# Optimizing Distributed Transactions via Modern AI, Storage and Networking Technologies

Tamer Eldeeb

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2025

© 2025

Tamer Eldeeb

All Rights Reserved

## Abstract

Optimizing Distributed Transactions via Modern AI, Storage and Networking Technologies

Tamer Eldeeb

Distributed ACID transactions, once declared as hopelessly unscalable and unnecessary, are back by popular developer demand. Unfortunately, designing on-disk database systems that support distributed transactions remains quite challenging. Designers of such systems typically face a difficult choice. They can either use an expensive commit protocol like two-phase commit (2PC) to guarantee atomicity, and suffer from slow distributed transactions, or forgo 2PC, which leads to weaker semantics, limitations to the programming model, or constrained scalability, making the system less general. Therefore, there is a trade-off between speed and generality in distributed transactions database systems.

This thesis posits that it is time to revisit that trade-off. We argue that modern developments in AI, storage, and networking unlock the potential of database system designs that offer fast and general distributed transactions. We tackle the problem from different angles. First, we leverage low-latency networking, storage hardware and system software to directly reduce the cost of the commit protocol. Second, when such a low latency stack is unavailable, we design algorithms to mask latency and maintain high throughput in the face of slow 2PC. Finally, we explore applying Reinforcement Learning to the sharding problem so that we reduce the number of distributed transactions the system has to execute without sacrificing other important objectives.

# Table of Contents

Acknowledgments . . . x

Dedication . . . xii

Chapter 1: Introduction . . . 1

1.1 Background . . . 2

1.1.1 Shared-nothing Distributed Database System Architectures . . . 2

1.1.2 Overview of 2PC . . . 2

1.1.3 Other Distributed Database System Architectures . . . 4

1.2 Related work . . . 5

1.2.1 Avoiding 2PC via Sharding . . . 5

1.2.2 Deterministic Execution . . . 5

1.2.3 Early Lock Release . . . 6

1.3 Contributions . . . 6

Chapter 2: Optimizing 2PC in Chardonnay . . . 7

2.1 Requirements . . . 10

2.2 Measuring Contention Footprint . . . 11

2.3 Architecture . . . 13

2.3.1 Epoch Service . . . 14

2.3.2 KV Service . . . 15
2.3.3 Transaction State Store . . . 16
2.3.4 Client . . . 17
2.4 Snapshots . . . 18
2.4.1 Versioning . . . 18
2.4.2 Read Algorithm . . . 19
2.4.3 Garbage Collection . . . 20
2.5 Prefetching . . . 21
2.5.1 API . . . 22
2.5.2 Semantics . . . 23
2.5.3 Design . . . 23
2.5.4 Handling Resource Contention . . . 24
2.6 Deadlock Avoidance . . . 24
2.7 Evaluation . . . 26
2.7.1 Contention Microbenchmark . . . 26
2.7.2 Scalability . . . 29
2.7.3 Snapshot Read Latency . . . 31
2.7.4 Range Reads . . . 31

Chapter 3: Chablis: Extending Chardonnay to Multiple Regions . . . 33
3.1 Introduction . . . 33
3.2 Challenge . . . 34
3.3 Chablis Overview . . . 35

3.3.1 Global Epoch Service . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.3.2 Single-Region Read-Write Transactions . . . . . . . . . . . . . . . . . . . 37

3.3.3 Multi-Region Read-Write Transactions . . . . . . . . . . . . . . . . . . . 38

3.4 Snapshot Read . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.4.1 Global Record Versioning . . . . . . . . . . . . . . . . . . . . . . . 39

3.4.2 Multi-Region Read Algorithm . . . . . . . . . . . . . . . . . . . . . 39

3.4.3 Multi-Region Snapshot Read Consistency . . . . . . . . . . . . . . . . . 41

3.4.4 Single-Region Snapshots . . . . . . . . . . . . . . . . . . . . . . . . 43

3.4.5 Handling Regional Failure . . . . . . . . . . . . . . . . . . . . . . . 43

3.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

3.5.1 Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

3.5.2 Experiment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

Chapter 4: Masking 2PC Latency in Orleans . . . . . . . . . . . . . . . . . . . . 46

4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

4.1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

4.1.2 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

4.2 Background on Orleans . . . . . . . . . . . . . . . . . . . . . . . . . . 49

4.2.1 Runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.2.2 Grains . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.2.3 Activations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

4.2.4 Persistence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.2.5 Transaction Latency . . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.3 Requirements . . . 53
4.4 Programming Model . . . 54
4.4.1 Transactional Grain . . . 54
4.4.2 Transactional Methods . . . 54
4.4.3 Transactional Storage . . . 55
4.5 Transaction Execution . . . 56
4.5.1 Transaction Context . . . 57
4.5.2 Concurrency Control . . . 58
4.6 Commit Protocol . . . 59
4.6.1 Discussion . . . 60
4.7 Reconnaissance Queries . . . 61
4.7.1 Deadlock Avoidance . . . 62
4.8 Experiments . . . 63
4.8.1 Transaction Overhead . . . 64
4.8.2 Single Grain Microbenchmarks . . . 64
4.8.3 Smallbank Multi-Transfer . . . 65
4.9 Conclusions and Future Work . . . 66
Chapter 5: Effectively Avoiding 2PC with Neuroshard . . . 68
5.1 Introduction . . . 68
5.2 Horizontal Sharding . . . 71
5.2.1 Motivating Example . . . 71
5.2.2 The Sharding Problem: Formal Definitions . . . 72

5.2.3 Fanout minimization . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.2.4 Load Balancing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.3 Overview of Learned Sharding . . . . . . . . . . . . . . . . . . . . . . . . 75
5.3.1 Why Learned Sharding? . . . . . . . . . . . . . . . . . . . . . . . . 75
5.3.2 Deep Reinforcement Learning Primer . . . . . . . . . . . . . . . . . . 76
5.3.3 Sharding Problem Formulation for RL . . . . . . . . . . . . . . . . . . 78
5.3.4 Neuroshard RL Algorithmic Framework . . . . . . . . . . . . . . . . . 80
5.4 Reinforcement Learning Design . . . . . . . . . . . . . . . . . . . . . . . . 81
5.4.1 Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.4.2 Observable State . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.4.3 Actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.4.4 Fanout Reward . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.4.5 Load-balancing Reward . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.5 Neural Network Agent . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.5.1 Agent Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.5.2 Training Procedure . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.6 Learning with Multiple Objectives . . . . . . . . . . . . . . . . . . . . . . 89
5.7 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.7.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.7.2 Cost of Query Fanout . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.7.3 Baselines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.7.4 Multi-objective Microbenchmarks . . . . . . . . . . . . . . . . . . . . 94
5.7.5 Epinions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

5.8 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

5.9 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

# List of Figures


2.1 Contention footprint of YCSB read (left bar) and write (right bar) transactions. % represent the proportion of the data in DRAM. . . . . . . 12

2.2 Transaction Lifetime in Chardonnay. . . . . . . 13

2.3 Simplified Chardonnay Client API . . . . . . . 17

2.4 Contention Microbenchmark Throughput Results . . . . . . . 25

2.5 Abort rates for 10% distributed tx micro-benchmark. Chardonnay's deadlock aborts are 0%. . . . . . . 29

2.6 TPC-C New-Order transaction results. . . . . . . 30

3.1 Two region Chablis deployment. . . . . . . 36

3.2 Single-Region Transaction Lifetime in Chablis. . . . . . . 38

4.1 Actor-Oriented Database System. . . . . . . 50

4.2 Breakdown of time spent in transactions . . . . . . . 52

4.3 Transactional Grain State Interface . . . . . . . 54

4.4 Transactional State Storage Plugin Interface . . . . . . . 55

4.5 Transaction Execution . . . . . . . 57

4.6 Pipelined 2PC with two participants (left) and one participant (right). . . . . . . 58

4.7 Single Grain Throughput . . . . . . . 65

4.8 Smallbank Multi-Transfer Results . . . . . . . 66

5.1 Toy example of the distributed database partitioning problem. . . . . . . . . . . . . . 69

5.2 Toy example represented as a bipartite graph. . . . . . . . . . . . . . . . . . . . . . . . 74

5.3 Policy neural network architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

5.4 System throughput for different kinds of queries. . . . . . . . . . . . . . . . . . . . . 93

5.5 Multi objective micro-benchmark throughput with 2 shards . . . . . . . . . . . . . . 96

5.6 2nd micro-benchmark throughput with 4 shards . . . . . . . . . . . . . . . . . . . . . 98

5.7 Epinions workload throughput with 5 shards . . . . . . . . . . . . . . . . . . . . . . . 99

# List of Tables

2.1 Comparison of representative on-disk distributed database systems. . . . . . . . . . . 8

2.2 Range Read Results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.1 YCSB Regional Latency Results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

4.1 Single Silo Write Throughput. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

## Acknowledgements

First and foremost I would like to thank my primary advisors, Junfeng Yang and Asaf Cidon for their support, mentorship and what I hope will continue to be a life-long friendship. My doctoral journey has been unconventional in more ways than one, not least of which is that the Covid-19 pandemic struck right after I moved to New York. Had it not been for my advisors' open-mindedness, willingness to take a chance on me, and their immense support and trust, I do not believe it would have been possible to make it to the finish line. Junfeng is a world class scientist and a great human being. He taught me how to be a researcher, how to think big and aim for truly impactful research, and how to weather the disappointments and the roller-coaster emotional journey that is the doctoral research life. Asaf is a brilliant researcher and an amazing mentor who moves heaven and earth to ensure his students are successful, including but not limited to: brainstorming and pitching ideas, rewriting papers to make them comprehensible, making intros to other students and researchers, fostering a collaborative environment and managing a happy and productive group of amazing students. One simply cannot wish for better advisors.

I am also deeply indebted to my long-time mentor and collaborator Phil Bernstein. I first met Phil over a decade ago when I walked into his office at Microsoft in Redmond expressing my wish to work on some research projects. To my shock, he agreed and we started working on adding transactions to the Orleans project which would become a success and ship to production, and form the basis of future research that was part of this thesis. I learned a great deal about databases and transactions by collaborating with Phil, but perhaps more than anything else, the fact that I

was taken seriously by a researcher of such stature was what gave me the confidence that perhaps I can succeed as a computer science researcher.

I am grateful to many professors and colleagues who I met at Columbia, and who made my experience quite rich both as collaborators and friends. I thank professors Kostis Kaffes and Kenneth Ross for serving on my thesis defense committee, and Kostis for being my secondary advisor. I really enjoyed working with Yannis Zarkadas, Kelly Kostopoulou, Haoyu Li, Vahab Jabrayilov, Xincheng Xie, Haonan Wang, Zhengneng Chen, and Ashwini Raina on various projects throughout my time at Colubmia. I am also grateful to the people outside Columbia who provided valuable feedback on my work, including Ryan Stutsman, Jinyang Li, Irene Zhang and Kyle Raftogianis.

Finally, I have to acknowledge that none of the successes I had in this endeavor, or in life in general, would have been possible without the support and love of my dear family. To Emad, Noha, Salma, Hamza and Laila: Thank you and I love you.

والْحَمْدُ لِله رَبِ الْعَالَمِينَ

# Dedication

Dedicated to the memory of Ayman Ali Abdo.

# Chapter 1: Introduction

Many applications need databases that are distributed across many machines for various reasons such as scalability, high availability and fault tolerance. Exposing the distribution to the programmers makes developing applications much harder, hence a holy grail of distributed database management systems (DBMS) is to provide an abstraction of a single-server database that can execute ACID transactions with high throughput, while maintaining high availability. More formally, the system should be high performance, scalable and highly available while having the following properties:

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

The combination of these two properties is known as *strict serializability* [73] or *external consistency* [40].

Recent work (e.g., [34, 49, 81, 130, 153, 154, 158]) shows that ACID distributed transactions with strong isolation and consistency semantics can be made efficient and scalable on memory-resident databases in modern datacenters. On the other hand, distributed transactions on scale-out disk-resident databases are commonly believed to be very expensive to support [13, 72, 141, 153]. Unfortunately, DRAM costs significantly more than SSDs and the cost difference is expected to grow. Therefore, due to their significantly lower cost, many applications use distributed databases [40, 139, 152, 164], which store their data on disk-based storage engines such as RocksDB [48, 105, 124] or LevelDB [90]. The classic architecture for such systems [126], popularized by System R* [112], 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) [87] to ensure atomicity for distributed ACID transactions. However, distributed transactions within these systems traditionally suffer from significant performance limitations [13, 41, 72, 93, 99, 141, 153].

In this thesis, we demonstrate that these limitations can now be lifted, and that fast and general distributed on-disk database systems can be built by leveraging modern advances from AI, networking and storage.

## 1.1 Background

In this section we review the challenges facing classic on-disk distributed transaction systems, and the current approaches to deal with them along with a discussion of their disadvantages.

### 1.1.1 Shared-nothing Distributed Database System Architectures

A classic architecture [126] popularized by System R* [112] is to horizontally shard the database across a collection of shared-nothing machines and use a distributed commit protocol such as two-phase commit [87] (2PC) to ensure atomicity for distributed ACID transactions. Spanner [40], and CockroachDB [139] are prominent modern examples of systems that utilize shared-nothing architecture. Distributed transactions within this architecture traditionally suffer from significant limitations, as we discuss in the next section.

### 1.1.2 Overview of 2PC

Two-phase commit (2PC) is a classic commit protocol with many variants [87]. 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 persists all of the transaction's 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.

### The Penalty of 2PC

The classic shared nothing architecture utilizing 2PC had many problems:

- **High overhead**. At least two network round trips and two synchronous log writes are required per transaction in 2PC, which incurs significant IO overhead, and traditionally, high CPU usage by the TCP/IP stack [153].
- **Blocking**. The failure of the coordinator at inopportune moments prevents participants from resolving their transactions. Non-blocking commit protocols [133] have been proposed, but they make impractical assumptions and have even higher overhead, so they are not widely adopted in practice. Instead, the problem can be addressed by replicating the coordinator state for availability [13, 40, 68].
- **Contention**. The coordination necessary to guarantee isolation can be very expensive, and can significantly decrease concurrency which leads to performance degradation as well as high abort rates [11]. This is not unique to distributed transactions, and in fact many single-node database systems run with lower isolation levels than serializability precisely to mitigate this issue [11]. But 2PC can drastically exacerbate contention, particularly for short transactions common in OLTP workloads, because of its high latency relative to the time it takes to execute the transaction logic [141]. The impact of contention is easy to see in locking-based concurrency control schemes such as two-phase locking [57], but other schemes such as optimistic concurrency control (OCC) are also not immune, and can in fact perform worse under high contention [71, 92, 154].

Because of these challenges, many scale-out disk-resident systems avoid providing any multi-key ACID transaction support at all [32], or limit it to transactions accessing keys within a single

machine or partition [29, 119]. Other systems opt to offer weaker semantics [12, 43, 95]. Nevertheless, because of strong developer demand [10], many modern DBMSes now fully support distributed ACID transactions [40, 58, 139, 152]. Systems can overcome the blocking problem in practice by using a consensus protocol such as Paxos [86] or Raft [117] to replicate the transaction coordinator state for high availability [13, 40, 139]. And modern datacenter network stacks make the first problem somewhat less of an issue [153]. However, this does not address the contention problem (which can in fact be exacerbated because of the added latency due to consensus rounds). Therefore, distributed transactions in modern System R*-style systems are *slow*. They are a lot more expensive than local transactions and often need an effort to cluster the application's data to minimize distributed transactions [41, 54, 56, 118, 138].

To overcome 2PC's slow performance, recent research proposed alternatives that offer support for distributed transactions, but forgo 2PC and sacrifice *generality* in one or more ways, e.g. by offering weaker semantics [12, 95, 146, 147], restricting the programming model [58, 141] or employing an architecture that limits system scalability [8, 23, 88, 163].

### 1.1.3 Other Distributed Database System Architectures

The focus of this thesis is the shared-nothing architecture, as it is the one with no fundamental limits to scaling. Nonetheless, it is useful to review other popular architectures along with their limitations.

**Shared Disk**. Shared-disk [8] is another classic DBMS architecture [31, 77] that has become popular in recent years in systems such as Amazon Aurora [146, 147], Socrates [8], and Google's AlloyDB [5]. 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. Additionally, for this mode to work well the workload should be easily partitionable with little cross-partition activity.

**Shared Log**. Hyder [23] and Tango [14] 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.

## 1.2 Related work

### 1.2.1 Avoiding 2PC via Sharding

Schism [41] pioneered modelling database sharding and replication as a graph partitioning problem with reducing the number of distributed transactions (and thus, 2PC) the system needs to execute as the the main goal. The general approach works by collecting a trace of past accesses to the database, model it as a (hyper)graph that links the rows accessed together in a query, and then compute the row-to-shard assignments by solving a graph partitioning problem with the objective of minimizing fanout. SWORD [118] builds up on SCHISM's approach but uses coarser granularity and applies hypergraph partitioning instead.

This approach can be fairly effective in practice. Unfortunately, it can come at the expense of other important system performance objectives such as load balancing [118]. On the other hand, commonly-used random, hash, range and round-robin partitioning [45] are good at balancing load but ignore fanout minimization completely.

### 1.2.2 Deterministic Execution

Deterministic execution has been explored as an alternative to distributed commit in systems such as Calvin [141] and Aria [98]. The key idea behind deterministic database system design is to agree on the order of transactions prior to execution, typically by appending transaction inputs to a log then having all replicas execute the transactions in a deterministic fashion so that they all reach the same state. A major benefit of using determinism is eliminating transaction aborts due to deadlocks [122]. On the other hand, deterministic database systems 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. A detailed discussion of the pros and cons of determinism can be found at Thompson et. al. [122].

### 1.2.3 Early Lock Release

Early Lock Release [47, 62] is a technique that involves releasing the transaction locks after it finishes execution, but before its writes are durably flushed to the transaction log. This can be quite beneficial to transaction throughput when combined with the well-known group commit optimization, particularly in the cases where transaction execution times are much smaller than persistent storage IO times.

As discussed earlier, the latency of 2PC is a major part of its performance challenge. Applying early lock release to 2PC provides the opportunity to apply a distributed form of the group commit optimization to mask the latency of 2PC, but comes with many challenges not present in single-node systems [66].

## 1.3 Contributions

In this thesis, we tackle the problems of distributed transactions for on-disk databases, and specifically the classic problems of the shared-nothing architecture with two-phase commit from different angles. First, we leverage low-latency networking, storage hardware and system software to directly reduce the cost of the commit protocol and build a system that can take advantage of modern datacenter environments. Second, when such a low latency stack is unavailable, we propose algorithms to mask latency and maintain high throughput in the face of slow 2PC. Finally, we explore applying Reinforcement Learning to the sharding problem so that we reduce the number of distributed transactions the system has to execute without sacrificing other important objectives.

# Chapter 2: Optimizing 2PC in Chardonnay

The performance challenges of supporting distributed transactions on scale-out on-disk systems that we discussed in the previous chapter have led many to avoid providing any multi-key ACID transaction support at all [32, 43], or limit it to transactions accessing keys within a single machine or partition [29, 119]. Other systems offer support for distributed transactions, but forgo 2PC and sacrifice *generality* in one or more ways, e.g., by offering weaker semantics [12, 95, 146, 147], restricting the programming model [141], or employing an architecture that limits the system scalability [8, 88, 163]. Nevertheless, due to strong developer demand [10], many popular SQL DBMSes now support general distributed ACID transactions [40, 139, 152], despite them being a lot slower than local transactions. Table 2.1 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 traditionally has been due to the high latency of RPCs and flushing log entries to disk. Fortunately, this no longer needs to be the case. Modern datacenter networks are fast [26], and systems such as eRPC [80] have demonstrated that RPCs can be run at single-digit microsecond latency within the datacenter even without using RDMA. Moreover, NVMe devices based on low-latency NAND [144] or 3DXpoint [1] also provide single digit microsecond latencies [76, 142, 144], making them ideal for database log storage.[1] Furthermore, there has been a large body of recent work [79, 127, 157, 160, 161] that focuses on cutting latency across the Linux software stack.

