Abstract of “Distributed Transaction Processing on Modern RDMA-enabled Networks” by Erfan Zamanian Dolati, Ph.D., Brown University, April 2020.

The development of modern RDMA-enabled networks in data centers calls for a fundamental re-evaluation of how distributed data stores should be designed. Conventionally, many data structures, algorithms, and on the higher level, the architecture of these systems were primarily centered around the assumption that network communication is the dominant bottleneck and thus should be avoided as much as possible. However, in the light of recent advances in modern interconnects, this assumption is no longer valid, as RDMA functionality, together with high bandwidth and low latency of these networks enables direct remote access with significantly more efficiency and lower overhead compared to conventional networks.

In this work, we first present a novel distributed OLTP system and show that it can truly scale even when all transactions span multiple machines. Motivated by this insight, we then argue that the conventional data partitioning techniques are no longer optimal in this new environment. To that end, we propose a new data partitioning scheme that optimizes for contention, which is the real bottleneck in the next generation of transactional databases. Finally, we tackle the problem of high availability in OLTP systems, and propose a new replication scheme that efficiently leverages RDMA to completely eliminate the processing redundancy in existing techniques.
Distributed Transaction Processing on Modern RDMA-enabled Networks

by
Erfan Zamanian Dolati
B. S., Sharif University of Technology, Iran, 2010
Sc. M., ETH Zurich, Switzerland, 2013

A dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the Department of Computer Science at Brown University

Providence, Rhode Island
April 2020
© Copyright 2020 by Erfan Zamanian Dolati
This dissertation by Erfan Zamanian Dolati is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ________________  ____________________________________________

Tim Kraska, Director

Recommended to the Graduate Council

Date ________________  ____________________________________________

Rodrigo Fonseca, Reader

Date ________________  ____________________________________________

Carsten Binnig, Reader
(TU Darmstadt, Germany)

Approved by the Graduate Council

Date ________________  ____________________________________________

Andrew G. Campbell
Dean of the Graduate School
Acknowledgements

First and foremost, I would like to thank my advisor, Tim Kraska. Throughout my years at Brown University, I received endless guidance and support from Tim. Working alongside him and under his supervision was indeed a unique experience. He always instilled confidence in our ideas and research agenda. I am greatly indebted to him for teaching me how to start from a simple, raw idea and realize it to a mature and presentable piece of work through dedication and hard work.

I am also thankful to my other committee members, Carsten Binnig and Rodrigo Fonseca for their insightful feedback on my work. My collaboration with Carsten started prior to joining Brown, and has continued to this very moment. In all these years, he has always been an invaluable friend and a truly inspiring researcher. I give special thanks to my former advisor, Donald Kossmann, for leading me to the field of data management systems during my master studies at ETH Zurich. Donald offered unceasing support in my master thesis project, and I have always considered him my academic role model.

Throughout my years in Ph.D., I had the privilege of collaborating with so many inspiring minds, including Micheal Stonebraker, Julian Shun, Xiangyao Yu and Tim Harris. I learned invaluable lessons from each and every one of these diligent and hard working people.

I thank all my fellow lab mates, Amir, Yeounoh, Kayhan, Ani, Alex, Andrew, Eric, Sam, John, Leonhard, and Phillip. I am also thankful to so many other people I met at Brown. Thanks to Kavosh, Lauren, Ahmad, Tobias and Mohammad.

I am especially thankful to my wife, Elnaz, for her tireless support and endless love, and more importantly for never giving up on me even when, at times, I was struggling to not give up on myself. Elnaz kept me sane throughout all these years of ups and downs, and my Ph.D. journey would be impossible without her.

Finally, I would like to thank my parents, Mohsen and Sedigheh, and my two brothers, Ehsan and Keyhan, for their constant support and encouragement. They gave me the strength to pursue my dreams.
# Contents

List of Tables ix

List of Figures x

1 Introduction 1
   1.1 Modern Online Transaction Processing . 2
      1.1.1 Challenges . 3
   1.2 Emerging Hardware: Opportunities and Requirements . 3
   1.3 Summary of Thesis Contributions and Outline . 4

2 Background on RDMA 6
   2.1 RDMA and its Advantages . 6
   2.2 RDMA-enabled Networks . 7
   2.3 RDMA Details . 8
      2.3.1 The Verb API and the Supported Operations . 8
      2.3.2 Queue Pairs and their Different Types . 8
      2.3.3 RDMA Workflow . 9
   2.4 RDMA-enabled Network vs. Local Memory . 10
   2.5 Micro-Benchmarks . 11
      2.5.1 Experimental Setup . 11
      2.5.2 Throughput and Latency . 11
      2.5.3 CPU Overhead . 12
      2.5.4 Conclusion . 13

3 Scalable Distributed Transaction Processing 14
   3.1 Motivation . 14
      3.1.1 On Un-scalability of Distributed Transactions . 15
      3.1.2 The Need for a System Redesign . 15
   3.2 Contributions and Chapter Organization . 16
   3.3 Network-Attached-Memory Architecture . 16
      3.3.1 Main Idea . 16
4 Handling Data Contention

4.1 Introduction .............................................. 39
   4.1.1 Motivating Example ................................. 40
   4.1.2 Challenges ......................................... 40

4.2 Contributions and Chapter Organization ..................... 41

4.3 Overview .................................................. 41
   4.3.1 Transaction Processing with 2PL & 2PC ............... 41
   4.3.2 Contention-Aware Transactions ....................... 42
   4.3.3 Discussion ......................................... 42

4.4 Two-region Execution ....................................... 43
   4.4.1 General Overview .................................. 43
   4.4.2 Constructing a Dependency Graph ................... 44
   4.4.3 Run-Time Decision .................................. 45
   4.4.4 Challenges ......................................... 46

4.5 Contention-aware Partitioning ............................... 46
4.5.1 Overview of Partitioning ........................................ 47
4.5.2 Contention Likelihood ........................................ 47
4.5.3 Graph Representation ........................................ 48
4.5.4 Partitioning Algorithm ........................................ 48
4.5.5 Discussion .................................................... 49
4.6 Fault Tolerance ................................................... 51
4.6.1 Replication Protocol ......................................... 51
4.6.2 Failure Recovery ............................................. 52
4.7 Implementation ................................................... 53
4.8 Evaluation ........................................................ 54
4.8.1 Setup .......................................................... 54
4.8.2 Baselines ....................................................... 54
4.8.3 Workloads ..................................................... 55
4.8.4 TPC-C Results ................................................. 56
4.8.5 YCSB Results .................................................. 58
4.8.6 Instacart Results .............................................. 61
4.9 Related Work ..................................................... 62
4.10 Main Takeaways ................................................... 63
5 High Availability with RDMA Networks 65
5.1 Motivation ......................................................... 65
5.2 Contributions and Chapter Organization ......................... 66
5.3 High Availability in Existing OLTP Systems ...................... 67
  5.3.1 Active-Passive Replication ................................ 68
  5.3.2 Active-Active Replication ................................ 69
5.4 The Case for Replication with RDMA Networks .................. 70
  5.4.1 Bottleneck Analysis ........................................ 71
  5.4.2 Background for RDMA ..................................... 72
5.5 Active-Memory: RDMA-based Replication ....................... 72
  5.5.1 Concurrency Control and Replication Assumptions .......... 73
  5.5.2 Overview ................................................... 73
  5.5.3 Design Challenges ......................................... 74
  5.5.4 Undo Log Buffers .......................................... 75
  5.5.5 Replication Algorithm ..................................... 75
5.6 Fault Tolerance ................................................... 78
  5.6.1 Recovery from Backup Failures ............................ 79
  5.6.2 Recovery from Primary Failures ........................... 79
  5.6.3 Recovery of Multi-Partition Transactions .................. 81
5.7 Evaluation ........................................................ 81
  5.7.1 Experiment Settings ....................................... 81

vii
5.7.2 Single-Partition Transactions ................................................. 83
5.7.3 Multi-Partition Transactions ................................................. 85
5.7.4 Network Bandwidth ......................................................... 85
5.7.5 Replication Factor .......................................................... 87
5.7.6 Impact of Contention ....................................................... 87
5.7.7 Read-Write Ratio ............................................................ 88
5.8 Related Work ................................................................. 89
5.8.1 Replication in Conventional Networks ..................................... 89
5.8.2 RDMA-based OLTP .......................................................... 90
5.9 Main Takeaways ............................................................... 91

6 Conclusion and Future Work ................................................ 92
6.1 Summary of Contributions .................................................... 92
6.2 Suggested Directions for Future Research ............................... 93

Bibliography .............................................................................. 94
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The supported operations for each queue pair transport mode.</td>
<td>9</td>
</tr>
<tr>
<td>5.1</td>
<td>Different scenarios for the coordinator failure. Each row describes a scenario and how trans-</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>action state and records are recovered.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>The median latency per transaction.</td>
<td>84</td>
</tr>
<tr>
<td>5.3</td>
<td>The network traffic per transaction in each replication protocol in our unified platform.</td>
<td>86</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Processing steps of an RDMA Write</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Network Throughput and Latency</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>CPU Overhead for Network Operations</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>The NAM Architecture</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Naïve RDMA-based SI-Protocol</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Version Management and Record Layout</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Scalability of NAM-DB</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>Latency and Breakdown</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>Scalability of Oracle</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>Effect of Locality</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Effect of Contention</td>
<td>34</td>
</tr>
<tr>
<td>3.9</td>
<td>Scalability of QPs</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Traditional Execution and Partitioning.</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Chiller Execution and Partitioning.</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>The lifetime of a distributed transaction. The green dots denote when the server commits the transaction. The blue lines represent the contention span for each server.</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Two-region execution of a simplified ticket purchasing transaction. In the dependency graph, primary key and value dependencies are shown in solid and dashed lines, respectively (blue for conditional constraints, e.g., an “if” statement). Assuming the flight record is contended (red circles), the red box in (c) shows the operations in the inner region (Step 4). The rest of the operations will be performed in the outer region (Steps 3 and 5).</td>
<td>44</td>
</tr>
<tr>
<td>4.5</td>
<td>An example workload and how partitioning techniques with different objectives will partition it into two parts.</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Replication algorithm for the inner region.</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Comparison of different concurrency control methods and Chiller for the standard TPC-C workload.</td>
<td>57</td>
</tr>
<tr>
<td>4.8</td>
<td>Impact of distributed transactions in TPC-C.</td>
<td>57</td>
</tr>
<tr>
<td>4.9</td>
<td><strong>YCSB local</strong> (all single-partition transactions)</td>
<td>58</td>
</tr>
</tbody>
</table>
4.10 **YCSB distributed** with increasing cluster sizes. 59
4.11 Varying the database coverage of the lookup table for **YCSB local** workload. 60
4.12 Instacart with different execution models. 61

5.1 Active-passive replication using log shipping. 68
5.2 Active-active replication in H-store/VoltDB 69
5.3 Active-active replication in Calvin 70
5.4 The blue arrows indicate the throughput is increased by reducing which axis (i.e. reducing the network communication in conventional networks, and reducing processing redundancy in RDMA-enabled networks). 71
5.5 The structure of the log buffer and the log entries. Each node has a private log buffer on every other node. 73
5.6 The linked list of RDMA messages sent to an active node 73
5.7 Active-Memory replication: Step 1 is replicating the undo logs and in-place updates to the backups (Section 5.5.5). Step 2 is marking the log entries as committed (Section 5.5.5). 76
5.8 **Scalability** – The throughput of different replication protocols on different cluster sizes. All transactions are single-partition, with each reading and modifying 10 records. 82
5.9 CPU time breakdown for log shipping and Active-Memory with varying percentage of write transactions. Cluster size is 5 with 2-safety. Total system utilization in all cases is 100%. 83
5.10 **Multi-partition transactions** – Eight records are read and modified locally, and two records are read and modified remotely. Cluster size is 5 with 2-safety. 85
5.11 **Network Bandwidth** – The measured throughput of the replication schemes on 5 servers, with replication factor set to 2-safety, and 10 read-modify records per transaction. 86
5.12 **Replication Factor** – Impact of replication factor. 87
5.13 **Contention** – The throughput on 5 machines with varying skew factor. Records are chosen according to a Zipfian distribution from the entire cluster with different contention levels. 88
5.14 **Read-Write Ratio** – the throughput of different protocols on 5 machines with single-partition transaction, with varying number of write operations per transaction. 89
Chapter 1

Introduction

Data-intensive applications are increasingly growing in prevalence and impact in today’s world. Database systems are the backbone of these applications, from traditional banking and accounting softwares, to more modern applications such as social networks and a multitude of web-based services.

Database management systems (DBMSs) can be broadly divided into two major categories: An on-line transaction processing (OLTP) system, which is the focus of this dissertation, is specialized in handling the operational workload of a business, such as retrieving the information for a given customer, updating the stock number for a product, or deleting an item from the product catalog. The updates made by an OLTP system are then aggregated and merged into the backend data warehouse, enabling the business user to perform on-line analytical processing (OLAP) on the entire dataset.

OLTP systems emerged at least five decades ago with the goal of providing the database user with an interactive way to read, modify and store changes to data records. In fact, the emergence of ATM was among the first use cases of these systems, where it was introduced by banks as a new and faster means for their customers to withdraw or deposit money.

Since then, OLTP systems have been constantly evolving to accomodate the ever-increasing requirements of applications. On one hand, the average throughput that modern applications demand is orders of magnitude higher than the traditional applications. On the other hand, scalability is of crucial importance for these applications that are mostly run on private-owned or public cluster-computing platforms.

To address these requirements, more and more OLTP systems in recent years are becoming in-memory, where the entire of the operational data reside in the main memory of the database servers. Moreover, many of these systems have adopted the shared-nothing architecture [87] as their distributed-computing environment, where nodes do not share computing and storage resources with each other. This proved to be immensely instrumental in providing better scalability.

While shared-nothing databases, such as Oracle Timesten [51], Google Spanner [19], and VoltDB [91] have gained popularity in recent years, they still fall short when in comes to scalability in the general case. In particular, many of these systems are able to deliver true scale-out as long as the data they operate on can be split into disjoint partitions to a great extent, and therefore each server node can separately work on its own portion of the database without needing to coordinate with the remote nodes. In short, existing OLTP systems
are known to have very poor scalability in the presence of distributed transactions, which are transactions that span multiple partitions.

In the following of this chapter, we will take a closer look at the characteristics of modern OLTP workloads and their scalability challenges for the existing OLTP systems in Section 1.1. Then, in Section 1.2, we will turn our attention to the recent advances in the area of networking, and discuss their great potentials to help solve the challenge of scalability of OLTP systems. Finally, Section 1.3 provides the structure for the remainder of this thesis.

1.1 Modern Online Transaction Processing

At the core of any OLTP system lies the notion of transaction. A database transaction is simply a logical representation of a unit of work performed inside an OLTP system. This abstract notion helps developers simplify reasoning about concurrent programs. More specifically, a transaction consists of multiple statements, which transforms the database state, while it maintains the following four properties (collectively known as the ACID properties) [36]: (1) atomicity, (2) consistency, (3) isolation, and (4) durability. Atomicity guarantees that the updates to the database occur in completion, and not partially. Consistency ensures that a transaction will bring the database from one valid state to another. Isolation specifies that concurrent transactions are executed as if they are run sequentially. Finally, durability guarantees that even in the presence of any type of system failure, the final verdict for a transaction (which is either “Commit” or “Abort”) must always be recoverable.

Transactions are typically short-lived, and generally contain fairly simple operations (as opposed to more complex OLAP queries) [75]. Lookuping up a record by its ID or a secondary index, updating a few of its columns, inserting new records and deleting existing records are examples of the typical operations inside an OLTP system.

Historically, the use of these systems was mostly limited to a few traditional domains, such as banking or travel agencies, where for the most part, transactions were coming from the system operators. However, the popularity of web has played a crucial role in setting new demands and hence steering the evolution of OLTP systems. The performance requirement for many modern web-based applications far exceed that of those traditional domains. Examples of these modern applications include online multi-player games, gambling websites, dynamic Web content generation, real-time online advertising services and social networking websites [88].

The OLTP workloads for modern applications generally demand the following two requirements:

1. **Scalable processing**: The ubiquity of cloud computing has pronounced the importance of scalability for modern applications whose success highly depends on their ability to meet their incoming transactional demands by scaling their distributed database computing platform, regardless of the partitionability of their workload.

2. **High availability (i.e. “Always-on service”)**: Database systems experience failures for different reasons, such as hardware failures, network communication failures, software bugs, or human errors.
Highly available database systems ensure that even in the face of such failures, the system remains operational with close to zero downtime.

In the following, we discuss the challenges for existing existing OLTP systems in meeting all these requirements.

1.1.1 Challenges

The main bottleneck in disk-based databases was reading from and storing to disk, which is due to the mechanical nature of an HDD’s moving parts. Distributed in-memory OLTP systems eliminate the need to access disk by storing the entire database in the collective main memory of the cluster, delivering orders of magnitude higher throughput than disk-based databases. With disk removed from the critical path of transaction processing, the new bottleneck in in-memory OLTP systems is network [44]. When executing a transaction, accessing data hosted on remote partitions over a conventional network is much more expensive than accessing local data, mainly due to three reasons:

1. **Higher latency**: A network roundtrip within the same datacenter on an 1Gbps takes about 2 to 3 orders of magnitude compared to random main memory access. This is quite significant for the otherwise short-lived transactions.

2. **Lower bandwidth**: The maximum bandwidth on local memory is close to 100x compared to the network bandwidth. Even though that transactions usually have small footprints, but a high-performance OLTP system which aims to performs millions transactions per second may eventually saturate the network bandwidth.

3. **High CPU overhead**: The TCP/IP has non-negligible CPU overhead for OLTP workloads which generally consist of a few simple operations. It is not uncommon that the CPU spends most of the time processing network messages, leaving little room for the actual work. In addition to accessing remote records, the participants of a distributed transaction also have to engage in an atomic commit protocol (such as 2PC) which involves multiple network roundtrips before being able to commit [67].

Due to these reasons, the common wisdom is that in-memory OLTP systems deliver scalable performance as long as the partitioning of the data is done in a way that prevents distributed transactions.

Many real-world workloads, however, are not as partitionable as one may hope, and may constitute of transactions that span multiple partitions. For example, the social networks’ workloads are notoriously difficult to partition.

1.2 Emerging Hardware: Opportunities and Requirements

The area of networking have observed an increasingly fast advancement in recent years. This has resulted in a significant reduction in the latency and bandwidth gap between local main memory and network. Compared to only a decade ago with the then common 1Gbps Ethernet, the datacenter roun-trip latency on modern
networks has been lowered more than $200\times$ to only $2\mu s$. The bandwidth has also increased about 2 orders of magnitude to now 100Gbps, and its increase pace does not seem to be slowing down [40].

Despite such lower latency and higher bandwidth, merely deploying an existing distributed data stores on the fast networks are not going to remove the network bottleneck [33], largely due the high CPU overhead of the traditional TCP/IP protocols, especially for the common OLTP workloads which involves a few small random record accesses. Interestingly, in fact doing so in an OLTP system would result in performance degradation, in some cases [12].

The next-generation networks, such as Infiniband, with their RDMA (Remote Direct Memory Access) feature offer a promising solution to this problem. Using RDMA, a machine is able to reliably read from and write to remote memory, eliminating all the overhead associated to TCP/IP communications. The key to this low overhead is that the networking machinery in the RDMA model bypasses the kernel and is entirely performed in user-level, without ever needing to copy buffers from kernel space to user space and vice versa. Moreover, RDMA allows for direct one-sided operations, where the CPU of the remote side is also not involved. The RDMA feature, together with ultra low latency and high bandwidth of these networks, have great potential to help making scalable distributed transactions.

However, leveraging RDMA, as we will show in the next chapter, requires a careful rethinking of the architecture of OLTP systems and their different components. First, the distributed setting is no longer a pure shared-nothing architecture, as main memory is not only controlled by its local machine. However, it is not a shared-memory architecture either, since memory semantics in RDMA are not the same as the ones in a shared-memory systems. Second, fully exploiting RDMA in transaction processing calls for designing vastly different data structures which are suitable for this new memory access method. Finally, various important database algorithms, such as concurrency control, data partitioning techniques and replication protocols, were devised in a time when the network used to be the dominant bottleneck. These algorithms need to be re-evaluated, too.

1.3 Summary of Thesis Contributions and Outline

In the previous sections, we explained the reasons which hinder the scalability of distributed OLTP systems, and how the emerging networking technologies have opened new possibilities to eliminate this problem. However, as we will show later in this thesis through both theoretical calculations and experimental evaluations, merely deploying an existing OLTP system on top of the modern RDMA-enabled network does not yield better scalability. Therefore, the central thesis of this work is the following:

*By taking advantage of modern RDMA-enabled networks, it is possible to build a new high-performance distributed OLTP system which offers much better scalability compared to existing systems. This requires fundamentally re-designing many components of an OLTP system such as data structures, transaction processing techniques, data partitioning schemes and replication algorithms.*
In particular, we make the following contributions in this dissertation:

1. **A scalable distributed in-memory OLTP system**: As shared-nothing architecture is not able to fully capture the memory access model of an RDMA-based distributed environment, we propose network-attached memory (NAM) architecture, which is a new abstraction model for distributed data stores and treat RDMA operations as first-class citizens. Based on this abstraction, we present a detailed explanation of NAM-DB, a new distributed OLTP system that leverages RDMA efficiently to deliver scalable performance.

2. **Automatic Contention-aware Execution and Data Partitioning**: We propose a new distributed transaction execution and partitioning protocol which aims to minimize contention on popular database records, and hence maximize the throughput. We show that the objective of minimizing contention requires a new way of partitioning database records, which is different than existing partitioning techniques whose goal is to minimize the number of distributed transactions. To that end, we present a detailed explanation of our novel data partitioning.

3. **Low-overhead RDMA-based replication**: We present a novel strongly consistent replication algorithm which fully exploits one-sided RDMA operation to completely eliminate the processing redundancy in existing replication schemes. Our proposed primary-backup replication algorithm allows an RDMA-based OLTP system to maintain its high performance and correctness in the presence of failures.

This dissertation is organized as following:

In Chapter 2, we cover the background of RDMA, the various design choices it provides, along with a discussion of the different types of RDMA-enabled networks. We also report the results of several micro-benchmarks.

Then in Chapter 3, we propose a new architecture called network-attached memory (NAM), which captures the RDMA memory semantics. We then present the design of our novel scalable OLTP system called NAM-DB in Chapter. The main design goal of NAM-DB is to remove the centralized components of a transaction processing system, such as timestamp oracle or lock service, and instead rely on data structures and protocols that can be efficiently read and modified using RDMA operations.

Having removed the communication over network as the dominant bottleneck in the scalability of OLTP systems, in Chapter 4 we show that the real bottleneck which hinders the scalability of OLTP systems is the data contention inherent in some workloads. We then present our proposed solution Chiller, which is a new approach to transaction execution and database partitioning that aims to minimize data contention for both local and distributed transactions, and hence outperforms existing traditional data partitioning techniques.

In Chapter 5, we first explain why existing high availability solutions for OLTP systems are not optimal for RDMA-enabled networks. Then, we propose a primary-backup replication technique called Active-Memory, which leverages one-sided RDMA operations to eliminate the processing redundancy of existing replication techniques and produce higher scalability and performance.

