Any committed write to a pinned record updates the value in the buffer, so that it becomes a write-through consistent cache for pinned records, and any transaction that needs to read a pinned key can just get its value from the cache and not have to go to the database.

To efficiently support range-queries, the prefetching buffer tracks key ranges not just individual keys; if a key range is pinned to the buffer, and a new transaction inserts (or deletes) a record whose key falls within that range, that new record is pinned too. Hence, a transaction that sees a range is pinned to the buffer can satisfy a range read from the cache without going to the database.

When a transaction is running in dry run mode, it reads the committed, snapshot value from the database without acquiring any locks, requests pinning the key (or key range), and then returns the committed value to the client to continue executing the query. In most cases both the snapshot and latest versions can be read using a single IO, so this does not typically increase the IO overhead. Writes made by the dry run query never make it to the KV Service, and are discarded at the client after the dry run. After the dry run completes, the client library reruns the transaction in normal mode. When that transaction finishes (i.e. commits or aborts), the keys are unpinned and become eligible for eviction.

It is possible that a request to pin a record cannot be satisfied because the range leader has run out of memory. In this case the dry run query could be delayed until memory frees up, or just be aborted. This serves as effective admission control prior to acquiring any locks. Some care is needed to avoid a potential live-lock situation, but in the worst case transactions can skip prefetching.

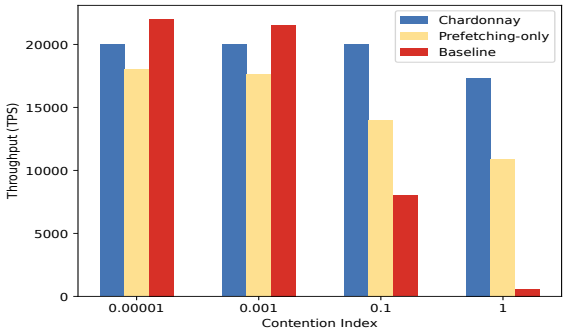
7.4 Handling Resource Contention

Dry run queries execute most of the transaction logic in Chardonnay, so that when the actual transaction executes it only needs to perform minimal work. However, if we are not careful, the activity from dry run queries and other background tasks such as garbage collection and compaction can compete with transactions for resources on the machines running the KV-service ranges. As a side effect, this could slow down the lock-acquiring transaction and increase data contention. Therefore, we dedicate resources on each machine to transactions to ensure they are insulated from lower-priority activity that does not hold locks.

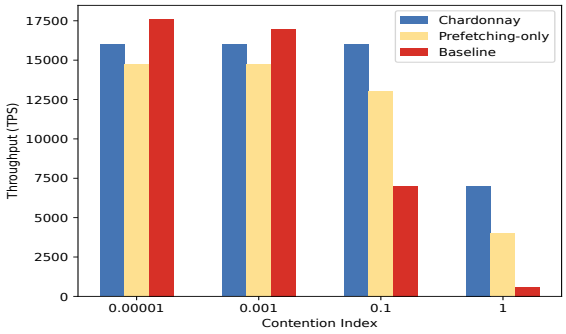
8 Deadlock Avoidance

Since Chardonnay uses 2PL, it has to deal with the problem of deadlocks. An easy solution is transaction timeouts, since they are needed anyway to deal with various possible failures. Unfortunately, a deadlocked transaction would be holding locks for the entire timeout duration before these locks are released. Another popular choice is a deadlock prevention scheme such as Wait-Die or Wound-Wait [64], but they can be too conservative (i.e., aborting transactions that are not deadlocked) which can become problematic under high contention. A more common choice in practice is deadlock detection [63] via detecting cycles in the wait-for graph [38, 72] and breaking the cycle by choosing a transaction to abort. This requires significant overhead for maintaining the global wait-for graph state, and potentially frequent aborts.

By making all transactions acquire their locks in the same order, we can prevent deadlocks. In Chardonnay, the read and write sets of the transaction are (approximately) computed by dry run queries, prior to acquiring any locks. We acquire the locks in ascending key order prior to actually executing the transaction. A naive implementation of this idea would require adding $|\text{read\_set} \cup \text{write\_set}|$ round-trips to the contention footprint to acquire the locks. Instead, the client uses an approach similar to RPC Chains [71], which cuts the round-trips required roughly in half compared to the naive approach. The client in Chardonnay sends one RPC to the first range from which it needs keys. The range acquires all the local locks, performs all the necessary local reads, and then forwards the request to the next range. The client immediately sends an RPC to the last range in the



(a) 10% Distributed Transactions.



(b) 100% Distributed Transactions.