We believe that due to these developments, it is time to revisit the assumptions inherent in many scale-out on-disk database system designs. However, simply replacing the RPC framework

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

| System | Serializabile | Linearizable | General API | Distributed TX | Contention |
|---|---|---|---|---|---|
| Spanner [40] | ✓ | ✓ | ✓ | Slow | X |
| Calvin [141] | ✓ | ✓ | X | Fast | ✓ |
| FoundationDB [163] | ✓ | ✓ | ✓ | Fast | X |
| Hyder [23] | ✓ | X | ✓ | N/A | X |
| Aurora (Multi-Master) [146] | X | X | ✓ | N/A | Partitionable |
| Chardonnay | ✓ | ✓ | ✓ | Fast | ✓ |

Table 2.1: Comparison of representative on-disk distributed database systems.

and log storage by low latency alternatives in existing systems is not sufficient. Indeed, as we show in §2.2, even eliminating the entire latency of the commit protocol would not be enough to achieve performance comparable to in-memory systems for high contention workloads, since transactions may still need to hold locks while fetching cold items from storage. At this point the bottleneck moves to reading the data from disk, since reading data from typical SSD storage can be an order of magnitude or more slower than the network.

We present Chardonnay [55], 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 (we show how to extend it for geo-partitioned databases that are deployed across multiple datacenters in §3.1). 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 [139]). Chardonnay uses the classic shared-nothing architecture, and uses strict two-phase locking (2PL) [57] to guarantee strict serializability [73] 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 [14, 163], 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 (§3.4). 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:

**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.

**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 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 [58, 140, 141], Chardonnay's prefetching mechanism works for scans, and neither requires changes to the user code, nor incurs significant additional latency or

contention.

**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 §2.7, 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 [143]. 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 [11]. Our results show that Chardonnay's techniques can be useful for them too.

## 2.1 Requirements

Chardonnay's objective is to provide *fast* and *general* transactions for on-disk databases. In this section we define what we mean by fast and general. 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.

## 2.2 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 [67, 141]. 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 [129]. Indeed, data contention is a bottleneck that hinders truly scalable transaction processing, even in RDMA-enabled in-memory distributed database systems [154], and on multi-core single-node systems [120].

Following the terminology of Calvin [141], 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 [38] 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 [65]. Both the write-ahead log (WAL) and the database are placed on a directly attached SSD.
- **Baseline-Fast**. Uses (fast) eRPC (with FlatBuffers [59] 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

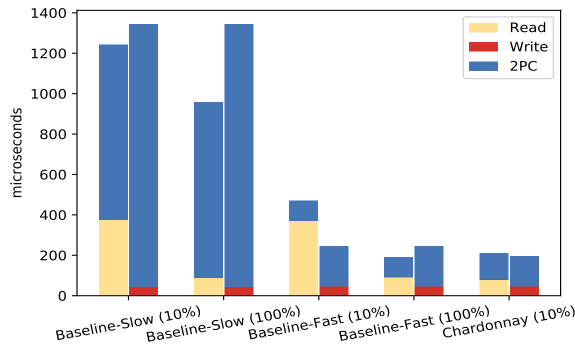



Figure 2.1: Contention footprint of YCSB read (left bar) and write (right bar) transactions. % represent the proportion of the data in DRAM.

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.

We show how the average latency of read and write operations each contribute to the contention footprint in Figure 2.1. 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.

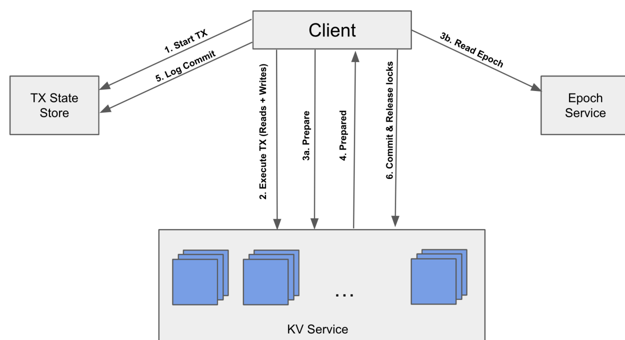



Figure 2.2: Transaction Lifetime in Chardonnay.

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.

## 2.3 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 (§3.4). 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 [40, 139, 152].

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.

4. **Client Library**. Applications link this library to access Chardonnay. It is the 2PC coordinator, and provides APIs (Figure 2.3) for executing transactions.

Figure 2.2 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.

### 2.3.1 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 §3.4. 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 [80]. 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.

### 2.3.2 KV Service

The key universe is partitioned into disjoint contiguous subsets called **ranges**. Each range is assigned to a number of range servers (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 [89, 97]. All reads and writes go through the leader.

To simplify the description in this chapter 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 [29, 32, 40, 139], which we leave for future work.

#### 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 §2.3.4 we explain how we use it to validate transaction locks, and later in §3.4 we describe its role in the correctness of our lock-free snapshot reads.

### 2.3.3 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 [112], 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 enables a 2PC latency optimization.

Recall that the client in Chardonnay acts as the 2PC coordinator. If the client crashes after

```
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 2.3: Simplified Chardonnay Client API

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.

### 2.3.4 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 (§3.4). 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.

## 2.4 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.

### 2.4.1 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.

#### 2.4.2 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 $T_2$.

**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 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. □

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 §2.3.4 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.

### 2.4.3 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).

### 2.5 Prefetching

Our experiments in §2.2 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 [21]. 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 [141]. This contradicts Chardonnay's goal of a general programming model (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 (§2.5.3). 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 [141]; 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 [141], 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 when to do prefetching based on the characteristics of the workload.

### 2.5.1 API

The API shown in Figure 2.3 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.

### 2.5.2 Semantics

The dry run query runs under snapshot isolation using our snapshot read mechanism that we described in §3.4, and can be configured to be strictly serializable if desired. Running under a lower isolation level such as *read committed* [17] 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.

### 2.5.3 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.

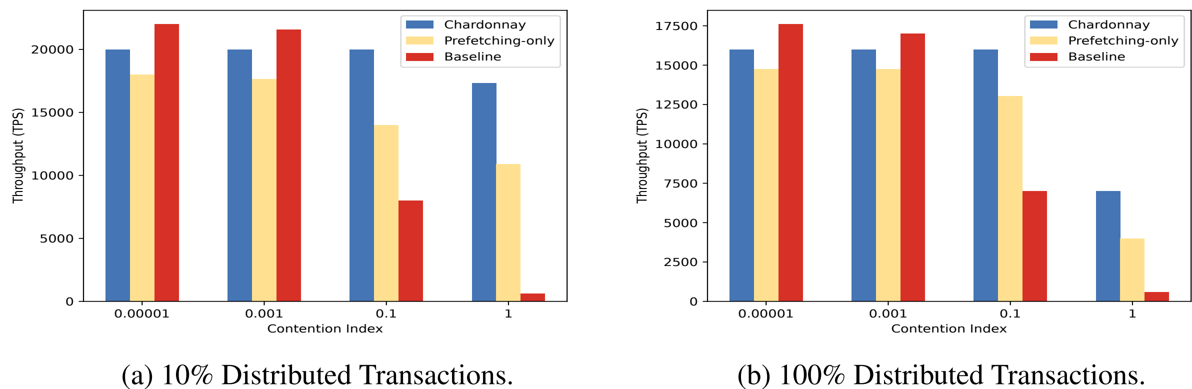
It is possible that a request to pin a record cannot be satisfied because the range leader has run out of memory. In this case the dry run query could be delayed until memory frees up, or just be aborted. This serves as effective admission control prior to acquiring any locks. Some care is needed to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

### 2.5.4 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.

## 2.6 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




(a) 10% Distributed Transactions.

(b) 100% Distributed Transactions.

Figure 2.4: Contention Microbenchmark Throughput Results

Wound-Wait [125], 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 [122] via detecting cycles in the wait-for graph [63, 135] 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 |read_set ∪ write_set| round-trips to the contention footprint to acquire the locks. Instead, the client uses an approach similar to RPC Chains [134], 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 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.

## 2.7 Evaluation

In this section we study how Chardonnay performs under contention (§2.7.1), its scalability (2.7.2), and its snapshot read performance (§2.7.3). To evaluate contention, we use a benchmark used by Calvin [141], 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 [38]. 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.

### 2.7.1 Contention Microbenchmark

We use a benchmark introduced in Calvin [141] 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 (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 [139] 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 2.4 and 2.5.

**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 2.5. 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.

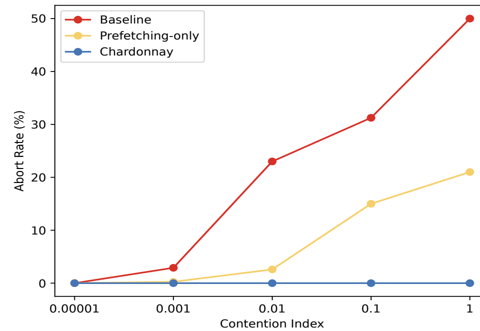



Figure 2.5: Abort rates for 10% distributed tx micro-benchmark. Chardonnay's deadlock aborts are 0%.

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 §4.7.1.

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 [41, 54, 56, 118, 138]. 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.

### 2.7.2 Scalability

The scalability of the System R*-style shared-nothing architecture is well established [151], but Chardonnay introduces the *read-epoch* operation during each transaction's 2PC. Therefore, we

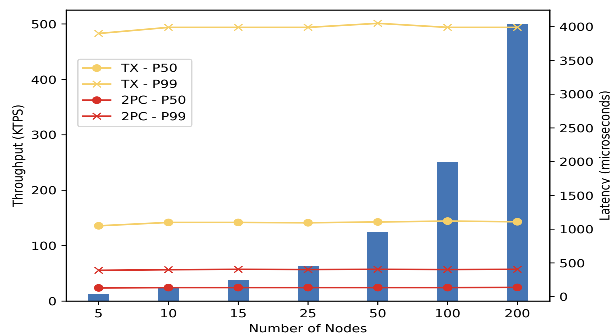



Figure 2.6: TPC-C New-Order transaction results.

need to ensure that the epoch service can keep up with an increasing scale.

**TPC-C New-Order.** Similar to prior work [141], 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 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 2.6, 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 [141], *without Calvin's significant programming model restrictions*. 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.

| | Chardonnay | Baseline |
|---|---|---|
| Insert TPS | 914 | 197 |

Table 2.2: Range Read Results.

**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.

### 2.7.3 Snapshot Read Latency

We use YCSB with 50% write and 50% read to evaluate snapshot read latency, using a setup similar to §2.2. 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.

### 2.7.4 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 §2.7.1. The results are in Table 2.2. 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.

# Chapter 3: Chablis: Extending Chardonnay to Multiple Regions

## 3.1 Introduction

The design of Chardonnay targets single datacenter deployments. However, 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 [121, 139]. Many workloads have locality in their access patterns [121, 139], 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.

Unfortunately, offering strong semantics like strict serializability in a geo-distributed setting has many well-known challenges [43, 64, 72, 96, 121], leading many popular systems to either disallow multi-region transactions completely [39], or offer weaker semantics instead [12, 95, 96, 108]. However, these weaker semantics expose anomalies to programmers making it harder to develop applications [13, 40, 121], and could lead to security vulnerabilities [149]. As a result, developer demand for strict serializability increased; e.g., Google's Spanner [40] is widely used both within Google and as a cloud offering, and has inspired many open source products [139, 152].

Existing geo-distributed systems that offer strong semantics [40, 85, 114, 115, 116, 121, 150, 156, 159] compromise either in speed or generality. Many systems [85, 115, 156] incur at least

one cross-region round-trip for **every** write, which slows all writes in the system. Spanner [40] 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 [121] and Detock [116] 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.

In this section, we show how to extend Chardonnay 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 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.

To show the practicality of the technique, we implemented it in Chablis [52], 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.

### 3.2 Challenge

Chardonnay makes each transaction read the (monotonically increasing) epoch after it finished

[1] Systems like CockroachDB [139] offer weaker consistency to avoid this dependency.

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.

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

### 3.3 Chablis Overview

Chablis has three components, illustrated in Figure 3.1:

1. **Client Library**. Applications link this library to access Chablis. It is the 2PC coordinator, and provides APIs (Figure 2.3) 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 (§3.3.1). 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 pair) 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

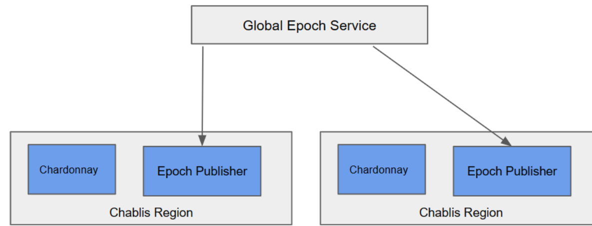



Figure 3.1: Two region Chablis deployment.

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 [139] in other regions that can serve older snapshots.

### 3.3.1 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 §3.4, we explain how Chablis clients can achieve linearizability of snapshot reads given this weaker invariant.

### 3.3.2 Single-Region Read-Write Transactions

Figure 3.2 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 (§3.4).

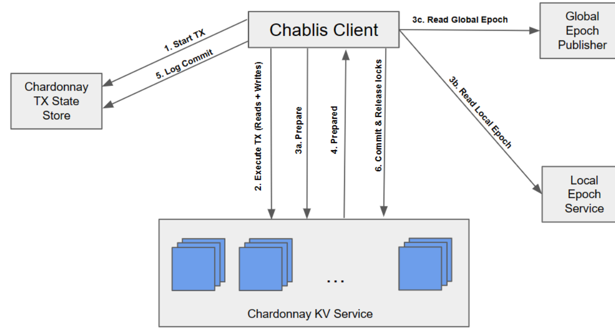



Figure 3.2: Single-Region Transaction Lifetime in Chablis.

### 3.3.3 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 instead, executing their reads using the snapshot read algorithm described in §3.4.

## 3.4 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.

### 3.4.1 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 (step 3c in Figure 3.2), since that value could be one less than the true epoch as we discussed in §3.3.1. 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.

### 3.4.2 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 §3.4.1. 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 ⟨k, VID⟩ 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 ← 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 ← 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 ← 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] (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 (see proof in the previous chapter on Chardonnay). 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.

### 3.4.3 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* [139], which is weaker than strict serializability. This

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

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 §3.4.3.

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.

#### 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 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.

### 3.4.4 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 [55]. 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.

### 3.4.5 Handling Regional Failure

Chablis is a CP system in the CAP theorem [60] sense. Modern datacenter networks make arbitrary partitions exceedingly rare [27] 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.

| | Read | Write |
|---|---|---|
| P50 (µs) | 214 | 199 |

Table 3.1: YCSB Regional Latency Results.

## 3.5 Evaluation

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

### 3.5.1 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.

### 3.5.2 Experiment

We run YCSB-A [38] 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 trans-

actions 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.

#### 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 3.1

#### Comparison to Cloud Spanner

Cloud Spanner [37] 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 3.1). 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.

# Chapter 4: Masking 2PC Latency in Orleans

## 4.1 Introduction

In the previous chapter, we showed how to leverage modern low-latency networking and storage technologies to drastically cut the latency of 2PC. Unfortunately, these technologies are not always available. In this chapter, we will describe how we designed efficient distributed ACID transactions mechanism in one context in which we could not use a fast RPC stack and low-latency storage: the Orleans [28, 109] virtual actor system. The basis of this chapter is published in [53].

### 4.1.1 Motivation

Many modern cloud services have a 3-tier architecture with a stateless front-end, a stateful middle-tier that implements business logic, and a storage layer. The stateful middle-tier is needed due to heavy CPU and memory requirements to execute the business logic, which does much more than simply read or write the database. For example, apps often manage a lot of state in the middle-tier, such as a knowledge base or image cache. Some of this state needs to be read and written at high rates. These apps may also perform heavy computation, such as rendering images or computing over large graphs. Such requirements make it infeasible to embed the application logic as stored procedures in the storage layer [25]. The architecture also allows computation and storage to scale independently.

The actor model [4] has become a popular choice for building stateful middle-tier applications in the modern cloud, especially interactive ones such as games, social networks, Internet of Things, and telemetry [9, 18]. Actors are single-threaded objects that do not share memory and interact only via asynchronous message passing. The single-threaded nature of actors simplifies their implementation, and applications are typically made of many actors, which are natural units

of scaling that are spread over many servers for scalability. Actors allow building a stateful middle tier with data locality and the semantic and consistency benefits of encapsulated entities via application-specific operations [18].

Orleans [28, 109] is an actor-based platform targeting stateful middle-tier applications with a primary focus on scalability and programmability. It simplifies the process of writing .NET scalable stateful middle-tier services, making it accessible to developers who are not necessarily distributed system experts. Orleans invented the abstraction of virtual actors [18], where actors are transparently loaded on demand, like pages in a virtual memory system. This solves a number of the complex distributed systems problems, such as reliability and distributed resource management, liberating the developers from dealing with those concerns. The Orleans runtime implements the virtual actor model, enabling applications to attain high performance, reliability and scalability. Over time, Orleans added support for automating the process of storing actor state durably to the programmer's choice of cloud storage systems.

By viewing an Orleans application as a collection of stateful actors, one can think of it as an actor-oriented database (AODB) [19, 20]. The distinguishing features of an AODB are that it scales out elastically to hundreds of servers, can use a variety of cloud storage services, and is compatible with the actor framework's programming model. Since application developers want to avoid being locked into a specific storage service, such a database must be able to use a wide variety of storage systems, such as page servers, BLOB servers, key-value stores, JSON stores, and SQL databases. An actor-oriented database needs to implement its own database abstractions, to compensate for the lack of such abstractions in the storage system and to ensure it integrates smoothly with the actor programming model. This perspective has led to the evolution of the system to add support for database abstractions as first class concepts, such as geo-replication [24] and indexing [20], as depicted in Figure 4.1. Prior work also investigated using the actor model for query serving [84].

Applications often need to perform logically atomic operations that span multiple actors. Because actors do not share memory, adding general support for such multi-actor operations with isolation and fault tolerance guarantees requires an ACID transaction mechanism [21]. Since ac-

tors in Orleans are randomly distributed across many servers for scalability and availability, most transactions that access multiple actors will access multiple servers, and hence must be distributed. On top of the standard challenges involved in supporting distributed transactions which we previously discussed, additional difficulties arise from implications of the virtual actor model and requirements from Orleans users (which we describe in detail in §4.3). However, opting not to offer distributed transactions would put the burden on developers to use ad-hoc methods to obtain cross-actor consistency, which is hard to do well. ACID transactions are a key database abstraction, and supporting them as a first-class concept in Orleans has been a major step in its evolution into a fully-fledged actor-oriented database system.

#### 4.1.2 Overview

In this chapter we describe the design and implementation of ACID transactions in Orleans. We utilize a classic design that combines two-phase locking [57] (2PL) for serializable isolation, with two-phase commit [87] (2PC) for atomicity, to implement distributed transactions, but with novel and unique twists. Our design does not introduce any centralized components to Orleans such as a shared log or an independent transaction manager system. Instead, all durable transaction state is maintained in decentralized, per-actor cloud storage accessible only via transactional storage drivers, allowing developers to pick any cloud storage they prefer.

The high latency of cloud storage presents many performance challenges and would typically limit transaction throughput. We introduce two main techniques to mask the latency of cloud storage. First, we apply a distributed form of *early lock release* [47, 62] to 2PC by allowing a transaction to release locks at the start of phase-one of the 2PC protocol. After a transaction T finishes executing, it will not acquire more locks, so holding locks after this point has no value from a 2PL perspective. By releasing its locks at the start of 2PC, T avoids blocking other transactions that access T's writeset while T is executing its high-latency commit process. However, since T releases write locks before it commits, subsequent transactions can read "dirty" (i.e., uncommitted) data that will be invalid if T aborts. To avoid this inconsistency, a transaction keeps track of its

dependencies on uncommitted transactions, and can only commit if and when all its dependencies commit.

Second, we utilize *reconnaissance queries* [141], which run transaction logic in a low isolation, dry run mode to prefetch all of the actor state in main memory prior to the actual transaction, so that transaction execution does not block waiting for slow reads from cloud storage while holding locks. Similar to our work in Chardonnay, the reconnaissance query also collects the identities of all the actors involved in a transaction so that lock acquisition requests can be ordered to avoid deadlocks.

While this work focuses on transactions in an actor-oriented database, we think these techniques are generally applicable in any situation where 2PC commit is high. A summary of the techniques described in this chapter:

1. Transactional extensions to the Orleans programming model (§4.4).

2. Transactional extensions to the Orleans runtime (§4.5).

3. Pipelined commit protocol and distributed early lock release (§4.6).

4. Reconnaissance queries for prefetching and deadlock avoidance (§4.7).

We also describe our experience and lessons learned from developing a prototype solution and taking it all the way to production in §4.3, including how feedback from users influenced the design and led to revisiting important aspects of the system.

## 4.2 Background on Orleans

Orleans is a framework for writing actors, as well as a platform that provides a set of runtime services to execute that actor code. In this section we describe the parts of Orleans that are necessary to explain how we added ACID transactions to it. More details can be found in the Orleans documentation [109].

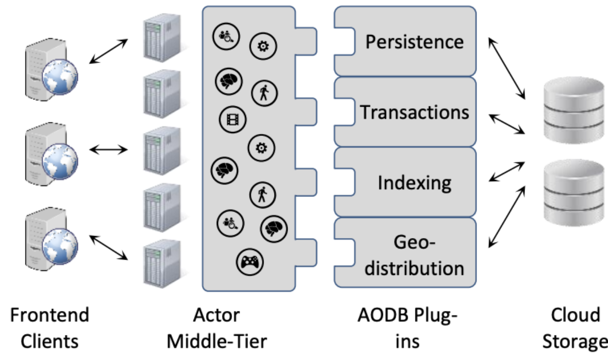



Figure 4.1: Actor-Oriented Database System.

### 4.2.1 Runtime

An Orleans cluster is a set of servers running an identical software service called a *silo*. The Orleans Runtime is a set of subsystems that run on each silo of a cluster.

### 4.2.2 Grains

Actors in Orleans are known as *grains*. Each grain has a location-transparent identity, called its *key*, which is the only way to reference it. Grains cannot share state.

Like regular .NET types, grain types are defined using interfaces and classes. A grain's public interface can have only async methods, and grains only communicate via these asynchronous method calls; a grain can perform a system call to the Orleans runtime to obtain a *reference* to another grain using the target grain's *key*, which is one of its member attributes. It then can use the reference to call any of the async methods on the target grain's interface. The reference returned by the runtime is in effect a *proxy* for the called grain, and allows the runtime to intercept all communication between grains in the system.

A method call immediately returns a promise, after which the caller can continue executing. It can also choose to wait for fulfillment of the promise (i.e., wait for the method call to finish executing and return) using the standard .NET *await* mechanism. Under the covers, this interaction is realized by messages in each direction.

Grains are single-threaded, and normally are non-reentrant. That is, a method call must execute

to completion before the next call is processed. Optionally, a grain can be reentrant. In this case, the steps of method calls can be interleaved. However, even in this case, only one method call is allowed to be actively executing inside the grain at any given time.

### 4.2.3 Activations

Since grains cannot share state and can only be referenced via the location transparent key, the Orleans runtime is able to place any grain on any server in the cluster. Typically, it distributes grains randomly across servers to minimize the chance that any server is a bottleneck, though users can customize grain placement policies using plug-ins.

If a grain is not currently running when one of its methods is invoked, the Orleans runtime *activates* the grain, which involves choosing a server on which to execute the grain and executing the grain's constructor. It then performs the method call. It retains a reference to the grain in its distributed fault-tolerant grain directory so that future invocations can be directed to it. If a grain is idle for too long, the Orleans runtime *deactivates* it by calling the grain's destructor and releasing its resources. Since, this model of activate-on-demand is very similar to the demand-paging model of virtual memory, Orleans calls it the Virtual Actor Model [18].

Notice that there is no notion of *creating* a grain in the Orleans programming model. Grains are assumed to always exist, and are instantiated only when referenced.

The mapping of grains to servers is dynamic. Each time a grain is activated, it may (and often does) execute on a different server than its previous activation.

Grains are fault tolerant. If a server fails, Orleans detects the failure and updates its grain directory accordingly. The next invocation of a grain that died on the failed server causes that grain to be re-activated on another server, just like any invocation of an inactive grain.

The grain directory is implemented as a decentralized, distributed hash table where each server in the cluster is responsible for a portion of the directory. The system strives to ensure there is at most one active instance of a grain at any point in time. However, this can be violated during periods of server failures, cluster reconfiguration, or network partitions, which has implications for

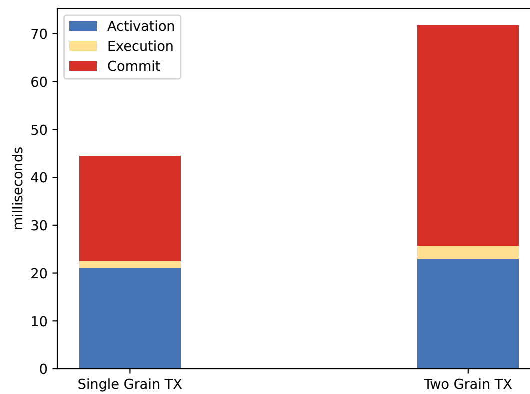



Figure 4.2: Breakdown of time spent in transactions

the correctness of our transaction implementation, which we discuss in §4.6.

### 4.2.4 Persistence

Orleans offers a simple declarative model of persistence, where a grain type identifies its persistent properties. Orleans maps those properties to persistent storage via a **storage provider** plug-in. The app specifies the storage provider (and hence the storage system) to use via a configuration attribute. Orleans uses the storage provider to populate a grain's state when the grain is activated. A grain can call WriteStateAsync to save its state at any time, e.g., just before returning from a method call that modifies its state or just before it is deactivated.

This approach to persistence decouples actor implementation from its storage. Developers can override this declarative persistence model with their own mechanism. For example, the developer can write custom code in the grain's constructor to initialize the grain state from any source, and can include code to save the grain's state in any method.

### 4.2.5 Transaction Latency

Figure 4.2 shows the breakdown of where the time is spent in the execution of a simple transaction that writes one or two randomly selected grains, whose storage is backed by Azure storage. As shown on the figure, the time to load the state and execute the commit protocol is much larger

than the time taken to execute the transaction logic itself.

### 4.3 Requirements

Here we discuss a set of requirements that we collected from Orleans users within and outside Microsoft.

First, to ensure the programming model for transactions is natural to Orleans users, transactions must be opt-in. They should affect only the programming model and performance of applications that use them.

Second, users want the ability to choose from a wide variety of cloud storage solutions. Hence all transaction storage must be external and pluggable.

Third, as reported in prior work [55, 141] and by our users, most workloads have low contention most of the time, but many workloads sometimes have a few very hot grains. Hence the transactions design needs to handle both high and low contention cases well.

We built an initial prototype satisfying these requirements [51]. However, when the Orleans team embarked on incorporating the prototype into their product, they identified additional requirements. First, they wanted an application opt into transaction functionality by using composition and dependency injection, rather than by extending a base class, as was required in the prototype. This allows better composability with other Orleans features that is difficult to achieve with an inheritance-based model.

Second, the prototype had a disadvantage where even single-grain transactions have to go through a two-phase commit process, and they were keen to eliminate this overhead both for performance reasons, and for integration with external one-phase systems that do not support 2PC.

Third, transaction aborts due to deadlock timeouts were identified as a major source of performance problems. The possibility of them occurring was often a surprise for users since locking is not explicit in the programming model. Users were keen for a way to avoid or reduce deadlocks.

Finally, they wanted there to be no additional components to deploy beyond that of the existing Orleans setup, which deploys one service (the Silo) per server. Users were quite unwilling to take