Finally, we conclude this dissertation in Chapter 6 and discuss some ideas for possible future research.
Chapter 2

Background on RDMA

In this chapter, we provide some background information on RDMA, its characteristics and terminology, its various types and the trade-offs between them. We also discuss different network protocols that support the RDMA feature. Additionally, to better characterize the design space for RDMA-based communication in OLTP systems, we present a set of micro-benchmarks that showcase the characteristics of this network technology and discuss how this will impact the design of distributed database systems.

The experimental evaluation presented at the end of this chapter comes from work published in our VLDB 2016 paper entitled “The End of Slow Networks: It’s Time for a Redesign” [12].

2.1 RDMA and its Advantages

Remote Direct Memory Access, or RDMA, is a mechanism that allows one machine to access the main memory of a remote machine without ever interrupting the remote CPU. As its name suggests, RDMA is basically the remote counterpart of DMA (Direct Memory Access), which is the ability of a device to directly read from or write to its host memory without the intervention of its CPU.

RDMA has important advantages over the conventional networking:

- **Zero-copy transfer:** The network adapter writes or reads the data to or from the remote memory without any excessive copying between the application memory and the data buffers in the operating system software stack. Without RDMA, a typical flow of data transfer entails making multiple copies of the application buffer in different network stack layers.

- **Kernel bypass:** The entire machinery of the data transfer takes place at the user-space. Therefore, no context switch takes place when sending data over wire, eliminating its overhead.

- **CPU offload:** An application can access or modify remote memory without involving the remote CPU in the process, freeing it up to do useful work. Also, its cache content will not be altered by the accessed data. In addition to remote CPU, the RDMA model also offloads much of the network processing from the CPU to the network interface cards and the switches.
• **Ultra low latency:** The end-to-end latency of RDMA-enabled networks is only a few micro seconds, which is two orders of magnitude lower than the traditional networks.

• **High bandwidth:** Existing RDMA-enabled networks provide a bandwidth which rival the bandwidth of one memory channel. Newer network models are able to far exceed that, providing more bandwidth over the wire than what the local memory does.

### 2.2 RDMA-enabled Networks

There are several RDMA-enabled network protocols in the market, among which three has found broader acceptance:

**InfiniBand (IB):** It is a self-contained new generation network protocol that was designed by the InfiniBand Trade Association in 1999 with the support of RDMA in mind from ground-up. The InfiniBand protocol specifies how to perform RDMA in an InfiniBand network, which is an alternative technology to Ethernet (for networking) and Fibre Channel (for storage) [73]. Therefore, it requires its own RDMA-aware NICs (known as RNIC) and switches. The IB protocol is developed by Open Fabrics Alliance ¹, consisting of Intel, HP, IBM, Cray and many other big companies.

In addition to offering RDMA, InfiniBand also offers another communication stack, called IP over InfiniBand (IPoIB), which implements a classical TCP/IP stack over the high-bandwidth InfiniBand network and makes it easy to run existing socket-based applications. Similar to Ethernet-based networks, data is copied by the application into operating system buffers (no zero-copy transfer), and the kernel processes the buffers by transmitting packets over the network (no kernel bypass). This allows existing socket-based applications to run on InfiniBand without modifications. While providing an easy migration path from Ethernet to InfiniBand, our experiments in Section 2.5 will reveal that IPoIB is unable to fully leverage the network fabric.

**RDMA over Converged Ethernet (RoCE):** It uses the Ethernet frames, while using the IB transport. More specifically, in RoCE, the transport layer of IB is placed on top of the physical and data-link layers of Ethernet. Therefore, RoCE is able to transfer RDMA messages over standard Ethernet-based network switches, and only requires that the NIC supports RoCE.

**Internet Wide Area RDMA Protocol (iWARP)**: This protocol specifies RDMA over congested-oriented networks such as the standard TCP/IP by adding three additional layers on top of them. Therefore, it can be employed in internet scale.

All these three protocols share the same API, and therefore codes written using this API can be executed in each of these networks with minimal changes, provided that the used functionalities exist in the target network.

For the rest of this dissertation, we use InfiniBand and its terminology to discuss the details of RDMA,

¹ [https://www.openfabrics.org/](https://www.openfabrics.org/)

² According to [17], iWARP is not an acronym. It seems that its longer name has been chosen to emphasize the potential of iWARP to be used in internet scales.
mainly because compared to the other technologies, InfiniBand provides the highest performance and therefore has been prevalent in high-performance datacenters. For this reason, we also built, tested, and benchmarked all our softwares present in the subsequent chapters on top of an InfiniBand infrastructure. However, most of these topics are still very relevant to most of the other protocols.

2.3 RDMA Details

RDMA uses a vastly different API compared to the traditional socket API. In this section, we will briefly discuss its main ideas and see how RDMA can be used in real distributed applications.

2.3.1 The Verb API and the Supported Operations

The Verb API describes the different operations (equivalent to functions) that can be used in the RDMA programming model. The operations in the Verb API (e.g. the libibverbs library) all take place in the user-space. This is key in delivering low latency and zero kernel overhead.

The most important verbs for the purpose of this writing can be categorized in two groups:

1. **One-sided verbs:** These are the verbs where the remote CPU is entirely passive in the process of data transfer. RDMA Write, Read and Atomics (namely, compare-and-swap and fetch-and-add) are the important operations in this group.

2. **Two-sided verbs:** Unlike the former group, the operations in this category require the remote side to actively be involved. RDMA Send and Receive are among the operations in this group.

2.3.2 Queue Pairs and their Different Types

All the listed operations in the previous part require a connection already established between the two machines, which is called a queue pair, in the InfiniBand terminology. The reason for its name is that it consists of a send and a receive queue, where operations are served in the first-in, first-out manner. Queue pairs are created by the application on the client and the server, after which the RNICs handle the state of the queue pairs.

When establishing a queue pair, its transport mode can be chosen using the following two independent attributes (and therefore there are in total four transport modes for queue pairs):

- **Reliable/Unreliable:** If the connection guarantees that all messages will be delivered in order and not corrupt, then the QP is called a reliable. Otherwise, it is referred to as unreliable.

- **Connected/Unconnected:** If this QP is only associated to exactly one remote QP (i.e. there is a 1-1 mapping), it is called connected. A 1-N QP is referred to as unconnected (also called datagram).

The choice of the transport mode determines the set of RDMA operations supported on that connection, as shown in Table 2.1. Out of the four possible combinations, only Reliable Datagram (RD) is not supported by today most NICs. The other three modes, i.e. RC, UC and UD, are supported.
Table 2.1: The supported operations for each queue pair transport mode.

<table>
<thead>
<tr>
<th>Operation</th>
<th>UD</th>
<th>UC</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send/Receive</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RDMA Write</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RDMA Read</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RDMA Atomics</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.1: Processing steps of an RDMA Write

2.3.3 RDMA Workflow

Once a queue pair is established between two machines, the general communication pattern is that the requester creates a request called a Work Queue Element (WQE) which specifies the verb (e.g. RDMA Read), along with other parameters such as the remote memory location on the target machine. The requester puts the WQE into its send queue and informs its local RNIC to perform the operation.

For one-sided verbs, the WQEs are handled by the remote NIC without interrupting its CPU using a DMA operation on the remote side. In case of two-sided verbs, however, the remote side must use a Receive verb to put a request into its receive queue to be able to handle an incoming Send request.

Figure 2.1 shows the processing steps of an RDMA Write:

1. The client sends the WQE to the local RNIC using PIO,
2. The local RNIC reads the data directly via DMA read from a registered local memory region.
3. The local RNIC sends the data to the remote RNIC.
4. The remote RNIC then writes the data to a given location in a registered remote memory region using a DMA write on the server side.
5. In a reliable connection, the remote RNIC sends an RDMA ack to the client.
6. For a signaled Write, the local RNIC pushes the completion event into the CQ once the RDMA ack has arrived from the remote RNIC.
2.4 RDMA-enabled Network vs. Local Memory

Even today, with the new generation of InfiniBand (e.g., FDR or EDR), the bandwidth available to transfer data across network is in the same ballpark as the bandwidth of one memory channel. For instance, DDR3 memory bandwidth currently ranges from 6.25 GB/s (DDR3-800) to 16.6 GB/s (DDR3-2133) per channel [3], whereas InfiniBand has a specified bandwidth of 1.7 GB/s (FDR 1x) to 37.5 GB/s (EDR 12x) [2] per NIC port, as illustrated in Figure 2.2. At the same time, particularly for smaller deployments, InfiniBand is becoming increasingly affordable.

However, modern systems typically support 4 memory channels per socket, leading to a higher aggregated bandwidth of 51.2 GB/s for a machine with 12.8GB/s per socket. Thus, today 4 dual-port FDR NICs with 4x (link aggregation) provide roughly the same bandwidth. It is worth noting that the CPU-memory bandwidth is half-duplex, while PCIe-based NICs are full-duplex. That is, if the same amount of data is written as read, only two NICs would be sufficient to match the CPU-memory bandwidth. In addition, new InfiniBand EDR and HDR NICs achieve 25GB/s and 50GB/s on using dual-port cards, respectively; enough to achieve the same bandwidth even with a single NIC.

Another important factor is that with major advances in RDMA, the network latency also improves quickly. Our recent experiments with InfiniBand FDR 4x showed that the system requires ≈ 1 micro seconds to transfer 1KB of data using RDMA, compared to ≈ 0.08 micro seconds required by the CPU to read the same amount of data from memory. With only 256KB, there is virtually no difference between the access time since the bandwidth starts to dominate the transfer time. However, due to laws of physics, one can expect that such gap will never be closed. In fact, we believe that cache- and memory-locality will remain important even for distributed in-memory databases since the network is becoming less of a bottleneck and CPU-efficiency becomes more important.

Furthermore, memory prices continue to drop, making it feasible to keep even large data sets entirely in memory with just a few machines, removing the disk-bottleneck. With fast networks, the whole system becomes more balanced and much more attention has to be given to exploit locality and efficient CPU usage and caching.
2.5 Micro-Benchmarks

In this section, we show the results of our micro-benchmarks for comparing different RDMA verbs to their socket-based counterparts. The main goal here is to dissect the basic properties, such as throughput, latency and CPU utilization of the two InfiniBand-based communication stacks (IPoIB and RDMA) to derive the guidelines underlying the design decisions in the subsequent chapters. Moreover, we also compare IPoIB and RDMA to a classical TCP/IP stack over 1Gbps Ethernet (IPoEth). It is clear that a 1Gbps Ethernet network will have much lower throughput than our InfiniBand network. However, an interesting question is whether IPoIB behavior is closer to a classical Ethernet-based TCP/IP stack or if it can efficiently leverage the high bandwidth/low-latency of the RDMA-based stack with low CPU overhead.

2.5.1 Experimental Setup

In our micro-benchmarks, we used two machines with an Intel Xeon E5-2660 v2 processor and 256GB RAM. Both machines are equipped with a Mellanox Connect IB FDR 4x dualport RNIC connected to the same InfiniBand switch. Each port of the RNIC has a bandwidth of 54.54Gbps (6.8 GB/s) and supports a full-duplex mode. Additionally, each machine has a 1Gbps Ethernet NIC (with only one port) also connected to the same Ethernet switch. As a software stack, each machine runs Ubuntu Server 14.04 and uses the OFED 2.3.1 driver for the RNIC.

In all our experiments, we use one port on the RNIC to better compare the InfiniBand results to the results of the classical Ethernet stack (especially the overhead per message call). Moreover, all experiments have been executed in a single-threaded mode. Multi-threaded experiments are shown in the subsequent sections since here we want to compare only the basic network properties here.

2.5.2 Throughput and Latency

As shown in Figure 2.3, this experiment shows the results for the throughput and latency of the two InfiniBand communication stacks (RDMA and IPoIB) for different message sizes ranging from 32B up to 64MB.
to simulate the characteristics of different workloads. As mentioned in the previous chapter, database transactions for modern applications have usually small footprints. Therefore, the reported results for up to 1KB message sizes are of special interest for our purpose. For RDMA, we show the results for signaled Reads, Writes, Sends and Receives. We also measured the RDMA atomic operations, but since they only support a maximal message size of 8B and show the same latency and throughput as 8B Reads, we omitted the results from this figure.

While all RDMA verbs saturate our InfiniBand network bandwidth of approximately 6.8GB/s for message sizes greater than 2KB, IPoIB only achieves a maximal throughput of 3.5GB/s even though it uses the same InfiniBand hardware as RDMA. Moreover, the message delay (i.e., 1/2 network round-trip) over IPoIB is also higher than for RDMA. In fact, for small message sizes, the latency of IPoIB is much closer to the latency of the 1Gbps Ethernet network (IPoEth). For example, for a message size of 8B the latency is 20µs for IPoIB and 30µs for IPoEth while an RDMA Write operation only takes 1µs. A possible explanation (though we could not verify it) is that the TCP/IP stack for IPoIB has a very high CPU overhead per message for small messages, as we will show in the next experiment. For larger message sizes (≥ 1MB), the latency of IPoIB is again closer to RDMA; however, it is still a factor of 2.5× higher than for RDMA. For example, a 1MB message has a latency of 393µs on IPoIB while it has only 161µs for RDMA verbs. For RDMA atomic operations, the latency is the same as for an RDMA Read of 8B.

An interesting result is that RDMA Write and Send operations take only 1µs for message sizes less than 256B while an RDMA Read needs 2µs. The reason is that for RDMA Writes and Sends, a payload of less than 256B can be inlined into the initial PIO (see step 1 in Figure 2.1) and thus the DMA read (step 2) can be skipped [61].

2.5.3 CPU Overhead

In this experiment, we compare the per-message overhead (in CPU cycles) incurred by the communication operations of the two communication stacks on both the client and the server side. Similar to the previous experiment, we vary the message sizes.

Figure 2.4 shows that RDMA has a constant overhead on the client and the server side that is independent of the message size. The reason is that the costs of registering a WQE on the RNIC is independent of the message size. The actual data transfer is executed by the RNIC which acts as a co-processor to handle the given WQE. On the client side the overhead is around 450 cycles independent of the RDMA verb used. The CPU overhead for atomic operations is actually the same. Moreover, as expected, on the server side only the Receive verb causes a CPU overhead. All the other verbs that are one-sided (RDMA Read/Write and the atomic operations) do not cause any overhead on the server side.

On the other hand, the overhead of IPoIB is very different from that of RDMA. In fact, it is much more similar to the overhead of the classical Ethernet-based TCP/IP stack (IBoEth). The major difference to RDMA is that for IPoEth and IPoIB the overhead per message actually grows linearly with the message size once exceeding the TCP window size (which was the default value of 1488B for IPoEth and 21888B for IPoIB in our experiment). Even more interestingly, the overhead per message for small messages for IPoIB is even higher than for IPoEth. For example, an 8B message needs $7.5 \cdot 10^3$ cycles for IPoEth and $13 \cdot 10^3$ cycles
for IPoIB.

### 2.5.4 Conclusion

Our micro-benchmarks showed even though that relying merely on IPoIB for transferring messages in distributed systems would provide an easy migration without changing its software code, but it would not suffice to leverage the full potential of a modern RDMA-enabled network. Even worse, the high CPU overhead of IPoIB for small message sizes may in some cases (which are most relevant to OLTP workloads) cause a negative impact when running an existing distributed database system using IPoIB.

In the following chapter, we will use the features of modern networks to build a scalable distributed OLTP system.
Chapter 3

Scalable Distributed Transaction Processing

Existing distributed OLTP systems are built on the assumption that the network is a dominant bottleneck [4] and therefore distributed transactions must be avoided as much as possible. However, a large group of workloads, such as social applications, are not as partitionable as one may hope and, no matter what sharding scheme is used, would end up having many transactions that span multiple partitions.

With the emerging modern network technologies and their new features, most importantly RDMA, the assumption that the network is the dominant bottleneck is losing its relevance in distributed OLTP systems. Naturally, therefore, many design decisions of these systems have to be critically re-evaluated in the light of the new advancements.

In this chapter, we first make the case for a re-design of the well-established shared-nothing architecture for OLTP systems. We then propose a novel distributed architecture called Network-Attached Memory (NAM), which unlike a shared-nothing architecture where the only way of accessing remote data is through messaging, RDMA is treated as a first-class citizen access type.

Then, based on this architecture, we propose a new OLTP system called NAM-DB, a truly scalable distributed OLTP database which leverages RDMA efficiently. All the data structures and algorithms in NAM-DB are designed specifically to leverage the features of modern networks, including their high bandwidth.

Our published work in VLDB 2017 titled “The End of a Myth: Distributed Transactions can Scale” [108] is the source for most of the content in this chapter.

3.1 Motivation

The common wisdom is that distributed transactions do not scale [94, 44, 89, 21, 81]. As a result, many techniques have been proposed to avoid such transactions ranging from locality-aware partitioning [80, 21, 109, 83] and speculative execution [76] to new consistency levels [49] and enforcing determinism [44, 95]. Some
of these techniques are not completely transparent to the application developer and impose strong application-level limitations. Transactions spanning multiple partitions usually observe very inferior performance (e.g. higher abort rate and much lower throughput) compared to single-partition transactions [13].

An OLTP system without such limitation on distributed transactions would have great benefits. First, the application developer would no longer need to worry about co-partitioning schemes in order to achieve decent performance. Instead of being a necessity to achieve a scalable system, good partitioning layout becomes a second-class design consideration in order to improve the performance of a few selected queries, similar to how an index is used in this manner. Second, the system would scale out linearly when adding more machines as the load increases, rather than sub-linearly because of the impact of distributed transactions. This makes it much easier to provision how much hardware is needed and hence increases the predictability of system maintenance.

3.1.1 On Un-scalability of Distributed Transactions

We first discuss the reasons why existing OLTP systems consider distributed transactions as hurdles in achieving scalability.

First, in many workloads, transactions are short-lived, where only a small number of attributes for a few records are accessed and modified. In such a setting, the CPU overhead of the conventional networking stacks such as TCP/IP prevents the processing power of the cluster to be dedicated to processing transactions.

Second, even though that a single transaction typically has small footprints, the available bandwidth in traditional Ethernet-based networks limits the aggregate number of distributed transactions that can be run simultaneously.

Third, the high latency of traditional networks, which is up to three times as high as the latency of local memory access, may significantly intensify any contention in the workload or on shared resources (lock tables, latches, counters, etc.). Such an intensified contention has negative impact on the performance of the system, and may either increase the abort rate or result in longer waiting times.

For these reasons, existing OLTP systems are designed with the aim of minimizing distributed transactions. When such layout does not exist, or varies too more frequently than can be captured by automatic partitioning algorithms, the performance of these system degrades significantly.

3.1.2 The Need for a System Redesign

The emerging RDMA-enabled networks eliminate/significantly mitigate the dominant limiting factors mentioned in Section 3.1.1 for scalable distributed transaction.

Yet, it is wrong to assume that the hardware alone solves the problem. In order to avoid the CPU message overhead with RDMA, many database-specific algorithms and data structures (such as lock management, indexing, concurrency control, the replication algorithms, etc) have to be re-designed accordingly to become RDMA-friendly.

More specifically, RDMA-enabled networks change the architecture to a hybrid shared-memory and message-passing system: it is neither a distributed shared-memory architecture (due to different memory
3.2 Contributions and Chapter Organization

In this chapter, we make the following main contributions:

1. We present the full design of a truly scalable system called NAM-DB and propose scalable algorithms specifically for Snapshot Isolation (SI) with (mainly one-sided) RDMA operations.

2. We present a novel RDMA-based and scalable global counter technique which allows for efficiently reading the latest consistent snapshot in a distributed SI-based protocol.

3. We show that NAM-DB is truly scalable using a full implementation of TPC-C. Most notably, for the standard configuration of TPC-C benchmark, we show that our system scales linearly to over 3.6 million transactions per second on 56 machines, and 6.5 million transactions with locality optimizations.

The remainder of this chapter is organized as follows: In Section 3.3 we give an overview of our novel RDMA-based architecture for distributed transaction processing and describe the naive RDMA-based SI protocol as implemented in our vision article [12] to show the downsides of heavily relying on RDMA atomics for the timestamp counter which results in un-scalable timestamp generation. From the lessons learned from this naive design, in Section 3.5 we explain the design of our new timestamp oracle to generate read and commit timestamps in a scalable manner. In Section 3.6, we propose a RDMA-optimized data layout, specifically designed for efficient storage of multiple versions per record. Then, in Section 3.7, we will present our transaction execution scheme. Afterwards, in Section 3.8, we present the results of our experimental evaluation using the TPC-C benchmark. Finally, we conclude this chapter with related work in Section 3.9 and the main takeaway message in Section 3.10.

3.3 Network-Attached-Memory Architecture

In the following, we first give a brief overview of the network-attached-memory (NAM) architecture [12] that was designed to efficiently make use of RDMA. We then discuss the core design principles of NAM-DB, which builds upon NAM and enables scalable transaction processing without an inherent bottleneck other than the workload itself.

3.3.1 Main Idea

The NAM architecture is based on two main ideas:

- First, it logically decouples compute and memory nodes.
- Second, it uses RDMA (one-sided and two-sided) for communication between all nodes as shown in Figure 3.1.
The idea is that memory servers provide a shared distributed memory pool that holds all the data, which can be accessed via RDMA from compute servers that execute transactions. This design already highlights that locality is a tuning parameter. In contrast to traditional architectures which physically co-locate the transaction execution with the storage location from the beginning as much as possible, the NAM architecture separates them. As a result, all transactions are by default multi-partition. However, this design still allows users to add locality as an optimization like an index, as we will explain in Section 3.7.

In the following, we give an overview of the tasks of memory and compute servers in a NAM architecture.

### 3.3.2 Memory Servers

In a NAM architecture, memory servers hold all data of a database system such as tables, indexes as well as all the necessary state for transaction execution, including logs and metadata.

From the transaction execution perspective, memory servers are “passive” since they provide only memory capacity to compute servers. However, memory servers still have important tasks such as memory management to handle remote memory allocation calls from compute servers, as well as garbage collection to ensure that enough free space is always available for compute servers, e.g. to insert new records.

Durability of the data stored by memory servers can be achieved in a similar way as described in [26] by using an uninterruptible power supply (UPS). When a power failure occurs, memory servers use the UPS to persist a consistent snapshot to disks. On the other hand, hardware failures are handled through replication as discussed in Section 3.7.2.

### 3.3.3 Compute Servers

The main task of compute servers is to execute transactions over the data items stored in the memory servers. This includes finding the storage location of records on memory servers, performing the operations inside transactions (inserting, modifying and deleting records), as well as committing or aborting transactions.
Moreover, compute servers are in charge of performing other tasks, which are required to ensure that transaction execution fulfills all ACID properties such as logging and consistency control.

Again, the strict separation of transaction execution in compute servers from managing the transaction state stored in memory servers is what distinguishes our design from traditional distributed database systems. As a result, the performance of the system is independent of the location of the data.

### 3.3.4 NAM-DB Design Principles

Here, we describe challenges in designing NAM-DB to achieve a scalable system design using the NAM architecture.

**Separation of Compute and Memory:** Segregation of compute and storage has been proposed before [14, 54, 60, 26]. However, existing database systems that follow this design typically push data access operations into the storage layer. When scaling out the compute servers and pushing data access operations from multiple compute servers into the same memory server, memory servers are likely to become a bottleneck. Even worse, with traditional socket-based network operations, every message transfer consumes additional CPU cycles.