Figure 4: Contention Microbenchmark Throughput Results

request, which holds the RPC until the request arrives. After finishing its local work, the last range replies to the client's RPC with all the read results. When this is done, the client runs the transaction logic. If the transaction invokes a read for a key or key range (that the client already has), the client returns it immediately since it has the lock on the data (and will detect and abort the transaction if that lock is lost before commit). If transaction's read or write set changes between the dry run and actual transaction, the client cannot serve the reads from its local cache and has to send the read requests to the ranges. We fall back to Wound-Wait for these locks.

If most transactions are likely to perform multiple read operations involving multiple network round-trips and reads from slow storage, then a developer might be tempted to parallelize those accesses, if possible, to reduce the contention footprint. Whether this is done with parallel threads or asynchronous APIs, it adds complexity to the programming model. Our scheme gets the same benefit without this complexity. On the other hand, the scheme can actually increase a transaction's contention footprint, because lock acquisition has to be serialized. There is no overhead for the common case of transactions accessing a single range. We allow the programmer to disable ordered lock acquisition per transaction. In the future, we plan to adaptively apply the technique.

9 Evaluation

In this section we study how Chardonnay performs under contention (§9.1), its scalability (9.2), and its snapshot read performance (§9.3). To evaluate contention, we use a benchmark used by Calvin [76], which is inspired by TPC-C's New-Order transaction. For scalability experiments, we use the standard TPC-C benchmark, and for read latency we use YCSB [28]. In all experiments the KV service range leaders use Standard_L8s_v2 Azure VMs, which provide 8 vCPU cores and 64GB of memory and support accelerated networking necessary for eRPC. We place the database on directly-attached SSD for high IOPS. For the WAL, we emulate NVMe on DRAM via RAMdisk, since it is not currently offered on Azure. We advance the epoch every 10ms. All results are 10 minute averages unless otherwise stated.

9.1 Contention Microbenchmark

We use a benchmark introduced in Calvin [76] to evaluate Chardonnay's performance under high contention. Each transaction in the benchmark reads 10 records, performs a constraint check on the result, and if the check passes, updates a counter in each record. The records in each KV-service range are divided into two disjoint sets: cold and hot. Each transaction can either be local or distributed. A local transaction accesses 9 records chosen at random from the cold set in the target range, and 1 record chosen at random from the hot set. A distributed transaction is similar, except it accesses 8 cold records and an additional hot record in a different range. The number of cold records in each range is much larger than available memory so cold records will be mostly served from disk. The number of hot records is determined by a parameter called the ***contention index***, which is set to be the inverse of the number of hot records and represents the probability that two transactions accessing the same range will conflict. Hence, a contention index of 0.01 means that there are 100 hot records per range, while a contention index of 1 means that there is 1 hot record (which is accessed by *all* transactions touching that range). The contention index controls the degree of parallelism within each range (e.g., a contention index of 1 means that all transactions within a range are serialized).

We wrote each transaction using simple, synchronous APIs. This means that all reads are executed sequentially.

This is not a requirement, but it highlights the additional benefits of Chardonnay's dry run and deadlock avoidance schemes, which move sequential operations outside of the contention footprint. The ordering of the reads done by each transaction is random, so there is variance in the time hot records spend under lock.

Setup. We use 6 ranges, and each range leader is assigned its own VM. We evaluate the following configurations of Chardonnay:

- **Baseline**. All transactions run without dry-run queries, so they do not perform prefetching or ordered lock acquisition. This is essentially a classic shared-nothing system architecture with a fast 2PC implementation, and Wound-Wait for deadlocks.
- **Prefetching-only**. Transactions run with dry-run queries, but only do prefetching and not ordered lock acquisition.
- **Chardonnay**. All transactions perform prefetching and ordered lock acquisition.

Initially, we planned to compare against CockroachDB [74] as a representative for a modern shared-nothing system. However, we realized that retrofitting the system with eRPC would be a very significant engineering effort. Running the (full SQL) system unchanged on the same experimental setup yielded low throughput (TPS per node is 90% less than Chardonnay). Hence, we use our baseline configuration for apples-to-apples comparison, as it is a good representative of the shared-nothing architecture.

We plot the throughput and abort rates under different values of contention index in Figures 4 and 5.

Analysis. As expected, under low contention, the dry run queries in Chardonnay are mostly wasteful and consequently the baseline configuration has slightly better throughput. Notably, full Chardonnay performs better than prefetching-only even under low contention. This is because ordered lock acquisition issues Lock & Read requests in a batched, efficient manner, as opposed to sequentially issuing an RPC per read during the transaction execution in the prefetching-only configuration. This further supports our intuition that Chardonnay's ordered lock acquisition scheme enables writing the transactions in a simple, synchronous manner without a significant performance penalty. As contention increases, the overall throughput becomes constrained by the contention footprint, and in particular, the length of time locks on hot records also increases. The baseline configuration has the sharpest drop in throughput, since it has to issue multiple RPCs and reads from slow storage while holding locks. The full Chardonnay configuration performs best under high contention and has *zero* aborts. Prefetching-only fares much better than baseline, even though it suffers a significant drop in throughput due to increased deadlock avoidance abort rates under contention, as well as increased contention footprint due to RPCs.