```
public interface ITransactionalState<TState>
   where TState : class, new()
{
   Task<TResult> PerformRead<TResult>(
      Func<TState, TResult> readFunction);

   Task<TResult> PerformUpdate<TResult>(
      Func<TState, TResult> updateFunction);
}
```

Figure 4.3: Transactional Grain State Interface

on the many complications of the deployment, versioning and rollout story that would be necessary due to additional components. This rules out architectures that include a dedicated transaction manager service or a centralized sequencer [14, 55].

## 4.4 Programming Model

In this section we describe the extensions we added to the Orleans programming model to support transactions.

### 4.4.1 Transactional Grain

A transactional grain is a stateful grain whose state is protected by ACID transactions. Any grain type can become transactional by declaring a field of type **ITransactionalState** in the class implementing the grain, which is a wrapper providing transactional read and write access to the grain state. The interface of ITransactionalState is shown in Figure 4.3.

### 4.4.2 Transactional Methods

Transactions in Orleans are bracketed by method tags, similar to the programming model of Java EE or .NET's COM+. A method on any grain interface can be declared as transactional by annotating it with the **Transaction** attribute and specifying a **TransactionOption** value indicating how this method behaves within a transaction. We list the most common TransactionOptions here.

```
public interface ITransactionalStateStorage<TState>
      where TState : class, new()
   {
      Task<TransactionalStorageLoadResponse<TState>> Load();