In the NAM architecture, we therefore follow a different route. Instead of pushing data access operations into the storage layer, the memory servers provide a fine-grained byte-level data access. In order to avoid any unnecessary CPU cycles for message handling, compute servers exploit one-sided RDMA operations as much as possible. This makes the NAM architecture highly scalable since all the computation required to execute transactions can be farmed out to compute servers.

Finally, for the cases where the aggregate main memory bandwidth is the main bottleneck, this architecture also allows us to increase the bandwidth by scaling out the memory servers.

**Data Location Independence:** Another design principle is that compute servers in a NAM architecture are able to access any data item independent of its storage location. As a result, the NAM architecture can easily move data items to new storage locations. Moreover, since every compute server can access any data item, it enables the implementation of work-stealing techniques for better distributed load balancing.

This does not mean that compute servers can not exploit data locality. In fact, the notion of compute and memory refer to the computer processes, and therefore they can still be co-located on the same physical machine. However, it lessens the importance of data locality from a necessity and a first-class design decision, to just an optimization which can be added on top of our scalable system, much similar to an index.

**Partitioned Data Structures:** As discussed before, in the NAM architecture every compute server should be able to execute any functionality by accessing the externalized state on the memory servers. However, this does not prevent a single memory region (e.g., a global read or timestamp counter) from becoming a bottleneck. Therefore, it is important that data structures are design in a way that can be partitioned. For instance, following this design principle, we invented a new decentralized data structure to implement a partitionable read and commit timestamp as shown in Section 3.5.
3.4 The First Attempt: A Counter-based SI

Our first attempt in building a transaction processing protocol based on the NAM architecture revealed important lessons which we later used to build our scalable system. Since the focus of this chapter is to present the design of the final NAM-DB, we mainly gloss over the details here and instead focus on the aspects relevant to our final proposal (the details can be found in our vision paper [12]).

3.4.1 Implementation Details

Simply put, with Snapshot Isolation (SI), a transaction reads the most recent snapshot of a database that was committed before the beginning of the transaction. Furthermore, transactions only abort if it would overwrite a concurrent change. For distributed systems, Generalized SI (GSI) [29] is more common as it allows any committed snapshot (and not only the most recent one) to be read. While we also use GSI for NAM-DB, our goal is still to read a recent committed snapshot to avoid high abort rates.

Listing 3.1 and the corresponding Figure 3.2 show a na"ıve SI protocol using only one-sided RDMA operations that are based on a global timestamp oracle as implemented in commercial systems [13].

To better focus on the interaction between compute and memory servers, we made the following simplifying assumptions: First, we did not consider the durability guarantee and the recovery of transactions. Second, we assume that there is an up-to-date catalog service, which helps compute servers to find the remote address of a data item in the pool of memory servers; the remote address is returned by the $\&_r$ operator in our pseudo-code. Finally, we consider a simple variant of SI where only one version is stored for each record (i.e. no multi-versioning). Note that these assumptions are only made in this section and we tackle each one later in this chapter.
For executing a transaction, the compute server first fetches the read-timestamp $rts$ using an RDMA read (step 1 in Figure 3.2, line 3 in Listing 3.1). The $rts$ defines a valid snapshot for the transaction. Afterwards, the compute server executes the transaction, which means that the required records are read remotely from the memory servers using RDMA read operations (e.g., the record with $ckey = 3$ in the example) and updates are applied locally to these records; i.e., the transaction builds its read- and write-set (step 2 in Figure 3.2, line 5 in Listing 3.1). Once the transaction has built its read- and write-set, the compute server starts the commit phase.

For committing, a compute server fetches a unique commit timestamp ($cts$) from the memory server (step 3 in Figure 3.2, line 7 in Listing 3.1). In the na"ive protocol, fetching a unique $cts$ counter is implemented using an atomic RDMA fetch-and-add operation that returns the current counter and increments it in the memory server by 1. Afterwards, the compute server verifies and locks all records in its write-set on the memory servers using one RDMA compare-and-swap operation (line 10-15 in Listing 3.1). The main idea is that each record stores a header that contains a version number and a lock bit in an 8-Byte memory region. For example, in Figure 3.2, $(3, 0)$ stands for version 3 and lock-bit 0 (0 means not locked). The idea of the compare-and-swap operation is that the compute server compares the version in its read-set to the version installed on the memory-server for equality and checks that the lock-bit is set to 0. If the compare succeeds, the atomic operation swaps the lock bit to 1 (step 4 in Figure 3.2, line 13 in Listing 3.1).

If compare-and-swap succeeds for all records in the write-set, the compute server installs its write-set using RDMA writes (line 19-20 in Listing 3.1). These RDMA writes update the entire record including updating the header, installing the new version and setting the lock-bit back to 0. For example, $(6, 0)$ is remotely written on the header in our example (step 5 in Figure 3.2). If the transactions fails, the locks are simply reset again using RDMA writes (line 24-28 in Listing 3.1).

Finally, the compute server appends the outcome of the transaction (commit or abort) as well as the commit timestamp $cts$ to a list ($ctsList$) in the memory server (step 6 in Figure 3.2, line 32 in Listing 3.1). Appending this information can be implemented in different ways. However, for our na"ive implementation we simply use an unsignaled RDMA send operation; i.e., the compute server does not need to wait for the $cts$ to be actually sent to the server, and gives every timestamp a fixed position (i.e., timestamp value - offset) to set a single bit indicating the success of a transaction. This is possible, as the fetch and add operation creates continuous timestamp values.

Finally, the timestamp oracle is responsible for advancing the read timestamp by scanning the queue of completed transactions. It therefore scans $ctsList$ and tries to find the highest commit timestamp (i.e., highest bit) so that every transactions before that timestamp are also committed (i.e., all bits are set). Since advancing the read timestamp is not in the critical path, the oracle uses a single thread that continuously scans the memory region to find the highest commit timestamp and also adjusts the offset if the servers run out of space.

### 3.4.2 Lessons Learned

While the previously-described protocol achieves some of our goals (e.g., it heavily uses one-sided RDMA to access the memory servers), it is still not scalable. The main reason is that global timestamps have inherit
```java
runTransaction(Transaction t) {
    // get read timestamp
    rts = RDMA_Read(&r(rts));

    // build write-set
    t.execute(rts);

    // get commit timestamp
    cts = RDMA_FetchAndAdd(&r(cts), 1);

    // verify write version and lock write-set
    commit = true;
    parfor i in size(t.writeSet) {
        header = t.readSet[i].header;
        success[i] = RDMA_CompAndSwap(&r(header), header, setLockBit(header))
        ;
        commit = commit && success[i];
    }

    // install write-set
    if(commit) {
        parfor i in size(t.writeSet) {
            RDMA_Write(&r(t.readSet[i]), t.writeSet[i]);
        }
    } // reset lock bits
    else {
        parfor i in size(t.writeSet) {
            if(success[i])
                header = t.readSet[i].header;
                RDMA_Write(&r(header), header);
        }
    }

    RDMA_Send([cts,commit]); //append cts and result to ctsList
} //Listing 3.1: Transaction Execution in a Compute Server
```
scalability issues [30], which are emphasized further in a distributed setting.

First, for every transaction, each compute server uses an RDMA atomic fetch-and-add operation to the same memory region to get a unique timestamp. Obviously, atomic RDMA operations to the same memory location scale poorly with the number of concurrent operations since the network card uses internal latches for the accessed memory locations. In addition, the oracle is involved in message passing and executes a timestamp management thread that needs to handle the result of each transaction and advance the read timestamp. Although this overhead is negligible for a few thousands transactions, it shows its negative impact when millions of transactions are running per second.

The second problem with the naïve protocol is that it likely results in high abort rates. A transaction’s snapshot can be “stale” if the timestamp management thread can not keep up to advance the read timestamp. Thus, there could be many committed snapshots which are not included in the snapshot read by a transaction. The problem is even worse for hot spots, i.e. records which are accessed and modified frequently.

A third problem is that slow workers also contribute to high abort rate by holding back the most recent snapshot from getting updated. In fact, the oracle only moves forward the read timestamp \( rts \) as fast as the slowest worker. Note that long transactions have the same effect as slow workers.

Finally, the last problem is that the naïve implementation does not have any support for fault-tolerance. In general, fault-tolerance in a NAM architecture is quite different (and arguably less straight-forward) than in the traditional architecture. The reason is that the transactions’ read- and write-sets (including the requested locks) are managed directly by the compute servers. Therefore, a failure of compute servers could potentially result in undetermined transactions and thus abandoned locks. Even worse, in our naïve implementation, a compute server that fails can lead to “holes” in the \( ctsList \) that cause the read timestamp to not advance anymore.

In the remaining sections, we will address these open issues and our simplifying assumptions listed at the beginning.

### 3.5 A Scalable Timestamp Oracle

In this section, we first describe how to tackle the issues outlined in Section 3.4.2 that hinder the scalability of the timestamp oracle as described in our naïve SI-protocol implementation. Afterwards, we discuss some optimizations.

#### 3.5.1 Timestamp Generation

The main issues with the current implementation of the timestamp oracle are: (1) The timestamp management thread that runs on a memory server does not scale well with the number of transactions in the system. (2) Long running transactions/slow compute servers prevent the oracle from advancing the read timestamp, further contributing to the problem of too many aborts. (3) High synchronization costs of RDMA atomic operations when accessing the commit timestamp \( cts \) are stored in one common memory region.

In the following, we explain how we tackle these issues to build a scalable timestamp oracle. The main idea is using a data structure called the **timestamp vector**, similar to a vector clock, which represents the
read timestamp as the following:

\[ T_R = \langle t_1, t_2, t_3, \ldots, t_n \rangle \]

Here, each component \( t_i \) in \( T_R \) is a unique counter that is assigned to one transaction execution thread \( i \) in a compute server, where \( i \) is a globally unique identifier. This vector can either be stored on one of the memory servers or also be partitioned across several servers as explained later. However, in contrast to vector clocks, we do not store the full vector with every record, but instead store only the timestamp of the compute server that did the latest update:

\[ T_C = \langle i, t_i \rangle \]

Here, \( i \) is the global transaction execution thread identifier and \( t_i \) the corresponding commit timestamp. This helps to mitigate one of the most fundamental drawbacks of vector clocks, the high storage overhead per record.

**Commit Timestamps:** Each component \( t_i = T_R[i] \) represents the latest commit timestamp that was used by an execution thread \( i \). Creating a new commit timestamp can be done without communication since one thread \( i \) executes transactions in a closed loop. The reason is that each thread already knows its latest commit timestamp and just needs to increase it by one to create the next commit timestamp. It then uses the previously-described protocol to verify if it is safe to install the new versions in the system with timestamp \( T_C = \langle i, t + 1 \rangle \) where \( t + 1 \) is the new timestamp.

At the end of the transaction, the compute server makes the updates visible by increasing the commit timestamp in the vector \( T_R \). That is, instead of adding the commit timestamp to a queue (line 32 of Listing 3.1), it uses an RDMA Write to increase its latest timestamp in the vector \( T_R \). No atomic operations are necessary since each transaction thread \( i \) only executes one transaction at a time.

**Read Timestamps:** Each transaction thread \( i \) reads the complete timestamp vector \( T_R \) and uses it as read timestamp \( rts \). Using \( T_R \), a transaction can read a record, including its header from a memory server, and check if the most recent version is visible to the transaction. The check is simple: as long as the version of the record \( \langle i, t \rangle \) is smaller or equal to \( t_i \) of the vector \( T_R \), the update is visible. If not, an older version has to be used to meet the condition. We will discuss details of the memory layout of our multi-versioning scheme in Section 3.6.

It is important to note that this simple timestamp technique has several important characteristics. First, long running transactions, stragglers, or crashed machines do not prevent the read timestamp to advance. The transaction threads are independent of each other. Second, if the timestamp is stored on a single memory server, it is guaranteed to increase monotonically. The reason is that all RDMA Writes are always materialized in the remote memory of the oracle and not cached on its NIC. Therefore, it is impossible for one transaction execution thread in a compute server to see a timestamp vector like \( \langle \ldots, t_n, \ldots, t_m + 1, \ldots \rangle \) while another observes \( \langle \ldots, t_n + 1, \ldots, t_m, \ldots \rangle \). As a result the timestamps are still progressing monotonically, similar to a single global timestamp counter. However, in the case where the timestamp vector is partitioned, this property might no longer hold true as explained later.
3.5.2 Further Optimizations

In the following, we explain further optimizations to make the timestamp oracle even more scalable:

**Dedicated Fetch Thread:** Fetching the most recent $T_R$ at the beginning of each transaction can cause a high network load for large transaction execution thread pools on large clusters. In order to reduce the network load, we can have one dedicated thread per compute server that continuously fetches $T_R$, and allow all transaction threads to simply use the pre-fetched $T_R$. At a first view, this seems to increase the abort rate since pre-fetching increases the staleness of $T_R$. However, due to the reduced network load, the runtime of each transaction is heavily reduced, leading instead to a lower abort rate.

**Compression of $T_R$:** The size of $T_R$ currently depends on the number of transaction execution threads, which could rise up to hundreds or even thousands entries when scaling out. Thus, instead of having one slot per transaction execution thread, we can compress $T_R$ by having only one slot $t_i$ per compute server; i.e., all transaction execution threads on one machine share one timestamp slot $t_i$. One alternative is that the threads of a compute server use an atomic RDMA fetch-and-add operation to increase the counter value. Since the number of transaction execution threads per compute server is bounded (if we use one dedicated thread per core) the contention will not be too high. As another alternative, we can cache $t_c$ in a compute server’s memory. Increasing $t_c$ is then implemented by a local compare-and-swap followed by a subsequent RDMA Write.

**Partitioning of $T_R$:** In our evaluation, we found that with those two optimizations, a single timestamp server is already able to sustain over 140 million txns/sec using a single dual-port FDR 4x NIC. In other words, we could scale our cluster to $\approx 500$ machines for TPC-C with two FDR 4x ports before the network bandwidth of the server becomes a bottleneck. In case a single server becomes the bottleneck, it is easy to partition $T_R$ across several memory nodes, since a transaction execution thread needs to update only a single slot $t_i$. This will improve the bandwidth per server as every machine now only stores a fraction of the vector. Unfortunately, partitioning $T_R$ no longer guarantees strict monotonicity. As a result, every transaction execution thread still observes a monotonically increasing order of updates, but the order of transactions between transaction execution threads might be different. While we believe that this does not impose a big problem in real systems, we are currently investigating if we can solve this by leveraging the message ordering guarantees provided by InfiniBand for certain broadcast operations. This direction represents an interesting avenue of future work.

3.6 Memory Servers

In this section, we first discuss the details of the multi-versioning scheme implemented in the memory servers of NAM-DB, which allows compute servers to install and find a version of a record. Afterwards, we present further details about the design of table and index structures in NAM-DB as well as memory management including garbage collection. Note that our design decisions are made to make distributed transactions scalable rather than optimize for locality.
3.6.1 Multi-Versioning Scheme

The scheme to store multiple versions of a database record in a memory server is shown in Figure 3.3. The main idea is that the most recent version of a record, called the current version, is stored in a dedicated memory region. Whenever a record is updated by a transaction (i.e., a new version needs to be installed), the current version is moved to an old-version buffer and the new current version is installed in-place. As a result, the most recent version can always be read with a single RDMA request. Furthermore, as we use continuous memory regions for the most recent versions, transferring the most recent versions of several records is also only a single RDMA request, which dramatically helps scans. The old-version buffer has a fixed size to be efficiently accessible with one RDMA read. Moreover, the oldest versions in the buffers are continuously copied to an overflow region. That way, slots in the old-version buffer can be re-used for new versions while keeping old versions available for long running transactions.

In the following, we first explain the memory layout in more detail and then discuss the version management.

Record Layout: For each record, we store a header section that contains additional metadata and a data section that contains the payload. The data section is a full copy of the record which represents a particular version. Currently, we only support fixed-length payloads. Variable-length payloads could be supported by storing an additional pointer in the data section of a record that refers to the variable-length part, which is stored in a separate memory region. However, when using RDMA, the latency for messages of up to 2KB remains constant, as shown in Section 2.5.2, as long as the bandwidth is not saturated. Therefore, for many workloads where the record size does not exceed this limit, it makes sense to store the data in a fixed-length field that has the maximal required length inside a record.

The header section describes the metadata of a record. In our current implementation, we use an 8-byte
value that can be atomically updated by a remote compare-and-swap operation from compute servers. The header encodes different variables: The first 29 bits are used to store the thread identifier $i$ of the transaction execution thread that installed the version (as described in the section before). The next 32 bits are used for the commit timestamp. Both these variables represent the version information of a record and are set during the commit phase. Moreover, we also store other data in the header section that is used for version management, each represented by a 1-bit value: a moved-bit, a deleted-bit, and a locked-bit. The moved-bit indicates if a version was already moved from the old-version buffer to the overflow region, and thus its slot can be safely reused. The deleted-bit indicates if the version of a record is marked for deletion and can be safely garbage collected. Finally, the locked-bit is used during the commit phase to avoid concurrent updates after the version check.

**Version Management:** The old-version buffer consists of two circular buffers of a fixed size as shown in Figure 3.3; one that holds only headers (excluding the current version) and another one that holds data sections. The reason for splitting the header and the data section into two circular old-version buffers is that the size of the header section is typically much smaller than the data section. That way, a transaction that needs to find a particular version of a record only needs to fetch all headers without reading the payload. Once the correct version is found, the payload can be fetched with a separate read operation using the offset information. This effectively minimizes the latency when searching a version of a record.

For installing a new version, we first validate if the current version has not changed since reading it and set the lock-bit using one atomic RDMA compare-and-swap operation (i.e., we combine validation and locking). If locking and validation fails, we abort. Otherwise, the header and the data of the current version is then copied to the old version buffer. In order to copy the current version to the buffers, a transaction first needs to determine the slot which stores the oldest version in the circular buffer and find out if that slot can be overwritten (i.e., the moved-bit is set to 1). In order to identify the slot with the oldest version, the circular buffers store an extra counter which is called the next-write counter.

Another issue is that the circular buffers have only a fixed capacity, the reason that we want to efficiently access them with one-sided RDMA operations and avoid pointer-chasing operations. However, in order to support long-running transactions or slow compute servers, the number of versions stored in the buffer might not be sufficient since they might need to read older versions. As a solution, a version-mover thread that runs in a memory server continuously moves the header and data section of the old-version buffers to an overflow region and sets the moved-bit in the header-buffer to 1. This does not actually mean that the header and data section are removed from the old-versions buffers. It only means that it can be safely re-used by a transaction that installs a new version. Keeping the moved versions in the buffer maximizes the number of versions that can be retrieved from the old-version buffers.

### 3.6.2 Table and Index Structures

In the following, we discuss how table and index structures are implemented in memory servers.

**Table Structures:** In NAM-DB, we only support one type of table structure that implements a hash table design similar to the one in [66]. In this design, compute servers can execute all operations on the hash table (e.g., put or a get) by using one-sided RDMA operations. In addition to the normal put and get operations
to insert and lookup records of a table, we additionally provide an update to install a new record version, as well as a delete operation.

The hash tables in NAM-DB stores key-value pairs where the keys represent the primary keys of the table. Moreover, the values store all information required by our multi-versioning scheme: the current record version and three pointers (two pointers to the old-version buffers as well as one pointer to the overflow region).

In contrast to [66], hash tables in NAM-DB are partitioned to multiple memory servers. In order to partition the hash table, we split the bucket array into equally-sized ranges and each memory server stores only one of the resulting sub-ranges as well as the corresponding keys and values. In order to find a memory server which stores the entries for a given key, the compute server only needs to apply the hash function which determines the bucket index (and thus the memory server which holds the bucket and its key-value pairs). Once the memory server is identified, the hash table operation can be executed on the corresponding memory server.

**Index Structures:** In addition to the table structure described before, NAM-DB supports two types of secondary indexes: a hash-index for single-key lookups and a $B^+$-tree for range lookups. Both types of secondary indexes map a value of the secondary attribute to a primary key that can then be used to lookup the record using the table structure dicussed before (e.g., a customer name to the customer key). Moreover, secondary indexes do not store any version information. Thus, retrieving the correct version of a record requires a subsequent lookup on the table structure using the primary key.

For NAM-DB’s secondary hash indexes, we use the same hash table design that we have described before for the table design. The main difference is that for values in a secondary hash index, we store only primary keys and no pointers (e.g., to old-version buffers etc.) as discussed before. For the $B^+$-tree index, we follow a different route. Instead of designing a tree structure that can be accessed purely by one-sided RDMA operations, we use two-sided RDMA operations to implement the communication between compute and memory servers. The reason is that operations in $B^+$-trees need to chase multiple pointers from the root to the leaf level, and we do not want to pay the network overhead for pointer chasing. While pointer chasing is also an issue for hash-tables, which use linked lists, [66] shows that clustering keys in a linked list into one memory region largely mitigates this problem (i.e., one RDMA operation can read the entire linked list). Moreover, for scaling-out and preventing individual memory servers from becoming a bottleneck, we range partition $B^+$-trees to different memory servers. In the future, we plan to investigate alternative indexing designs for $B^+$ trees.

### 3.6.3 Memory Management

Memory servers store tables as well as index structures in their memory as described before. In order to allow compute servers to access tables and indexes via RDMA, memory servers must pin and register memory regions at the RDMA network interface card (NIC). However, pinning and registering memory at the NIC are both costly operations which should not be executed in a critical path (e.g., whenever a transaction created a new table or an index). Therefore, memory servers allocate a large chunk of memory during initialization and register it to the NIC. After initialization, memory servers handle both allocate and free calls from compute
Allocate and Free Calls: Allocate and free calls from compute servers to memory servers are implemented using two-sided RDMA operations. In order to avoid many small memory allocation calls, compute servers request memory regions from memory servers in extends. The size of an extend can be defined by a compute server as a parameter and depends on different factors (e.g., expected size and update rate). For example, when creating a new table in NAM-DB, a compute server that executed the transaction allocates an extend that allows the storage of an expected number of records and their different versions. The number of expected records per table can be defined by applications as a hint.

Garbage Collection: To prevent old versions from taking up all the space, the job of the garbage collection is to determine old version records which can be safely evicted. In NAM-DB, garbage collection is implemented by having a timeout on the maximal transaction execution time \( E \) that can be defined as a parameter by the application. Transactions that run longer than the maximal execution time might abort since the version they require might already be garbage collected.

In order to implement our garbage collection scheme, we capture a snapshot of the timestamp vector \( T \) that represents the read timestamp of the timestamp oracle (see Section 3.5) in regular intervals. We currently create a snapshot of \( T \) every minute and store it together with the wall-clock time in a list sorted by the wall-clock time. That way, we can find out which versions can be safely garbage collected based on the maximal transaction execution time. For garbage collecting these versions, each memory server has a garbage collection thread which continuously scans the overflow regions and sets the deleted-bit of the selected versions of a record \( 1 \). These versions are then truncated lazily from the overflow regions once contiguous regions can be freed.

3.7 Compute Servers

In this section, we first discuss how compute servers execute transactions and then present techniques for logging and recovery as well as fault-tolerance.