In the 10% distributed transactions case, transactions essentially never deadlock since they can only conflict on one record in the vast majority of cases. Yet, the Wound-Wait deadlock avoidance scheme is too conservative and results in many unnecessary aborts as contention increases; see Figure 5. Note that because the base configuration's transactions have a much larger contention footprint, even a relatively modest 0.001 contention index is affected by these superfluous aborts. A less conservative scheme such as deadlock detection would not suffer from this, at the cost of taking much longer to resolve the deadlock when an actual one appears. In Chardonnay, we largely avoid deadlock aborts and only use Wound-Wait as a fall-back mechanism, as discussed in §8.

One interesting property of Chardonnay is that distributed transactions are not dramatically more expensive than local transactions. The peak throughput under low contention with 100% distributed transactions is roughly 22% lower than with only 10% distributed transactions. This makes the importance of reducing cross-partition transactions less significant, thus relieving database administrators and developers from the requirement to continually re-partition the application data to minimize cross-partition transactions [30, 35, 36, 60, 73]. The big difference in throughput between 10% and 100% ratio of distributed transactions under higher contention index values is largely because each transaction in the 100% distributed case accesses two items from the hot set, not because the transaction is distributed. This is in part because 2PC is highly optimized in Chardonnay, but also because non-distributed transactions have to go through a phase of reading the epoch before committing. Our results for the 10% distributed case show that the benefits of Chardonnay are not limited to distributed transactions.

9.2 Scalability

The scalability of the System R*-style shared-nothing architecture is well established [80], but Chardonnay introduces the *read-epoch* operation during each transaction's 2PC. Therefore, we need to ensure that the epoch service can keep up with an increasing scale.

TPC-C New-Order. Similar to prior work [76], we limited our TPC-C implementation to the New Order transaction, which is the bulk of the TPC-C workload including almost all distributed transactions that require high isolation. We would expect similar results if we were to run the full TPC-C benchmark. We assign each KV service range leader to a dedicated VM and have it

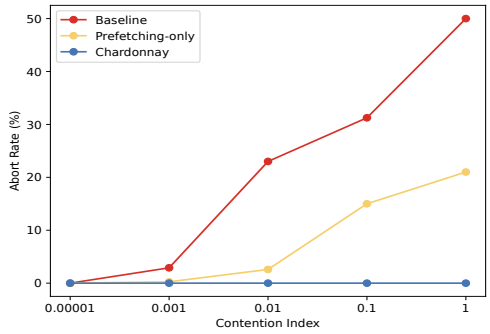


Figure 5: Abort rates for 10% distributed tx micro-benchmark. Chardonnay's deadlock aborts are 0%.

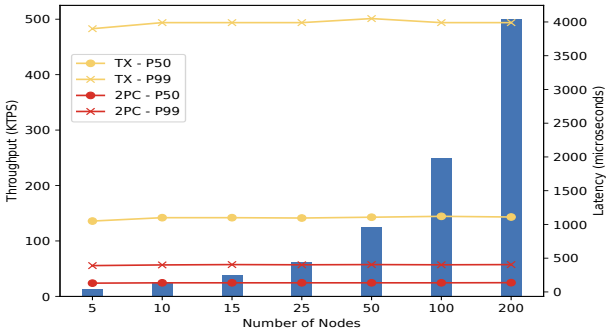


Figure 6: TPC-C New-Order transaction results.

host 10 warehouses. We limit the overall throughput to 2500 TPS per node, since we aim to evaluate the scalability of the system rather than the raw per-node peak throughput. The clients run on dedicated VMs, separate from Chardonnay nodes. (Recall that in Chardonnay, the execution of the transaction logic happens on the client.) We plot the results in Figure 6, which show stable 2PC latency as the system scales linearly.

Comparison with Calvin. Chardonnay is able to reach similar New-Order throughput scale as reported by Calvin [76], *without Calvin's significant programming model restrictions* (described in §10). Calvin's reported single-datacenter latency is much higher than Chardonnay (∼100ms), but the comparison is not meaningful since it does not use fast RPC and storage. However, with 10ms epoch duration as in our setup, we expect Calvin adds 5ms to the median latency since it groups transactions into batches at the start of each epoch. Therefore, even with fast RPC and storage, we expect Calvin's median latency to be higher than Chardonnay's P99 latency.