      Task<string> Store(
         string expectedETag,
         TransactionalStateMetaData metadata,
         List<PendingTransactionState<TState>> statesToPrepare,
         long? commitUpTo,
         long? abortAfter
      );
   }
```

Figure 4.4: Transactional State Storage Plugin Interface

The Orleans documentation [109] contains a comprehensive list.

- **Create**. Every call to the method starts a new transaction, T, and completes T on exit.
- **Join**. The method can be called only within an already executing transaction.
- **CreateOrJoin**. If the method's caller is executing within a transaction, T, then it becomes part of T. If not, then the call starts a new transaction, T′, and completes T′ on exit.

Once a method that starts a transaction completes without throwing any exceptions, the Orleans runtime will attempt to commit the transaction. If it succeeds, the method returns normally. Otherwise, an exception will be thrown to the caller. Transactional methods should not have any side effects beyond changing the state of transactional grains, so that if they need to be aborted they can be rolled back cleanly. This is not enforced by Orleans, and left to programmer discipline.

All methods accessing transactional grain state must be transactional, but transactional methods can exist on non-transactional grains as well.

### 4.4.3 Transactional Storage

As we discussed in §4.3, Orleans users require the flexibility to use a cloud storage solution of their choice to store durable grain state, including transactional grains. To this end, we augment

the Orleans framework with a pluggable transactional storage interface **ITransactionalStateStorage** that users can implement, see Figure 4.4. The interface requirements are quite minimal; it accommodates any highly available cloud storage solution that supports conditional writes based on an ETag check (which in practice means any cloud storage can be used).

## 4.5 Transaction Execution

Orleans uses dependency injection to populate the transactional state field of a transactional grain. In addition to the methods on ITransactionalState's public API to read and write the transactional state (Figure 4.3), the injected object also has methods to *prepare*, *commit* and *abort*, which are required for the 2PC protocol. In effect, a transactional grain acts as a mini-database. It is the unit of access, meaning that a transaction that reads or writes any part of the grain's state is considered to have read or written its entire state. This is to simplify the bookkeeping required during transaction execution; while it is conceptually possible that a transaction only needs to access a part of the grain's transactional state, grains are meant to be small and developers naturally divide large state across many grains. Orleans serialization features are used to generate a deep copy of the state of arbitrary type TState so that working copies can be created for transactions that can later be rolled back if the transaction aborts.

Each silo in the Orleans cluster has a component called the **Transaction Agent** (TA), which provides transaction functionality within the Orleans runtime. It has APIs to start, commit, and abort a transaction. The TA assigns a globally unique transaction ID to each transaction.

Recall from §4.2.2 that the Orleans runtime intercepts all grain method calls via grain reference objects. When a transactional method M is invoked, the runtime uses M's transaction tag to determine whether to start a transaction or propagate the caller's transaction to M. Figure 4.5 illustrates how a transaction starts and propagates. To start a transaction, the runtime calls the local TA on the server running M. Once M completes, the runtime calls its local TA to coordinate running the commit protocol to commit the transaction, which we describe in §4.6. Optionally, the runtime might also run M in reconnaissance mode prior to starting the transaction, as explained in §4.7.

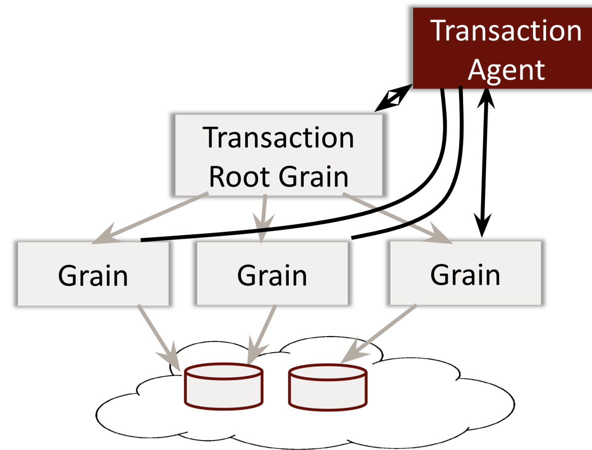



Figure 4.5: Transaction Execution

### 4.5.1 Transaction Context

Each transactional method call carries a hidden parameter called the *transaction context* to record information about the transaction's execution. The transaction context includes the identity of the caller's transaction, T, and of the grains and versions accessed by T. A transaction context is created when the runtime starts a new transaction.

The transaction context is passed back and forth between the grain that started T and other grains T accesses (which all happen via grain async method calls). The transaction context supports a Union method, which accepts another transaction context with the same transaction id and unions its readset, writeset. After a method call is completed, the callee returns an updated transaction context which the caller unions with its own. If a method M1 running within a transaction T calls another method M2 within T, M1 must await M2's return so that it can union the updated context down M2's path. Otherwise, the call down M2 could have made some changes that are not recorded and updates by the transaction can be lost, breaking atomicity. If M1 fails to await M2, we call this an *orphaned call*. The system is able to detect such calls and abort the transaction if they occur.

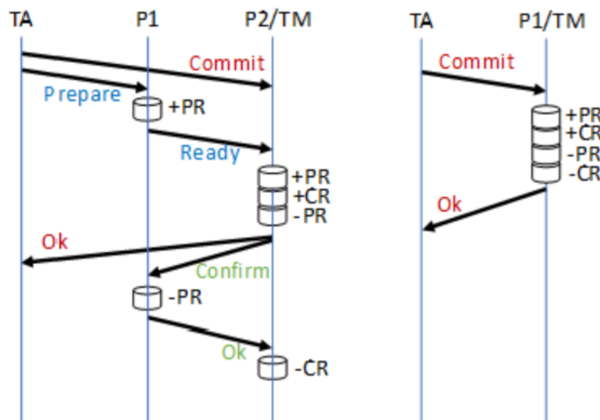



Figure 4.6: Pipelined 2PC with two participants (left) and one participant (right).

### 4.5.2 Concurrency Control

We use 2PL with grain-granularity for concurrency control. When the runtime propagates a transaction, T, to a transactional grain, G, it locks G on behalf of T. While T holds the lock, only method calls that are part of T may execute. Other calls are queued. This is susceptible to deadlocks. Our reconnaissance queries (§4.7) allow us to avoid deadlocks in most cases. When this fails, we use timeouts as the fallback mechanism.

Since locks are not persisted, if a server hosting a transactional grain G fails, a transaction T holding the lock on G will lose the lock and T's in-flight updates of G. If G recovers before T starts 2PC, then T must avoid being fooled into committing and thereby breaking atomicity. As we explain in §4.6, the prepare-phase ensures this by checking whether T still holds its locks; if not, it aborts T. It is also possible for T to lose a lock due to a failure and then re-acquire the lock by accessing G again before it finishes, perhaps via a different callpath. In this case, T's context will refer to two versions of G. To handle this, if the union operation on T's transaction context has seen more than one version of any grain, it will abort T. This also handles the rare cases mentioned in §4.2.3, where there are multiple activations of G simultaneously.

### 4.6 Commit Protocol

After a method that starts a transaction T completes without throwing any exceptions, the runtime initiates the commit process by invoking the local Transaction Agent (TA) and passing it the complete transaction context, which contains the full readset and writeset (i.e., participant transactional grains) of T. Figure 4.6 illustrates the full commit protocol.

Recall that the transactional grain state object has methods to implement 2PC. The TA designates one of the participant grains as the *transaction manager* (TM) of the transaction, which will be the authority on T's outcome. The TA then issues *prepare* RPCs to all participants, except the TM. The RPC includes the identity of the TM as one of its parameters. Each participant grain G that receives a prepare RPC first validates that T still holds its lock, and if it is part of the writeset, validates that it still has all the writes made by T. If this validation passes, G immediately releases the transaction's lock on it, which allows other transactions to access G and take a dependency on T (since T is not yet committed). T is now said to be *pending* at G. Asynchronously, G will submit a write to its transactional storage to record a *prepare record* for T, which persists T's writes as well as the identity of the TM to durable storage. If and when that write returns successfully, and G has committed all its pending transactions prior to T, G notifies the TM that it is done with its prepare phase. Alternatively, if the validation fails, G rolls back the writes of T and any subsequent pending transaction.

In parallel to the prepare RPCs, the TA also sends a *commit* RPC to the TM. The TM handles that call similarly to how other participants handle the prepare RPC, except it also has the additional responsibility of deciding T's outcome. The TM waits for all the other participants to successfully finish their prepare phase. After the TM receives notifications from all participants that they have successfully prepared, it asynchronously submits a write to its transactional storage to store the *commit record* for T. After that write is successful, the TM replies to the TA so it can notify the client that the transaction completed. Additionally, the TM notifies all the participants of T's outcome so they can clean up their logs and mark T as no longer pending. The persisted commit

record includes the identity of all the transaction participants that need to be notified, so that the TM grain can perform this duty despite failures. Once all participants have been notified of the transaction commit, the TM can remove the commit record from its storage to reclaim space.

As illustrated in Figure 4.6, transactions that only touch a single grain only need one RPC to commit, instead of the usual two rounds of 2PC.

One subtle issue in Orleans' commit protocol is that even read-only participants have to go through the process of persisting a prepare record. Recall from §4.2.3 that multiple instances of the same grain might be active simultaneously in some failure cases. It is therefore necessary to do a write to transaction storage to perform an ETag check and ensure that the grain version read by the transaction is indeed the latest and not a stale version.

Apart from the above considerations that arise from the early release of locks, the recovery actions required when participant fails, a storage write fails, or a message is lost are the same as in other 2PC protocols. A detailed description appears in Chapter 7 of Bernstein et al. [21].

### 4.6.1 Discussion

Orleans' pipelined 2PC variant enables two major benefits. First, transaction locks need not be held during the high latency 2PC. Second, since persisting prepare and commit records to storage is asynchronous, writes can be efficiently batched, enabling a distributed form of the *group commit* optimization. Together, these reduce the contention of transactions dramatically and allow for much higher transaction throughput despite the high latency of 2PC over cloud storage compared to a standard 2PL/2PC implementation.

On the other hand, a potential drawback of this design is that it allows cascading aborts. However, since a transaction only releases its locks after it has already finished executing, transaction program failures and deadlocks are no longer possible. Hence, cascading aborts only happen only due to server failures, e.g., a hardware or operating system failure, which are relatively rare. Additionally, when a server S fails, there is significant delay in cluster reconfiguration, since other servers need to wait long enough to be sure that S failed and is not simply slow. Then they must

work around S's failure until S recovers. Thus, independent of the existence of transactions, application execution will be disrupted. Cascading aborts will add to the period of unavailability, but we argue that the effect is incremental, not a fundamentally new effect to be coped with.

### 4.7 Reconnaissance Queries

A guiding principle in the design of Orleans transactions is to avoid holding locks while waiting on high latency cloud storage access. Orleans' pipelined 2PC removes the latency of the commit protocol from a transaction contention period. The other cause of cloud storage access is reading grain state from storage when activating a grain. Hot grains will typically have been activated by the system and hence have in-memory instances. However, this does not fully address the issue because a transaction might access hot grains along with other cold grains that have not been recently activated. There are several well-known techniques to work around this problem [21]. For example, the programmer could manually prefetch grains before executing the transaction. Another technique is to ensure hot grains are the last to be accessed. This is beneficial because it minimizes the execution time during which access to the hot grains causes a conflict. Unfortunately, these are are not always applicable, and they push a lot of complexity to programmers.

To deal with this problem in a general way, we added reconnaissance queries [55, 141], which is currently an experimental feature. When a grain method is supposed to start a new transaction T, the system can first start T in *reconnaissance mode*. In this mode, T executes in lock-free *repeatable read* isolation. Transactional grains, which are multi-versioned and keep track of stable, committed versions, return a known committed value to serve the read operations without waiting on any locks. Any writes made by T in reconnaissance mode are staged at the grain state, and discarded at the end of execution. After the reconnaissance phase, the system then starts T normally in lock-acquiring mode. Note that this does not require any changes to the application code, and is done completely transparently.

As shown in prior work [55, 141], it is rare for the readset of a transaction to change between the reconnaissance query and the actual transaction. As a result, running the reconnaissance query

serves the important purpose of activating the grains in the transaction's readset which ensures their state is loaded from cloud storage prior to acquiring any locks. In the uncommon case where the readset does change, transactions will potentially have to hold locks while reading from slow cloud storage, but the system does not face any correctness problems.

Reconnaissance queries have two main disadvantages. First, they add to the query latency, although this additional latency does not lead to increased the contention. Second, they require executing the transaction logic twice before committing. While transactions tend to be short, this could still be wasteful if the transaction is compute-intensive, particularly in low contention cases. Hence, we allow users to opt-out of reconnaissance queries on a per transactional method basis.

### 4.7.1 Deadlock Avoidance

Since transactions in Orleans use locking for concurrency control, the system must deal with the potential for deadlocks. Deadlocks in Orleans transactions have been identified as a major source of performance issues by prior work [94], and by our users. Furthermore, that they can occur at all is often a surprise for Orleans users since locking is not explicit in the programming model.

The most general mechanism used in the system to handle deadlocks is transaction timeouts, since these are required to handle other possible failures. However, requiring transactions to wait for the entire timeout duration to resolve deadlocks can be too slow and in practice leads users to set their transaction timeouts to conservatively short values, resulting in many transaction aborts and restarts.

By making all transactions acquire their locks in the same order, we can prevent deadlocks. The system leverages the fact that the readsets and writesets of the transaction are (approximately) computed by the reconnaissance queries, prior to acquiring any locks. After the reconnaissance query, the runtime will issue RPCs to acquire the locks on the participant grains in a defined order before executing the transaction. A naive implementation of this idea would require adding $|\text{readset} \cup \text{writeset}|$ round-trips to time under locks, which increases contention. Instead, as in

Table 4.1: Single Silo Write Throughput.

| | Single Grain | Two Grains |
|---|---|---|
| Writes (KTPS) | 430 | 212 |
| Persistent Writes (KTPS) | 126 | 61 |
| Transactions (KTPS) | 46 | 11 |

prior work [55], the Orleans runtime uses an RPC Chains [134] style approach, which cuts the round-trips required roughly in half compared to the naive approach. The way this works is that the lock acquisition RPC to a grain contains an ordered list of subsequent grains that need to be locked. When the silo locks a grain, it forwards the remainder of the list to the next grain and so on until the last grain is locked, which will then notify the first grain that all the locks have been acquired, at which point the transaction execution can start.

It is again possible that the transaction's readset or writeset changes between the reconnaissance query and actual transaction. We fall back to timeouts to resolve any potential deadlocks that might arise as a result.

This ordered lock acquisition scheme can actually increase a transaction's contention period, because lock acquisition has to be serialized. We allow the programmer to disable the scheme on a per transaction basis. Note that there is no overhead for the common case of transactions accessing a single grain.

### 4.8 Experiments

In this section we study the overhead of transactions by comparing transactional operations to regular persistent non-transactional grains (§4.8.1) and the effectiveness of the pipelined commit protocol (§4.8.2) using micro-benchmarks. We also evaluate effectiveness of reconnaissance queries in Orleans using the Smallbank benchmark (§4.8.3).

All Silos and clients are running on Azure Standard D8as v5 VMs, which promise 8 cores and 32GB of RAM. We use Orleans 7. Unless otherwise stated, we use Azure Storage for grain persistent storage. Throughput numbers are computed by running the workload for 5 minutes and

taking the average.

### 4.8.1 Transaction Overhead

Transactions incur overhead since transactional state has to create copies of itself to support rollbacks and multi-versioning, in addition to the RPCs and write needed for the commit protocol. To measure that overhead, we devise a simple micro-benchmark. Grains in this workload have a very simple 64-bit integer state. Each operation can either write the state of one or two grains, selected at random from a large universe of grains. In this experiment, storage for grain state is in-memory, since we want to measure the overhead of Orleans' transactions operations, not the cost of IO.

The results are shown in Table 4.1. The first row shows the total throughput for regular non-persistent grain operations, the second row shows the throughput for the same operations when performed with persistence, and the third row shows the throughput when performed within a transaction.Transactions have significant overhead compared to plain grain operations and even persistent grains. One notable thing about the results is that the throughput of single grain workload is significantly more than double the throughput of the two-grain transaction workload. This shows that single-grain transactions are significantly more efficient than multi-grain ones, which is due to the transfer of coordination from TA to TM in the single-grain case, as described in §4.6.

### 4.8.2 Single Grain Microbenchmarks

Hot data is a worst case for transaction performance and often arises in practice. Orleans' commit protocol is designed to support high throughput for hot data. In this experiment we evaluate its performance for a single hot grain and compare it to a standard 2PL/2PC baseline as well as non-transactional persistent grains. The workload involves writing a single grain which has 1KB state. We plot the results in Figure 4.7.

With its early lock release and pipelined commit, Orleans is able to sustain much higher throughput than a baseline 2PL/2PC implementation. Furthermore, it also greatly outperforms

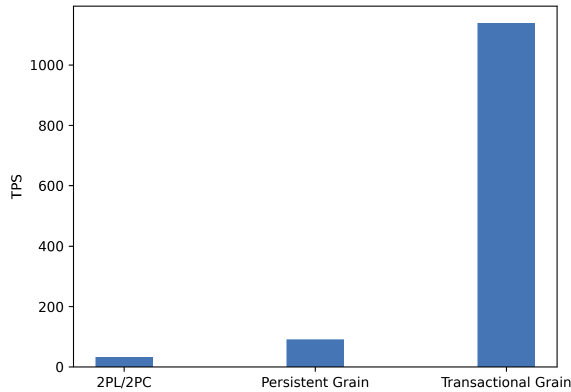
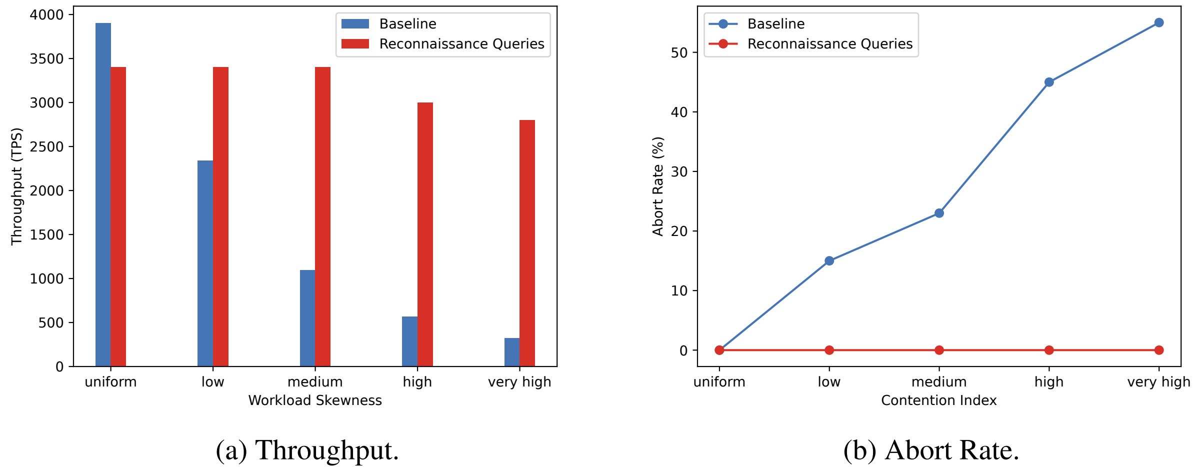



Figure 4.7: Single Grain Throughput

even the throughput of non-transactional grain writes, which still have to lock the grain for every individual write during the entire duration of performing the write to cloud storage.

### 4.8.3 Smallbank Multi-Transfer

Here we use the Smallbank [6] benchmark to evaluate the effectiveness of reconnaissance queries in addressing high contention and deadlock aborts, by comparing the performance when configured to use reconnaissance queries vs. a baseline where they are disabled. SmallBank is an OLTP benchmark simulating basic operations on bank accounts, and is a good fit for simulating actor workloads which are write-intensive and interactive [94]. We use a similar setup to prior work [94]; in particular, we also use a MultiTransfer transaction that withdraws money from one account and deposits money to multiple other accounts in parallel. In this workload, each account is modeled as a separate grain, and there are 10k accounts in total. The grains accessed by each transaction are selected randomly using zipfian distribution. We vary the zipf parameter to generate different levels of *skewness* to measure the effect of contention. In a highly skewed workload, transactions access only a small set of grains, which causes them to be highly contended. We plot the throughput and abort rates under different workload skewness in Figure 4.8.




(a) Throughput.

(b) Abort Rate.

Figure 4.8: Smallbank Multi-Transfer Results

**Analysis.** As expected, under low contention, the reconnaissance queries in Orleans are mostly wasteful and consequently the baseline configuration has better throughput. As contention increases, however, they become crucial. Throughput of the baseline drops significantly and deadlock aborts become frequent, even with a modest increase in skewness, whereas the performance of the variant with reconnaissance queries holds steady and is able to avoid deadlock aborts completely.

## 4.9 Conclusions and Future Work

We presented the design of ACID transactions mechanism in Orleans, an actor framework and platform that has evolved into an actor-oriented database system. Orleans transactions have to be distributed over external high-latency cloud storage, yet we have shown how to mask the high latency of cloud storage using early lock release, pipelined 2PC and reconnaissance queries to achieve high throughput and good performance. We shared many experiences from our journey to productionize transactions in Orleans, including how much considerations like extensibility, ease of deployment and ease of integration with existing workflows often trump pure performance once the system achieves performance acceptable to customers.

There is much that can be done to extend this work. On the research side, one could experiment

with variations of our technique to identify further optimizations. For example, one could try multi-version optimistic concurrency control, so transactions can read or overwrite data that was last written by a still active transaction. This should increase the maximum throughput when the transaction conflict rate is low. On the practical side, it would be beneficial to avoid deep copying the entire object state when a small update is made to a big structure, e.g., a dictionary. One way is to implement a custom transactional variation of the data structure that can log and undo incremental updates.

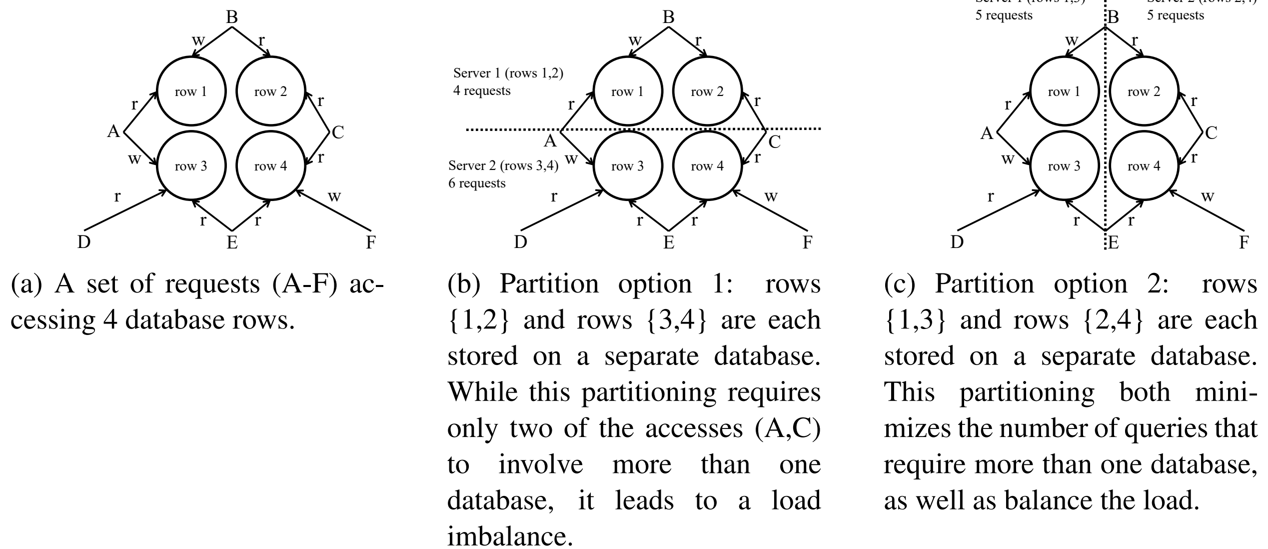
# Chapter 5: Effectively Avoiding 2PC with Neuroshard

## 5.1 Introduction

Horizontal sharding is a decades-old technique to scale production databases [123]. When a database's load or storage capacity overwhelms a single server, operators split the rows in the database and store them in multiple servers. Notable examples of horizontal sharding are Facebook's social graph [106] and Google's ad serving database [3, 131].

The choice of assigning rows to servers affects many aspects of a system's performance, so operators often need to simultaneously optimize for multiple, sometimes conflicting, objectives. One example objective is to assign a roughly equal number of rows to each server, balancing storage load. Another is to assign rows such that the servers receive roughly the same number of queries, balancing compute and network load. This problem is exacerbated by real-world requirements, such as variable-sized objects, servers with heterogeneous capacities, and queries with variable complexity. Beyond load balancing, another class of objectives is to minimize the fanout – the number of shards that a query touches – because a distributed query is typically much more expensive than a single-server query. However, fanout minimization requires clustering the rows assessed together frequently into the same shard, which is at odds with load balancing objectives if many rows in the cluster are hot.

Given these intricate, combinatorial objectives, an ideal sharding scheme should flexibly adapt and optimize for them simultaneously. Unfortunately, existing sharding algorithms are designed primarily for single objectives. For instance, commonly-used random, hash, range and round-robin partitioning [45, 46] are good at balancing load but ignore fanout minimization completely. To minimize average query fanout, recent work, including Schism [41], SWORD [118] and SHP [78], collect a trace of past accesses to the database, model it as a (hyper)graph that links the rows




(a) A set of requests (A-F) accessing 4 database rows.

(b) Partition option 1: rows {1,2} and rows {3,4} are each stored on a separate database. While this partitioning requires only two of the accesses (A,C) to involve more than one database, it leads to a load imbalance.

(c) Partition option 2: rows {1,3} and rows {2,4} are each stored on a separate database. This partitioning both minimizes the number of queries that require more than one database, as well as balance the load.

Figure 5.1: Toy example of the distributed database partitioning problem.

accessed together in a query, and then compute the row-to-shard assignments by solving a partitioning problem. In general, graph partitioning is an NP-Hard problem [7], so existing work uses solvers that utilize hand-designed heuristics that require considerable expertise and experimentation to produce good solutions for just the objective of minimizing fanout. Given the diverse and competing objectives, designing hand-crafted heuristics that work well for each objective and their combinations is a painstaking task.

This chapter presents *Neuroshard*, a sharding system that tackles the decades-old horizontal scaling problem using a novel learned approach. Neuroshard can automatically optimize for multiple classic sharding objectives, such as fanout minimization and load balancing, by learning effective partitioning heuristics directly from the input trace. This work is based on [54].

Similar to prior work [78, 118] Neuroshard works by collecting a trace of past queries and representing it as a hypergraph, where the rows are vertices and the queries are hyperedges, and then partitions that hypergraph. This process is done regularly offline at some frequency. The intuition is that computing a sharding assignment that works well for past accesses will most likely serve to improve the performance on future accesses as well. This offline approach is orthogonal and complementary to online sharding in systems such as Clay [129] and E-store [56, 138], in which the

system actively moves rows to improve the performance. A database can use Neuroshard to precompute sharding assignments for a workload trace and then dynamically adjust the assignments via online sharding upon the workload shifts.

Neuroshard's goal is to use Deep Reinforcement Learning (RL) to partition the hypergraph of past queries, in order to achieve various sharding objectives. However, the challenge in directly applying RL to our setting is that the query hypergraph may contain millions of queries (or hyperedges), which would make the the state and action space of the RL too large.

Inspired by prior work on graph partitioning [155], we make the observation that if at each stage of partitioning we only consider adding the "neighboring" rows (or the rows that were accessed in the same query), we can achieve good partitions, while also significantly limiting the number of vertices we consider at each step of the algorithm. This structure naturally lends itself to a reinforcement learning formulation. Therefore, Neuroshard uses a trained RL agent to score each one of the candidate vertices, where the reward is a function of multiple sharding objectives. Neuroshard uses techniques from Multi-Task learning to optimize for multiple objectives in parallel.

We implement Neuroshard on a distributed database based on MariaDB [103], and compare it to three heuristic baselines: Neighbor Expansion [155], hMetis [82] and hash partitioning. We compare the different algorithms on several microbenchmarks and on the Epinions [104] social network dataset. Our evaluation shows that while Neuroshard does not always provide the best performance, it consistently provides close to the best performance on all the traces, while the heuristic schemes' performance varies. For example, hMetis and Neighbor Expansion perform well in workloads where fanout minimization is a primary objective and the load is spread relatively evenly across queries, but perform poorly in skewed workloads where load balancing is an important objective. In contrast, Neuroshard is able to balance multiple objectives simultaneously (*i.e.,* fanout and load balancing) in both of these types of workloads.

We make three primary contributions:

1. **Learned sharding.** Neuroshard is the first system to use a learned approach for directly

assigning rows to shards.

2. **RL formulation.** A novel RL framework for the hypergraph partitioning problem (§5.4). Our design uses ideas from the Neighbor Expansion [155] algorithm to restrict the state and action spaces so that the RL agent only needs to learn how to make good local decisions based on a small subset of the hypergraph (the neighborhood). We formulate two popular sharding objectives as RL rewards: fanout minimization (§5.4.4) and load balancing (§5.4.5)

3. **Multiple objectives.** A general approach for multi-objective sharding using Multi-Task Learning (§5.6) that can automatically incorporate new objectives as RL rewards.

## 5.2 Horizontal Sharding

This section provides a background on the problems tackled by Neuroshard. It first provides a motivating example for sharding (§5.2.1) and formally defines the sharding problem (§5.2.2) on two popular sharding objectives: fanout minimization (§5.2.3) and load balancing (§5.2.4).

### 5.2.1 Motivating Example

A common problem in the distributed database setting, where the entirety of the data cannot be fit in a single database server, is how to shard the data (*e.g.,* rows, keys or tables) across servers. The overall performance of the system is affected by how the data is sharded. For example, in many cases, it is beneficial to minimize *fanout*, or the number of servers that on average need to be accessed to satisfy a single query. In addition, the system may also need consider the *load* on each server, or how many total queries it is serving, or the average *ratio of writes* seen by each server, since writes are typically more expensive than reads. Finally, the system may have other real-world constraints, such as network or disk bandwidth limitations.

Balancing between these different objectives and constraints can be challenging. Consider the following toy motivating example, depicted in Figure 5.1a.

In this example, we assume we have only four database rows, which need to be equally split

across two physical servers. We assume the system sees six different requests (marked A-F in the figure), each of which issue one or two read (marked r) or write (w) operations to one of the rows. For example, query B issues a write request to row 1, and a read request to row 2, while query F issues a read request to row 4. Assume this request pattern repeats in an infinite loop.

Suppose our objective is only to consider fanout minimization across these six requests, or to minimize the number of servers that on average need to be accessed to satisfy each query. One possible solution would be to assign rows 1 and 2 to one database, and rows 3 and 4 to the second database, depicted in Figure 5.1b. In this solution, four of the queries would be satisfied by only one database (queries B, D, E and F) with a fanout of 1, and two of them would have a fanout of 2: query A that needs to access rows 1 and 3, which are stored on separate servers, and query C that needs to read from rows 2 and 4, which are also stored on separate servers. Therefore, the total average fanout would be 1.3, which is the optimal solution in this particular setting.

However, this row partitioning would create a load imbalance: server 1 would need to handle a total of 4 reads and writes, while server 2 would handle 6 reads and writes in each loop iteration. Instead, if we assign rows 1 and 3 to one server, and rows 2 and 4 to the second server (depicted in Figure 5.1c), we can achieve the same fanout (1.3), while achieving a perfect load balance. However, in this case, we still have not considered the fact that reads and writes have different costs, or that different servers may have different amounts of storage or network bandwidth available.

While our toy example is easy to solve, since there are only three ways the four rows can be equally partitioned, solving this problem for a large number of queries and rows is difficult (in fact, it is NP-Hard for fanout minimization [7], as we show below).

### 5.2.2 The Sharding Problem: Formal Definitions

The sharding problem can be formally modeled as a hypergraph partitioning problem [41, 78, 118]. A hypergraph is a generalization of a graph for which each edge (called hyperedge) can connect any number of vertices rather than just two. In the case of database sharding, a trace of recent past queries to the database that records the rows accessed by each query can be thought of

as a hypergraph, with the rows as the vertices (*e.g.,* rows 1-4 in our toy example) and the queries (*e.g.,* queries A-F in our example), which may involve more than 2 rows, as the hyperedges.

The goal of hypergraph partitioning is to divide the vertices of a hypergraph into a number of equal size partitions, usually with a goal such as minimizing the number of hyperedges that cross partitions (fanout), or equalizing the sum of the degrees of the vertices in a partition (load), which we formalize next.

### 5.2.3 Fanout minimization

We follow the notation of SHP [78] and describe the hypergraph fanout minimization problem in the equivalent terms of bipartite graphs, where the vertices are either requests and rows, and edges are drawn between each request and the rows it is accessing. As a reference, our toy example is depicted in a bipartite graph form in Figure 5.2.

Let $G = (Q \cup R, E)$ be an undirected bipartite graph with disjoint sets of query vertices, Q, and row vertices, R. The goal is to partition R into k parts, i.e. find a collection of k disjoint subsets $V_1,...,V_k$ (also called partitions) covering R that minimizes an objective function. The output partitions should be balanced, that is,

$$|V_i| \leq (1+\epsilon)\frac{n}{k}$$

for all $1 \leq i \leq k$ and some $\epsilon \geq 0$, where $n = |R|$. Intuitively, this captures the requirement that partitions should all be assigned a similar number of vertices (or rows in our case). Given a partitioning $P = \{V_1,...,V_k\}$ and a query vertex $q \in Q$, informally the fanout of q is the number of distinct partitions that contain row vertices adjacent to q. Formally, we can define

$$fanout(P,q) = |\{V_i : V_i \cap N(q) \neq \emptyset\}|$$

Where N(q) is the set of vertices adjacent to q in G. Note that we have $N(q) \subseteq R$, since the graph is bipartite. We can now define the quality of the partitioning P as the average query fanout:

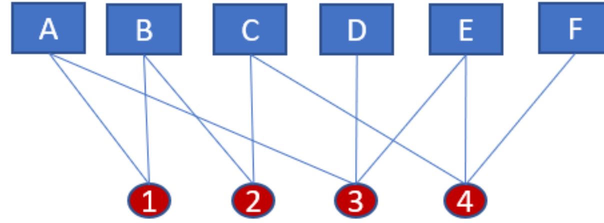



Figure 5.2: Toy example represented as a bipartite graph.

$$fanout(P) = \frac{1}{|Q|} \sum_{q \in Q} fanout(P, q)$$

The fanout minimization problem is, given a bipartite graph G, an integer k > 1, and a real number $\epsilon \geq 0$, find a partitioning of G into k partitions with the minimum average fanout.

### 5.2.4 Load Balancing

Another very common objective for sharding is load balancing. The setting for the load balancing problem is similar to that of fanout minimization. We are also given a hypergraph represented as a bipartite graph $G = (Q \cup R, E)$, and the goal is still to partition R into k partitions subject to the same constraints on the number of vertices assigned to each partition. Additionally, each vertex r in R has an associated value $\mathbf{W_r}$, which we call the *load weight*. This is meant to represent how much load is caused by processing queries that access a particular row r. It is up to the application to specify the load weight for each row; a natural scheme is to use the degree of the vertex in r as its load weight (which Neuroshard uses by default).

Given a partitioning $P = \{V_1,...,V_k\}$ and a partition $i \leq k$ we can define the load of partition i as the sum of the load weight of all vertices in $V_i$. Formally,

$$load(P, i) = \sum_{r \in V_i} W_r$$

One possible measure of load imbalance is the difference between the most and least loaded partitions. Let $\mathbf{p_{max}}$ be the partition with the highest value of load. Similarly, let $\mathbf{p_{min}}$ be the

partition with the lowest value. We can now define the quality of the partitioning P as

$$Imbalance(P) = load(P, p_{max}) - load(P, p_{min})$$

Hence, the load balancing problem is, given a graph bipartite G, an integer k > 1, and a real number $\epsilon \geq 0$, find a partitioning of G into k partitions with the minimum imbalance.

We note that we made a simplifying modelling decision by assuming that the weight load of a vertex is static and does not rely on the partitioning P. In reality, this may not be entirely accurate. As we mentioned earlier and show empirically in §5.7, the amount of work involved in a distributed query or transaction can be significantly more than a local one. As a result, the amount of load generated by an access to a row does depends on whether that access is distributed or local which is clearly affected by the partitioning.

## 5.3 Overview of Learned Sharding

This section overviews key design decisions in Neuroshard. First, we motivate for a learned sharding approach (§5.3.2). Second, we give a primer on deep reinforcement (RL) learning to provide background. Third, we describe requirements of RL and an algorithm amenable to these requirements. Finally, we present the high-level RL algorithm in Neuroshard.

### 5.3.1 Why Learned Sharding?

The conventional approach to sharding employs various heuristics, which are difficult to adapt. For instance, since fanout minimization is NP-hard, prior work [78, 107, 118] has focused on devising hand-tuned heuristic sharding schemes based on hypergraph partitioning. These schemes involve collecting a trace of past accesses to the database, modelling it as a hypergraph, and then solving the fanout minimization problem using heuristics to create new row-to-shard assignments.

While such schemes are often effective at fanout minimization, in order to achieve optimal performance, the sharding algorithm needs to balance multiple objectives and constraints simul-

taneously. Incorporating new objectives and constraints into heuristic algorithms is difficult and time-consuming, and may require a complete rewrite of the algorithm.

Therefore, we instead pursue a different approach and build a framework to automatically *learn* effective sharding heuristics for multiple objectives directly from the workload.

### 5.3.2 Deep Reinforcement Learning Primer

A natural follow-up question is which learning method to use to realize learned sharing. We utilize Deep reinforcement learning (RL) in Neuroshard so here we give a primer on deep RL, highlighting its fit for the sharding problem.

RL [136] is concerned with the development of *agents* that learn from direct interaction with their environment. In an RL setting, an agent starts out knowing nothing about the given task, and learns by taking incremental actions, observing how these actions affect the environment, and receiving a reward that depends on its performance on the task. Despite not having any prior knowledge about the task, RL training algorithms allow the agent to improve its performance at it. This aligns very well with our goal of designing a general sharding framework that is able to perform well on various workloads and objectives, without needing to hand design heuristics for each one. By casting the sharding problem into an RL training problem, we only need to design the reward signal, and then leverage RL to train the sharding agent.

Combining classical RL with Deep Learning [61] techniques (dubbed Deep Reinforcement Learning [111]) has been key to many recent breakthroughs such as in game playing [132] and many applications in computer systems [33, 100, 101]. For the rest of this section we give an overview of the Deep RL concepts that we use in this chapter, and in particular a family of techniques called policy gradient methods [137]. For a comprehensive treatment of the subject please refer to [136].

**Setting** The usual setting of Reinforcement Learning is the discrete-time Markov Decision Process (MDP), in which an *agent* is interacting with an *environment*. At each time step **t** the agent

observes some environment state $\mathbf{s_t}$, and performs an action $\mathbf{a_t}$. As a result of the action taken, the agent receives a reward $\mathbf{r_t}$ and the environment state transitions to $\mathbf{s_{t+1}}$. The state transitions and rewards are stochastic and are assumed to have the Markov property; i.e. the probability of receiving a reward $r_t$ and the probability of transitioning to a state $s_{t+1}$ depend only on the state of the environment $s_t$ and the action taken by the agent $a_t$. The goal of the agent is to maximize the expected cumulative discounted reward:

$$\mathbb{E}[\sum_t \gamma^t r_t]$$

Where $\gamma \in (0, 1]$ is a factor discounting future rewards. At each step the agent takes a decision based on a *policy* $\boldsymbol{\pi}$, which defined as the conditional probability distribution of actions given states. In other words, $\pi$(s, a) is the probability that action a is taken when the state is s. For all but the most trivial of applications, the size of the state space makes it infeasible to store the policy explicitly in a tabular form. Hence, a common approach is to use function approximators to approximate $\pi$ using a manageable number of paramters $\theta$. Deep Neural Networks have emerged as a popular choice for the function approximator used to represent the parameterized policy $\pi_\theta$.

**Policy gradient methods** Like prior work [100], we utilize a class of RL algorithms in Neuroshard learns by performing *gradient descent* over the policy parameters. In Neuroshard we use the REINFORCE [137] algorithm, which works by obtaining an estimate for the gradient of the cumulative discounted reward empirically by sampling (*i.e.,* running the MDP by following the current policy $\pi_\theta$) then updates the parameters using that estimate, according to the following formula:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta log \pi_\theta(s_t, a_t) v_t$$

Where $\alpha$ is the learning rate and $v_t$ is the empirically computed cumulative reward. Like prior work [100], we use a variant [128] of the REINFORCE algorithm that subtracts a baseline value from each return $v_t$, which is useful to reduce the variance of gradient estimates. We describe the

training procedure in more detail in §5.5.2.

### 5.3.3 Sharding Problem Formulation for RL

We now need to formulate the sharding problem in RL terms. We have three main design considerations. First, problems with an incremental structure are good fits for RL because the agent can learn to perform a specific action and get an incremental reward at each step. Second, it is ideal if the agent needs to learn only how to make good local decisions. If the agent requires much global knowledge, training tends to be difficult and inference performance overhead tends to be high. Worse, a model trained on a graph may not generalize to a different graph. Third, it is desirable to reduce the action search space such that the agent model can stay simple.

These design considerations lead us to solution structure based on Neighbor Expansion (NE) [155], an algorithm for *graph* edge partitioning. The goal of the graph partitioning problem is to divide the *edges* of a graph evenly into equal-size partitions while minimizing the average number of partitions that are incident to a vertex (called the *replication factor*). It is analogous to the fanout minimization problem we defined previously on hypergraphs in §5.2.3. The NE algorithm then exploits graph structure and can produce high-quality partitions for the replication factor objective, and automatically balances the number of edges partitions in each partition. It also has the nice properties of being fast to run [155] and highly scalable [70], making it an effective approach for graph edge partitioning.

As explained in §5.2.2, the sharding problem is modelled as a hypergraph partitioning problem. Therefore we design a Hypergraph Neighbor Expansion (HNE) algorithm, an adaptation of Neighbor Expansion to hypergraph partitioning.

We show the pseudo code of HNE in Algorithm 2. The basic idea is fairly simple; the algorithm proceeds by building partitions sequentially. It maintains two hyperedge sets, the core set **C**, and the set **S** which is the set of all hyperedges incident to the vertices assigned to the current partition. We call a hyperedge q *unassigned* if it has not yet been added to the core set C of any partition. At every step (starting with a random seed hyperedge), one of the hyperedges in S (that is not already

a member of C) is selected as a core hyperedge and added to the set C, then all its remaining vertices are added to the partition. All of C's unassigned neighbors (two hyperedges are *neighbors* or *adjacent* if they are both incident to at least one common vertex) are then added to the set S. This is repeated until the partition is filled up. A hand-crafted greedy heuristic determines which hyperedge to select at every step of the process to add to C.

Algorithm 2 describes an algorithmic framework that meets our design considerations. it has an *incremental* structure of assigning candidate hyperedges gradually. The decision to assign a hyperedge to a partition is *local*, depending on the neighboring relationship. It is *hyperedge-centric*, meaning that each step selects a hyperedge and then assigns all vertices on the hyperedge to a partion. This coarsens the action space compared to a *vertex-centric* approach that adds one vertex at a time. In an extremely skewed graph, a hyperedge may connect to many vertices, posing a problem for this treatment, but in practice, OLTP workloads which we target do not typically have such skewed distributions.