3.7.1 Transaction Execution

Compute servers use multiple so called transaction execution threads to execute transactions over the data stored in the memory servers. Each transaction execution thread \( i \) executes transactions sequentially using the complete timestamp vector \( T \) as the read timestamp, as well as \( (i, T[i]) \) as the commit timestamp, to tag new versions of records as discussed in Section 3.5. The general flow of executing a single transaction in a transaction execution thread is the same workflow as outlined already in Section 3.4.1. Indexes are updated within the boundaries of the transaction that also updates the corresponding table using RDMA operations (i.e., we pay additional network roundtrips to update the indexes).

One import aspect that we have not discussed so far is how the database catalog is implemented such that transactions can find the storage location of tables and indexes. The catalog data is hash-partitioned and stored in memory servers. All accesses from compute servers are implemented using two-sided RDMA operations since query compilation does not result in a high load on memory servers when compared to
the actual transaction execution. Since the catalog does not change too often, the catalog data is cached by compute servers and refreshed in two cases. In the first case, a requested database object is not found in the cached catalog, and the compute server requests the required metadata from the memory server. The second case is if a database object is altered. We detect this case by storing a catalog version counter within each memory server that is incremented whenever an object is altered on that server. Since transaction execution threads run transactions in a closed loop, this counter is read from the memory server that stores the metadata for the database objects of a given transaction before compiling the queries of that transaction. If the version counter has changed when compared to the cached counter, the catalog entries are refreshed.

3.7.2 Failures and Recovery

NAM-DB provides a fault-tolerance scheme that handles failures of both compute and memory servers. In the following, we discuss both cases. At the moment, we do not handle failures resulting from network partitioning since the events are extremely rare in InfiniBand networks. These types of failures could be handled using a more complex commit protocol than 2PC (e.g., a version of Paxos based on RDMA), which is an interesting avenue of future work. Moreover, it is also important to note that high-availability is not the design goal of NAM-DB, which could be achieved in NAM-DB by replicating the write-set during the commit phase.

Memory Server Failures: In order to tolerate memory server failures, each transaction execution thread of a compute server writes a private log journal to a memory server using RDMA writes. In order to avoid the loss of a log, each transaction execution thread writes its journal to more than one memory server. The entries of such a log journal are \(<T, S>\) where \(T\) is the read snapshot used by thread \(i\) and \(S\) is the executed statement with all its parameters. Commit timestamps that have been used by a transaction execution thread are stored as parameters together with the commit statement and are used during replay. The log entries for all transaction statements are written to the database log before installing the write-set on the memory servers.

Once a memory server fails, we halt the complete system and recover all memory servers to a consistent state from the last persisted checkpoint (discussed below). For replaying the distributed log journal, the private logs of all transaction execution threads need to be partially ordered by their logged read timestamps \(T\). Therefore, the current recovery procedure in NAM-DB is executed by one dedicated compute server that replays the merged log for all memory servers.

In order to avoid long restart phases, an asynchronous thread additionally writes checkpoints to the disks of memory servers using a dedicated read-timestamp. This is possible in snapshot isolation without blocking other transactions. The checkpoints can be used to truncate the log.

Compute Server Failures: Compute servers are stateless and thus do not need any special handling for recovery. However, a failed compute server might result in abandoned locks. Therefore, each compute server is monitored by another compute server called a monitoring compute server. If a monitoring compute server detects that a compute server is down, it unlocks the abandoned locks using the log journals written by the transaction execution threads of this compute server.


3.8 Evaluation

The goal of our experiments is to show that distributed transactions can indeed scale and locality is just an optimization. In the following, we first discuss the benchmarks and the setup used for this evaluation and then present the results of the individual experiments.

**Benchmark:** We used TPC-C [97] as our main benchmark without any modifications unless otherwise stated for specific experiments. We generated 50 warehouses per memory server and created all required secondary indexes. All these indexes were implemented using our hash- and $B^+$-tree index as discussed in Section 3.6. Moreover, to show the effect of locality, we added a parameter to TPC-C that allows us to change the degree of distribution for new-order transactions from 0% to 100% (10% is the TPC-C specified configuration). As defined by the benchmark we only report the throughput of new-order transactions, which roughly make up 45% of all queries.

**Setup:** For executing the experiments, we used two different clusters, both with an InfiniBand network:

*Cluster A* has 57 machines, each with Mellanox Connect-IB card, and all connected through a single InfiniBand FDR 4X switch. The cluster contains two types of machines: the first 28 machines (type 1) have two Intel Xeon E7-4820 processors (each with 8 cores) and 128 GB RAM, the other 29 machines (type 2) have two Intel Xeon E5-2660 processors (each with 8 cores) and 256 GB RAM. All machines in this cluster run Oracle Linux Server 6.5 (kernel 2.6.32) and use the Mellanox OFED 2.3.1 driver for the network.

*Cluster B* has 8 machines connected to a single InfiniBand FDR 4X switch using a Mellanox Connect-IB card. Each machine has two Intel Xeon E5-2660 v2 processors (each with 10 cores) and 256GB RAM. The machines run Ubuntu 14.01 Server Edition (kernel 3.13.0-35-generic) as their operating system and use the Mellanox OFED 2.3.1 driver for the network.

For showing the scalability of NAM-DB, we used Cluster A. However, since we only had restricted access to that cluster, we executed the more detailed analysis (e.g., the effect of data locality) on Cluster B.
3.8.1 System Scalability

To show that NAM-DB scales linearly, the number of servers were increased from 2 to 56 on Cluster A. We used two configurations of NAM-DB, with and without locality. For the setup without locality optimization, we deployed 28 memory servers on type-2 machines and 28 compute servers on type-1 machines, the latter using 60 transaction execution threads per machine. For the setup with the locality optimization, we deployed 56 compute and 56 memory servers (one pair per physical machine). In this deployment, each compute server was running only 30 transaction execution threads to have the same total number in both deployments. Finally, in both deployments we used one additional dedicated memory server on a type-2 machine to store the timestamp vector.

Figure 3.4 shows the throughput of NAM-DB on an increasing cluster size both without exploring locality (blue) and with adding locality (purple) and compares them against a more traditional implementation of Snapshot Isolation (red) with two-sided message-based communication. The results show that NAM-DB scales nearly linearly with the number of servers to 3.64 million distributed transactions over 56 machines. However, if we allow the system to take advantage of locality, we achieve 6.5 million TPC-C new-order transactions. This is 2 million more transactions than the current scale-out record by Microsoft FaRM [26], which achieves 4.5 million TPC-C transactions over 90 machines with comparable hardware and using as much locality as possible. It should be noted though that FaRM was deployed on a cluster with ConnectX-3 NICs, not ConnectIB, which can have an performance impact if the number of queue pairs is large [43]. However, as Section 3.8.5 will show, for TPC-C this should make almost no difference. Furthermore, FaRM implements serializability guarantees, whereas NAM-DB supports snapshot isolation. While for this benchmark it makes no difference (there is no write-skew), it might be important for other workloads. At the same time, though, FaRM never tested their system for larger read queries, for which it should perform particularly worse as it requires a full read-set validation.

The traditional SI protocol in Figure 3.4 follows a partitioned shared-nothing design similar to [53] but using two-sided RDMA for the communication. As the figure shows, this design does not scale with the number of servers. Even worse, the throughput even degrades when using more than 10 machines. The degrade results from the high CPU costs of handling messages.
Figure 3.6: Scalability of Oracle

Figure 3.5a shows that the latency of new-order transactions. While NAM-DB almost stays constant regardless of the number of machines, the latency of the classic SI implementation increases. This is not surprising; in the NAM-DB design the work per machine is constant and is not related to the number of machines in the cluster, whereas the classical implementation requires more and more message handling.

When looking more carefully into the latency of NAM-DB w/o locality and its break-down (Figure 3.5b), it reveals that the latency increases slightly mainly because of the overhead to install new versions. In fact, we know from profiling that NAM-DB for TPC-C is network bandwidth bound. That is also the main reason why locality improves the overall performance, and we speculate that a system with the next generation of network hardware, such as EDR, would be able to achieve even higher throughputs. Finally, Figure 3.5b shows, that the latency for the timestamp oracle does not increase, indicating the efficiency of our new technique (note that we currently do not partition the timestamp vector).

### 3.8.2 Scalability of the Oracle

To test the scalability of our novel timestamp oracle, we varied the number of compute servers that concurrently update the oracle. As opposed to the previous experiment, however, compute servers do not execute any real transaction logic. Instead, each compute server thread executes the following three actions in a closed loop: (1) reads the current timestamp, (2) generates a new commit timestamp, and (3) makes the new commit timestamp visible. We call this sequence of operations a timestamp transaction, or simply t-txn. For this experiment, we used cluster B with eight nodes. The compute servers were deployed on seven machines, where we scaled the number of threads per machine. The remaining one node runs a memory server that stores the timestamp vector.

As a baseline, we analyze the original timestamp oracle of Section 3.4 (red line in Figure 3.6), which only achieved up to 2 million t-txns/sec. As shown in the graph, our old timestamp oracle did not scale. In fact, when scaling to more than 20 clients, the throughput starts to degrade due to high contention on the oracle. However, it should be noted that for smaller clusters, the threshold of 2 million t-txns/sec might be enough. As shown before, when running the full mix of TPC-C transactions, our system can execute up to 14 million transactions on 56 nodes (6.5 million of transactions are new-order). This load could not be handled by the
Figure 3.7: Effect of Locality

As shown in Figure 3.6, the new oracle (blue line) can easily sustain the above mentioned load. For example, the basic version with no optimization achieves 20 million t-txns/sec. However, the basic version still does not scale linearly because the size of the timestamp vector grows with the number of transaction execution threads (i.e., clients) and makes the network bandwidth of the timestamp server the main bottleneck.

While 20 million t-txns/sec is already sufficient that the basic new oracle (blue line) does not become a bottleneck in our experiments, we can push the limit even further by applying the optimizations discussed in Section 3.5.2. One of the optimizations is using a dedicated background fetch thread per compute server (instead of per transaction execution thread) to read the timestamp vector periodically. This reduces the load on the network. When applying this optimization (black line, denoted by “bg ts reader”), the oracle scales up to 36 million t-txns/sec. Furthermore, when applying compression (green line), where there is one entry in the timestamp vector per machine (instead of per transaction thread), the oracle scales even further to 80 million t-txns/sec. Finally, when enabling both optimizations (yellow line), the oracle scales up to 135 million t-txns/sec on only 8 nodes.

It is worth noting that even the optimized oracle reaches its capacity at some point when deployed on clusters with hundreds or thousands of machines (we speculate that with these two optimizations and the given hardware we could support a cluster size of 500 machines for TPC-C). At that point, the idea of partitioning the timestamp vector (see Section 3.5.2) could be applied to remove the bottleneck. Therefore, we believe that our proposed design for the timestamp oracle is truly scalable.

3.8.3 Effect of Locality

As described earlier, we consider locality an optimization technique, like adding an index, rather than a key requirement to achieve good scale-out properties. This is feasible with high-speed networks since the impact of locality is no longer as severe as it is on slow networks. To test this assumption, we varied the degree of distribution for new-order transactions from 0% up to 100%. The degree of distribution represents the likelihood that a transaction needs to read/write data from/to a warehouse that is stored on a remote server. When exploiting locality, transactions are executed at those servers that store the so-called home warehouse.
In this experiment, we only executed the new-order transaction and not the complete mix in order to show the direct effect of locality.

For the setup, we again use cluster $B$ with one server acting as the timestamp oracle and the remaining seven machines physically co-locating one memory and one computer server each. The TPC-C database contained 200 warehouses partitioned to all memory servers. When running w/o locality, we executed all memory accesses using RDMA. When running w/ locality, we directly accessed the local memory if possible. Since our HCA's atomic operations are not atomic with respect to the attached processor, all the atomic operations were issued as RDMA atomics, even in locality mode.

Figure 3.7 shows that the performance benefit of locality is roughly 30% in regard to throughput and latency. While 30% is not negligible, it still demonstrates that there are no longer orders-of-magnitude differences between them if the system is designed to achieve high distributed transaction throughput.

We also executed the same experiment on a modern in-memory database (H-Store [44]) that implements a classical shared-nothing architecture which is optimized for data-locality. We choose H-Store as it is one of the few freely available distributed in-memory transactional database systems. We used the distributed version of H-Store without any modifications over InfiniBand (IB) using IP over IB as communication stack. Overall, we observed that H-Store only achieves 11K transactions per second (not shown in Figure 3.7) on a perfectly partitionable workload. These numbers are in line with the ones reported in [75]. However, at 100% distributed transactions the throughput of H-Store drops to only 900 transactions per second, a 90% drop in its performance, while our system still achieves more than 1.5M transactions under the same workload. This clearly shows the sensitivity of the shared-nothing design to data locality.

### 3.8.4 Effect of Contention

As mentioned earlier, the scalability is influenced by the intrinsic scalability of the workload. In order to analyze the effect of contention on the scalability, we increased the number of machines with different levels of contention. That is, we varied the likelihood that a given product item is selected by a transaction by using a uniform distribution as well as different zipf distributions with low skew ($\alpha = 0.8$), medium skew ($\alpha = 0.9$), high skew ($\alpha = 1.0$) and very-high skew ($\alpha = 2.0$).
Figure 3.8 shows the results in regard to throughput and abort rate. For the uniform and zipf distribution with low skew, we can see that the throughput per machine is stable (i.e., almost linearly as before). However, for an increasing skewness factor the abort rate also increases due to the contention on a single machine. This supports our initial claim that while RDMA can help to achieve a scalable distributed database system, we can not do something against an inherently non-scalable workload that has individual contention points. The high abort rate can be explained by the fact that we immediately abort transactions instead of waiting for a lock once a transaction does not acquire a lock. It is important to note that this does not have a huge impact on the throughput, since in our case the compute server directly triggers a retry after an abort.

3.8.5 Scalability of RDMA Queue Pairs

Recent work [43] pointed out that a high number of queue pairs per NIC might limit the scalability. The reason is that the NIC cache is limited, so a high number of queue pairs (QPs) may overflow the cache, potentially causing the performance to degrade.

To investigate the impact of the number of queue pairs on our design, we dedicated one machine as server and seven machines as clients on Cluster \textit{B} and varied the number of queue pairs per client thread while running 20 threads per client. That way, we scaled the number of total queue pairs to approximately 4000. Moreover, at every round, a client thread chooses one of its queue pairs randomly, and issues a fixed-size RDMA Read to the server.

Figure 3.9 shows the total number of performed operations for three different sizes of one-sided RDMA Reads. The results show that queue pairs indeed have an impact on the overall performance, but mainly for small messages. For example, we observed a 40\% (25\%) drop in the throughput of 8-byte (64-byte) Reads. However, with 256-byte Reads, the number of queue pairs has almost no impact. In this case, the network bandwidth limits the maximum throughput, not the queue pairs. Thus, we argue that for many workloads (as well as benchmarks such as TPC-C) the number of queue pairs does not play an important role.

Also note that queue pairs are not needed to be established between all cores. For example, in the NAM architecture only queue pairs between servers and client-threads are required. This can be further reduced by not using a queue pair for every client-thread (as currently done in NAM-DB) but rather a few dedicated “communicator” threads at the potential cost of additional coordination overhead.
Finally, while in its simplest case, a queue pair is needed per core to enable RDMA and thus the increasing number of cores per CPU might again become a bottleneck, we observe that – at least currently – the cache sizes of NICs increase much faster than the number of cores per CPU. If this trend continues, this might further mitigate the problem in the future.

3.9 Related Work

Most related to our work is FaRM [25, 25]. However, FaRM uses a more traditional message-based approach and focuses on serializability, whereas we implemented snapshot isolation, which is more common in practice because of its low-overhead consistent reads. More importantly, in this work we made the case that distributed transactions can now scale, whereas FaRMs design is centered around locality.

FaSST [43] is another related which, similar to NAM-DB, focuses on scalability but the authors took a different approach by building an efficient RPC abstraction on top of 1-to-many unreliable datagrams using two-sided RDMA Send/Receive verbs. This design minimizes the size of queue pair state stored on NIC cache with the goal of better scalability with the size of cluster. However, as Section 3.8.5 showed, for many realistic workloads and cluster sizes, the number of queue pairs may not be that influential on performance. Due to their decision of abandoning one-sided RDMA verbs in favor of unreliable datagrams, their system is not able to take full advantage of leveraging the NIC as co-processors to access remote memory, and the design is likely more sensitive to data locality. Finally, and most importantly, FaSST implements serializability guarantees, whereas we show how to scale snapshot isolation, which provides better performance for read-heavy workloads and is more common in practice than serializability (e.g., Oracle does not even support it).

Another recent work [60] is similar to our design since it also separates storage from compute nodes. However, instead of treating RDMA as a first-class citizen, they treat RDMA as an afterthought. Moreover, they use a centralized commit manager to coordinate distributed transactions, which is likely to become a bottleneck when scaling out to larger clusters. Conversely, our NAM-DB architecture is designed to leverage one-sided RDMA primitives to build a scalable shared distributed architecture without a central coordinator to avoid bottlenecks in the design.

Industrial-strength products have also adopted RDMA in existing DBMSs. For example, Oracle RAC [74] has RDMA support, including the use of RDMA atomic primitives. However, RAC does not directly take advantage of the network for transaction processing and is essentially a workaround for a legacy system. Furthermore, IBM pureScale [9] uses RDMA to provide high availability for DB2 but also relies on a centralized manager to coordinate distributed transactions. Finally, SQLServer [58] uses RDMA to extend the buffer pool of a single node instance but does not discuss the effect on distributed databases at all.

Other projects in academia have also recently targeted RDMA for data management, such as distributed join processing [10, 82, 34]. However, they focus mainly only on leveraging RDMA in a traditional shared-nothing architecture and do not discuss the redesign of the full database stack. SpinningJoins [34] suggest a new architecture for RDMA. Different from our work, this work assumes severely limited network bandwidth (only 1.25GB/s) and therefore streams one relation across all the nodes (similar to a block-nested loop join).
Another line of work is on **RDMA-enabled key value stores** [71, 66, 42]. We leverage some of these results to build our distributed indexes in NAM-DB, but transactions and query processing are not discussed in these papers.

Furthermore, there is a huge body of work on distributed transaction processing over slow networks. In order to reduce the network overhead, many techniques have been proposed ranging from locality-aware partitioning schemes [80, 44, 21, 109] and speculative execution [76] to new consistency levels [49, 5, 6] and the relaxation of durability guarantees [50].

Finally, there is also recent work on high-performance OLTP systems for many-core machines [48, 101, 78]. This line of work is largely orthogonal to ours as it focuses on scale-up rather than scale-out. However, our timestamp oracle could be used in a scale-up solution to achieve better scalability.

### 3.10 Main Takeaways

In this chapter, we tackled the problem of making a scalable distributed database on modern networks even when all transactions are distributed. We proposed a new architecture for RDMA-enabled distributed systems called Network-Attached Memory, in which the storage and compute are logically decoupled, and compute nodes can access any storage node using RDMA operations.

Based on this overall architecture, we designed an OLTP system called NAM-DB, which provides the most common isolation level, Snapshot Isolation, and showed that although adding locality (i.e., making transactions single-partition) does increase the throughput in our system, but it is not essential in scalability of the system. This is in sharp contrast to the common belief in the database community that the very notion of distributed transactions equals poor scalability. As we saw, modern hardware is a key enabler in this change.

Each component of the system is carefully designed to be as scalable as possible. In particular, the main ideas in the three major components in NAM-DB can be summarized as follows:

- **Memory nodes**: we proposed an RDMA-friendly data layout suitable for multi-versioning databases which enable reading a suitable version with at least one (in the usual case) and at most two RDMA Read operations (in case an older version is needed). The two main ideas are (i) to store all records’ newest versions contiguously for more efficient RDMA access, and (ii) to encapsulate all the state for each record (e.g., its lock state, its latest timestamp, etc.) in the record’s header to allow remote compute nodes efficiently modify a record meta-data using RDMA atomic operations.

- **Compute nodes**: The key design principle in scalable compute nodes is to make them state less. All the state is stored on the memory nodes, and only the temporary buffers are kept on the compute side, where they heavily rely on RDMA operations to read and modify the state on remote memory nodes.

- **Timestamp oracle**: We found that scalable timestamp generation is essential in making the entire system scalable, and that a centralized timestamp generator soon reaches its capacity. To that end, we proposed a lock-free timestamp data structure for Snapshot Isolation. This data structure is a vector with each logical compute node having one entry, denoting the latest commit timestamp of that node.
The timestamp (i.e. the entire vector) can be read using one RDMA Read and updated using one RDMA Write. Using only RDMA means that no local active processing is needed to maintain the state of the timestamp oracle. Therefore, it easily scales with increasing number of compute nodes.
Chapter 4

Handling Data Contention

In Chapter 3, we proposed a new RDMA-based transactional system which exhibits perfect scalability even when all transactions are multi-partition.

In this chapter, we make the case that the new bottleneck which hinders truly scalable transaction processing in databases on modern networks is data contention. We will see that optimizing for data contention requires rethinking the way records should be partitioned, which turns out different than what existing partitioning techniques produce. In addition, minimizing contention demands a radically different method of executing transactions in database systems.

To this end, we present our solution called Chiller, a novel approach to data partitioning and transaction execution. Chiller aims to minimize data contention for both single- and multi-partition transactions, and therefore massively outperforms existing partitioning techniques.

This chapter is primarily based on our published work in SIGMOD 2020 with the title “Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks” [110].

4.1 Introduction

Since distributed transactions were conventionally associated with poor scalability, existing partitioning schemes split OLTP workloads such that the number of multi-partition transactions is minimized [21, 98, 75, 83, 92].

However, as we already showed in Chapter 3, thanks to modern high-bandwidth RDMA-enabled networks, avoiding multi-partition transactions is not necessary to achieve scalability and good performance. This raises the fundamental question of how data should be partitioned across machines given these modern networks. In this chapter, we argue that the new optimization goal should be to minimize contention rather than to minimize distributed transactions as is done in traditional schemes.

In this chapter, we present Chiller, a new partitioning scheme and execution model based on 2-phase-locking which aims to minimize contention. Chiller is based on two complementary ideas: (1) a novel commit protocol based on re-ordering transaction operations with the goal of minimizing the lock duration
4.1.1 Motivating Example
Assume a simple scenario with three servers in which each server can store up to two records, and a workload consisting of three transactions \( t_1, t_2, \) and \( t_3 \) (Figure 4.1a). All transactions update \( r_1 \). In addition, \( t_1 \) updates \( r_2 \), \( t_2 \) updates \( r_3 \) and \( r_4 \), and \( t_3 \) updates \( r_4 \) and \( r_5 \).

The common wisdom would dictate partitioning the data in a way that the number of cross-cutting transactions is minimized; in our example, this would mean co-locating all data for \( t_1 \) on a single server as shown in Figure 4.1b, and having distributed transactions for \( t_2 \) and \( t_3 \).

However, we argue that we can achieve a better partitioning, by first re-ordering the operations per transactions, so that the updates to the most contended items, here \( r_1 \) and \( r_4 \), are done last in the transactions, as shown in Figure 4.2a. Second, we argue that it is better to place \( r_1 \) and \( r_4 \) on the same machine, as in Figure 4.2b. At first this might seem counter-intuitive as it increases the total number of distributed transactions from two to three (\( t_1 \) is no longer a local transaction). However, this partitioning scheme can decrease the likelihood of conflicts and therefore increase the total transaction throughput.