Epoch microbenchmark. To test the limits of the Epoch service, we wrote a micro-benchmark where each thread simulates a Chardonnay client node running 2PC. We run 30 client nodes with 8 threads each, where each thread is issuing 5000 read-epoch calls per second for a total of *1,200,000* calls per second. The median latency is below 60 µs, which is less than the median for the full Prepare phase. Since *read-epoch* runs in parallel to Prepare, this does not increase the overall 2PC latency.

9.3 Snapshot Read Latency

We use YCSB with 50% write and 50% read to evaluate snapshot read latency, using a setup similar to §4. Read operations run with snapshot isolation for this purpose. The dataset fits in DRAM since our focus is measuring protocol overhead, not IO latency. When running with a uniform distribution of keys, the median latency of reads is roughly 220 µs. On the other hand, when running with Zipfian 0.99 distribution it increases to nearly 355 µs. This is because most reads in the Zipfian case are going to write-hot records and hence almost always have to wait for locks to be released before they can execute. We also run the read operations with strict serializability. The median latency of the read operations increases by ∼5ms since they need to wait for the epoch to advance.

9.4 Range Reads

We devise a simple experiment to demonstrate Chardonnay's effectiveness for range reads. The experiment involves a single range that contains 100 records. There are two client threads, one is a writer thread that is continuously deleting and then re-inserting a random record in the range, and the other is a reader that is executing a range query to read all records. Even though the reader thread is not doing any writes, its range read query is not declared as read-only so that it runs as a read-write transaction, not as a snapshot read. We compare the number of insert operations per second in Chardonnay and the baseline from §9.1. The results are in Table 2. Without prefetching, the baseline has to execute the range read against the database each time while holding the lock on the entire range, resulting in a longer contention period and thus slowing down the writer.

10 Related Work

Shared-nothing. Spanner [29], and CockroachDB [74] are prominent modern examples of systems that utilize shared-nothing architecture, both primarily targeting inter-datacenter operations with globally-distributed workloads. Spanner uses specialized hardware to

	Chardonnay	Baseline
Insert TPS	914	197

Table 2: Range Read Results.

achieve clock synchronization guarantees that are necessary for its external consistency support. Many design choices in the system make it hard to support fast transactions within the datacenter. For instance, Spanner guarantees correctness of readers by introducing a delay for writers during the commit protocol until the clock uncertainty interval has passed, which can add many milliseconds to a transaction's contention footprint. In contrast, Chardonnay guarantees correctness by having readers potentially wait for write locks so that the contention footprint for writers does not have to increase. CockroachDB is a system with similar emphasis on global distribution. It does not guarantee external consistency to avoid requiring specialized clocks, and instead provides the weaker single-key linearizability (which still relies on bounded clock skew). Its concurrency control protocol has optimistic components optimized for low contention.

Shared Disk. Shared-disk [13] is another classic DBMS architecture [25, 44] that has become popular in recent years in systems such as Amazon Aurora [78, 79], Socrates [13], and Google's AlloyDB [2]. These systems are single-master, in which only one node actively writes the database, limiting scalability. Aurora also has a multi-master mode which does not offer serializability, and works well for partitionable workloads with little cross-partition activity. In contrast, Chardonnay can horizontally scale both reads and writes with strict serializability and supports fast cross-partition transactions.

Shared Log. Hyder [21] and Tango [18] scale-out compute without partitioning by utilizing a shared log that is accessed by all compute nodes. Appending to and replaying the log can be a bottleneck limiting scalability.

Deterministic Systems. Deterministic execution has been explored as an alternative to distributed commit in systems such as Calvin [76] and Aria [56]. A major benefit of using determinism is eliminating transaction aborts due to deadlocks [63], which Chardonnay largely achieves using its lock ordering scheme. Deterministic execution databases typically have to restrict the programming model to one-shot transactions. They also group incoming transactions into batches before executing them, which can add tens of milliseconds to latency.

Another significant limitation of most deterministic database systems is they require knowing a transaction's read and write set upfront [75, 76]. To support dependent queries, a programmer can precede a transaction with a lower isolation *reconnaissance* query to compute its read/write sets (e.g., OLLP [75, 76]). However, compared to dry run queries in Chardonnay, reconnaissance queries require changes to the client transaction code and run at low isolation level, exposing the code to anomalies that do not appear in the real execution. Furthermore, if the read or write set of the transaction changes between running the reconnaissance query and actual transaction, the transaction must abort. Fauna [3], a DBMS based on Calvin's design, eliminates the need for manual reconnaissance queries at the cost of adding a round of Raft consensus to the transaction's contention footprint, and using OCC, degrading performance under contention. Snapper [53] is a transaction library for single-node systems based on the Actor model, which enables deterministic execution for transactions that can be labeled with their read and write sets, while simultaneously supporting non-deterministic execution for transactions where this is not possible.