```
C ← ∅;
S ← ∅;
while Partition is not full do
    candidates ← S \ C;
    if candidates = ∅ then
        Seed candidates with a random, unassigned hyperedge;
    end
    Select the best candidate h based on scoring heuristic.;
    Add h to C.;
    Add all of h's unassigned adjacent hyperedges to S.;
    Add all of h's unassigned vertices to partition, and remove them from the hypergraph.;
end
```

**Algorithm 2:** Hypergraph Neighbor Expansion (HNE)

The scoring heuristic we use for HNE is also inspired by the neighborhood heuristic in the original Neighbor Expansion algorithm [155]. Let **HN(q)** be the set of unassigned neighbors of hyperedge q. The neighborhood heuristic we use is to select the candidate hyperedge that has the minimum value of $|$ HN(q) \ S $|$. This choice is greedy in that it selects the hyperedge that results in the smallest increase in fanout at this step.

To simplify the presentation of Algorithm 2 we omitted the handling of the following corner case: If the partition fills up while adding h's vertices, we do not just add all of h's vertices and violate the balancing constraints. Instead, we initialize the procedure for the next partition by making S = {h} instead of the empty set. We also omit discussing how to handle vertices that do not have any incident hyperedges. These can be handled straightforwardly in various ways, we generally preprocess the hypergraph to remove such vertices before running Algorithm 2. After finishing, we shuffle the vertices then assign them to the computed partitions in a round-robin fashion. Hence, for the rest of this chapter we always assume that each vertex appears in at least one hyperedge.

### 5.3.4 Neuroshard RL Algorithmic Framework