The idea is that re-ordering the transaction operations minimizes the lock duration for the “hot” items and subsequently the chance of conflicting with concurrent transactions. More importantly, after the re-ordering, the success of a transaction relies entirely on the success of acquiring the lock for the most contended records. That is, if a distributed transaction has already acquired the necessary locks for all non-contended records (referred to as the outer region), the outcome of the transaction only depends on the success of updating the contended records (referred to as the inner region). This allows us to make all updates to the records in the inner region without any further coordination. Note that this partitioning technique primarily targets high-bandwidth low-latency networks, which mitigates the two most common bottlenecks for distributed transactions: message overhead and limited network bandwidth.

4.1.2 Challenges
In order to provide such a contention-aware transaction execution scheme, we need to address two inter-related challenges:
1. A contention-aware data partitioning algorithms: Different from existing partitioning algorithms that aim to minimize the number of distributed transactions (such as Schism [21]), Chiller’s partitioning algorithm must explicitly take record contention into account to co-locate hot records.

2. An operation-reordering execution scheme: At runtime, we need new scheme which, by taking advantage of the co-location of hot records, reorders operations such that it can release locks on hot records early and thus reduce the overall contention span on those records.

4.2 Contributions and Chapter Organization

In summary, we make the following contributions in this chapter:

1. We propose a new contention-centric partitioning scheme.

2. We present a new distributed transaction execution technique, which aims to update highly-contended records without additional coordination.

3. We show that Chiller outperforms existing techniques by up to a factor of 2 on various workloads.

The remainder of this chapter is organized as follows. In Section 4.3, we take a closer look at the main reasons for poor performance of distributed transactions in the presence of high contention in the workload. Then, in Section 4.4, we thoroughly present our novel protocol for executing transactions that deal with contended records. As already alluded by our motivating example, our proposed transaction execution protocol requires a new way of partitioning data, which we present in Section 4.5. Our proposed transaction execution also requires a new fault-tolerance scheme, explained in Section 4.6. After a brief discussion of the implementation details of our system in Section 4.7, we evaluate the effectiveness of our proposed technique through experiments in Section 4.8. Finally, we conclude this chapter with the related works in Section 4.9, and the main takeaway message of this chapter in Section 4.10.

4.3 Overview

The throughput of distributed transactions is limited by three factors: (1) message overhead, (2) network bandwidth, and (3) increased contention [12]. The first two limitations are significantly alleviated with the new generation of high-speed RDMA-enabled networks. However, what remains is the increased contention likelihood, as message delays are still significantly longer than local memory accesses.

4.3.1 Transaction Processing with 2PL & 2PC

To understand the impact of contention in distributed transactions, let us consider a traditional 2PL with 2PC. Here, we use transaction \( t_3 \) from Figure 4.1, and further assume that its coordinator is on Server 1, as shown in Figure 4.3a. The green circle on each partition’s timeline shows when it releases its locks and commits. We refer to the time span between acquisition and release of a record lock as the record’s
Figure 4.3: The lifetime of a distributed transaction. The green dots denote when the server commits the transaction. The blue lines represent the contention span for each server.

contention span (depicted by thick blue lines), during which, all concurrent accesses to the record would be conflicting, resulting in longer waiting time or aborts and thus negatively impacting the system performance. In this example, the contention span for all records is 2 messages long with piggybacking optimization (if the messages of the prepare phase can be merged onto the last step of the execution) and 4 without it.

While our example used 2PL, other concurrency control (CC) methods suffer from this issue, though to different extents [37]. For example in OCC, transactions must pass a validation phase before committing. If another concurrent transaction has modified the data accessed by a validating transaction, it has to abort and all its work will be wasted [22, 37].

4.3.2 Contention-Aware Transactions

As part of Chiller, we propose a new execution scheme that aims to minimize the contention span for contended records. The data partitioning layout shown in Figure 4.2b opens new possibilities for this purpose. As shown in Figure 4.3b, the coordinator requests locks for all the non-contended records in $t_3$, which is $r_5$. If successful, it will send a request to the partition hosting the hot records, Server 3, to perform the remaining part of the transaction. This request message will also include the values for the transaction read-set so far (i.e. $r_5$). Server 3 will attempt to acquire the lock for its two records, complete the read-set, and perform the transaction logic to check if the transaction can commit. If so, it commits the changes to its records. Server 3’s commit point, therefore, happens earlier than the other two involved partitions. Depending on the response from Server 3, the coordinator then sends a message to the other partitions to either apply their updates and commit, or roll back their changes and abort.

The reason that Server 3 can unilaterally commit or abort is that it contains all necessary data to perform the transaction logic. Therefore, the part of the transaction which has the hottest records is treated as if it were an independent local transaction. This effectively makes the contention span of $r_1$ and $r_4$ much shorter (just local memory access, as opposed to at least one network roundtrip).

4.3.3 Discussion

There are multiple details and simplifications hidden in the execution scheme presented above.

First, after sending the request to Server 3, neither the coordinator nor the rest of the partitions is allowed
to abort the transaction and this decision is only up to Server 3. For this reason, our system currently does not support triggers, which may cause the transaction to abort at any arbitrary point in its execution time. In that matter, its requirement is very similar to that of VoltDB [91] (and its academic predecessor H-Store [44]), Calvin [95], and MongoDB [7]. Also, the required determinism to disallow transactions to abort after a certain point in their life cycles is realized through the combination of Chiller’s novel execution, replication and recovery protocols, which will be discussed in Section 4.6.

Second, for a given transaction, the number of partitions for the inner region has to be at most one. Otherwise, multiple partitions cannot commit independently without coordination. This is why the execution of transactions in this manner requires a new partitioning scheme to ensure that contended records that are likely to be accessed together are located in the same physical partition, which will be formally presented in Section 4.5.

Finally, our execution model needs to have access to the transaction logic in its entirety to be able to reorder its operations. Our prototype achieves this by running transactions through invoking stored procedures, though it can be realized by other means such as implementing it as a query compiler (similar to Quro [105]). Due to the low overhead of our re-ordering algorithm, ad-hoc transactions can also be supported, as long as all operations of a transaction between BEGIN and END statements are issued in one shot. The main alternative model, namely interactive transactions, in which there may be multiple back-and-forth rounds of communication over a network between a client application and the database is extremely unsuitable for applications that deal with contended data yet demand high throughput, because the database cannot reason about the boundaries of transactions upfront, and therefore all locks and latches have to be held for the entire scope of the client interaction which may last multiple roundtrips [96].

4.4 Two-region Execution

We now present the two-region transaction protocol. We assume that we already have a contention-avoiding partitioning of the records, which will later be presented in Section 4.5.

4.4.1 General Overview

The goal of two-region execution scheme is to minimize the duration of locks on contended records by postponing their lock acquisition until right before the end of the expanding phase of 2PL, and performing their lock release right after they are read/modified, without involving them in the 2PC protocol. More specifically, the execution engine re-orders operations into cold operations (outer region) and hot operations (inner region); the outer region is executed as normal. If successful, the records in the inner region are accessed. The important point is that the inner region commits upon completion without coordinating with the other participants. Because of the way that the inner region is not involved in 2PC, fault tolerance requires a complete re-visit, otherwise many failure scenarios may sacrifice the system’s correctness or liveness. Until we discuss our fault tolerance algorithm in Section 4.6, we present the execution and partitioning schemes under a no-failure assumption.
Figure 4.4: Two-region execution of a simplified ticket purchasing transaction. In the dependency graph, primary key and value dependencies are shown in solid and dashed lines, respectively (blue for conditional constraints, e.g., an “if” statement). Assuming the flight record is contended (red circles), the red box in (c) shows the operations in the inner region (Step 4). The rest of the operations will be performed in the outer region (Steps 3 and 5).

To help explain the concepts, we will use an imaginary flight-booking transaction shown in Figure 4.4a. Here, there are four tables: flight, customer, tax and seats. In this example, if the customer has enough balance and the flight has an available seat (line 12), a seat is booked (lines 14 and 16) and the ticket fee plus state-tax is deducted from their account (line 15). Otherwise, the transaction aborts (line 19).

The remainder of this section is structured as follows. Section 4.4.2 describes how we extract the constraints in re-ordering operations from the transaction logic and model it as a dependency graph. Using such a graph, a five-step protocol, described in Section 4.4.3, is used to execute a two-stage transaction. Finally, in Section 4.4.4, we present optimizations, and look at challenges for the protocol to be correct and fault tolerant. Our solution to these challenges is then presented throughout the subsequent sections.

4.4.2 Constructing a Dependency Graph

There may be constraints on data values that must hold true (e.g., there must be an available seat in the flight for the booking transaction). Furthermore, operations in a transaction may have dependencies among each other. The goal is to reflect such constraints in the dependency graph. In our prototype, this graph is built when registering a new stored procedure. Here, we distinguish between two types of dependencies. A primary key dependency (pk-dep) is when accessing a record r2 can happen only after accessing record r1, as the primary key of r2 is only known after r1 is read (e.g., the read operation for the tax record in line 7 must happen after the read operation for the customer record in line 6). In a value dependency (v-dep), the new values for the update columns of a record r2 are known only after accessing r1 (e.g., the update operation in line 14). To determine the dependencies, we are only concerned about the pk-deps, and not the v-deps. This is because v-deps do not restrict the order of lock acquisition, while pk-deps do put restrictions on which re-orderings are possible.

Each operation of the transaction corresponds to a node in the dependency graph. There is an edge from
node $n_1$ to $n_2$ if the corresponding operation of $n_2$ depends on that of $n_1$. The dependency graph for our running example is shown in Figure 4.4b. For example, the insert operation in line 15 ($s_{\text{ins}}$ in the graph) has a pk-dep on the read operation in line 5 ($f_{\text{read}}$), and has a v-dep on the read operation in line 6 ($c_{\text{read}}$). This means that getting the lock for the insert query can only happen after the flight record is read (pk-dep), but it can happen before the customer is read (v-dep). Please refer to the figure’s caption for the explanation of the color codes.

4.4.3 Run-Time Decision

Given the dependency graph, we describe step-by-step how the protocol executes a two-region transaction.

1) Decide on the execution model: First, it must find the list of candidate operations for the inner region. An operation can be a candidate if the record(s) accessed by it is marked as contended in the lookup table, and it does not have any pk-dep on operations on other partitions, since if it does, it would make early commit of the inner region impossible. In Figure 4.4b, if the insert operation $s_{\text{ins}}$ belongs to a different partition than $f_{\text{read}}$, the latter cannot be considered for the inner region because there is a pk-dep between them.

Note that finding the hosting partition of an operation which accesses records by their primary keys is quite straightforward. However, finding this information for operations which access records by attributes other than their primary keys may require secondary indexes. In case no such information is available, such operations will not be considered for the inner region.

2) Select the host for the inner region: If all candidate operations for the inner region belong to the same host, then it is chosen as the inner host, otherwise, one has to be picked. Currently, we choose the candidate with the highest number of hot operations as the inner host.

3) Read records in the outer region: The transaction attempts to lock and read the records in its outer region. In our example, an exclusive lock for the customer record and a shared lock for the tax record are acquired. If either of these lock requests fails, the transaction aborts.

4) Execute and commit the inner region: Once all locks have been acquired for the records in the outer region, the coordinator delegates processing the inner region to the inner host by sending a message with all information needed to execute its part (e.g., transaction ID, input parameters, etc.). Having the values for all of the records in the read-set allows the inner host to check if all of the constraints in the transaction are met (e.g., there are free seats in the flight). This guarantees that if operation in the outer region should result in an abort, it will be detected by the inner host and the entire transaction will abort.

Once all locks are successfully acquired and the transaction logic is checked to ensure the transaction can commit, the inner host updates the records, replicates its changes to its replicas (Section 4.6.1) and commits. In case any of the lock requests or transaction constraints fails, the inner host aborts the transaction and directly informs the coordinator about its decision. In our example, the update to the flight record is applied, a new record gets inserted into the seats table, the partial transaction commits, and the value for the cost variable is returned, as it will be needed to update the customer’s balance by another partition.

5) Commit the outer region: If the inner region succeeds, the transaction is already considered committed and the coordinator must commit all changes in the outer region. In our example, the customer’s balance gets updated, and the locks are released from the tax and customer records.
4.4.4 Challenges

There are two main challenges for efficiently implementing the two-region execution model. First, it will not be useful if the hot records of a transaction are scattered across different partitions. No matter which partition becomes the inner host, the other contended records will observe long contention spans. Therefore, frequently co-accessed hot records must be co-located. To accomplish this goal, we present a novel partitioning technique in Section 4.5. Second, the inner host removes its locks earlier than the other participants (steps 4 and 5). For this reason, fault tolerance requires a revisit, which will be presented in Section 4.6.

4.5 Contention-aware Partitioning

To fully unfold the potential of the two-region execution model, the objective of our proposed partitioning algorithm is to find a horizontal partitioning of the data which minimizes the contention. To better explain the idea, we will use 4 transactions shown in Figure 4.5. The shade of red corresponds to the record hotness (darker is hotter), and the goal is to find two balanced partitions. To keep things simple, we define “balanced” as a partitioning that splits the set of records in half. A formal definition of balance will ensue in Section 4.5.4. Existing partitioning schemes try to minimize the number of distributed transactions. For example, Schism [21] would create the partitioning shown in Figure 4.5b. However, this partitioning scheme would increase the contention span for records 3 or 4, and 6 in transaction $t_2$, because $t_2$ will have to hold locks on either 3 or 4, and 6 as part of an outer region.

The challenge in creating a contention-aware partitioning is that in addition to determining the record-to-partition mapping, it must also help specify which operations should go inside the inner region of a given transaction. For example, a better split is shown in Figure 4.5c since the contended part for each transaction can be accessed in an inner region.

Figure 4.5: An example workload and how partitioning techniques with different objectives will partition it into two parts.
Not only the main objective of our proposed partitioning is different than the one in existing partitioning techniques, but also their commonly used graph representation of the workload (with records as vertices and co-accesses as edges, as in Figure 4.5b) does not capture the essential requirements of our problem, i.e., the distinction of inner and outer region and differences in their execution. This necessitates a new workload representation.

4.5.1 Overview of Partitioning

To measure the hotness of records, servers randomly sample the transactions’ read- and write-sets during execution. These samples are aggregated over a pre-defined time interval by the partitioning manager server (PM). PM uses this information to estimate the contention of individual records (Section 4.5.2). It then creates the graph representation of the workload which captures these contention estimates and accommodates the requirements for the two-region execution model (Section 4.5.3). Based on this representation, it then uses a graph partitioning tool to partition the records with the objective of minimizing the overall contention of the workload (Section 4.5.4). Finally, it updates servers’ lookup tables with new partition assignments.

For the sake of clarity, in the remainder of this section we assume that records the units of locking, and hence, partitioning. However, the same concepts can be used for more coarse grained lock units, such as pages or hash buckets.

4.5.2 Contention Likelihood

Using the aggregated samples, PM calculates the total access frequencies of records to determine their conflict likelihoods. More specifically, we define the probability of a conflicting access for a given record as:

\[
P_c(X_w, X_r) = P(X_w > 1)P(X_r = 0) + P(X_w > 0)P(X_r > 0)
\]

Here, \(X_w\) and \(X_r\) are random variables corresponding to the number of times a given record is read or modified within the lock window, respectively (A lock window is defined as the average time a lock is held on a record in our experiments). The equation consists of two terms to account for the two possible conflict scenarios: (i) write-write conflicts, and (ii) read-write conflicts. Since (i) and (ii) are disjoint, we can simply add them together.

Similar to previous work [106, 49], we model \(X_w\) (\(X_r\)) using a Poisson process with a mean arrival time of \(\lambda_w\) (\(\lambda_r\)), which is the time-normalized access frequency. This allows us to rewrite the above equation as follows:

\[
P_c(X_w, X_r) = (1 - \left(\frac{\lambda_w^0 e^{-\lambda_w}}{0!} + \frac{\lambda_w^1 e^{-\lambda_w}}{1!}\right)\frac{\lambda_r^0 e^{-\lambda_r}}{0!}) + \left(1 - \frac{\lambda_w^0 e^{-\lambda_w}}{0!}\right)\left(1 - \frac{\lambda_r^0 e^{-\lambda_r}}{0!}\right)
\]

\[
= 1 - e^{-\lambda_w} - \lambda_w e^{-\lambda_w} e^{-\lambda_r}
\]
Note that the arrival rates and the contention probability are defined per lock unit, i.e., record. We use $P_c(\rho)$ to refer to the contention likelihood of record $\rho$. In the equation above, when $\lambda_w$ is zero, meaning no write has been made to the record, $P_c(\rho)$ will be zero, since shared locks are compatible with each other so do not cause any conflicts. With a non-zero $\lambda_w$, higher values of $\lambda_r$ will increase the contention likelihood due to the conflict of read and write locks.

### 4.5.3 Graph Representation

There are three key properties that a graph representation of the workload should have to properly fit in the context of our execution model. First, record contentions must be captured in the graph as this is the main objective. Second, the relationship between records must also be modeled, due to the requirement that there can be only one inner region for a transaction, and hence the frequently co-accessed contended records should be co-located. Third, the final partitioning should also make it possible to determine the inner region for each transaction. Therefore, Chiller models the workload quite differently than existing partitioning algorithms.

As shown in Figure 4.5c, we model each transaction as a star; at the center is a dummy vertex (referred to as t-vertex, denoted by squares) with edges to all the records that are accessed by that transaction. Thus, the number of vertices in the graph is $|T| + |R|$, where $|T|$ is the number of transactions and $|R|$ is the number of records. The number of edges will be the sum of the number of records involved per transaction.

All edges connecting a given record-vertex (r-vertex) to all of its t-vertex neighbors have the same weight. This weight is proportional to the record’s contention likelihood, as defined before in Section 4.5.2. More contended records will have edges with higher weights. In the context of the two-region execution model, the weight of the edge between an r-vertex and a connected t-vertex reflects how bad it would be if the record is not accessed in the inner region of that transaction.

Applying the contention likelihood formula to the our running example and normalizing the weights will produce the graph with the edge weights in Figure 4.5c. Note that there is no edge between any two records. Co-accessing of two records is implied by having a common t-vertex connecting them. Next, we will describe how our partitioning algorithm takes this graph as input and generates a partitioning scheme with low contention.

### 4.5.4 Partitioning Algorithm

As we are able to model contention among records using a weighted graph, we can apply standard graph partitioning algorithms. More formally, our goal is to find a partitioning, which minimizes the contention:

$$
\min_S \sum_{\rho \in R} P_c^{(S)}(\rho)
$$

s.t. $\forall p \in S : L(p) \leq (1 + \epsilon) \cdot \mu$

Here, $S$ is a partitioning of the set of records $R$ into $k$ partitions, $P_c^{(S)}(\rho)$ is the contention likelihood of record $\rho$ under partitioning $S$, $L(p)$ is the load on partition $p$, $\mu$ is the average load on each partition, and $\epsilon$
is a small constant that controls the degree of imbalance. Therefore, \( \mu = \frac{\sum_{p \in P} L(p)}{|P|} \). The definition of load will be discussed shortly.

Chiller makes use of METIS [45], a graph partitioning tool which aims to find a high-quality partitioning of the input graph with a small cut, while at the same time respecting the constraint of approximately balanced load across partitions. The resulting partitioning assigns each vertex to a partition.

For our specific problem, the interpretation of the partitioning is as follows: A cut edge \( e \) connecting a r-vertex \( v \) in one partition to a t-vertex \( t \) in another partition implies that \( t \) will have to access \( v \) in its outer region, and thus observing a conflicting access with a probability proportional to \( e \)'s weight. To put it differently, the partition to which \( t \) is assigned determines the inner host of \( t \); all r-vertices assigned to the same partition can be executed in the inner region of \( t \). As a result, finding the partitioning which minimize the total weight of all cut edges also minimizes the contention.

In our example, the sum of the weights of all cut edges (which are portrayed as green lines) is 1.3. Transaction \( t_1 \) will access record 3 in its inner region as its t-vertex is in the same partition as record 3, while it will access records 1 and 2 in its outer region. Note that even though compared to the partitioning in Figure 4.5b, multi-partition transactions is increased by two, this split results in a much lower contention.

While the objective function minimizes the contention, the load constraint ensures that the partitions are approximately balanced. The load \( L \) for a partition can be defined in different ways, such as the number of executed transactions, hosting records, or record accesses. The weights of the vertices in the graph will depend on the chosen load metric. For the metric of number of executed transactions, t-vertices have a weight of 1 while r-vertices will have a weight of 0. The weighting is reversed for the metric of number of hosting records. Finally for the metric of the number of record accesses, r-vertices are weighed proportionally to the sum of reads and writes to them. METIS generates a partitioning such that the sum of vertices weights in each partition is approximately balanced.

### 4.5.5 Discussion

**Scalability of Partitioning.**

There are two issues every partitioning scheme has to address: (1) the graph size and the cost of partitioning it, and (2) the size of the lookup table.

Even with the recent advances in graph partitioning algorithms, it is still expensive to partition a large graph with possibly billions of nodes. However, Chiller has a unique advantage compared to existing partitioning techniques: it produces graphs with significantly fewer edges for most workloads. For example, the most commonly used workload graph (used by Schism [21], Clay [83], JECB [98], etc.) introduces one edge for every new co-accessed data item, for a total of \( n(n-1)/2 \) edges for a transaction with \( n \) records. However, because of our star representation, we only introduce \( n \) edges per transaction; one to connect every r-vertex to the t-vertex of that transaction. Such representation accounts for a much smaller graph compared to the existing tools. For example, we found that on average, constructing the workload graph and applying the METIS partitioning tool take up to 5 times longer on Schism compared to Chiller for the datasets we used in our experiments.
Furthermore, our approach provides a unique opportunity to reduce the size of the lookup table. As we are mainly interested in reducing contention, we can focus our attention on the records with a contention likelihood above a given threshold. Hence, the lookup table only needs to store where these hot records are located. All other records can be partitioned using an orthogonal scheme, for example, hash- or range-partitioning, which literally takes almost no lookup-table space. Note that this technique would increase multi-partition transactions but not contention. We study this technique in more depth using an experiment (Section 4.8.5).

Re-Partitioning and Data Migration.

While the partitioning process described in Section 4.5.1 can be done periodically for the purpose of re-partitioning and cope with changing characteristics of the workload, our current prototype is based on an offline version of the Chiller partitioner. However, our experience

In our experiments, running the partitioning algorithm on a set of 100 thousands sampled transactions for a workload with as many as 30 millions records (YCSB in the experiments) took less than ten minutes on one of the machines we used in our evaluations (Section 4.8.1). This time includes calculating the contention likelihoods, building the workload graph, and partitioning it using METIS. In addition, we also found that the graph pruning techniques proposed above are quite effective in reducing the partitioning time without significantly impacting the throughput. Therefore, we envision that the off-line re-partitioning scheme presented in this chapter would be sufficient for many workloads, and re-partitioning can be as simple as running the algorithm from scratch on one or multiple partitioning managers. For other workloads with more frequently changing hot spots, however, it is possible that constantly relocating records in an incremental way is more effective compared to doing it in an offline fashion. Extending this work to support online re-partitioning is an interesting direction of future work.

Another related topic is when a re-partitioning of the database happens, how the system relocates records to their new partitions while still maintaining ACID guarantees. In our current prototype, the partitioning algorithm produces a record relocation list, which can be used by the system to move records transactionally (each tuple migration is performed as one individual transaction). As data migration is not specific to our partitioning algorithm, and is a general requirement by every production OLTP partitioning tool, there are many automatic tools which perform this task more efficiently [28, 102, 27]. We are planning to extend our prototype to use Squall [27], which is a live data migration tool.