The Calvin paper proposes using the read set to prefetch data prior to sequencing the transaction. Prefetching in Calvin requires precisely estimating I/O latency [76]. It also happens after the reconnaissance query, adding to query latency. Notably, Calvin's designers do not discuss range reads. Presumably even if a transaction's entire readset is in memory, it still needs to run the range query against the database to ensure no other transaction has inserted or deleted records within that range.

Distributed epoch-based commit. Coco [57] is an in-memory system that applies epoch-based group commit in a distributed setting. It uses a centralized coordinator to *synchronously* commit transactions in epoch order. This requires adding many milliseconds of latency to read-write transactions. In contrast, Chardonnay is an on-disk system that guarantees the equivalence to epoch ordering, but transactions commit out of epoch order for low latency.

11 Conclusions

This paper presents Chardonnay, a scale-out, general-purpose, multi-versioned, on-disk transactional key-value store optimized for single datacenter deployments with fast 2PC. Chardonnay takes advantage of fast RPCs to support strictly serializable snapshot reads without relying on specialized clocks or assumptions about maximum clock skew. Chardonnay achieves high performance for high contention workloads by automatically and transparently loading and pinning data from slow storage to main memory prior to acquiring any locks, and avoids deadlocks by ordering its lock requests. We believe that the design principles of Chardonnay can also be applied in other settings, such as multi-core single-node systems for high contention workloads.

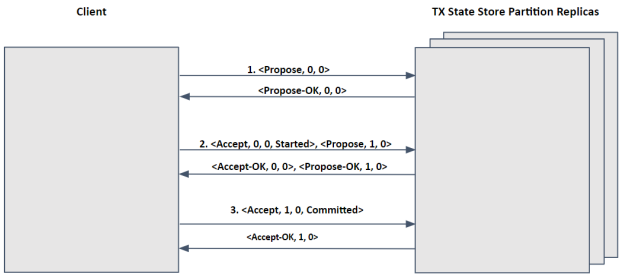


Figure 7: Recording Transaction State.

12 Acknowledgments

We thank our shepherd, Marc Shapiro, and the anonymous reviewers for their many useful comments and suggestions. We also thank Ryan Stutsman, Irene Zhang and Kyle Raftogianis for their feedback on earlier drafts of this work. This work was supported in part by a GE/DARPA grant, a CAIT grant, NSF awards CNS 2106530, CNS 2104292 and CNS 2143868, and gifts from JP Morgan, DiDi, and Accenture.

A Appendix

A.1 Optimizing 2PC

Pipelined WAL. Since RPCs and log writes are cheap in our system and low latency is paramount, we do not batch multiple operations into a single WAL entry. Instead, each operation (e.g., a transaction's Prepare) has its own WAL entry (hence runs its own instance of the Paxos state-machine). Furthermore, appends to the WAL are pipelined [67] (i.e., we do not wait for the previous entries to be completely written and applied before starting a new one). Log entries are still applied to the database in log order for correctness, however. Note that a Prepare must go through the range leader, which drives appending it to the log. The long-lived leader design allows the leader to complete a log append using one RPC in the common case. Hence the latency of a Prepare operation is roughly the sum of the latencies of two RPCs and one NVMe write.

Client-driven Commit. As mentioned earlier, we use Paxos to replicate the state of each transaction in the transaction state store. However, to minimize the number of required round-trips, we do not designate any of the replicas as a leader. Leaders in Paxos are an optimization used in part to avoid the dueling proposers problem. Since we carefully designed the state of each transaction to be an independent Multi-Paxos log, the client is the only proposer in the vast majority of cases. So requiring it to go through a leader to run the Paxos protocol to commit (or abort) the transaction adds the latency of a superfluous RPC to the Commit operation (which happens under transaction locks). Furthermore, the client utilizes a variant of the well-known technique of chaining Paxos instances together [24]. As illustrated in Figure 7, when performing the RPC to run the second (Accept) phase of Paxos to append log entry 0 (i.e., recording transaction start), the client simultaneously runs the first (Propose) phase of Paxos entry number 1 (i.e., reserving the right to propose the value of proposal number 0). Thus, it incurs the latency of only one RPC to append the decision.