```
C ← ∅;
S ← ∅;
while Partition is not full do
    candidates ← S \ C;
    while |candidates| < threshold do
        Seed candidates with a random, unassigned hyperedge;
    end
    candidate_probabilities ← DNN(candidates, partition_state);
    Select a candidate hyperedge h based on probabilities;
    Add h to C.;
    Add all of h's adjacent hyperedges to S.;
    Add all of h's unassigned incident vertices to the partition.;
end
```

**Algorithm 3:** Neuroshard RL algorithmic framework.

Instead of using a hand-crafted scoring heuristic, Neuroshard adopts a trained agent, represented as a Neural Network, that assigns a probability to each hyperedge at each step of the algorithm. This agent is trained using RL, with a reward being a function of multiple objectives. As we show in §5.7.5, by using this approach Neuroshard is able to train directly on larger traces than prior work (e.g. [42]) and generalize to much larger hypergraphs. Algorithm 3 provides the pseudocode for the Neuroshard RL algorithmic framework. We highlight one other minor, but

significant, difference from NE/HNE in Neuroshard: If the size of the candidate set drops below a threshold, the algorithm will seed with random unassigned hyperedges (if available) to keep the size of candidates set at the threshold. This is important, to avoid getting stuck with candidates that would be good only for only a single objective (e.g., fanout minimization) but hurt other objectives.

## 5.4 Reinforcement Learning Design

In this section, we describe the parts of training Neuroshard as a single-objective Reinforcement Learning problem, namely the **environment**, **state**, **actions**, and **rewards**.

### 5.4.1 Environment

In Neuroshard's RL setting, the environment is made up of 3 components: the hypergraph, the partition state and the assignment state.

**The Hypergraph** This part of the environment is static and does not change throughout a training episode. It includes the structure of the hypergraph, i.e. the set of vertices and hyperedges, or the mapping of requests to data rows. The hypergraph is also augmented with important information, such as the degree of each vertex and hyperedge, whether a hyperedge is a read-only or read-write query, and the load weight of each vertex (in our experiments we use the vertex degree as the load weight, but this can be up to the application).

**Partition state** This is information about the state of the partitions. It is represented by the vector [$\mathbf{T_v}$, $\mathbf{C_v}$, $\mathbf{C_l}$, $\mathbf{P}$, $\mathbf{Max_l}$, $\mathbf{Min_l}$]. These values are defined as follows: $\mathbf{T_v}$ is the target number of vertices per partition. This is a hard constraint. $\mathbf{C_v}$ is the number of vertices in the partition that's currently being built. $\mathbf{C_l}$ is the load weight of all the vertices in the current partition. $\mathbf{P}$ is the number of partitions left to be built after finishing the current partition. $\mathbf{Max_l}$ is the highest load of a complete partition so far. Similarly, $\mathbf{Min_l}$ is the minimum load of a complete partition. Both $\mathbf{Max_l}$ and $\mathbf{Min_l}$ are initialized to the target total load weight per partition (i.e. the sum of the load weight of all vertices divided by the number of partitions) at the beginning of the episode. Once

a partition is complete, the new value of $Max_l$ is computed by maxing the previous value with the new partition's load. The value of $Min_l$ is updated in a similar fashion. This choice of initial value and how to update is significant; as we'll discuss in §5.4.5.

**Assignment state** This is state associated with each vertex and hyperedge indicating their partition assignment status. Specifically, each element can be in one of three states: Unassigned, Assigned, or Assigned to current partition. For hyperedges, being assigned to a partition i means having been added to the core set C of i (note that, due to the corner case described at the end of §5.3.3, a hyperedge q can actually be assigned to more than 1 core set, but this does not require any special handling). Additionally, each hyperedge is also associated with a bit indicating whether it is in the set S of the current partition or not. For each hyperedge $q \in S$, we also maintain its HNE heuristic score, i.e. $| HN(q) \setminus S |$.

### 5.4.2 Observable State

Technically, the agent has access to all of the environment state. However, it only makes use of a small portion to decide on the action to take, namely the neighborhood of the candidates set, as well as the partition state. Theoretically, this makes the problem harder since the agent's inability to take into account the full state results in a partially-observed MDP [113], but like prior work [100], we find that the approach works well in practice nevertheless. For efficiency, the agent is also passed the candidates set explicitly even though it can technically compute it from the environment state.

### 5.4.3 Actions

At every step the agent selects one hyperedge from the candidates set. Hence, its action space at every step is the same as the set of candidates. Once the agent makes a selection, the state of the environment transitions as described in Algorithm 3.

Restricting the Size of the Candidates Set

As noted by prior work [107], applying neighbor expansion to hypergraph partitioning can run into the challenge where the candidates set grows very large very quickly. A very large candidate set makes training very expensive in terms of memory and computational requirements for each episode, as well as much harder in terms of the ability of the agent to learn useful rules as it would need to run more episodes to sample more trajectories. This is more common in hypergraphs that represent social networks since these can have extremely large hyperedges which are not very common in the OLTP workloads we target. Nevertheless, we found it helpful to limit the size of the candidates set considered by the agent at each step to an upper bound $\mathbf{Cand_{max}}$. This is a trade-off since that this adds a hyperparameter to the model that needs tuning for best performance. We find that randomly selecting a subset of the candidate hyperedges of size $\mathrm{Cand_{max}}$ at each step works well in our evaluation, but other more disciplined strategies such as described by prior work [107] are also possible.

### 5.4.4 Fanout Reward

To guide the agent towards producing good solutions for the fanout objective, we design the reward signal as follows: Suppose the agent chose hyperedge $q_t$ as its action in time-step t. It receives a reward

$$r_t = \frac{-1}{|Q|}|HN(q_t) \setminus S|$$

Recall from §5.3.3 that HN(q) is the set of *unassigned* hyperedges that are adjacent to hyperedge q. Furthermore, we say that a hyperedge q is *incident* to a partition i iff $V_i \cap N(q) \neq \emptyset$. In other words, q is incident to partition i if any of the row vertices accessed by q are assigned to i. Note that the numerator of $r_t$ is the (negative of the) number of hyperedges that became incident to the partition as a result of the agent's action $a_t$, *i.e.,* the number of queries that are incident to the partition at time t+1 but were not already incident to the partition at time t. Hence, $r_t$ is the incremental change in the fanout objective as a result of $a_t$. In other words, we have

$$fanout(P) = -\sum_{t} r_t$$

.

RL training works to *maximize* the sum of expected rewards, which is the same as *minimizing* its negative. Hence, our reward signal design causes training to minimize fanout(P).

Distinguishing Between Reads and Writes

So far we ignored the difference between read-only and read-write queries in our modeling of fanout. However, as we show in §5.7, it is generally the case that distributed writes are significantly more expensive than reads. This is easy to account for in Neuroshard by adjusting the reward signal to weigh the reward differently based on whether $h_t$ represents a read-only or a read-write query.

### 5.4.5 Load-balancing Reward

Now we discuss how we design the load-balancing reward signal so that the agent can produce good solution for the imbalance objective. Suppose the agent chose a hyperedge $\mathbf{q_t}$ as its action in time-step t. This causes a set of unassigned row vertices adjacent to $q_t$ to be assigned to the partition. We call this set of newly assigned row vertices $\mathbf{A_t}$. Let $\mathbf{WA_t}$ be the sum of the load weights for each vertex in $A_t$, that is,

$$WA_t = \sum_{v \in A_t} W_v$$

Recall from §5.4.1 that $\mathbf{Max_l}$ is the highest load of a complete partition so far, *i.e.,* up to time-step t. Likewise, $\mathbf{Min_l}$ is the lowest load of a complete partition so far. Let $\mathbf{load_t}$ be the load of the current partition at the start of time-step t. Note that $load_{t+1} = load_t + WA_t$.

The agent will receive a reward $r_t$ that is made of two different components. We start by defining the components:

- $\mathbf{rmax_t}$, defined as: $\max(0, WA_t - \max(0, load_t - Max_l))$. Note that if $load_{t+1} \leq Max_l$ this

sets $rmax_t$ to 0. Otherwise, the $rmax_t$ is set to the incremental increase over $Max_l$ caused by action $a_t$.

- **$rmin_t$.** The value of $rmin_t$ is always 0 if action $a_t$ does not cause the current partition to fill up and start a new partition. Otherwise, it is defined as: $|\max(0, Min_l - (load_t + WA_t))|$. In other words, if the newly-finished partition has at least the same load as $Min_l$, $rmin_t$ is set to zero. Otherwise, it is set to the difference in load.

Recall from §5.4.1 that both $Max_l$ and $Min_l$ are initialized to the same value $\mathbf{Target_l}$ at the beginning of the training episode. Also recall from §5.2.4 that, given a complete partitioning P, $\mathbf{p_{max}}$ and $\mathbf{p_{min}}$ are the most and least loaded partitions, respectively. Consider the way we defined $rmax_t$. Every time the agent's action causes the current partition's load to increase over the prior value of $Max_l$, we set $rmax_t$ to the difference. Hence, we have

$$\sum_t rmax_t = load(P, p_{max}) - Target_l$$

Similarly, we can show that

$$\sum_t rmin_t = Target_l - load(P, p_{min})$$

We can now define the reward $r_t$ that the agent receives:

$$r_t = -(rmax_t + rmin_t)$$

This has the following property:

$$\sum_t r_t = -(\sum_t rmax_t + \sum_t rmin_t) = -(load(P, p_{max}) - load(P, p_{min}))$$

.

Giving

$$\sum_t r_t = -Imbalance(P)$$

We now briefly discuss the choice of using $\text{Target}_\text{l}$ as the initial value for both $\text{Max}_\text{l}$ and $\text{Min}_\text{l}$. Initially, we chose not to define these values while processing the first partition, then use the load of the first partition to initialize the values starting from the second partition. However, we quickly realized that giving the agent the target load at the start is very helpful and makes attributing rewards to actions significantly easier. This way, the agent gets penalties (i.e. the negative-valued rewards) only and as soon as its choices start causing deviation from the ideal value.

## 5.5 Neural Network Agent

In this section, we describe the internal design of the agent including how it represents the policy as a neural network, and finish by describing the training procedure. We leverage a powerful technique called Graph Neural Network to capture key features of the underlying hypergraph, which enables a simple, three-layer policy network that yields effective sharding results (see §5.7).

### 5.5.1 Agent Design

The agent's main function is to choose a hyperedge from the candidate set at every time step t. Let $\mathbf{s_t}$ be the observable state at time t (as described in §5.4.2), and $\mathbf{candidates_t}$ be the set of candidate hyperedges. Conceptually, the agent goes through three steps: First, it computes an embedding vector representation for each hyperedge $q \in \text{candidates}_\text{t}$. Second, it uses these embedding vectors to compute a score for each hyperedge q. Finally, these scores are converted to probabilities and a hyperedge $q_t$ is selected based on these probabilities. A deep neural network representing a parameterized policy $\boldsymbol{\pi_\theta}\mathbf{(s, q)}$ is used to accomplish all these steps. As depicted in Figure 5.3, its architecture consists of three components, which we now describe.

**Embedding Layer** Graph Neural Networks [162] (GNNs) are the standard tool used to create embedding vectors for graph elements such as vertices or edges. The goal is to encode the high-

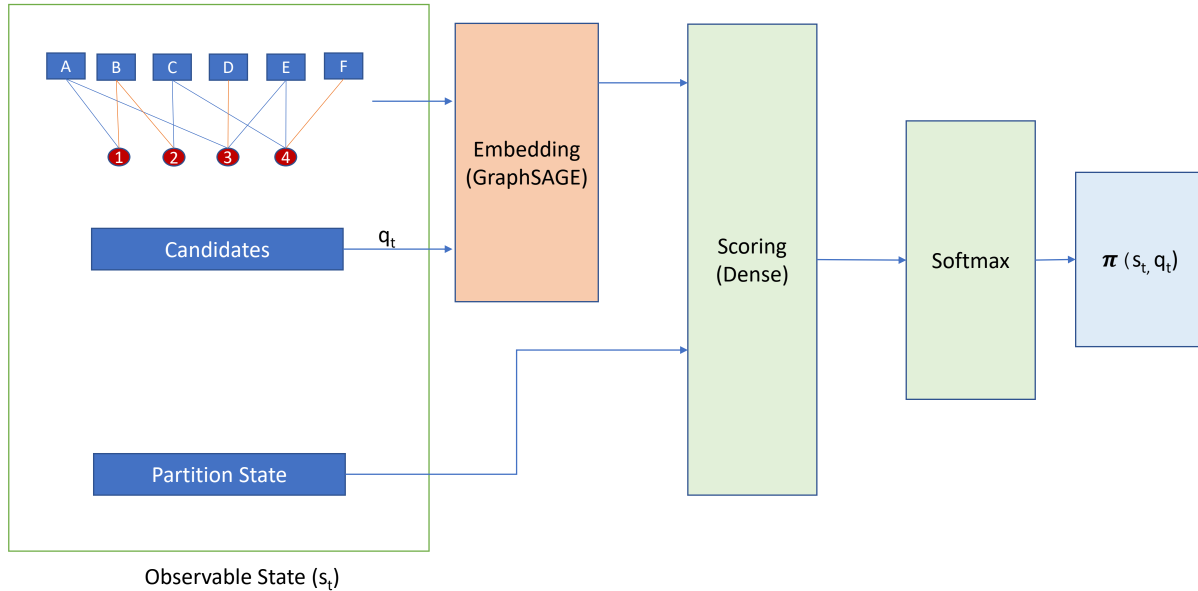



Figure 5.3: Policy neural network architecture

dimensional information about a vertex's graph neighborhood into a dense vector embedding that is suitable as input to downstream layers [69]. Given our bipartite graph representation of the hypergraph, we are able to leverage graph neural network architectures to generate an embedding vector representation for each hyperedge q in $\text{candidates}_t$. We use GraphSAGE [69], a popular GNN architecture in Neuroshard as the embedding layer. We set the hyper-parameter K to 2, which means the computed embedding vectors for a vertex contain information from its 2-hop neighborhood. This means each embedding vector for a hyperedge q encodes information about its adjacent hyperedges, not just the row vertices it accesses. We found that using the LSTM aggregator in GraphSAGE works well in our evaluation. Finally, GraphSAGE requires each vertex in the graph to be initialized with a feature vector. We use the static and dynamic components of the environment (§5.4.1) to create the initial feature vector for each vertex in the bipartite graph. To simplify the implementation, we use the same feature representation for both query and row vertices.

**Scoring Layer** This layer takes as input the embedding vector produced by the embedding layer, as well as the partition state vector (§5.4.1) and produces a single real number representing a "score" for the input hyepredge q. We use a dense fully-connected neural network (with 2 hidden layers) for the scoring layer for its simplicity and efficiency. Note that our network evaluates each hyperedge separately to produce a score without taking the other hyperedges in the candidates set as input. The intuition behind this design decision is that the vectors produced by the embedding layer already encode information about adjacent hyperedges (which are often going to be candidates themselves). Nevertheless, a more sophisticated architecture able look at all the candidate hyperedges simultaneously (such as LSTM [75] or Transformer [145]) might perform better for other more complex workloads, which we leave for future work.

**Softmax Layer** We use a standard Softmax [61] function to map the scores to probabilities.

An action is then sampled based on the probabilities computed for each candidate. The neural network representing $\pi_\theta$(s, q) is trained end-to-end using the training procedure we describe in the following section.

### 5.5.2 Training Procedure

We use a fairly standard training procedure, similar to prior work [100]. The training proceeds in *iterations* until convergence (which can be set as a fixed number of iterations, or until partitioning quality reaches a user-defined threshold). Each iteration consists of running N *episodes*. An episode consists of completely partitioning the hypergraph using Algorithm 3 with the current value of $\theta$, and recording the state, action and reward for each time step in the episode. After completing all episodes, we use this information to apply the modified REINFORCE equation to compute the gradient (as explained in §5.3.2), and take one step in its direction at the end of the iteration. The pseudocode for the training procedure is presented in 4. Note that for simplicity, we assume all episodes take the same number of time-steps L.

Training is an expensive process, but it only needs to be done offline and with less frequency