Minimizing Distributed Transactions.

Finally, it is also possible to co-optimize for contention and minimizing distributed transactions using the same workload representation. To do this, one only needs to assign a minimum positive weight to all edges in the graph. The bigger the minimum weight, the stronger the objective to co-locate records from the same transaction. Such co-optimization is still relevant even in fast RDMA-enabled networks since a remote access through RDMA is about $10 \times$ slower than a local access. However, as we argue in this chapter, the optimal partitioning objective should shift in the direction of minimizing contention, and therefore minimizing distributed transactions is just a secondary optimization.
4.6 Fault Tolerance

The two-region execution model presented in Section 4.4 modifies the typical 2PC for transactions accessing contended records. In this model, a transaction is considered committed once its processing is finished by the inner host, after which, it must be able to commit on the other participants even if any of the involved nodes fail. To put it differently, a participant in the outer region cannot unilaterally abort a transaction once it has granted its locks in the outer region and informed the coordinator, since the transaction may have already been committed by the inner host.

Chiller employs write-ahead logging to non-volatile memory. However, similar to 2PC, while logging enables crash recovery, it does not provide high availability. The failure of the inner host before sending its decision to the coordinator may sacrifice the availability, since the coordinator would not know if the inner region is already committed or not, in which case it has to wait for the inner host to recover.

To achieve high availability, Chiller relies on a new method of replication, which is based on synchronous log-shipping, and explained in Section 4.6.1. In Section 4.6.2, we discuss how this replication protocol achieves high availability while still maintaining consistency.

4.6.1 Replication Protocol

In conventional synchronous log-shipping replication, the logs are replicated before the transaction commits. Since in Chiller, the transaction commit point (i.e., when the inner region commits) happens before the outer region participants commit their changes (see Figure 4.3b), the inner region replication cannot be postponed to the end of the transaction, otherwise its changes may be lost if the inner host fails.

To solve this problem, Chiller employs two different algorithms for the replication of the inner and outer regions. The changes in the outer region are replicated as normal—once the coordinator finishes performing the operations in the transaction, it replicates the changes to the replicas of the outer region before making the changes visible. The inner region replication, however, must be done before the transaction commit point, so that the commit decision will survive failures. Below, we describe the inner region replication in terms of the different roles of the participants:

**Inner host:** As shown in Figure 4.6, when the inner host finishes executing its part, it sends an RPC message to its replicas containing the new values of its records, the transaction read-set, and the sequence ID of the replication message. It then waits for the acks from its NIC hardware, which guarantee that the messages have been successfully sent to the replicas. Finally, it safely commits its changes. In case the inner host decides to abort, the replication phase will not be needed and it can directly reply to the coordinator.

**Replicas of the inner host:** Each replica applies the updates in the message in the sequence ID order. This guarantees that the data in the replicas synchronously reflect the changes in the primary inner host partition. When all updates of the replication message are applied, each replica notifies the original coordinator of the transaction, as opposed to responding back to the inner host. This saves one network message delay.

**Coordinator:** The coordinator is allowed to resume the transaction only after it has received the notifications from all the replicas of the inner host.

In the following, we describe our failure recovery protocol, and how it guarantees both safety and liveness.
4.6.2 Failure Recovery

For failure detection and membership reconfiguration, Chiller relies on a cluster manager such as Zookeeper [39]. When a machine is suspected of failure, all of the other nodes close their communication channels to the suspected node to prevent it from interfering in the middle of the recovery process.

The recovery procedure is as follows: First, each partition \( p \) probes its local log, and compiles a list of transactions which have been started but are not yet committed or aborted on \( p \). For each transaction, its coordinator, inner host, and the list of outer region participants are retrieved. These lists are aggregated at a designated node to make a global list of pending transactions. Below, possible failure scenarios for a pending two-region transaction along with how the fault tolerance is achieved are discussed.

**Failure of inner host:** If none of the surviving replicas of a failed inner host has received the replication message, the transaction can be safely aborted, because it indicates that the inner host has not committed either. However, if at least one of its replicas has received such a message, that transaction can commit, even though that it might have not yet replicated on all the replicas. In this case, the coordinator finishes the transaction on the remaining inner host replicas and the outer region participants, and commits.

**Failure of coordinator:** If a node is found to be the inner host (or one of its replicas, in case the inner host is failed too), it will be elected as the new coordinator, since it already has the values for the transaction read-set. Otherwise, the transaction can be safely aborted because its changes are not yet received/committed by its inner host.

**Failure of an outer region participant:** If the failure of participant \( i \) happens before the coordinator initiates the inner region, then the transaction can be safely aborted. Otherwise, one of \( i \)'s replicas which has been elected as the new primary will be used to take over \( i \)'s role in the transaction.

**Proof of Correctness** — We now provide some intuition on the protocol correctness in terms of safety and liveness.

It is easy to see why the two-region execution model with the described replication protocol maintains safety, since transactions are serialized at the point when their inner host commits. Also, similar to 2PC, if even one participant commits (aborts), no other participant is allowed to abort (commit). This is due to the “no turning back” concept of the commit protocol of the inner region, guaranteeing that all participants will agree on the same decision.

To support liveness, the system first needs to detect failures and repair/replace faulty nodes. For this purpose, Chiller relies on the existence of a fault tolerant coordinator, such as Zookeeper [39] or Chubby [15]. So
long as at most $f$ out of $f + 1$ replicas fail, the protocol guarantees liveness by satisfying these two properties:

1. A transaction will eventually commit at all its participants and their replicas once it has been committed by the inner host: The inner host commits only when its changes are replicated on its replicas. Therefore, there will be at least one replica containing the commit decision which causes the transaction to commit during the recovery process.

2. A transaction which is not yet committed at its inner host will eventually either abort or commit: If the inner host does not fail, it will eventually process the inner region. However, if it encounters a failure, the transaction will be discovered during the failure recovery, and handled by a new inner host.

4.7 Implementation

We now briefly present the implementation of the system we used to evaluate Chiller's partitioning and two-region execution schemes.

In our system, tables are built on top of a distributed hash table, and are split horizontally into multiple partitions, with each execution server hosting one partition in its main memory. The unit of locking is hash bucket, and each hash bucket encapsulates its lock metadata in its header, eliminating the need for a centralized lock manager per partition. Our current implementation performs locking on bucket granularity and does not prevent the phantom problem. However, the core idea of Chiller also works with range locks.

An execution server utilizes all its machine’s processing cores through multi-threading, where each execution thread has access to the hosted partition on that machine. To minimize inter-thread communication, each transaction is handled from its beginning to the end by one execution thread. To extract maximum concurrency, each worker thread employs multiple co-routine workers, such that when one transaction is waiting for a network operation to be completed, it yields to the next co-routine worker which processes a new transaction. The communication needed for distributed transactions is done either through direct remote memory operations (RDMA Read, Write, and atomic operations), or via RPC messages implemented using RDMA Send and Receive. The access type for each table is specified by the user when creating that table.

Bucket to partition mappings are stored in a lookup table, which is replicated on all servers so that execution threads would know where each record is. It can be either defined by the user in the form of hash or range functions on some table attributes, or produced individually for all or a subset of buckets using the partitioning algorithm. In addition to storing the partition assignments, the lookup table also contains the list of buckets with a contention above a given threshold, which is used to determine the inner region for each transaction.

As discussed in Section 4.6, to guarantee high availability in the presence of failures, we use log-shipping with 2-safety, where each record is replicated on two other servers. The backup replicas get updated synchronously before the transaction commits on the primary replica. In addition, like other recent high-performance OLTP systems [26, 43, 108], crash recovery is guaranteed by relying on the persistence of execution servers’ logs on some form of non-volatile memory (NVM), such as battery-backed DRAM.
4.8 Evaluation

We evaluated our system to answer two main questions:

1. How does Chiller and its two-region execution model perform under various levels of contention compared to existing techniques?

2. Is the contention-aware data partitioning effective in producing results that can efficiently benefit from the two-region execution model?

4.8.1 Setup

The test bed we used for our experiments consists of 7 machines connected to a single InfiniBand EDR 4X switch using a Mellanox ConnectX-4 card. Each machine has 256GB RAM and two Intel Xeon E5-2660 v2 processors with 2 sockets and 10 cores per socket. In all experiments, we use only one socket per machine where the NIC is directly attached and disable hyper-threading to minimize the variability in measurements caused by same-core threads interference, which is a typical setup used in similar works [86, 107, 111]. The machines run Ubuntu 14.04 Server Edition as their OS and Mellanox OFED 3.4-1 driver for the network.

4.8.2 Baselines

To assess the ability of the two-region execution model in handling contention, we evaluate how it holds up against alternative commonly used concurrency control (CC) models, more specifically these protocols:

**Two-Phase Locking (2PL):** we implemented two widely used variants of distributed 2PL with deadlock prevention. In **NO_WAIT**, the system aborts a transaction once it suspects of a deadlock, i.e., when a lock request by a transaction is denied. Therefore, waiting for locks is not allowed. In **WAIT_DIE**, transactions are assigned unique timestamps before execution. An older transaction is allowed to wait for a lock which is owned by a younger transaction, otherwise it aborts. Timestamp ordering ensures that no deadlock is possible. Note that while one could also implement 2PL with deadlock detection, it demands significant network synchronization between servers to detect cycles in the lock-request graph, and is therefore very costly in a distributed setting [37, 1]. Therefore, we did not include it in our evaluation.

We based the implementation of Chiller’s locking mechanism on **NO_WAIT** due to its lower overhead (no need to manage lock queues), although **WAIT_DIE** could also be used.

**Optimistic (OCC):** We based our implementation on the MaaT protocol [62], which is an efficient and scalable algorithm for OCC in distributed settings [37]. Each transaction is assigned a range for its commit timestamp, initially set to $[0, \infty]$. Also, the DBMS stores for each record the list of pending reader IDs and writer IDs, and the ID of the last committed transaction which accessed the record. Each time a transaction reads/modified a record, it modifies its timestamp range to be in compliance with the read/write timestamp of that record, and adds its unique timestamp to the list of the record’s read/write IDs. At the end, each participant of the transaction attempts to validate its part by changing the timestamp ranges of the validating
transaction and the other conflicting transactions, and votes to commit if the final range of the transaction is valid. The coordinator commits a transaction only if all the participants vote to commit.

In addition to these CC schemes, we evaluate two common partitioning schemes:

**Hash-partitioning** is the simple strategy of applying a hash function to a record’s primary key(s) to determine its partition assignment.

**Schism** is the most notable automatic partitioning technique for OLTP workloads. It first uses Metis to find a small cut of the workload graph, then compares this record-level partitioning to both a decision tree-learned range partitioning and a simple hash partitioning and picks the one which results in the minimum number of distributed transactions, or if equal, requires a smaller lookup table. We include the results for different CC schemes for Schism partitioning, and report only **NO_WAIT** for hash partitioning as a simple baseline.

### 4.8.3 Workloads

For our experiments, we use the following workloads and measure throughput as number of committed transactions per second (i.e., excluding aborted transactions).

**TPC-C:** This is the de facto standard for evaluating the performance of OLTP systems. This benchmark is centered around an order-processing application with multiple warehouses. It consists of 9 tables and 5 types of transactions. The majority of transactions access records belonging to a single warehouse. Therefore, the obvious partitioning layout is by warehouse ID. Despite being highly partitionable, the standard TPC-C contains two severe contention points. First, each *new-order* transaction does an increment on one out of 10 records in the district table of its warehouse. Second, every *payment* transaction updates the total balance of a warehouse and one of its 10 districts, creating an even more severe contention point. These two transactions comprise more than 87% of the workload.

We used one warehouse per server (i.e., 7 warehouses in total) which translates to a high contention workload. This allows us to focus on the differences in the execution models of Chiller and traditional schemes.

**YCSB:** The Yahoo Cloud Serving Benchmark (YCSB) [18] simulates large-scale applications of Internet-based companies. YCSB consists of a single table with 1KB records. For all experiments, we generated 5 million records (∼5 GB) per server. To generate read and write-sets of transactions with a desired level of locality, we used a mapping function from records to partitions when generating the workload trace. Since the benchmark does initially not specify any transactions, we group multiple read/write-operations into one transaction as discussed next.

To explore different aspects of the problem in more depth, we used the following two workloads.

**YCSB Local:** This workload represents a perfectly partitionable dataset. Each transaction reads and modifies 16 records stored on a single partition using a Zipfian distribution with varying skew factor $\theta$. Also, in the last experiment, we will explore the entire spectrum of locality from complete random record selection to complete single-partition record selection.

**YCSB Distributed:** Many real OLTP workloads are not as partitionable as **YCSB Local** on the transaction level (i.e. all records accessed by the transaction must reside in exactly one partition), but still exhibit some locality on the record level. For example, a purchase that contains one Harry Potter book is likely to contain
a few other volumes of the Harry Potter franchise, while still including any other non-related item. To model such cases, we generated a workload where each transaction reads 4 records across different partitions of the entire database uniformly, and reads and modifies 2 other records from a single partition using a Zipfian distribution.

**InstaCart:** To assess the effectiveness of our approach to deal with difficult to partition workloads, we used a real-world data set released by Instacart [41], which is an online grocery delivery service. The dataset contains over 3 million grocery orders for around 50K items from more than 200K of their customers. On average, each order contains 10 grocery products purchased in one transaction by a customer. To model a transactional workload based on the Instacart data, we used the TPC-C’s NewOrder where each transaction reads the stock values of a number of items, subtracts each one by 1, and inserts a new record in the order table. However, instead of randomly selecting items according to the TPC-C specification, we used the actual Instacart data set. Unlike the original TPC-C, this data set is actually difficult to partition due to the nature of grocery shopping, where items from different categories (e.g., dairy, produce, and meat) may be purchased together. More importantly, there is a significant skew in the number of purchases of different products. For example, 15% of transactions contain banana.

### 4.8.4 TPC-C Results

As common in all TPC-C evaluations, all tables are partitioned by warehouse ID, except for the Items table which is read-only and therefore replicated on all servers. Both Chiller and Schism produce this partitioning given the workload trace, therefore in the following experiments, we mainly focus on the two-region execution feature of Chiller, and evaluate it against the other CC schemes.

**Impact of Concurrency Level.**

In the first experiment, we measured the throughput and abort rate of Chiller, 2PL (NO_WAIT and WAIT_DIE), and OCC with increasing number of worker threads per server. Although this increase provides more CPU power to process transactions, it also increases the contention. Studying this factor is therefore of great importance since many modern in-memory databases are designed for systems with multi-core CPUs.

As Figure 4.7a shows, with only one worker thread running in each machine (i.e., no concurrent data access), NO_WAIT and WAIT_DIE perform similarly, and has 10% higher throughput than Chiller. This is accounted by the two-region execution overhead in Chiller. However, as we increase the number of worker threads, the likelihood that transactions conflict with each other increases, negatively impacting the scalability of 2PL and OCC. Chiller, on the other hand, automatically minimizes the lock duration for the two contention points in TPC-C (warehouse and district records) and thus, scales much better with increasing worker threads. With 10 threads, the throughput of Chiller is 2× and 3× higher than that of NO_WAIT and WAIT_DIE, respectively.

Figure 4.7b shows the corresponding abort rates (averaged over all threads). With more than 4 threads, OCC’s abort rate is even higher than NO_WAIT, which is attributed to the fact that many transactions are executed to the validation phase and then are forced to abort. Compared to the other techniques, the abort
rate of Chiller increases much more slowly as the level of concurrency per server increases.

This experiment shows the inherent scalability issue with traditional CC schemes when deployed on multi-core systems, and how Chiller manages to significantly alleviate it.

Impact of Distributed Transactions.

For this experiment, we restricted the transactions to \texttt{NewOrder} and \texttt{Payment}, each making up 50\% of the mix (In the standard TPC-C workload, these two transactions are the only ones which can be multi-partition). For \texttt{Payment}, we varied the probability that the paying customer is located at a remote warehouse, and for \texttt{NewOrder} we varied the probability that at least one of the purchased items is located in a remote partition.

Figure 4.8 shows the total throughput with a varying fraction of distributed transactions. As the percentage of distributed transactions increases, the already existing conflicts become more pronounced due to the prolonged duration of transactions, since a higher ratio of transactions must wait for network roundtrips to access records on remote partitions. This observation clearly shows why having good partitioning layout is a necessity for good performance in traditional CC protocols, and why existing partitioning techniques aim to minimize the percentage of distributed transactions.
Also, compared to the traditional concurrency protocols, Chiller degrades the least when the fraction of distributed transactions increases. More specifically, the performance of Chiller drops only by 26%, while NO\_WAIT and WAIT\_DIE both observe close to 50% drop in throughput, and the throughput of OCC has the largest decrease, which is about 73%. This is because the execution threads for a partition always have useful work to do; when a transaction is waiting for remote data, the next transaction can be processed. Since in Chiller, conflicts are handled sequentially in the inner region, concurrent transactions have a much smaller likelihood of conflicting with each other. Therefore, an increase in the percentage of distributed transactions only means higher latency per transaction, and not much increased contention, therefore has much less impact on the throughput. This highlights our claim that minimizing the number of multi-partition transactions should not be the primary goal in the next generation of OLTP systems that leverage fast networks, but rather that optimizing for contention should be.

### 4.8.5 YCSB Results

#### Single-Partition Transactions.

We begin by examining the impact of contention on single-partition transactions. We use the YCSB local workload and vary the skew level $\theta$ from 0.5 (low skew) to 0.99 (high skew, the default in the original YCSB). The aggregated throughput and the average abort rate are shown in Figures 4.9a and 4.9b. For this workload, both Chiller and Schism can produce the same split as the ground truth mapping function we used to generate transactions. As explained before, under traditional CC schemes, distributed transactions significantly intensify any contention in the workload, which explains the steep increase in the abort rate of the hash partitioning baseline in Figure 4.9b.

As the contention increases, all traditional CC schemes face high abort rate, reaching more than 50% with $\theta = 0.85$. Chiller, on the other hand, is able to minimize the contention and hence reduce the abort rate. When the skew is high ($\theta = 0.99$), the throughput of Chiller is more than 85% higher than the second best baseline, NO\_WAIT, while its abort rate is about half of WAIT\_DIE.

This experiment shows that even for a workload with only single-partition transactions which is considered the sweet spot of the traditional partitioning and CC techniques, high contention can result in a major
Scalability of Distributed Transactions.

We next compare the scalability of different schemes in the presence of distributed transactions. For this purpose, we used YCSB distributed workload, in which each transaction reads 4 records from the entire database, and modifies 2 skewed records from the same partition. Schism gives up fine-grained record-level partitioning and chooses simple hash partitioning, because in this workload, co-locating the hot records is not advantageous to simple hashing in terms of minimizing distributed transactions. Thereby, we also show the results for the original min-cut partitioning produced by Metis, which minimizes the number of cross-partition record accesses.

Figure 4.10 shows the throughput of the different protocols as the number of partitions (and servers) increases. To avoid cluttering the graph, we only show the performance of the best CC scheme for Schism, which is \texttt{NO\_WAIT}. Also, similar to before, we used the replication factor of two. However, for the case of one and two machines, the replication factor is zero and one, respectively, since it is not reasonable to replicate a record on the same machine it is hosted.

At first, all schemes drop in throughput from one to two partitions due to the introduced replication and distributed transactions. Surprisingly, as the number of partitions increases, all the CC schemes which use Metis partitioning outperforms the one which uses Schism, even though that almost all transactions are distributed in both cases. This is because in Metis, the contended records are co-located, and this drastically reduces the negative impact of aborting transactions in our system. More specifically, a transaction which fails to get one of the two contended locks would release the other one immediately, whereas in the partitioning produced by Schism, these two records are likely to be placed in different partitions, and releasing the locks for an aborted transaction may take one network roundtrip, further intensifying the contention problem. \texttt{WAIT\_DIE} performs better than \texttt{NO\_WAIT} since its waiting mechanism is able to overcome the lock thrashing issue in \texttt{NO\_WAIT}, though we note that we also observed this phenomenon for \texttt{WAIT\_DIE} for workloads where transactions access a higher number of hot records (e.g. see Figure 4.9b). Compared to all the other
schemes, Chiller scales much better since not only its partitioning co-locates the contended records together, but also its two-region execution model is able to access those records in inner regions of transactions and therefore significantly reduces the contention. On 7 machines, the performance of Chiller is $4 \times$ and $40 \times$ of NO\_WAIT on Metis and Schism partitions, respectively, while close to $2 \times$ of the second best baseline, WAIT\_DIE.

**Lookup Table Size.**

In this experiment, we investigate the performance of our proposed scheme in situations where a full-coverage lookup table cannot be either obtained or stored, as discussed in Section 4.5.5. This can mainly happen when the number of database records is too large to store a complete lookup table on each machine.

We used YCSB local, and fixed the skew parameter $\theta$ to 0.8 to represent a moderately skewed workload. Since all transactions in this workload are single-partition with respect to the ground truth mapping, both Schism and Chiller are able to find the optimal partitioning which makes all transactions single-partition, but this requires to cover the entire database in their resulting lookup tables. To measure the impact of lookup table coverage, we vary the percentage of the records which are partitioned according to the optimization goal of each partitioning algorithm. We used hash partitioning for the remaining records which, as a result, do not take up any lookup table entry, but result in a significant increase in the number of multi-partition transactions.

The results are shown in Figure 4.11. When all records are hash partitioned (and so the size of the lookup table is 0), Chiller and all the other schemes achieve similar throughput. As the lookup table increases in size, Chiller starts to diverge from the other schemes. With a coverage of only 20%, Chiller achieves close to half of its peak throughput, whereas NO\_WAIT and OCC achieve less than 0.1 of their peak throughput. In contrast, NO\_WAIT relies on a 80% coverage to achieve half of its max throughput. The wide gap between Chiller and the other protocols is due to the way that Chiller handles contention. Placing the contended records (which are often a small fraction of the entire database) in the right partitions and handling them in inner regions are enough to remove most of the contention in the workload. The rest of the records can be randomly assigned to partitions without increasing contention.
This experiment supports our claim in Section 4.5.5 that, compared to partitioning schemes aiming to minimize distributed transactions, Chiller requires a much smaller lookup table to achieve a similar throughput. In addition, while for this particular workload there exists a split of the database where each transaction accesses only a single partition, for many real workloads such partitioning does not exist. Therefore, this experiment also shows how Chiller compares against the other schemes for workloads with different degrees of partitionability.

4.8.6 Instacart Results

In our final experiment, we analyze the benefits of combining the Chiller’s partitioning scheme with the two-region execution model. We use a real-world Instacart workload (as introduced in Section 4.8.3), which is harder to partition than TPC-C and YCSB. Furthermore, we use the same replication factor of 2 as for the previous experiments.

In order to understand whether or not the two-region execution model of Chiller is beneficial for the overall performance, we compare full Chiller (Chiller) to Chiller partitioning without the two-region execution model (ChP) and Chiller partitioning using Quro* (ChP+Quro*). In contrast to ChP which does not re-order operations, ChP+Quro* re-orders operations using Quro [105], which is a recent contention-reduction technique for centralized database systems. Moreover, we compare full Chiller to two other non-Chiller baselines (Hash-partitioning and Schism-partitioning). For both ChP and ChP+Quro* as well as the non-Chiller baselines (Hash and Schism), we only show the results for a WAIT_DIE scheme as it yielded the best throughput compared to NO_WAIT and OCC for this experiment.