A.2 Proof Sketch of Epoch Ordering

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 and sometime later T_1 acquired a lock. However, since $e_1 < e_2$, the monotonic epoch invariant implies T_1 finished execution (and acquired all its locks) before T_2 did so, a contradiction as transactions do not release any locks until commit. Hence, T_1 precedes T_2 in the equivalent order. □

A.3 Scaling the Epoch Service

Here we discuss briefly how to scale-out the epoch service while maintaining the correctness of our snapshot reads. The basic idea is introduce intermediary *epoch publishers* between the epoch service and its clients. 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 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 highly available). Each client is assigned to one of the publishers, and uses the same procedure to read the epoch from that publisher exactly as it would from the epoch service itself.

This design requires slightly weakening the monotonic epoch invariant, since it is possible for a client to read a value of the epoch that is one less than the true epoch. Furthermore, when a client is assigning an epoch to a transaction, it needs to ensure the epoch is at least as high as that of any record in its read and write sets, even if the version it reads from the publisher is lower. Linearizability of snapshot reads can be ensured at the cost of additional latency, by waiting for the epoch to advance twice instead of just once.

References

[1] 3D Xpoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html, 2023.

[2] AlloyDB for PostgreSQL. https://cloud.google.com/alloydb, 2023.

[3] Fauna. https://fauna.com/, 2023.

[4] FlatBuffers. https://google.github.io/flatbuffers/, 2023.

[5] gRPC. https://grpc.io/, 2023.

[6] Intel® OptaneTM SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html, 2023.

[7] LevelDB. https://leveldb.org/, 2023.

[8] RocksDB. A persistent key-value store for fast storage environments. https://rocksdb.org/, 2023.

[9] Toshiba memory introduces XL-FLASH storage class memory solution. https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html, 2023.

[10] Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf, 2023.

[11] yugabyteDB. https://yugabyte.com/, 2023.

[12] ZippyDB: Facebook's key value store. https://engineering.fb.com/2021/08/06/core-data/zippydb/, 2023.

[13] ANTONOPOULOS, P., BUDOVSKI, A., DIACONU, C., HERNANDEZ SAENZ, A., HU, J., KODAVALLA, H., KOSSMANN, D., LINGAM, S., MINHAS, U. F., PRAKASH, N., PUROHIT, V., QU, H., RAVELLA, C. S., REISTETER, K., SHROTRI, S., TANG, D., AND WAKADE, V. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1743–1756.

[14] BACON, D. F., BALES, N., BRUNO, N., COOPER, B. F., DICKINSON, A., FIKES, A., FRASER, C., GUBAREV, A., JOSHI, M., KOGAN, E., LLOYD, A., MELNIK, S., RAO, R., SHUE, D., TAYLOR, C., VAN DER HOLST, M., AND WOODFORD, D. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, Association for Computing Machinery, p. 331–343.

[15] BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow. 7*, 3 (nov 2013), 181–192.

[16] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. HAT, not CAP: Towards highly available transactions. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)* (Santa Ana Pueblo, NM, May 2013), USENIX Association.

[17] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J. J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data Systems Research (CIDR 2011)* (2011).

[18] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCK, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 325–340.

[19] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ansi sql isolation levels. SIGMOD '95, Association for Computing Machinery, p. 1–10.

[20] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[21] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder - a transactional record manager for shared flash. In *CIDR* (2011), www.cidrdb.org, pp. 9–20.

[22] BINNIG, C., CROTTY, A., GALAKATOS, A., KRASKA, T., AND ZAMANIAN, E. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow. 9*, 7 (mar 2016), 528–539.

[23] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., HARIDAS, J., UDDARAJU, C., KHATRI, H., EDWARDS, A., BEDEKAR, V., MAINALI, S., ABBASI, R., AGARWAL, A., HAQ, M. F. U., HAQ, M. I. U., BHARDWAJ, D., DAYANAND, S., ADUSUMILLI, A., MCNETT, M., SANKARAN, S., MANIVANNAN, K., AND RIGAS, L. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, Association for Computing Machinery, p. 143–157.

[24] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live - an engineering perspective (2006 invited talk). In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing* (2007).

[25] CHANDRASEKARAN, S., AND BAMFORD, R. Shared cache - the future of parallel databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)* (2003), pp. 840–850.

[26] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (USA, 2006), OSDI '06, USENIX Association, p. 15.

[27] CHEN, Y., YU, X., KOUTRIS, P., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SHU, J. Plor: General transactions with predictable, low tail latency. In *Proceedings of the 2022 International Conference on Management of Data* (New York, NY, USA, 2022), SIGMOD '22, Association for Computing Machinery, p. 19–33.

[28] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, Association for Computing Machinery, p. 143–154.

[29] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst. 31*, 3 (aug 2013).

[30] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow. 3*, 1–2 (sep 2010), 48–57.

[31] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, Association for Computing Machinery, p. 205–220.

[32] DONG, S., KRYCZKA, A., JIN, Y., AND STUMM, M. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (Feb. 2021), USENIX Association, pp. 33–49.