```
for each iteration do
    Δθ ← 0;
    for episode i := 1 ... N do
        Fully partition the hypergraph and record {s_1^i, a_1^i, r_1^i, ...s_L^i, a_L^i, r_L^i};
        for t := 1 ... L do
            // Compute cumulative reward from t
            // onwards
            v_t^i ← ∑_{s=t}^{L} r_s^i
        end
    end
    for t := 1 ... L do
        b_t ← (1/N) ∑_{i=1}^{N} v_t^i // Compute baseline
        for i := 1 to N do
            Δθ ← Δθ + α ∇_θ logπ_θ(s_t^i, a_t^i)(v_t^i − b_t)
        end
    end
    θ ← θ + Δθ
end
```

**Algorithm 4:** Training Procedure

than the resharding process. Theoretically, the agent can be trained only once and never be updated after that, but retraining regularly has the benefit of being able to adapt to workload changes over time. Neuroshard is primarily designed to be trained on a single workload so that it can discover and exploit workload specific properties, but another possibility is to train on traces collected from different workloads simultaneously, with the purpose of creating a robust, general-purpose sharding agent and avoid overfitting. We think this could be an interesting direction to explore, but we leave that for future work.

### 5.6 Learning with Multiple Objectives

In the previous sections we showed how to model each of the fanout minimization and the load balancing problems as an RL problem individually. While this is potentially useful on its own, a key feature of Neuroshard is the ability to optimize for both simultaneously. In this section we describe how we accomplish this, as well as how we augmented Neuroshard with general support for multi-objective training so that additional objectives can be added as long as a suitable reward

signals can be defined for them.

**Architecture** We use the same policy network (Figure 5.3) when training with multiple objectives. This requires the embedding layer to extract useful features relevant for optimizing all objectives. We considered using multiple embedding layers then aggregating their output into one vector that is then fed to the scoring layer, which could be useful to allow capturing different features independently, but we found that this adds complexity to training, in terms of implementation and speed, without much benefit in our evaluation.

**Rewards** Initially we attempted the following straightforward approach: Simply define the reward $r_t$ as the sum of the individual objective rewards, *i.e.,* at each time-step t, we add the rewards of fan-out minimization and load balancing together. This did not work well in practice, as it suffers from many problems. First, the units of the reward components are very different, which makes it easy for one of the objectives to completely dominate the training. Our attempts at normalizing the units of the rewards (*e.g.,* by using a weighted sum of reward components with tunable weights instead of just giving each component an equal weight of 1) were also not very successful: It was brittle, requires manual tuning and experimentation for each workload and would be very hard to generalize. We also found that certain hyper-parameters (*e.g.,* the learning rate) that work well for one task may not work well for the other. Nevertheless, we gained a valuable insight from this approach: By making the overall reward a linear combination of the individual component rewards, the overall objective function, or loss function in machine learning parlance, is a linear combination of the individual component loss functions, *i.e.,*

$$L = \sum_i w_i L_i$$

This realization made it possible to cast the multi-objective Neuroshard training as a multi-task learning problem, and leverage existing techniques that solve that problem. Multi-Task learning techniques are popular in computer vision, but to the best of our knowledge this is the first time this

is applied to a multi-objective combinatorial optimization problem like hypergraph partitioning. GradNorm [35] is the multi-task learning technique that we use in Neuroshard. It is designed for problems where the overall loss is a linear combination of the individual task loss. Key in GradNorm is to dynamically tune the gradient magnitudes so that gradients for different tasks are placed on a common scale and dynamically adjust gradient norms to ensure that the training for different tasks progresses at similar rates. It is in contrast to joint optimization methods that use a hyperparameter to combine two objectives but the hyperparameter remains the same across training iterations. In order to apply GradNorm in Neuroshard we make the following choices:

- **W**. This is the subset of network weights to apply GradNorm to. In our case since all weights are shared among both tasks we just set W to all the weights in the policy network.
- $\alpha$. This is the "asymmetry" hyper-parameter. It is tunable and might benefit from hyper-parameter search, but intuitively the value of $\alpha$ should be high when the tasks are dissimilar to each other which is generally the case in our setting.

### 5.7 Evaluation

This section describes the implementation, experiments and analysis we performed to evaluate the effectiveness and versatility of Neuroshard. We first describe our implementation of Neuroshard in §5.7.1, and establish the relative cost of different operations using the system §5.7.2. We describe and analyze two micro-benchmarks we developed to illustrate the importance of multi-objective sharding in §5.7.4 and §5.7.4. Finally, we conclude with an analysis of an experiment using the real-world Epinions [104] dataset, which is a social network dataset used to evaluate sharding schemes [41]. We note that tree-structured schemas like those used in the TPC-C benchmark are straightforward to shard effectively, so we do not use it in our evaluation.

All throughput numbers we report are computed as follows: We first run each workload for 1 minute as a warm up, then measure the average queries per second (QPS) for a period of 5 minutes after that.

### 5.7.1 Implementation

In order to measure the performance of different sharding assignments, we use a shared-nothing distributed database where each node is a small Google Cloud VM of type n1-standard-1 (1 vCPU, 3.75 GB memory) running a MariaDB [103] instance. The instances themselves are not sharding-aware, and we load the subset of rows assigned to each instance at the beginning of each run. The clients that are running the queries know the mapping of rows to shards, and they route the queries to each node as appropriate, using the standard XA API [22] to co-ordinate the transaction if the query is distributed.

### 5.7.2 Cost of Query Fanout

To understand the characteristics of our system and the relative cost of different kinds of transactions, we first run different workloads and measure the overall system QPS. In all workloads, the database consists of 1 table which has a single integer-typed column (which is the primary key) and 5000 rows. Each query, regardless of the workload, touches two different rows. Results are in Figure 5.4. **Local RO** is a workload made entirely of read-only queries that run on one server. **Distributed RO** is made of read-only queries that access two rows on two different servers. **Distributed RO (XA)** is similar, but runs its queries in a transaction which needs bracketing via the XA API. **Distributed RW (XA)** queries run a transaction that reads one row from one server and writes another row on a different server, and runs XA two-phase commit. Finally, **Mixed** is made up of 90% Local RO queries and 10% Distributed RW (XA) queries.

**Analysis.** Unsurprisingly, distribution incurs a high cost. A distributed read that does not involve coordination via the XA API is about 2× more expensive than a purely-local read. Reads that involve one round of XA coordination are roughly 3× more expensive than local reads, and writes that use two-phase commit are close to 8× more expensive that local reads. A workload with only 10% distributed writes has just about 60% of the throughput of a workload that is composed of purely-local reads. These experiments highlight the importance of fanout minimization, as well as the different costs of reads and writes.

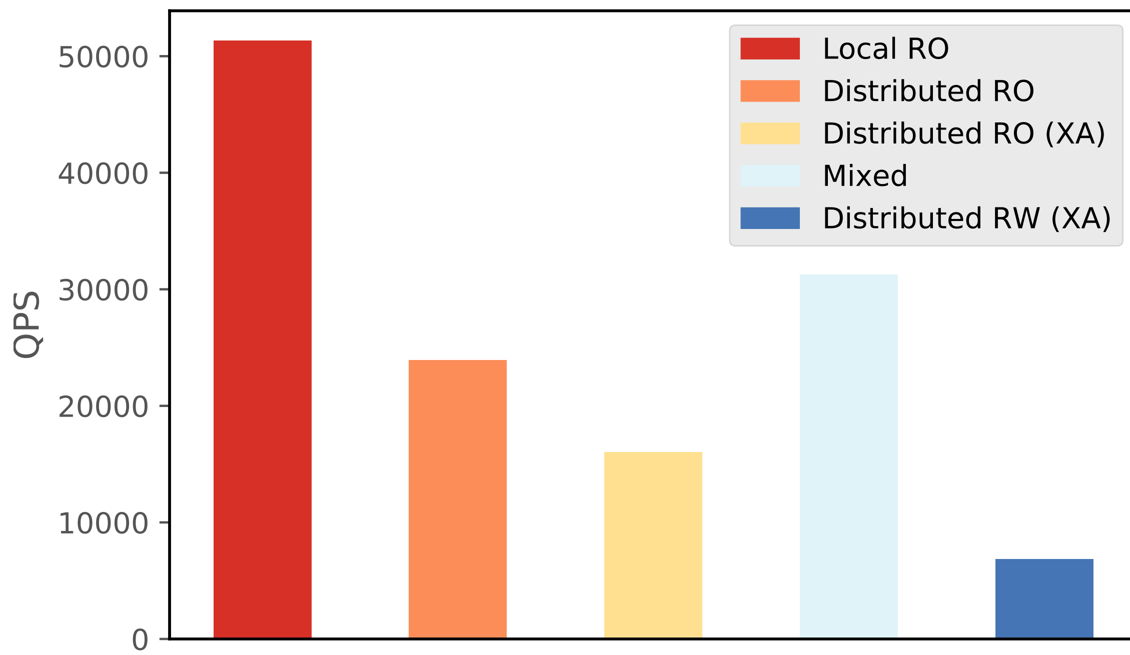



Figure 5.4: System throughput for different kinds of queries.

### 5.7.3 Baselines

For the rest of the section we compare the performance of Neuroshard's sharding with the following baseline approaches:

- **hMetis** [82]. A popular heuristic-based hypergraph partitioner. We use the standard flags by using the **shmetis** executable. For fairness, we set the value of balance flag (UBfactor) to the tightest possible. This is a baseline that is optimized for the fanout objective.
- **HNE**. We also use Algorithm 2 as a fanout optimizing baseline.
- **Random**. This scheme chooses a shard uniformly at random for each row. It is an idealization of hash partitioning, and is optimized for load balancing.

### 5.7.4 Multi-objective Microbenchmarks

To study the versatility of Neuroshard and the importance of optimizing for multiple objectives simultaneously, we created a simple but illustrative synthetic workload.

#### Microbenchmark with Two Shards

The database consists of 500 rows, where each row belongs to exactly one of the following sets:

- **A**. A single hot row
- **B**. A single hot row
- $\mathbf{W_a}$. 249 rows that get accessed together with row **A** in read-write transactions, and with row **B** in read-only transactions.
- $\mathbf{W_b}$. 249 rows that get accessed together with row **B** in read-write transactions, and with row **A** in read-only transactions.

The workload is made up of the following queries:

- 25% are read-only point queries for row **A**.
- 25% are read-only point queries for row **B**.
- 12.5% are read-write queries that access two rows: row **A**, and a row selected uniformly at random from the set $\mathbf{W_a}$.
- 12.5% are read-only queries that access two rows: row **A**, and a row selected uniformly at random from the set $\mathbf{W_b}$.
- 12.5% are read-write queries that access two rows: row **B**, and a row selected uniformly at random from the set $\mathbf{W_b}$.
- 12.5% are read-only queries that access two rows: row **B**, and a row selected uniformly at random from the set $\mathbf{W_a}$.

Our goal is to shard the database into two shards. Note that the ideal sharding in this case is known, it would be to assign both **A** and $\mathbf{W_a}$ to one of the shards, and assign **B** and $\mathbf{W_b}$ to the other one. This way the load is perfectly balanced, and there are no distributed writes, which as we show earlier, are the costliest operation. Also note that any assignment that puts **A** and **B** in the same shard will have identical (worst) performance, because that shard will be serving all of the point queries, and 12.5% of the queries will be distributed read-write transactions.

We generate a trace of 5000 queries and use it for training Neuroshard. We then generate another trace made of 5000 queries that follow the same distribution, and shard it using the different algorithms. We then generate a workload that follows the same distribution and use it to measure the QPS. The results are shown in Figure 5.5. In that figure, Neuroshard-FO refers to a version of Neuroshard that only uses the fanout (FO) reward, but not load balancing. Similarly, Neuroshard-LB only uses the load balancing measure as the reward, but not fanout. Random assigns rows to shards uniformly at random, but we start by assigning A and B to different shards.

**Analysis.** As HNE and Neuroshard-FO only optimize for fanout minimization, they usually place A and B together because they are very prominent in each other's neighborhoods, causing

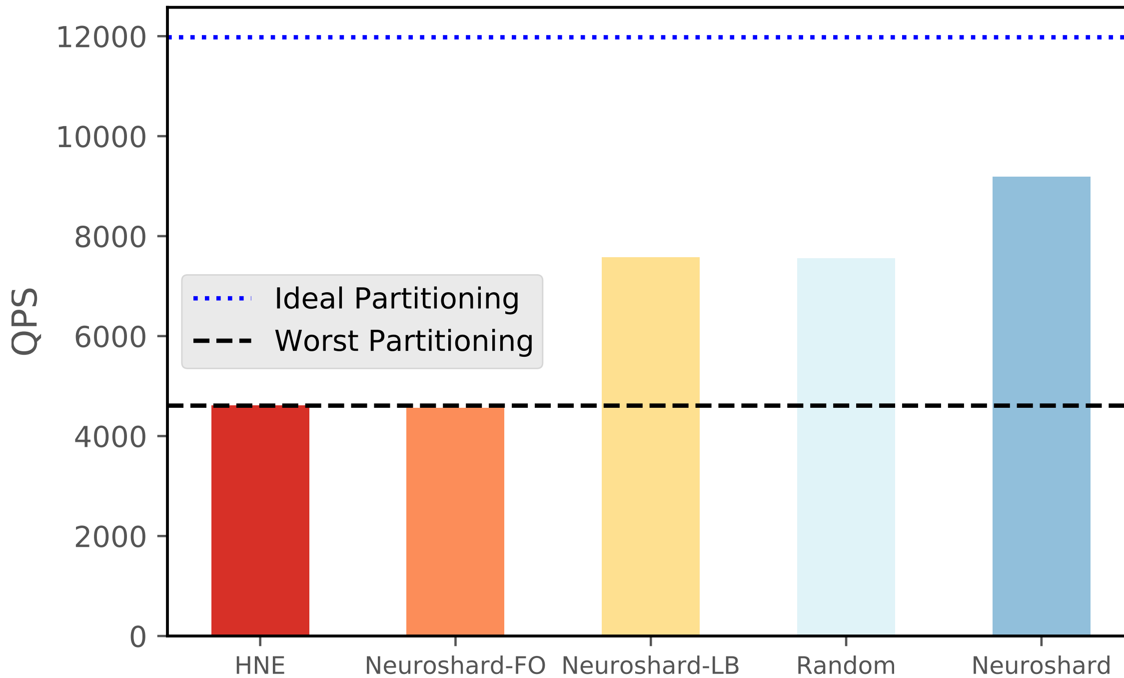



Figure 5.5: Multi objective micro-benchmark throughput with 2 shards

them to have the worst possible performance. Random doesn't have that problem because of our setup, but it can't e.g. prioritize putting the rows of $\mathbf{W_a}$ in the same shard as **A**, leading to a high number of distributed writes. Neuroshard-LB reliably avoids placing **A** and **B** together, but it is unable to distinguish between read-only and read-write queries so it suffers from the same problem as Random. Neuroshard, which optimizes for both objectives and takes hyperedge types into consideration, is therefore able to make the best decisions and its sharding assignments have the highest performance.

**Microbenchmark with Four Shards**

We designed another microbenchmark to compare Neuroshard to two popular baseline sharding approaches, hash partitioning (referred to as Random) and hMetis hypergraph partitioning.

The database consists of 400 rows, where each row belongs to exactly one of the following

sets:

- **Parent Rows**. 4 rows
- **Child Rows**. These are 96 rows in total. Each parent row is associated with 24 child rows.
- **The rest**. 300 rows that get accessed independently

The workload is made up of the following (read-only) queries:

- 1% of the queries read a subset of 2 or more of the parent rows.
- 49% read one parent row, and 0 or 1 of its child rows.
- 50% are point queries that read one of the 300 other rows.

Our goal in this experiment is to shard the database into four shards. Note that if we only cared about fanout minimization, then the perfect solution would be to put all of the Parent and Child rows in a single shard, and distribute the rest equally across the 3 remaining shards. This, however, would mean that 50% of the queries go to a single shard, creating a large load imbalance.

Similar to the prior experiment, we generate a trace of 5000 queries and use it for training. We then generate another trace made of 5000 queries that follow the same distribution, and shard it using different algorithms, then measure the QPS by generating a workload that follows the same distribution. The results are shown in Figure 5.6.

**Analysis.** Both HNE and hMetis compute the sharding that minimizes fanout perfectly by putting all of the Parent and Child rows in a single shard. This results in a large load imbalance as we mentioned previously, causing them to have the worst throughput on this microbenchmark. We find that Neuroshard usually computes a sharding that assigns the parent rows to two shards; two rows each. It almost perfectly co-locates the child rows with their parents, leading to close to zero distributed queries. This better load-balancing of parent rows is the reason why the QPS using Neuroshard is significantly better than HNE or hMetis. Using random hash partitioning on this workload actually has a lot of variance in terms of the overall number of rows assigned to each

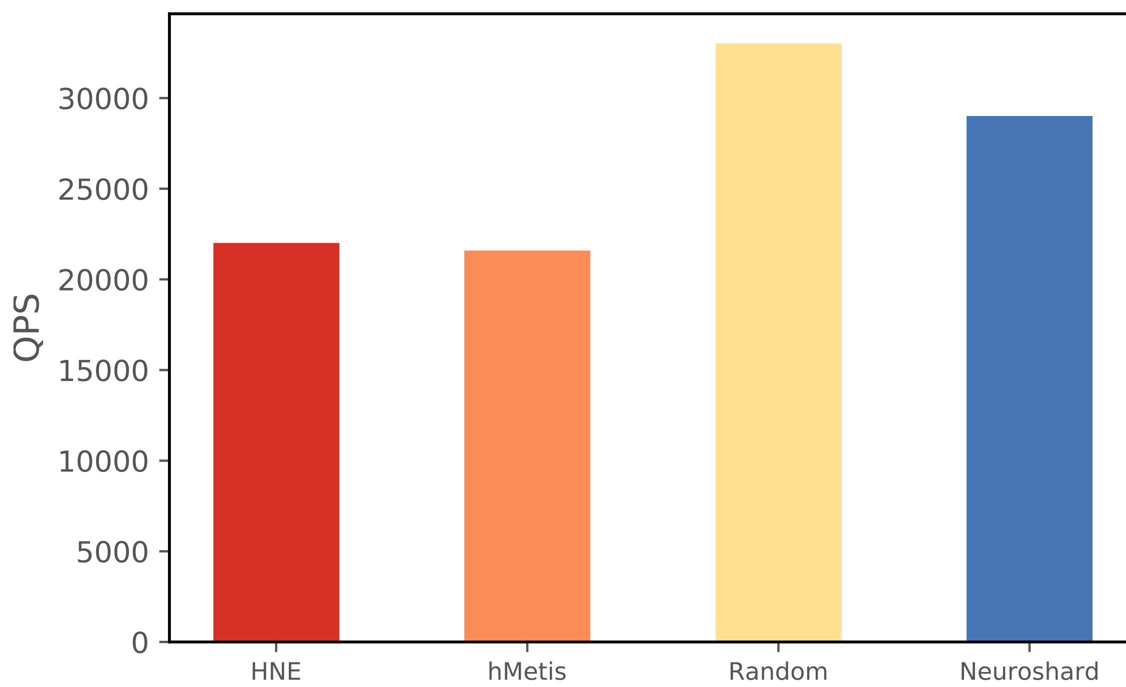



Figure 5.6: 2nd micro-benchmark throughput with 4 shards

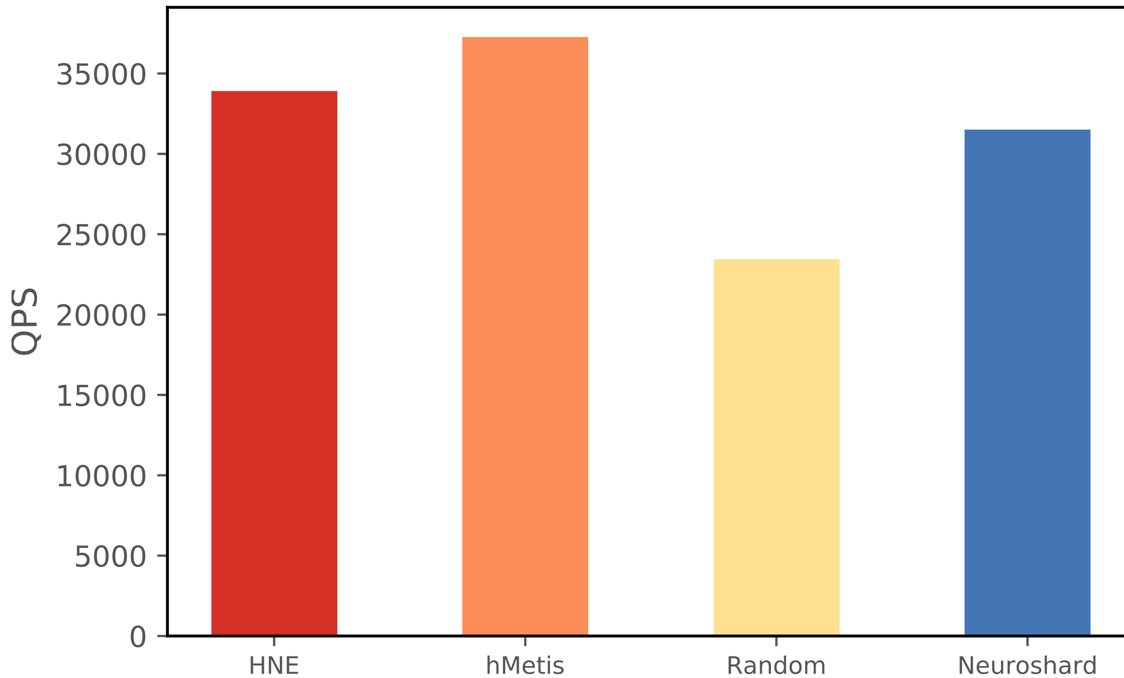



Figure 5.7: Epinions workload throughput with 5 shards

shard, as well as the distribution of parent/child rows among the shards. To reduce the noise we started by assigning one parent row to each shard then applying random partitioning to the rest of the rows. The resulting sharding scheme has the best load balancing in this workload which leads to it having the best QPS despite having a significant share of distributed queries; since this is a read-only workload, the cost of distribution is not quite high. Even though Neuroshard's QPS is not the highest in this workload, these results demonstrate the robustness of Neuroshard compared to single-objective HNE or hMetis.

### 5.7.5 Epinions

We now evaluate Neuroshard's scalability and ability to handle real-world datasets. We use a workload based on the Epinions [104] dataset, a real-world social network that is a popular choice for evaluating sharding schemes because its many-to-many relationships make it hard to shard [41].

The database is composed of two tables: **reviews** and **trust**. Each row in the reviews table has the schema <*user, item, rating*> and represents a review of an item by a user. Each row in the trust table has the schema <*user_a, user_b*> and records the fact that user_a trusts reviews by user_b. We shard the reviews table by the trusting user column (*i.e.,* user_a), and we shard the items table by the item column. In this experiment our goal is to shard the database on 5 MariaDB servers.

**Workload** . There is no available trace of queries from the epinions website, so we designed a workload based on the website's most popular functionality: users that view an item and want to retrieve a list of reviews for that item by the users they trust. Thus, each application-level query is represented by a pair <*user, item*>. Our workload generator can generate these application-level queries based on different distributions. We chose one such distribution in this section for illustration (we got similar results with other distributions): Items are sorted by popularity (*i.e.,* number of reviews), and the top 20% of items are considered *hot*, while the rest is considered *cold*. The workload generator generates a query as follows: First it selects an item **i**, with i being hot with probability 80% and cold with probability 20%. Then, a user **u** is selected uniformly at random from the set of users trusted by i's reviewers. Recall that we implement query routing in the client. If both the i and u are colocated on the same server only one query needs to be sent by the client. Otherwise, the client needs to send queries to two servers. Hence, sharding has a big impact on throughput (up to 50% reduction in SQL queries), and potentially also latency. Depending on the query plan selected by the client it might choose to serialize the requests or execute them in parallel, and then perform the join locally.

**Generating training and test traces** Similar to the other experiments we also generate two traces by the workload generator. The first is a training trace that is made up of 4000 queries, and is used to train Neuroshard. We then generate a partitioning trace of 100,000 queries. This is used by Neuroshard and the other baseline techniques to partition the database. We then measure the overall system QPS given by each technique by sending queries based on the workload. The results are displayed in Figure 5.7.

**Analysis.** Partitioning quality with respect to fanout minimization is the dominant factor in this workload, which is why it is unsurprising that single-objective hMetis and HNE have the best performance, while Random which completely ignores fanout minimization does the worst. Neuroshard appears to suffer a bit due to its attempts at jointly optimizing for load balancing, but its performance is still comparable to HNE. Once again, this shows that Neuroshard is robust to changes in the workload unlike single-objective approaches. Furthermore, we used a small trace for Neuroshard training, and yet it was still able to learn heuristics that were effective on the much larger test trace. This is very important, as it would be unfeasible to use very long traces to train Neuroshard; not only would that be computationally very expensive, but excessively long time horizons are known to diminish the effectiveness of RL algorithms [136, 148].

## 5.8 Related Work

**Hypergraph partitioning.** As an NP-Hard problem with many practical applications, many popular heuristic solvers for hypergraph partitioning have been developed such as hMetis[82] and Zoltan [44]. More recent work such as SHP [78] and HYPE [107] has focused on scalability and the ability to partition large hypergraphs. All these tools only optimize for fanout minimization (or the closely related cut-size). HYPE is also a generalization of Neighbor Expansion [155] but takes a vertex-centric approach compared to the hyperedge centric approach that we take. We adopt some of the performance optimizations they suggested such as limiting the size of the candidate set in our implementation.

**Sharding as (hyper)graph partitioning.** Schism [41] pioneered modelling database sharding and replication as a graph partitioning problem. SWORD [118] builds up on this approach but uses coarser granularity and applies hypergraph partitioning instead. They use heuristic solvers like Metis [83] and hMetis [82] which optimize for a single objective, unlike Neuroshard, but they also model replication which Neuroshard currently does not.

**Online sharding.** Clay [129], E-Store [138] are online heuristic sharding algorithms that actively monitor the system to identify hot tuples, and then potentially migrate them (along with

other jointly accessed row to a different server). However, these systems make some assumptions about the workload: Clay primarily target in-memory workloads with a high skew in load, E-Store assumes all tables form a tree-schema based on foreign key relationships. In contrast, Neuroshard aims to be broadly applicable to a variety of workloads and environments. It would be interesting to investigate how a similar RL approach can improve online sharding.

**Deep learning for combinatorial optimization.** A large recent body of work (e.g. [2, 15, 16, 91]) has explored using deep learning for solving NP-Hard combinatorial problems in general, and on graph problems in particular. Dai et. al. [42] proposed a general framework for solving optimization problems over graphs using RL. Their approach uses deep-Q-learning and requires learning a representation for the entire graph, as opposed to the policy methods we use that needs only to learn a localized policy. To the best of our knowledge, we are the first to explore using deep RL for multi-objective hypergraph partitioning.

**Deep RL for systems.** Deep RL has been applied successfully to a wide range of problems from video streaming [101] and network traffic scheduing [33, 36] to compute resource management [100] and even physical device placement [110]. We were inspired by DeepRM [100] and Decima [102], which apply Deep RL to the resource management and job scheduling domains. Both use policy methods, and Decima also uses GNNs to produce state representation. Unlike Neuroshard, the graphs in Decima's state are comparatively small, and while Decima and DeepRM support different objectives, users would have to select just one to use in training. Both systems do not optimize for multiple objectives simultaneously.

**Deep RL for Partitioning**. Recent work (e.g., [30, 50, 74, 165]) has explored using Deep RL methods for partitioning. GridFormation [50] proposes an online partitioning approach using Deep-Q-learning, which is applied to vertical partitioning in [30]. In contrast, Neuroshard is an offline approach for horizontal sharding. Hilprecht et. al. [74] use deep-Q-learning to build a general-purpose partitioning advisor that can recommend a table partitioning scheme for a new workload. It targets analytical workloads in which each table can be partitioned on one or more columns, while Neuroshard's focus is OLTP workloads and supports fine-grained row to shard

assignments. The state and action spaces in this setting are much smaller than in Neuroshard, but evaluating agent's solutions and computing the rewards is much more computationally challenging.

### 5.9 Conclusions

Neuroshard is the first system that uses a learned approach for directly learning horizontal sharding assignments. It models past accesses to rows as a hypergraph, and then applies deep RL to score a relatively small set of candidates derived from the rows accessed together as part of the same query. It uses GradNorm, a multi-task learning technique, to automatically tune the gradients of multiple different task losses, so that it can balance between multiple different objectives simultaneously.

This work takes the first steps in realizing a learned sharding system, but leaves several open problems. First, for large and complex production traces, we expect that we will need to devise sampling techniques to keep the episode length from being too large, while allowing Neuroshard to learn effective local heuristics. Second, while building one partition at a time is simple and efficient, we suspect extending Neuroshard to build more than one simultaneously would allow it exploit power of two choices more effectively. Finally, we do not address how to transition to a new partitioning scheme when one is computed, nor do we attempt to minimize data movement when computing a new partitioning.

# Conclusion

This work shows that modern advances in AI, networking and storage technologies unlock practical and efficient and *general* distributed ACID transaction systems beyond what was commonly believed possible. We designed and built 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. We showed how to workaround the unavailability of such a fast 2PC technology stack, by designing a high-performance distributed commit protocol in the Orleans system which takes advantage of early lock release. Finally, we designed Neuroshard to effectively partition workloads to minimize the need for distributed transactions in the first place, without sacrificing the important system objective of load balancing.