Figure 4.12 shows the results of this experiment for increasing cluster sizes. Compared to the Hash-partitioning baseline (black line), both ChP and ChP+Quro* (green and red lines) have significantly higher throughput. We found that this is not because the Chiller partitioning technique reduces the number of distributed transactions, but rather because contended records which are accessed together are co-located, which in turn reduces the cost of aborting transactions. More specifically, if a transaction on contented records needs to be aborted, it only takes one round-trip, leading to an overall higher throughput since the failed transaction can be restarted faster (cf. Section 4.8.5).

Furthermore, we see that ChP+Quro*, which re-orders operations to access the low contended records
first, initially increases the throughput by 20% compared to ChP but then its advantage decreases as the number of partitions increases. The reason for this is that the longer latency of multi-partition transactions offsets most of the benefits of operation re-ordering if the commit order of operations remains unchanged. In fact, with 5 partitions, Schism (yellow line) starts to outperform ChP+Quro*, even though Schism does not leverage operation re-ordering.

In contrast to these baselines, Chiller (blue line) not only re-orders operations but also splits them into an inner and outer region with different commit points, and thus can outperform all the other techniques. For example, for the largest cluster size, the throughput of Chiller is by approximately 1 million txns/sec higher than the second best baseline. This clearly shows that the contention-centric partitioning must go hand-in-hand with the two-region execution to be most effective.

4.9 Related Work

Data Partitioning: A large body of work exists for partitioning OLTP workloads with the ultimate goal of minimizing cross-partition transactions [21, 98]. Most notably, Schism [21] is an automatic partitioning and replication tool that uses a trace of the workload to model the relationship between the database records as a graph, and then applies METIS [45] to find a small cut while approximately balancing the number of records among partitions. Clay [83] builds the same workload graph as Schism, but instead takes an incremental approach to partitioning by building on the previously produced layout as opposed to recomputing it from scratch. E-store [92] balances the load in the presence of skew in tree-structured schemas by spreading the hottest records across different partitions, and then moving large blocks of cold records to the partition where their co-accessed hot record is located. Given the schema of a database, Horticulture [75] heuristically navigates its search space of table schemas to find the ideal set of attributes to partition the database. As stated earlier, all of these methods share their main objective of minimizing inter-partition transactions, which in the past have been known to be prohibitively expensive. However, in the age of new networks and much “cheaper” distributed transactions, such an objective is no longer optimal.

Transaction Decomposition: There has been also work exploring the opportunities in decomposing transactions into smaller units. Gemini [57] introduces a mixed consistency model called BlueRed in which transaction operations are divided into blue operations, which are eventually consistent with lower latency, and red operations, which are strongly consistent which require global serialization. Gemini optimizes for overall latency and requires data to be replicated at all servers, and therefore does not have the notion of distributed transactions. Chiller, on the other hand, optimizes for minimizing contention, and supports distributed transactions.

There has also been work on the theory of transaction chopping [84, 85, 112], in which the DBMS splits a transaction into smaller pieces and treats them as a sequence of independent transactions. In contrast to the idea of transaction chopping, our two-region execution not only splits a transaction into cold and hot operations, but re-orders operations based on which region they belong to. Also, we do not treat the outer region as an independent transaction and will hold the locks on its records until the end of the transaction. This allows us to our technique to abort a transaction later in the inner region. Transaction chopping techniques,
however, must adhere to rollback-safety, in which all operations with the possibility of rollback must be executed in the first piece, since subsequent pieces must never fail. This restricts the possible ways to chop the transaction.

Determinism and Contention-Reducing Execution: Another line of work aims to reduce contention through enforcing determinism to part or all of the concurrency control (CC) unit [20, 44, 95]. In Granola [20], servers exchange timestamps to serialize conflicting transactions. Calvin [95] takes a similar approach, except that it relies on a global agreement scheme to deterministically sequence the lock requests. Faleiro et al. [32, 31] propose two techniques for deterministic databases, namely lazy execution scheme and early write visibility, which aim to reduce data contention in those systems. All of these techniques and protocols require a priori knowledge of read-set and write-set.

There has also been a large body of work on optimizing and extending traditional CC schemes to make them more apt for in-memory databases. MOCC [101] targets thousand-core systems with deep memory hierarchies and proposes a new concurrency control which mixes OCC with selective pessimistic read locks on contended records to reduce clobbered reads in highly contended workloads. Recent work on optimistic CC leverages re-ordering operations inside a batch of transactions to reduce contention both at the storage layer and validation phase [24]. While Chiller also takes advantage of operation re-ordering, it does so at an intra-transaction level without relying on transaction batching. MV3C [22] introduces the notion of repairing transactions in MVCC by re-executing a subset of a failed transaction logic instead of running it from scratch. Most related to Chiller is Quro [105], which also re-orders operations inside transactions in a centralized DBMS with 2PL to reduce lock duration of contended data. However, unlike Chiller, the granularity of contention for Quro is tables, and not records. Furthermore, almost all these works deal with single-node DBMSs and do not have the notion of distributed transactions, 2PC or asynchronous replication on remote machines, and hence finding a good partitioning scheme is not within their scopes.

Transactions over Fast Networks: Our work on chiller continues the growing focus on distributed transaction processing on new RDMA-enabled networks [12]. The increasing adoption of these networks by key-value stores [66, 42, 55] and DBMSs [26, 108, 43, 103] is due to their much lower overhead for message processing using RDMA features, low latency, and high bandwidth. These systems are positioned in different points of the spectrum of RDMA. For example, FaSST [43] uses the unreliable datagram connections to build an optimized RPC layer, and FaRM [26] and NAM-DB [108] leverage the RDMA feature to directly read or write data to a remote partition. Though different in their design choices, scalability in the face of cross-partition transactions is a common promise of these systems, provided that the workload itself does not impose contention. Therefore, Chiller’s two-region execution and its contention-centric partition are specifically suitable for this class of distributed data stores.

4.10 Main Takeaways

In this chapter, we presented Chiller, a distributed transaction processing and data partitioning scheme that aims to minimize contention. Chiller is designed for fast RDMA-enabled networks where the cost of distributed transactions is already low, and the system’s scalability depends on the absence of contention in the
workload. Chiller partitions the data such that the hot records which are likely to be accessed together are placed on the same partition. Using a novel two-region processing approach, it then executes the *hot* part of a transaction separately from the *cold* part. Our experiments show that Chiller can significantly outperform existing approaches under workloads with varying degrees of contention.
Chapter 5

High Availability with RDMA Networks

In the previous chapters, our focus was on designing a distributed OLTP system which is scalable and deliver high performance. However, high performance and scalability only when all units of the system are functional and running are not the only requirements of real-world applications. Database systems, like any other computer system, experience failures for a wide range of reasons. If a database system does not show proper resiliency to failures, then it will be of very little use to today’s applications that are expected to deliver “always-on” service.

Highly available database systems use replication to ensure that even in the face of failures, the system remains operational with close to zero downtime.

In this chapter, we will review the existing solutions for high availability in OLTP systems, and see that they are not suitable to be used in distributed databases on next-generation networks as they were designed in a time when network was the dominant performance bottleneck.

We then propose a novel replication scheme called “Active-memory”, which efficiently leverages RDMA to brings high availability while maintaining high performance and correctness in the presence of failures.

The content of this chapter comes from the published work in VLDB 2018 titled “Rethinking Database High Availability with RDMA Networks” [111].

5.1 Motivation

A key requirement of essentially any transactional database system is high availability. A single machine failure should neither render the database service unavailable nor should it cause any data loss. High availability is typically achieved through distributed data replication, where each database record resides in a primary replica as well as one or multiple backup replicas. Updates to the primary copy propagate to all the backup copies synchronously such that any failed primary server can be replaced by a backup server.

The conventional wisdom of distributed system design is that the network is a severe performance bottleneck. Messaging over a conventional 10-Gigabit Ethernet within the same data center, for example, delivers 2–3 orders of magnitude higher latency and lower bandwidth compared to accessing the local main memory.
of a server [12]. Two dominant high availability approaches, active-passive and active-active, both adopt the optimization goal of minimizing network overhead.

With the rise of the next-generation networks, however, conventional high availability protocol designs are not appropriate anymore, especially in a setting of Local Area Network (LAN). The latest remote direct memory access (RDMA) based networks, for example, achieve a bandwidth similar to that of main memory, while having only a factor of $10 \times$ higher latency. Our investigation of both active-passive and active-active schemes demonstrates that with a modern RDMA network, the performance bottleneck has shifted from the network to CPU’s computation overhead. Therefore, the conventional network-optimized schemes are not the best fit anymore. This calls for a new protocol design to fully unleash the potential of RDMA networks.

To address this problem, we propose **Active-Memory Replication**, a new high availability protocol designed specifically for the next-generation RDMA networks in the LAN setting. The optimization goal in Active-Memory is to minimize the CPU overhead of performing data replication rather than minimizing network traffic. The core idea of Active-Memory is to use the one-sided feature of RDMA to directly update the records on remote backup servers without involving the remote CPUs. One key challenge in such design is to achieve fault tolerance when the CPUs on backup servers do not participate in the replication protocol. To address this problem, we designed a novel undo-logging based replication protocol where all the logic is performed unilaterally by the primary server. Each transaction goes through two serial phases: (1) undo logging and in-place updates and (2) log space reclamation, where each update is performed by a separate RDMA write. We have proved that the protocol has correct behavior under different failure scenarios.

We compared Active-Memory with both active-passive (i.e., log-shipping [65, 46]) and active-active (i.e., H-Store/VoltDB [44, 91] and Calvin [95]) schemes on various workloads and system configurations. Evaluation shows that Active-Memory is up to a factor of $2 \times$ faster than the second-best baseline protocol that we evaluated over RDMA-based networks.

### 5.2 Contributions and Chapter Organization

In this chapter, we make the following contributions:

- We revisit the conventional high availability protocols on the next-generation networks and demonstrate that optimizing for network is no longer the most appropriate design goal.

- We propose Active-Memory, a new replication protocol for RDMA-enabled high bandwidth networks, which is equipped with a novel undo-log based fault tolerance protocol that is both correct and fast.

- We perform extensive evaluation of Active-Memory over conventional protocols and show it can perform $2 \times$ faster than the second-best protocol that we evaluated.

The organization of this chapter is as follows: Section 5.3 describes the background of the conventional high availability protocols. Section 5.4 analyzes why conventional wisdom is no longer appropriate for modern RDMA-based networks. Section 5.5 describes the Active-Memory replication protocol in detail, and Section 5.6 demonstrates that the protocol is fault tolerant. In Section 5.7, we present the results of our
5.3 High Availability in Existing OLTP Systems

High availability is typically achieved through replication: every record of the database gets replicated to one or more machines. To survive \( k \) machine failures, the system must make sure that for each transaction, its effects are replicated on at least \( k + 1 \) machines. This is known as the \( k \)-safety rule. For example, for \( k = 1 \), each record is stored on two different machines, so that a failure of either of them does not disrupt the continuous operation of the system.

According to the widely-cited taxonomy of Gray el al. [35], replication protocols can be either *eager* or *lazy* (which pertains to when the updates are propagated to the replicas), and be either *primary copy* or *update anywhere* (which concerns where data-modifying transactions must be issued). Lazy replication is often used in data stores where strong consistency is not crucial, and the possibility of data loss can be accepted in exchange for possibly better throughput, such as in Amazon Dynamo [23] and Facebook Cassandra [52]. However, it has been acknowledged that abandoning consistency by lazy replication introduces complexity, overheads, and costs that offset its benefits for many applications [63].

Here, we target eager, or **strongly consistent** replication in shared-nothing architectures, which are suitable for databases which aim to offer high availability without compromising consistency and correctness. These databases make sure that all of the updates of a transaction reach the backup replicas before the transaction is considered committed [35]. Strong consistency makes it easy to handle machine failures, since all of the copies of a record are identical at all times. A failover can be as simple as informing every surviving server about the new change in the cluster and allowing them to reach an agreement on the current configuration. Reconfiguration of the cluster can be done by a cluster configuration manager such as Zookeeper [39].

Strongly consistent replication is implemented in databases using various schemes, which can be broadly categorized into two groups, namely *active-passive* and *active-active*, with each having their own many variations and flavours. Here, we abstract away from their subtleties and provide a simple and general overview of how each one delivers strongly consistent replication, and discuss their associated costs and limitations.
5.3.1 Active-Passive Replication

Active-passive replication is one of the most commonly used replication techniques. Each database partition consists of one active copy (known as the primary replica) which handles transactions and makes changes to the data, and one or more backup replicas, which keep their copies in sync with the primary replica. When a machine \( p \) fails, the system maintains its high availability by promoting one of \( p \)'s backup nodes as the new primary and fails over to that node. There are many different implementations of active-passive replication both in academic projects [26, 43, 47, 100] and in commercial databases [16] (such as Postgres Replication [46], Oracle TimesTen [51], and Microsoft SQL Server Always On [65]). Active-passive schemes are often implemented through log shipping where the primary executes the transaction, then ships its log to all its backup replicas. The backup replicas replay the log so that the new changes are reflected in their copy.

Figure 5.1 illustrates how log shipping is used in an active-passive replication scheme. Here, we assume that the database is split into two partitions (\( P_1 \) and \( P_2 \)), with each partition having a backup copy (\( B_1 \) is backup for \( P_1 \), and \( B_2 \) is backup for \( P_2 \)). In practice, \( P_1 \) and \( B_2 \) may be co-located on the same machine, while \( P_2 \) and \( B_1 \) may reside on a second machine. \( T_1 \) (colored in blue) is a single-partition transaction which touches data only on \( P_1 \). Therefore, \( P_1 \) executes this transaction, and before committing, it sends the change log to all its backup. Upon receiving all the acks, \( P_1 \) can commit. Transactions that span multiple partitions, such as \( T_2 \) (colored in orange), follow the same replication protocol, except that an agreement protocol such as 2PC is also needed to ensure consistency.

Wide adoption of active-passive replication is due to its simplicity and generality. However, two factors work against it. First, the log message contains each data record that the transaction has updated and is therefore may be big in size [79]. This becomes a serious bottleneck for conventional networks due to their limited bandwidth. Second, the communication between primaries and replicas not only imposes long delays to the critical path of transactions, but perhaps more importantly, it consumes processing power of machines for replaying logs and exchanging messages.
5.3.2 Active-Active Replication

The second group of eager replication techniques is update everywhere, or active-active protocols [35]. These systems allow any replica to accept a transaction and then broadcast the changes to all the other replicas. Synchronizing updates between replicas requires much more coordination than active-passive, due to possible conflicts between each replica’s transactions with the others. Main modern active-active databases solve this issue by removing coordination and replacing that with determinism of execution order among replicas [44, 90, 91, 95].

Deterministic active-active replication seeks to reduce the network communication needed to ship logs and coordinating with other replicas in an active-passive scheme. Specifically, the transactions are grouped into batches where each batch is executed by all replicas of the database in the same deterministic order, such that all the replicas end up with identical states once the batch is executed. Replicas in an active-active database only coordinate to determine the batches, but do not coordinate during the execution of transactions, which significantly reduces the network traffic between replicas. Two prominent examples of databases which use active-active replication are H-store [44] and its commercial successor VoltDB [91] and Calvin [95].

H-Store's replication protocol is illustrated in Figure 5.2. In H-Store, all transactions have to be registered in advance as stored procedures. The system is optimized to execute each transaction from the beginning to completion with minimum overhead. In H-Store, transactions do not get record locks, and instead only lock the partition they need. Transactions are executed in each partition sequentially, without getting pre-empted by other concurrent transactions. For a single-partition transaction such as $T_1$, the primary replicates the ID of the invoked stored procedure along with its parameters to all its backup replicas. All replicas, including the primary, start executing the transaction code in parallel. Unlike in log shipping, here the replicas do not need to coordinate, as they execute the same sequence of transactions and make deterministic decisions (commit or abort) for each transaction. For multi-partition transactions, one of the primaries acts as the transaction coordinator and sends the stored procedure and its parameters to the other participating partitions. At each partition, an exclusive lock is acquired, so that no other transaction is allowed to be executed on that partition. Each partition sends the stored procedure to all its backup replicas so they run the same transaction and build their write-set. Finally, the coordinator initiates a 2PC to ensure that all the other primaries are able to commit. H-Store performs extremely well if the workload consists of mostly single-partition transactions. However, its
performance quickly degrades in the presence of multi-partition transactions, since all participating partitions are blocked for the entire duration of such transactions.

Calvin [95] is another active-active system that takes a different approach than H-Store to enforce determinism in execution and replication (Figure 5.3). All transactions first arrive at the sequencer which orders all the incoming transactions in one single serial history. The inputs of the transactions are logged and replicated to all the replicas. Then, a single lock manager thread in each partition scans the serial history generated by the sequencer and acquires all the locks for each transaction, if possible. If the lock is already held, the transaction has to be queued for that lock. Therefore, Calvin requires that the read-set and write-set of transactions are known upfront, so that the lock manager would know what locks to get before even executing the transaction (This assumption may be too strict for a large category of workloads, where the set of records that a transaction is read or modified is known throughout executing the transaction). Those transactions which all their locks are acquired by the lock manager are then executed by the worker threads in each replica without any coordination between replicas. For multi-partition transactions, the participating partitions communicate their results to each other in a push-based manner (instead of pull-based, which is common in the other execution schemes).

Compared to Calvin with its sequencing and lock scheduling needs, H-store has a much lower overhead for single-partitioned transactions. Calvin, on the other hand, benefits from its global sequencing for multi-partition transactions.

## 5.4 The Case for Replication with RDMA Networks

With the fast advancement of network technologies, conventional log-shipping and active-active schemes are no longer the best fits. In this section, we revisit the design trade-offs that conventional schemes made and demonstrate why the next-generation networks call for a new design of high availability protocol in Section 5.4.1. We then provide some background on RDMA in Section 5.4.2.
5.4.1 Bottleneck Analysis

The replication schemes described in the previous section were designed in a time that network communication was the obvious bottleneck in a distributed main-memory data store by a large margin. Reducing the need for accessing the network was therefore a common principle in designing efficient algorithms. Both classes of techniques approach this design principle by exchanging high network demand with more processing redundancy, each to a different degree. This idea is illustrated in Figure 5.4a. In log shipping, the logs have to be replayed at each replica, which may not be much cheaper than redoing the transaction itself for some workloads. Active-active techniques reduce the need for network communication even further and thus improve performance when the network is the bottleneck but impose even more redundancy for computation.

In these networks, communication during replication is considered expensive mainly due to three factors. (1) **Limited bandwidth** of these networks would be easily saturated and become the bottleneck. (2) **The message processing overhead** by the operating system proved to be substantial [12], especially in the context of many OLTP workloads which contain simple transactions that read and modify only a few records. (3) **High latency** of network communication increases the transaction latency, contributing to contention and therefore impacts throughput.

With the emergence of the next-generation of RDMA-enabled networks, such as InfiniBand, these assumptions need to be re-evaluated. (1) **Network bandwidth** has increased significantly, and its increase rate does not seem to be slowing down [40]. For example, a Mellanox ConnectX-4 EDR card offers $100 \times$ bandwidth of a typical 1Gb/sec Ethernet found in many public cluster offerings (including our own private cluster). (2) The **RDMA feature** open new possibilities to design algorithms that eschew not only the message processing overhead of existing methods, but also the actual processing redundancy attached to replication (i.e. replaying logs in the log shipping scheme or executing transactions multiple times in deterministic schemes). (3) RDMA can achieve much **lower latency** compared to Ethernet networks, owing to its zero copy transfer.
and CPU bypass features. While the latency of RDMA is still an order of magnitude higher than main memory latency, it can be significantly masked by efficiently leveraging parallelism and concurrency in software [43].

Consequently, the modern RDMA-enabled networks have been shifting the bottleneck in the direction of CPU rather than the network, as depicted in Figure 5.4b; creating an identical memory copy can be done with very little CPU overhead. In this new environment, the old replication techniques are not optimal anymore, as their inherent processing redundancy underutilizes the processing power of the cluster, without efficiently leveraging modern networks. This calls for a new replication protocol which is designed specifically for the new generation of networks.

Note that in a shared-nothing environment, where nodes do not share memory or storage, nodes either have to replicate their new states to the replicas after performing one or a batch of transactions (resulting in relatively higher bandwidth requirement, as in the case with log shipping and Active-Memory) or they must enforce the same transaction schedule at all replicas (resulting in higher processing redundancy). This explains that shaded area in Figure 5.4 is very unlikely to achieve in a shared-nothing system.

5.4.2 Background for RDMA

The RDMA feature allows a machine to directly access the main memory of another machine without the involvement of the operating systems of either side, enabling zero-copy data transfer. RDMA recent popularity in the database community is mostly due to the fact that the network technologies which support RDMA have become cost competitive with Ethernet [12]. InfiniBand, RoCE, and iWarp are currently the main implementations of RDMA networks. Bypassing the OS and entirely offloading the networking protocol onto the network cards allow RDMA to have high throughput, low latency and low CPU utilizations. For example, the latest Mellanox ConnectX-6 RNICs can deliver 200Gb per second data transfer, have latency of below 1µs, and are able to handle up to 200 million messages per second.

The RDMA interface provides two operations types: one-sided (Read, Write, and atomic operations) and two-sided (Send and Receive). One-sided operations bypass the remote CPU and provide user-level memory access interface, where the remote memory is directly read from or written to. Two-sided operations, on the other hand, provide a message-passing interface for two user-level processes to exchange RPC messages. Unlike one-sided operations, two-sided communication involves the CPUs of both sides.

Two processes communicate to each other through queue pairs, which have different modes. In Reliable Connected (RC) queue pairs, packets are delivered in order and without any loss. These two properties are the key to our replication and fault-tolerance protocol, as we will see in Sections 5.5 and 5.6.

5.5 Active-Memory: RDMA-based Replication

In this section, we start with an overview of Active-Memory Replication, our RDMA-based high availability solution, and then present the replication algorithm in detail.
Figure 5.5: The structure of the log buffer and the log entries. Each node has a private log buffer on every other node.

Figure 5.6: The linked list of RDMA messages sent to an active node

5.5.1 Concurrency Control and Replication Assumptions

We present Active-Memory in the context of a partitioned and distributed shared-nothing database. Similar to many other data stores (such as FaRM [26], RAMCloud [70, 72], and FaSST [43]), we assume striped master partitions. Each data record has one primary copy on the master node and multiple replica copies on each of the backup nodes. Each node in the cluster is the master for a fraction of data records and the backup for some other records. A transaction accesses only the primary copy of a record. The backup copies are therefore only accessed and/or modified during the replication protocol. This is necessary to avoid having transactions reading uncommitted data on the backups.

While Active-Memory is orthogonal to the employed consistency protocol, our focus in this work is on two-phase locking (2PL) with a NO_WAIT policy [11]. However, we do require that every data structure change is made atomic within the boundaries of a transaction. Without this requirement, a direct memory copy with RDMA could replicate uncommitted changes. Our implementation guarantees this requirement using exclusive latches on the shared data structures (e.g., buckets in a hash-table). However, many alternative designs exist. Finally, we assume that each node is equipped with Non-Volatile Memory (NVM) similar to other recent high-performance OLTP systems [25, 43, 108].