[33] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Symposium on Operating Systems Principles (SOSP'15)* (October 2015), ACM – Association for Computing Machinery.

[34] EISENMAN, A., CIDON, A., PERGAMENT, E., HAIMOVICH, O., STUTSMAN, R., ALIZADEH, M., AND KATTI, S. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 65–78.

[35] ELDEEB, T., CHEN, Z., CIDON, A., AND YANG, J. Neuroshard: Towards automatic multi-objective sharding with deep reinforcement learning. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (New York, NY, USA, 2022), aiDM '22, Association for Computing Machinery.

[36] ELMORE, A. J., ARORA, V., TAFT, R., PAVLO, A., AGRAWAL, D., AND EL ABBADI, A. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), pp. 299–313.

[37] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (nov 1976), 624–633.

[38] GRAY, J. Notes on data base operating systems. In *Advanced Course: Operating Systems* (1978).

[39] GUO, Z., WU, K., YAN, C., AND YU, X. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data* (New York, NY, USA, 2021), SIGMOD '21, Association for Computing Machinery, p. 658–670.

[40] GUO, Z., ZENG, X., WU, K., HWANG, W., REN, Z., YU, X., BALAKRISHNAN, M., AND BERNSTEIN, P. A. Cornus: Atomic commit for a cloud DBMS with storage disaggregation. *Proc. VLDB Endow. 16*, 2 (2022), 379–392.

[41] HARDING, R., VAN AKEN, D., PAVLO, A., AND STONEBRAKER, M. An evaluation of distributed concurrency control. *Proc. VLDB Endow. 10*, 5 (jan 2017), 553–564.

[42] HELLAND, P. Life beyond distributed transactions: an apostate's opinion. In *Conference on Innovative Data Systems Research (CIDR 2007)* (2007).

[43] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (jul 1990), 463–492.

[44] JOSTEN, J. W., MOHAN, C., NARANG, I., AND TENG, J. Z. Db2's use of the coupling facility for data sharing. *IBM Systems Journal 36*, 2 (1997), 327–351.

[45] KAFFES, K., CHONG, T., HUMPHRIES, J. T., BELAY, A., MAZIÈRES, D., AND KOZYRAKIS, C. Shinjuku: Preemptive scheduling for µsecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, p. 345–359.

[46] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 1–16.

[47] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 185–201.

[48] LAMPSON, B. W., AND LOMET, D. B. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1993), VLDB '93, Morgan Kaufmann Publishers Inc., p. 630–640.

[49] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)* (January 2015).

[50] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. Multi-version range concurrency control in deuteronomy. *Proc. VLDB Endow. 8*, 13 (sep 2015), 2146–2157.

[51] LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, Association for Computing Machinery, p. 21–35.

[52] LIN, Q., CHANG, P., CHEN, G., OOI, B. C., TAN, K.-L., AND WANG, Z. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, p. 1659–1674.

[53] LIU, Y., SU, L., SHAH, V., ZHOU, Y., AND VAZ SALLES, M. A. Hybrid deterministic and nondeterministic execution of transactions in actor systems. SIGMOD '22, Association for Computing Machinery, p. 65–78.

[54] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. 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* (New York, NY, USA, 2011), SOSP '11, Association for Computing Machinery, p. 401–416.

[55] LOMET, D. B., AND MOKBEL, M. F. Locking key ranges with unbundled transaction services. *Proc. VLDB Endow. 2* (2009), 265–276.

[56] LU, Y., YU, X., CAO, L., AND MADDEN, S. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow. 13*, 12 (jul 2020), 2047–2060.

[57] LU, Y., YU, X., CAO, L., AND MADDEN, S. Epoch-based commit and replication in distributed oltp databases. *Proc. VLDB Endow. 14*, 5 (jan 2021), 743–756.

[58] MATSUNOBU, Y., DONG, S., AND LEE, H. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proc. VLDB Endow. 13*, 12 (aug 2020), 3217–3230.

[59] MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst. 11*, 4 (dec 1986), 378–396.

[60] QUAMAR, A., KUMAR, K. A., AND DESHPANDE, A. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology* (2013), EDBT '13, p. 430–441.

[61] RAO, J., SHEKITA, E. J., AND TATA, S. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow. 4*, 4 (jan 2011), 243–254.

[62] REN, K., FALEIRO, J. M., AND ABADI, D. J. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, p. 1583–1598.

[63] REN, K., THOMSON, A., AND ABADI, D. J. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow. 7*, 10 (jun 2014), 821–832.

[64] ROSENKRANTZ, D. J., STEARNS, R. E., AND LEWIS, P. M. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (jun 1978), 178–198.