# 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.

[2] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, "Solving np-hard problems on graphs by reinforcement learning without domain knowledge," *arXiv preprint arXiv:1905.11623*, 2019.

[3] A. Adya *et al.*, "Slicer: Auto-sharding for datacenter applications," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 739–753, ISBN: 978-1-931971-33-1.

[4] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986, ISBN: 0262010925.

[5] *AlloyDB for PostgreSQL.* https://cloud.google.com/alloydb.

[6] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The cost of serializability on platforms that use snapshot isolation," in *2008 IEEE 24th International Conference on Data Engineering*, 2008, pp. 576–585.

[7] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[8] P. Antonopoulos *et al.*, "Socrates: The new sql server in the cloud," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, 1743–1756, ISBN: 9781450356435.

[9] J. Armstrong, "Erlang," *Commun. ACM*, vol. 53, no. 9, 68–75, 2010.

[10] D. F. Bacon *et al.*, "Spanner: Becoming a sql system," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, 331–343, ISBN: 9781450341974.

[11] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, 181–192, 2013.

[12] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "HAT, not CAP: Towards highly available transactions," in *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, Santa Ana Pueblo, NM: USENIX Association, May 2013.

[13] J. Baker *et al.*, “Megastore: Providing scalable, highly available storage for interactive services,” in *CIDR*, 2011.

[14] M. Balakrishnan *et al.*, “Tango: Distributed data structures over a shared log,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13, Farminton, Pennsylvania: Association for Computing Machinery, 2013, 325–340, ISBN: 9781450323888.

[15] T. D. Barrett, W. R. Clements, J. N. Foerster, and A. Lvovsky, “Exploratory combinatorial optimization with reinforcement learning,” *arXiv preprint arXiv:1909.04063*, 2019.

[16] I. Bello, H. Pham, Q. Le, M. Norouzi, and S. Bengio, “Neural combinatorial optimization with reinforcement learning,” in *ICLR (Workshop)*, 2017.

[17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ansi sql isolation levels,” ser. SIGMOD ’95, San Jose, California, USA: Association for Computing Machinery, 1995, 1–10, ISBN: 0897917316.

[18] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, “Orleans: Distributed virtual actors for programmability and scalability,” Tech. Rep. MSR-TR-2014-41, 2014.

[19] P. A. Bernstein, “Actor-oriented database systems,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 13–14.

[20] P. A. Bernstein, M. Dashti, T. Kiefer, and D. Maier, “Indexing in an actor-oriented database,” in *CIDR*, 2017.

[21] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987, ISBN: 0-201-10715-5.

[22] P. A. Bernstein and E. Newcomer, *Principles of transaction processing*. Morgan Kaufmann, 2009, pp. 330–x336.

[23] P. A. Bernstein, C. W. Reid, and S. Das, “Hyder - a transactional record manager for shared flash.,” in *CIDR*, www.cidrdb.org, 2011, pp. 9–20.

[24] P. A. Bernstein *et al.*, “Geo-distribution of actor-based services,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 2017.

[25] P. B. Bernstein, “Resurrecting middle-tier distributed transactions,” *IEEE Data Eng. Bull.*, vol. 42, pp. 3–6, 2019.

[26] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, “The end of slow networks: It’s time for a redesign,” *Proc. VLDB Endow.*, vol. 9, no. 7, 528–539, 2016.

[27] E. Brewer, "Spanner, truetime and the cap theorem," Tech. Rep., 2017.

[28] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," in *ACM Symposium on Cloud Computing (SOCC 2011)*, ACM, 2011.

[29] B. Calder *et al.*, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, Cascais, Portugal: Association for Computing Machinery, 2011, 143–157, ISBN: 9781450309776.

[30] G. Campero Durand, R. Piriyev, M. Pinnecke, D. Broneske, B. Gurumurthy, and G. Saake, "Automated vertical partitioning with deep reinforcement learning," in *New Trends in Databases and Information Systems*, 2019, pp. 126–134.

[31] S. Chandrasekaran and R. Bamford, "Shared cache - the future of parallel databases," in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, 2003, pp. 840–850.

[32] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06, Seattle, WA: USENIX Association, 2006, p. 15.

[33] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proceedings of the 2018 conference of the ACM special interest group on data communication*, 2018, pp. 191–205.

[34] Y. Chen, X. Yu, P. Koutris, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Plor: General transactions with predictable, low tail latency," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, Philadelphia, PA, USA: Association for Computing Machinery, 2022, 19–33, ISBN: 9781450392495.

[35] Z. Chen, V. Badrinarayanan, C.-Y. Lee, and A. Rabinovich, "GradNorm: Gradient normalization for adaptive loss balancing in deep multitask networks," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 80, 2018, pp. 794–803.

[36] S. Chinchali *et al.*, "Cellular network traffic scheduling with deep reinforcement learning," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[37] *Cloud Spanner | Google Cloud*, `https://cloud.google.com/spanner`, 2023.

[38] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, 143–154, ISBN: 9781450300360.

[39] B. F. Cooper *et al.*, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, 1277–1288, 2008.

[40] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 2013.

[41] C. Curino, E. P. C. Jones, Y. Zhang, and S. R. Madden, "Schism: A workload-driven approach to database replication and partitioning," 2010.

[42] H. Dai, E. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 6348–6358.

[43] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, Stevenson, Washington, USA: Association for Computing Machinery, 2007, 205–220, ISBN: 9781595935915.

[44] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science & Engineering*, vol. 4, no. 2, pp. 90–96, 2002.

[45] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The gamma database machine project," 1990.

[46] D. J. DeWitt, S Ghanderaizadeh, and D. Schneider, "A performance analysis of the gamma database machine," in *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1988, pp. 350–360.

[47] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84, Boston, Massachusetts: Association for Computing Machinery, 1984, 1–8, ISBN: 0897911288.

[48] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "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)*, USENIX Association, Feb. 2021, pp. 33–49, ISBN: 978-1-939133-20-5.

[49] A. Dragojevic *et al.*, "No compromises: Distributed transactions with consistency, availability, and performance," in *Symposium on Operating Systems Principles (SOSP'15)*, ACM – Association for Computing Machinery, 2015.

[50] G. C. Durand *et al.*, “Gridformation: Towards self-driven online data partitioning using reinforcement learning,” in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, ser. aiDM’18, 2018.

[51] T. Eldeeb and P. Bernstein, “Transactions for distributed actors in the cloud,” Tech. Rep. MSR-TR-2016-1001, 2016.

[52] T. Eldeeb, P. A. Bernstein, A. Cidon, and J. Yang, “Chablis: Fast and general transactions in geo-distributed systems,” in *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*, www.cidrdb.org, 2024.

[53] T. Eldeeb, S. Burckhardt, R. Bond, A. Cidon, J. Yang, and P. A. Bernstein, “Cloud actor-oriented database transactions in orleans,” *Proc. VLDB Endow.*, vol. 17, no. 12, 3720–3730, Aug. 2024.

[54] T. Eldeeb, Z. Chen, A. Cidon, and J. Yang, “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*, ser. aiDM ’22, Philadelphia, Pennsylvania: Association for Computing Machinery, 2022, ISBN: 9781450393775.

[55] T. Eldeeb, X. Xie, P. A. Bernstein, A. Cidon, and J. Yang, “Chardonnay: Fast and general datacenter transactions for On-Disk databases,” in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA: USENIX Association, Jul. 2023.

[56] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi, “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.

[57] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, “The notions of consistency and predicate locks in a database system,” *Commun. ACM*, vol. 19, no. 11, 624–633, 1976.

[58] *Fauna.* `https://fauna.com/`.

[59] *FlatBuffers*, `https://google.github.io/flatbuffers/`, 2022.

[60] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, 51–59, 2002.

[61] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[62] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch, “Controlled lock violation,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of*

*Data*, ser. SIGMOD '13, New York, New York, USA: Association for Computing Machinery, 2013, 85–96, ISBN: 9781450320375.

[63] J. Gray, "Notes on data base operating systems," in *Advanced Course: Operating Systems*, 1978.

[64] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," ser. SIGMOD '96, Montreal, Quebec, Canada, 1996, 173–182, ISBN: 0897917944.

[65] *gRPC*. `https://grpc.io/`.

[66] H. Guo, X. Zhou, and L. Cai, "Lock violation for fault-tolerant distributed database system," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1416–1427.

[67] Z. Guo, K. Wu, C. Yan, and X. Yu, "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*, ser. SIGMOD '21, Virtual Event, China: Association for Computing Machinery, 2021, 658–670, ISBN: 9781450383431.

[68] Z. Guo *et al.*, "Cornus: Atomic commit for a cloud DBMS with storage disaggregation," *Proc. VLDB Endow.*, vol. 16, no. 2, pp. 379–392, 2022.

[69] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[70] M. Hanai, T. Suzumura, W. J. Tan, E. Liu, G. Theodoropoulos, and W. Cai, "Distributed edge partitioning for trillion-edge graphs," *Proc. VLDB Endow.*, 2379–2392, Sep. 2019.

[71] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *Proc. VLDB Endow.*, vol. 10, no. 5, 553–564, 2017.

[72] P. Helland, "Life beyond distributed transactions: An apostate's opinion," in *CIDR*, 2007.

[73] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, 463–492, 1990.

[74] B. Hilprecht, C. Binnig, and U. Röhm, "Learning a partitioning advisor for cloud databases," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, 2020, 143–157.

[75] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, 1997.

[76] *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.

[77] J. W. Josten, C. Mohan, I. Narang, and J. Z. Teng, “Db2’s use of the coupling facility for data sharing,” *IBM Systems Journal*, vol. 36, no. 2, pp. 327–351, 1997.

[78] I. Kabiljo *et al.*, “Social hash partitioner: A scalable distributed hypergraph partitioner,” *arXiv preprint arXiv:1707.06665*, 2017.

[79] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, “Shinjuku: Preemptive scheduling for μsecond-scale tail latency,” in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’19, Boston, MA, USA: USENIX Association, 2019, 345–359, ISBN: 9781931971492.

[80] A. Kalia, M. Kaminsky, and D. Andersen, “Datacenter RPCs can be general and fast,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, pp. 1–16, ISBN: 978-1-931971-49-2.

[81] A. Kalia, M. Kaminsky, and D. G. Andersen, “FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 185–201, ISBN: 978-1-931971-33-1.

[82] G. Karypis, “Hmetis 1.5: A hypergraph partitioning package,” *Technical Report*, 1998.

[83] G. Karypis and V. Kumar, “METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices,” 1997.

[84] P. Kraft, F. Kazhamiaka, P. Bailis, and M. Zaharia, “Data-Parallel actors: A programming model for scalable query serving systems,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA: USENIX Association, Apr. 2022, pp. 1059–1074, ISBN: 978-1-939133-27-4.

[85] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, “Mdcc: Multi-data center consistency,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys ’13, Prague, Czech Republic: Association for Computing Machinery, 2013, 113–126, ISBN: 9781450319942.

[86] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, 133–169, 1998.

[87] B. W. Lampson and D. B. Lomet, “A new presumed commit optimization for two phase commit,” in *Proceedings of the 19th International Conference on Very Large Data Bases*,

ser. VLDB '93, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, 630–640, ISBN: 155860152X.

[88] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang, "High performance transactions in deuteronomy," in *Conference on Innovative Data Systems Research (CIDR 2015)*, 2015.

[89] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang, "Multi-version range concurrency control in deuteronomy," *Proc. VLDB Endow.*, vol. 8, no. 13, 2146–2157, 2015.

[90] *LevelDB*. `https://leveldb.org/`.

[91] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Advances in Neural Information Processing Systems*, 2018, pp. 539–548.

[92] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, 21–35, ISBN: 9781450341974.

[93] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang, "Towards a non-2pc transaction management in distributed database systems," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: Association for Computing Machinery, 2016, 1659–1674, ISBN: 9781450335317.

[94] Y. Liu, L. Su, V. Shah, Y. Zhou, and M. A. Vaz Salles, "Hybrid deterministic and nondeterministic execution of transactions in actor systems," ser. SIGMOD '22, Philadelphia, PA, USA: Association for Computing Machinery, 2022, 65–78, ISBN: 9781450392495.

[95] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "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*, ser. SOSP '11, Cascais, Portugal: Association for Computing Machinery, 2011, 401–416, ISBN: 9781450309776.

[96] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for Low-Latency Geo-Replicated storage," in *NSDI 13*, Lombard, IL: USENIX Association, Apr. 2013, pp. 313–328, ISBN: 978-1-931971-00-3.

[97] D. B. Lomet and M. F. Mokbel, "Locking key ranges with unbundled transaction services," *Proc. VLDB Endow.*, vol. 2, pp. 265–276, 2009.

[98] Y. Lu, X. Yu, L. Cao, and S. Madden, "Aria: A fast and practical deterministic oltp database," *Proc. VLDB Endow.*, vol. 13, no. 12, 2047–2060, 2020.

[99] Y. Lu, X. Yu, L. Cao, and S. Madden, “Epoch-based commit and replication in distributed oltp databases,” *Proc. VLDB Endow.*, vol. 14, no. 5, 743–756, 2021.

[100] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, “Resource management with deep reinforcement learning,” in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, pp. 50–56.

[101] H. Mao, R. Netravali, and M. Alizadeh, “Neural adaptive video streaming with pensieve,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 197–210.

[102] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, “Learning scheduling algorithms for data processing clusters,” in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 270–288.

[103] *MariaDB Server: The open source relational database*, `https://mariadb.org/`, 2021.

[104] P. Massa and P. Avesani, “Controversial users demand local trust metrics: An experimental study on epinions.com community,” *AAAI’05*, 2005.

[105] Y. Matsunobu, S. Dong, and H. Lee, “Myrocks: Lsm-tree database storage engine serving facebook’s social graph,” *Proc. VLDB Endow.*, vol. 13, no. 12, 3217–3230, 2020.

[106] Y. Matsunobu, S. Dong, and H. Lee, “MyRocks: LSM-tree database storage engine serving Facebook’s social graph,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.

[107] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, and K. Rothermel, “Hype: Massive hypergraph partitioning with neighborhood expansion,” in *2018 IEEE International Conference on Big Data (Big Data)*, IEEE, 2018, pp. 458–467.

[108] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, “I can’t believe it’s not causal! scalable causal consistency with no slowdown cascades,” ser. NSDI’17, Boston, MA, USA: USENIX Association, 2017, 453–468, ISBN: 9781931971379.

[109] *Microsoft Orleans*, `docs.microsoft.com/dotnet/orleans`, 2023.

[110] A. Mirhoseini *et al.*, “Device placement optimization with reinforcement learning,” in *International Conference on Machine Learning*, PMLR, 2017, pp. 2430–2439.

[111] V. Mnih *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, 2015.

[112] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the r* distributed database management system," *ACM Trans. Database Syst.*, vol. 11, no. 4, 378–396, 1986.

[113] G. E. Monahan, "State of the art - a survey of partially observable markov decision processes: Theory, models, and algorithms," *Management Science*, 1982.

[114] S. Mu, L. Nelson, W. Lloyd, and J. Li, "Consolidating concurrency control and consensus for commits under conflicts," ser. OSDI'16, Savannah, GA, USA: USENIX Association, 2016, 517–532, ISBN: 9781931971331.

[115] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Minimizing commit latency of transactions in geo-replicated data stores," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, 1279–1294, ISBN: 9781450327589.

[116] C. D. T. Nguyen, J. K. Miller, and D. J. Abadi, "Detock: High performance multi-region transactions at scale," *Proc. ACM Manag. Data*, vol. 1, no. 2, 2023.

[117] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319, ISBN: 978-1-931971-10-2.

[118] A. Quamar, K. A. Kumar, and A. Deshpande, "Sword: Scalable workload-aware data placement for transactional workloads," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13, Genoa, Italy, 2013, 430–441.

[119] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *Proc. VLDB Endow.*, vol. 4, no. 4, 243–254, 2011.

[120] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design principles for scaling multi-core oltp under high contention," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: Association for Computing Machinery, 2016, 1583–1598, ISBN: 9781450335317.

[121] K. Ren, D. Li, and D. J. Abadi, "Slog: Serializable, low-latency, geo-replicated transactions," *Proc. VLDB Endow.*, vol. 12, no. 11, 1747–1761, 2019.

[122] K. Ren, A. Thomson, and D. J. Abadi, "An evaluation of the advantages and disadvantages of deterministic database systems," *Proc. VLDB Endow.*, vol. 7, no. 10, 821–832, 2014.

[123] D. R. Ries and R. S. Epstein, *Evaluation of Distribution Criteria for Distributed Data Base Systems*. Electronics Research Laboratory, College of Engineering, University of California, 1978.

[124] *RocksDB. A persistent key-value store for fast storage environments*, https://rocksdb.org/, 2022.

[125] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "System level concurrency control for distributed database systems," *ACM Trans. Database Syst.*, vol. 3, no. 2, 178–198, 1978.

[126] J. B. Rothnie *et al.*, "Introduction to a system for distributed databases (sdd-1)," *ACM Trans. Database Syst.*, vol. 5, no. 1, 1–17, 1980.

[127] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency," in *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA: USENIX Association, May 2011.

[128] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 37, 2015, pp. 1889–1897.

[129] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker, "Clay: Fine-grained adaptive partitioning for general database schemas," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 445–456, 2016.

[130] A. Shamis *et al.*, "Fast general distributed transactions with opacity," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, 433–448, ISBN: 9781450356435.

[131] J. Shute *et al.*, "F1: A distributed SQL database that scales," *Proc. VLDB Endow.*, vol. 6, no. 11, 1068–1079, Aug. 2013.

[132] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[133] D. Skeen, "Nonblocking commit protocols," in *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '81, Ann Arbor, Michigan: Association for Computing Machinery, 1981, 133–142, ISBN: 0897910400.

[134] Y. J. Song, M. K. Aguilera, R. Kotla, and D. Malkhi, "Rpc chains: Efficient client-server communication in geodistributed systems," in *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, USENIX, 2009.

[135] M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed ingres," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 188–194, 1979.

[136] R. S. Sutton and A. G. Barto, *Reinforcement Learning*. MIT Press, 2018.

[137] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, 1999, 1057–1063.

[138] R. Taft *et al.*, "E-store: Fine-grained elastic partitioning for distributed transaction processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 245–256, 2014.

[139] R. Taft *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020, 1493–1509, ISBN: 9781450367356.

[140] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proc. VLDB Endow.*, vol. 3, no. 1–2, 70–80, 2010.

[141] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, 1–12, ISBN: 9781450312479.

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

[143] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: Association for Computing Machinery, 2013, 18–32, ISBN: 9781450323888.

[144] *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`.

[145] A. Vaswani *et al.*, "Attention is all you need," in *Advances in neural information processing systems*, 2017.

[146] A. Verbitski *et al.*, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *SIGMOD 2017*, 2017.

[147] A. Verbitski *et al.*, "Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes," *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[148] H. Wang, H. He, M. Alizadeh, and H. Mao, "Learning caching policies with subsampling," in *NeurIPS Machine Learning for Systems Workshop*, 2019.

[149] T. Warszawski and P. Bailis, “Acidrain: Concurrency-related attacks on database-backed web applications,” ser. SIGMOD ’17, Chicago, Illinois, USA, 2017, 5–20, ISBN: 9781450341974.

[150] X. Yan *et al.*, “Carousel: Low-latency transaction processing for globally-distributed data,” in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD ’18, Houston, TX, USA: Association for Computing Machinery, 2018, 231–243, ISBN: 9781450347037.

[151] Z. Yang *et al.*, “Oceanbase: A 707 million tpmc distributed relational database system,” *Proc. VLDB Endow.*, vol. 15, no. 12, pp. 3385–3397, 2022.

[152] *yugabyteDB*. `https://yugabyte.com/`.

[153] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, “The end of a myth: Distributed transactions can scale,” *Proc. VLDB Endow.*, vol. 10, no. 6, 685–696, 2017.

[154] E. Zamanian, J. Shun, C. Binnig, and T. Kraska, “Chiller: Contention-centric transaction execution and data partitioning for modern networks,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20, Portland, OR, USA: Association for Computing Machinery, 2020, 511–526, ISBN: 9781450367356.

[155] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, “Graph edge partitioning via neighborhood heuristic,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 605–614.

[156] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, “Building consistent transactions with inconsistent replication,” in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP ’15, Monterey, California: Association for Computing Machinery, 2015, 263–278, ISBN: 9781450338349.

[157] I. Zhang *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.

[158] M. Zhang, Y. Hua, P. Zuo, and L. Liu, “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: USENIX Association, Feb. 2022, pp. 51–68, ISBN: 978-1-939133-26-7.

[159] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, “Transaction chains: Achieving serializability with low latency in geo-distributed storage systems,” ser. SOSP ’13, Farminton, Pennsylvania, 2013, 276–291, ISBN: 9781450323888.

[160] Y. Zhong *et al.*, “Bpf for storage: An exokernel-inspired approach,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS ’21, Ann Arbor, Michigan: Association for Computing Machinery, 2021, 128–135, ISBN: 9781450384384.

[161] Y. Zhong *et al.*, “XRP: In-Kernel storage functions with eBPF,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022.

[162] J. Zhou *et al.*, “Graph neural networks: A review of methods and applications,” *AI Open*, vol. 1, pp. 57–81, 2020.

[163] J. Zhou *et al.*, “Foundationdb: A distributed unbundled transactional key value store,” in *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, ACM, 2021, pp. 2653–2666.

[164] *ZippyDB: Facebook’s key value store*, `https://engineering.fb.com/2021/08/06/core-data/zippydb/`.

[165] J. Zou *et al.*, “Lachesis: Automated partitioning for udf-centric analytics,” *Proc. VLDB Endow.*, vol. 14, no. 8, pp. 1262–1275, 2021.