[65] ROTHNIE, J. B., BERNSTEIN, P. A., FOX, S., GOODMAN, N., HAMMER, M., LANDERS, T. A., REEVE, C., SHIPMAN, D. W., AND WONG, E. Introduction to a system for distributed databases (sdd-1). *ACM Trans. Database Syst. 5*, 1 (mar 1980), 1–17.

[66] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's time for low latency. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)* (Napa, CA, May 2011), USENIX Association.

[67] SANTOS, N., AND SCHIPER, A. Optimizing paxos with batching and pipelining. *Theoretical Computer Science 496* (2013), 170–183. Distributed Computing and Networking (ICDCN 2012).

[68] SERAFINI, M., TAFT, R., ELMORE, A. J., PAVLO, A., ABOULNAGA, A., AND STONEBRAKER, M. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow. 10*, 4 (nov 2016), 445–456.

[69] SHAMIS, A., RENZELMANN, M., NOVAKOVIC, S., CHATZOPOULOS, G., DRAGOJEVIĆ, A., NARAYANAN, D., AND CASTRO, M. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 433–448.

[70] SKEEN, D. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1981), SIGMOD '81, Association for Computing Machinery, p. 133–142.

[71] SONG, Y. J., AGUILERA, M. K., KOTLA, R., AND MALKHI, D. Rpc chains: Efficient client-server communication in geodistributed systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)* (April 2009), USENIX.

[72] STONEBRAKER, M. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering SE-5*, 3 (1979), 188–194.

[73] TAFT, R., MANSOUR, E., SERAFINI, M., DUGGAN, J., ELMORE, A. J., ABOULNAGA, A., PAVLO, A., AND STONEBRAKER, M. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment 8*, 3 (2014), 245–256.

[74] TAFT, R., SHARIF, I., MATEI, A., VANBENSCHOTEN, N., LEWIS, J., GRIEGER, T., NIEMI, K., WOODS, A., BIRZIN, A., POSS, R., BARDEA, P., RANADE, A., DARNELL, B., GRUNEIR, B., JAFFRAY, J., ZHANG, L., AND MATTIS, P. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 1493–1509.

[75] THOMSON, A., AND ABADI, D. J. The case for determinism in database systems. *Proc. VLDB Endow. 3*, 1–2 (sep 2010), 70–80.

[76] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, Association for Computing Machinery, p. 1–12.

[77] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 18–32.

[78] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD 2017* (2017).

[79] VERBITSKI, A., GUPTA, A., SAHA, D., COREY, J., GUPTA, K. K., BRAHMADESAM, M., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. *Proceedings of the 2018 International Conference on Management of Data* (2018).

[80] YANG, Z., YANG, C., HAN, F., ZHUANG, M., YANG, B., YANG, Z., CHENG, X., ZHAO, Y., SHI, W., XI, H., YU, H., LIU, B., PAN, Y., YIN, B., CHEN, J., AND XU, Q. Oceanbase: A 707 million tpmc distributed relational database system. *Proc. VLDB Endow. 15*, 12 (2022), 3385–3397.

[81] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow. 10*, 6 (feb 2017), 685–696.

[82] ZAMANIAN, E., SHUN, J., BINNIG, C., AND KRASKA, T. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 511–526.

[83] ZHANG, I., RAYBUCK, A., PATEL, P., OLYNYK, K., NELSON, J., LEIJA, O. S. N., MARTINEZ, A., LIU, J., SIMPSON, A. K., JAYAKAR, S., ET AL. The demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 195–211.

[84] ZHANG, M., HUA, Y., ZUO, P., AND LIU, L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 51–68.

[85] ZHONG, Y., LI, H., WU, Y. J., ZARKADAS, I., TAO, J., MESTERHAZY, E., MAKRIS, M., YANG, J., TAI, A., STUTSMAN, R., AND CIDON, A. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association.

[86] ZHONG, Y., WANG, H., WU, Y. J., CIDON, A., STUTSMAN, R., TAI, A., AND YANG, J. Bpf for storage: An exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics*

in Operating Systems (New York, NY, USA, 2021), HotOS '21, Association for Computing Machinery, p. 128–135.

[87] ZHOU, J., XU, M., SHRAER, A., NAMASIVAYAM, B., MILLER, A., TSCHANNEN, E., ATHERTON, S., BEAMON, A. J., SEARS, R., LEACH, J., ROSENTHAL, D., DONG, X., WILSON, W., COLLINS, B., SCHERER, D., GRIESER, A., LIU, Y., MOORE, A., MUPPANA, B., SU, X., AND YADAV, V. Foundationdb: A distributed unbundled transactional key value store. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* (2021), ACM, pp. 2653–2666.