5.5.2 Overview

Active-Memory adopts a primary-backup replication scheme that is specifically designed for RDMA-enabled networks. Instead of shipping logs to the backup nodes, the coordinator directly writes each of the transaction’s changes to the main memory of the backup nodes using the one-sided write operations that RDMA supports. Therefore, the CPUs at the backup nodes are no longer involved in the data replication logic and
therefore can be spent on processing new transactions. Specifically, for each transaction, the replication protocol involves two serial phases: (1) UNDO log transfer and in-place updates and (2) log space reclamation. Overall, Active-Memory has the following salient features.

- **Strong Consistency**: Following our discussion in Section 5.3, Active-Memory provides strong consistency replication. The data in the backup nodes always reflects the changes up to the last committed transaction and do not lag behind. Fresh backups enable fast and straight-forward fail-over.

- **Zero Processing Redundancy**: In contrast to log shipping and active-active schemes, Active-Memory entirely eliminates the processing redundancy in the replication protocol. The transaction logic is executed only once. The backup nodes neither execute the transaction code nor replay the logs. Their CPU time is used to process new transactions.

- **Zero Overhead of Message Handling**: By fully relying on one-sided RDMA operations, Active-Memory is able to remove much of the overhead caused by sending and receiving RPC messages, including the TCP/IP overhead, and message dispatching mechanism inside each database node.

- **Simple and Fast Fault-Tolerance**: No matter how well a replication protocol works in the normal execution mode, recovery from failures must be reliable and consistent, and preferably fast. As we will see in Section 5.6, our protocol takes advantage of RDMA to provide an intuitive and fast mechanism for fault-tolerance.

The reduction of CPU time in Active-Memory comes at the cost of increased network traffic. As we will demonstrate in Section 5.7, however, this is not a problem with the new-generation RDMA networks due to their high bandwidth. By efficiently leveraging RDMA, Active-Memory can significantly outperform both log shipping and active-active schemes.

### 5.5.3 Design Challenges

Although the key idea behind Active-Memory is simple, what makes the design challenging is supporting fault tolerance and non-blocking fast recovery. For correctness, the coordinator must make sure that either all its changes are replicated on all the backup nodes, or none of them has been replicated. Different from a conventional log-shipping protocol where the primary and backups coordinate to achieve this goal, the backup nodes in Active-Memory do not actively participate in the replication protocol. Therefore, the coordinator has to unilaterally guarantee fault tolerance properties, which makes it more challenging.

To achieve this goal, Active-Memory relies on an undo logging mechanism, rather than traditional redo logging. The coordinator writes undo log entries to a backup node before directly updating any memory state on that node. Section 5.5.4 describes how undo logging occurs in Active-Memory in more details.

Another challenge in Active-Memory is non-blocking recovery, which requires the system to quickly recover to a consistent state when one or multiple coordinators or backup nodes fail. Active-Memory ensure that there is always enough information available to at least one of the surviving nodes so that it is able to recover the correct state of the modified data records and the ongoing transactions. Section 5.5.5 describes more details on this topic.
5.5.4 Undo Log Buffers

As stated before, Active-Memory uses an RDMA-based undo logging mechanism to ensure failure atomicity. Each server node has a pre-allocated RDMA buffer hosted on every other machine, which contains a certain number of fixed-size log entries, and implements a circular buffer primitive, as shown in Figure 5.5. Each log buffer can only be modified by a single remote server node. Thereby, there is no concurrent updates to a log buffer. This significantly simplifies the replication algorithm of Active-Memory (Section 5.5.5) and its fault-tolerance mechanism (Section 5.6).

Each node maintains the list of its available undo log entries on every other server machine. That is, a server knows what the head and tail pointers are for each of its remote log buffers. A log entry is placed in a remote buffer by issuing an RDMA Write operation. Implementing a circular buffer primitive means that the log entries which are not needed anymore can be re-used. In other words, the log buffer merely needs to have as many entries as the number of open (i.e. live, uncommitted) transactions initiated by each node at a time (for example in our system, this number is at most 20).

The layout of a log entry is illustrated on the right side of Figure 5.5. Each entry stores the pre-update contents of the attributes of the records modified by the transaction. For example, a transaction that modifies two attributes of three records will contain 6 changed attributes ($\text{ChangesCnt} = 6$), and for each of them, the entry will contain its $\text{HostID}$, the attribute’s memory offset in that host, its size in bytes ($\text{Len}$), and its old content in $\text{Payload}$. Storing the values of changed attributes as opposed to the entire record content minimizes the required log size, and therefore minimizes the number of log entries to be sent per transaction. Each entry stores a locally unique identifier $\text{LogID}$. If a log exceeds the entry’s fixed size, a follow-up log entry is sent with the same $\text{LogID}$. An entry with $\text{IsLast}$ set to $\text{false}$ signifies that a follow-up entry should be expected. The coordinator sets the same log ID at the end of each entry as well ($\text{LogID}_{\text{Check}}$). A log entry is considered correct and usable only if $\text{LogID}$ and $\text{LogID}_{\text{Check}}$ have the same value, otherwise it is considered corrupt. This is because at the receiver side, most NICs guarantee that RDMA Writes are performed in increasing address order [25] (i.e. writing of $\text{LogID}_{\text{Check}}$ does not happen before writing of $\text{LogID}$). Therefore, this mechanism makes each log entry self-verifying. Finally, $\text{IsCommitted}$ indicates the commit status of a transaction.

For some workloads, it may be possible that some few transactions have such large write-sets that they would not fit in the undo log buffer. Such an uncommon case can be supported by relying on the RPC-based log shipping; the coordinator sends an RPC message to each replica, with all its log entries concatenated to each other. The replicas apply the updates and send back acks to the coordinator. In general, the system must make sure that such cases remain rare for a given workload, and if not, it should increase the size the log buffers to better accommodate the write set of most transactions.

5.5.5 Replication Algorithm

We now describe Active-Memory replication protocol. The coordinator initiates this protocol once it has built its read-set and write-set, which we refer to as the execution phase. Active-Memory assumes that for each transaction, its coordinator maintains a local write-set ($\text{WS}$), which contains a list of unique record keys and
their new values which the transaction intends to write. At the end of its execution, the transaction is ready to commit and apply the updates in $WS$ to the database. This is when the coordinator initiates the replication protocol which consists of two steps (Figure 5.7).

### Step 1: Undo Log and In-place Update

The goal of the first step is to 1) replicate the undo logs, and 2) directly modify the records in the write-set in-place. These two sub-steps must be performed for all the involved partitions in the transaction and their replicas, henceforth referred to as active nodes. The algorithm must make sure that for each active node, in-place data update does not happen without undo logging.

Listing 5.1 shows the algorithm for this step. In summary, the coordinator scans its write-set and forms a linked list of RDMA operations for each active node. The first message of this linked list is the undo log RDMA Write operation, and the rest are the RDMA Writes for in-place attribute updates for records hosted on that partition, as shown in Figure 5.6. Because of the in-order message delivery guarantee of reliable connected queue pairs, the log message is received by the destination NIC before the in-place update messages. Such a guaranteed delivery order is crucial in our fault tolerance protocol, as we shall see later in Section 5.6.

As shown in Listing 5.1, the coordinator performs this process for each active node $p$ in the transaction. It first retrieves and updates the tail pointer of the log buffer hosted on node $p$. Then, it initializes a log entry which is to be replicated later on $p$ and its replicas (the $IsCommitted$ field is set to false, indicating that the transaction has not yet committed). It retrieves the records in the write-set which are hosted on $p$ (line 6), and adds their changed attributes to the log entry (lines 8 to 13). It then initializes a list of RDMA messages by adding the undo log entry (lines 16 to 18) and the data updates (lines 20 and 25). The resulting list is similar to Figure 5.6. This linked list is then issued to $p$ and all its replicas (lines 27 to 30). Posting a linked list of RDMA operations in one call, as opposed to issuing them separately, allows the low-level driver to perform
for (p in active_nodes) {
    // update the tail pointer
    tail = undoBuffers[p].tail++;
    log = init_log_entry();
    // add the changes to the log msg
    recs = get_records_in_ws_in_partition(p);
    i = 0;
    for (r in recs) {
        for (attr in r.changedAttrs) {
            log.changes[i] = init_change(attr);
            i++;
        }
    }
    // prepare RDMA msgs
    msgs = linkedList();
    ml = prep_log_RDMA(log);
    // add the log entry as the *first* msg
    msgs.append(ml);
    // add all updates to the msgs list
    for (r in recs) {
        for (attr in r.changedAttrs) {
            md = prep_data_RDMA(attr);
            msgs.append(md);
        }
    }
    // send RDMA msg list to p and its replicas
    issue_RDMA(p, msgs);
    for (rep in replicas[p]) {
        issue_RDMA(rep, msgs);
    }
}

Listing 5.1: Step 1 of the replication protocol, i.e. undo logging and in-place data modification.

optimizations that result in less CPU usage on the sender side, and therefore improves performance [61, 8].

For the sake of brevity, here we assumed that all changed attributes can fit into one log message. Multi-
entry log messages are handled in the same manner with one difference. IsLast in all the log entries except
for the last one will be set to false.

Once the log and change messages are sent, the transaction has to wait until their acks are received. These
acks are indicators that the transaction log and its updates are replicated on all active nodes, so the coordinator
proceeds to the second step.

Step 2: Commit the Log Entries

The goal of this step is to ensure that the commit decision will survive $f - 1$ failures in a replica set of
size $f$ before reporting to the user (the $k$-safety property). To accomplish that, the coordinator first sets
IsCommitted attribute of the log entry at each active node to true using an RDMA Write, as shown in
Table 5.1: Different scenarios for the coordinator failure. Each row describes a scenario and how transaction state and records are recovered.

<table>
<thead>
<tr>
<th>Time of Coordinator Failure</th>
<th>Coordinator Status</th>
<th>Transaction Status in $T_P$</th>
<th>Transaction Recovery</th>
<th>Data Recovery</th>
<th>Case ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before step 1</td>
<td>Uncommitted</td>
<td>No log</td>
<td>Abort</td>
<td>Data is untouched.</td>
<td>1</td>
</tr>
<tr>
<td>During step 1</td>
<td>Uncommitted</td>
<td>No log</td>
<td>Abort</td>
<td>Data is untouched.</td>
<td>2a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Corrupt log</td>
<td>Abort</td>
<td>Data is untouched.</td>
<td>2b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logged</td>
<td>Abort</td>
<td>Data may be untouched, corrupt, or updated. Recover from log.</td>
<td>2c</td>
</tr>
<tr>
<td>After step 1 (received all acks) and before step 2</td>
<td>Uncommitted</td>
<td>Logged</td>
<td>Abort</td>
<td>Data is updated. Recover from log.</td>
<td>3</td>
</tr>
<tr>
<td>During step 2</td>
<td>Uncommitted</td>
<td>Logged</td>
<td>Abort</td>
<td>Data is updated. Recover from log.</td>
<td>4a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit-ready</td>
<td>Requires consensus: If all agree, commit. Otherwise, abort.</td>
<td>Data is updated. If the consensus is abort, recover from log. Otherwise, do nothing.</td>
<td>4b</td>
</tr>
<tr>
<td>After step 2 (received all acks)</td>
<td>Uncommitted</td>
<td>Committed</td>
<td>Commit-ready</td>
<td>Since all are committed, the consensus will be commit.</td>
<td>5a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit-ready</td>
<td>Commit-ready</td>
<td>Since all are committed, the consensus will be commit.</td>
<td>5b</td>
</tr>
</tbody>
</table>

Figure 5.7. Once the coordinator NIC finishes sending these RDMA Write messages, it locally increments the $head$ pointer of the corresponding undo log buffer for each involved node, indicating that this entry can be re-used by future transactions. It also logs to its local NVM, releases its locks (if the transaction is multi-partition, informing every participant to release their locks) and returns to the user.

In summary, Active-Memory takes advantage of the in-order message delivery of reliable connected queue pairs in RDMA. Using this connection type, it is guaranteed that messages are delivered to the receiver’s NIC in the same order that they are transmitted by the sender (even though that they may be applied to the receiver’s main memory out of order [38], but this does not pose a problem for our protocol, as will be discussed in Section 5.6). The algorithm leverages this fact by issuing the undo log RDMA message before the in-place updates. This guarantees that even if the coordinator fails in the middle of the replication protocol, the in-place update RDMA messages will not take effect on the receiver without the undo log entry being present.

5.6 Fault Tolerance

In this section, we discuss how our protocol guarantees fault tolerance in various failure scenarios without sacrificing either correctness or high availability. For the ease of explanation, we first describe our recovery mechanism for single-partition transactions, and later extend it to multi-partition transactions. For single partition transactions, the coordinator is the primary replica for all records that the transaction accesses. We explain the failure of primary replicas and backup replicas separately. In the event of a machine failure, each surviving node $S$ performs the procedure in Section 5.6.1 for transactions that $S$ is a primary of (and therefore is the coordinator), and the procedure in Section 5.6.2 for the transactions that $S$ is among the backups.
5.6.1 Recovery from Backup Failures

Handling failures of backup machines is fairly straightforward, since the backups do not perform any replication processing and the replication state is maintained by the coordinator, which is $S$. In our protocol, a coordinator returns the result to the client only when it has successfully replicated the updates on all its backups. Therefore, any backup failure which happens before $S$ has successfully completed step 2 of the replication process (i.e. commit and release its log entries) will prevent $S$ from returning to the client. Instead, it has to wait for the cluster manager to broadcast the new configuration, upon receiving which, $S$ will know either it can commit or it needs to replicate on more machines. Note that the coordinator never rolls back a transaction for which it is already in the middle or at the end of step 2.

5.6.2 Recovery from Primary Failures

Recovery of transactions whose coordinator has failed is more challenging. The undo log messages are the footprints of the coordinator on the backups. Using these logs, the backups are able to decide to either rebuild and commit or discard and abort the transaction in order to maintain the system’s transactional consistency. More specifically, after a machine $P$ is suspected of failure, $S$ will perform the following procedure.

1. $S$ closes its RDMA queue pair to $P$, so that even if $P$ returns from failure and issues new RDMA operations, it will not be successful once $S$ starts the recovery protocol.

2. $S$ checks the $P$-owned log buffer on $S$’s local memory, and records all the transactions in the buffer, committed or uncommitted, to which we refer as $T_P$.

3. For each transaction $t$ in $T_P$, $S$ checks the data records pointed by the Change entries in $t$’s undo logs.

4. $S$ constructs a status message and broadcasts it to all surviving nodes. This status message contains $P$’s and $S$’s ID and also contains two entries per each transaction $t$ in $T_P$: ($t$’s id, $t$’s status), where the first one is the unique ID of the transaction, and the second one is the current status of the transaction on $S$. $t$’s status can be one of the following 4 cases (the 4th one will be described later):
   i) Corrupt log: for at least one of the log entries of $t$, LogID_check does not match LogID (i.e. self-verifying check fails), or the log does not end with an entry with IsLast=true (i.e. not all logs have been received). In this case, the data records must be untouched, since the undo logs are delivered before updates.
   ii) Logged: the log entries are correct, but the value of CommitBit is 0. In this case, the data may be untouched (i.e., the update messages are not received), corrupt (i.e., the update messages are partially received), or fully updated (i.e., the update message are fully received).
   iii) Commit-ready: the value of CommitBit is 1. In this case, the undo logs must all be received and the data records are fully updated.

5. $S$ broadcasts its status message to all surviving nodes, and receives their statue messages in return, which may have some overlap with $T_P$, but also may contain new transactions. These transactions have left no footprint on $S$, so $S$ adds them to $T_P$ and set their statuses to the fourth state: No log.
6. Upon receiving all the status messages, \( S \) commits a transaction in \( T_P \) if all involved nodes of that transaction are in *Commit-ready* status. Otherwise, \( S \) aborts the transaction, reverts its modified data using the corresponding undo log, and frees the \( P \)-owned log buffer on its local memory.

Once all backup nodes recovered the state and commit (or abort) the ongoing transactions, they inform the cluster manager to elect a new primary and the cluster continues regular processing.

**Proof of Correctness** — We now provide some intuition on how the recovery protocol ensures correctness. Table 5.1 summarizes the different failure scenarios that may befall the coordinator. We will refer to each scenario by its case ID which is in the rightmost column. The coordinator may fail *before or during* either of the two steps of the replication protocol presented in Section 5.5 (cases 1-4), or it may fail *after* completing the second step (case 5). In all these 5 cases, the recovery protocol guarantees correctness by satisfying these properties:

1. **Unanimous agreement:** All involved nodes must reach the same agreement about the transaction outcome. This is achieved by steps 5 and 6 of the recovery protocol in Section 5.6.2. If there is at least one surviving node that has not gone through step 2 of the replication protocol, all nodes will unanimously abort the transaction. Otherwise, all nodes will reach the same decision.

2. **Consistency for aborted transactions:** For an aborted transaction, the data in all involved nodes must be reverted to its prior consistent state, in the presence of any failure. In-order RDMA message delivery of reliable connected channels guarantees that for any surviving active node, if there is any data modification (complete or corrupt), there must be a complete log present at that node. Combined with the first property (unanimous agreement), this ensures that the recovery protocol always brings the database back to the last consistent state.

3. **Consistency for committed transactions:** A transaction whose coordinator has failed can commit only in cases 4b (if all nodes agree), 5a, and 5b. What all these three cases have in common is that all active nodes have gone through step 2 of the replication protocol (even if the coordinator failed before receiving acks) and their logs are in *Commit-ready* status. Therefore, the data records on all the active nodes must be already successfully updated, as it is done in step 1 of the replication protocol.

4. **Consistency in responses to the client:** The new elected coordinator will not report a different outcome to the client than what the original failed coordinator might have already reported. This property is guaranteed by the following reasoning: First, the coordinator commits the transaction only if it has completed step 2 on all active nodes. Second, in the event of the coordinator failure, the surviving nodes will reach a commit consensus only if the failed coordinator has completed step 2 on all active nodes. As a result, the transaction outcome is always preserved; If the coordinator assumed the transaction as committed, the consensus will be commit. Otherwise, it will be abort (cases 5a and 5b in Table 5.1).
5.6.3 Recovery of Multi-Partition Transactions

A multi-partition transaction accesses data from multiple primaries, with one partition acting as the coordinator. During the replication phase, the coordinator is in charge of both constructing the log entry for each accessed data partition, and in-place updating the data records in all nodes involved in the transaction — both the primary and backup nodes. A multi-partition transaction commits only after all the nodes have acknowledged the coordinator.

The recovery process of a multi-partition transaction is largely the same as a single-partition transaction. If the failed node is not the coordinator of an active transaction, the coordinator decides whether to replicate on more machines after the cluster reconfiguration, which is the same as in Section 5.6.1. If the the failed node is the coordinator, all the other machines locally construct and broadcast transaction status messages in the same way described in Section 5.6.2, with the only difference being that the commit decision is made if nodes from all involved partitions (rather than one partition) are in Commit-ready status.

5.7 Evaluation

In this section, we evaluate Active-Memory and compare it to the three replication schemes that we introduced previously. In particular, we aim to explore the following two main questions:

1. How does the new-generations network change the design trade-off of high availability protocols?
2. how well does Active-Memory scale under different loads compared to the other active-passive and active-active replication protocols?

5.7.1 Experiment Settings

Setup

Our cluster consists of 8 machines connected to a single InfiniBand EDR 4X switch using a Mellanox ConnectX-4 card. Each machine is equipped with 256GB RAM and two Intel Xeon E5-2660 v2 processors, each with 10 cores. All processes were mapped to cores on only one socket which resides in the same NUMA region as the NIC. The machines run Ubuntu 14.04 Server Edition as their OS and Mellanox OFED 3.4-1 driver for the network.

Workload

For the experiments, we use YCSB (Yahoo Cloud Serving Benchmark [18]) which models the workload for large-scale online stores. It contains a single table, with each record containing a primary key and 10 string columns of size 100 bytes each. For all experiments, the YCSB table is hash partitioned by the primary key, where each physical machine contains 5 million records (~ 5 GB). Each YCSB transaction in our experiments consists of 10 operations. Unless otherwise stated, the set of records for each transaction are selected uniformly either from the entire database (for the distributed transactions) or from the local partition
Figure 5.8: **Scalability** – The throughput of different replication protocols on different cluster sizes. All transactions are single-partition, with each reading and modifying 10 records.

(for the single-node). Also, for most experiments, each operation in a transaction reads a record and modifies it. Also, the replication factor is set to 3 (i.e. 2-safety) for all experiments unless otherwise stated. We will inspect each of these settings in more detail.

**Implementation**

To achieve a fair comparison between the replication schemes, we implemented all of them in a unified platform.

The system consists of a number of server machines and one client machine. The database is partitioned horizontally into multiple partitions. Each machine is the primary for at least one partition and the backup for several other partitions, i.e. the striped master model. Furthermore, each machine contains the same number of primary and backup partitions.

Within each machine, there are multiple worker threads working on the same partitions, except for H-Store (in which each single thread owns one partition and has exclusive access to it). To extract maximum concurrency, each thread uses multiple co-routines, so that when one transaction is waiting for a network operation, the currently running co-routine yields to the next one, who will then work on a different transaction. This way, threads do not waste their time stalling on a network operation, and are always doing useful work. We found that using 5 co-routines per thread is sufficient to extract maximum concurrency in our implementation.

The implementations for all the replication protocols share the same storage layer and access methods. Besides, the RPC sub-system is implemented using RDMA Send and Receive. Therefore, all replication schemes use the same high-bandwidth RDMA-enabled network, and do not have the TCP/IP messaging overhead.

We now briefly describe our implementation for each of the replication protocols that we will compare in the following sub-sections.

**Active-Memory** uses two-phase locking for concurrency control, and 2PC for multi-partitioned transactions. During transaction execution, remote records are accessed by issuing an RPC request to the record’s primary
Figure 5.9: CPU time breakdown for log shipping and Active-Memory with varying percentage of write transactions. Cluster size is 5 with 2-safety. Total system utilization in all cases is 100%.

partition. Once the transaction has successfully built its read-/write-set, the replication protocol described in Section 5.5 is performed using one-sided RDMA write operations.

Log shipping shares everything with Active-Memory, except its replication protocol which is based on sending log messages via RPC and replaying those logs at the receiver side. The transaction coordinator returns the result to the client only when it has received acks from all backup replicas. For distributed transactions, the coordinator sends the log message to all the backup replicas of the participating nodes.

H-Store replicates the transaction statement to all the replicas. For single-node transactions, each replica executes the transaction to completion without needing to acquire any locks, and returns the outcome to the coordinator. For distributed transactions, the coordinator locks all the participating nodes and engages them using 2PC. While one transaction is executing, no other transaction can be run in any of these partitions.

Calvin uses a partitioned and distributed sequencer, where each client sends its transactions to one of the them. At the beginning of every epoch, each sequencer replicates its transaction inputs on its replicas, and then sends it to all the schedulers on all the machines in its replica. The lock scheduler thread on each machine then scans all the sequenced transactions, and attempt to get all the locks for each transaction. Transactions that have acquired all of their locks are then processed by the worker threads. As stated earlier, any communication between two nodes is done via two-sided RDMA Send and Receive.

5.7.2 Single-Partition Transactions

We begin by examining the throughput of different replication schemes as the number of servers increases. The result is shown in Figure 5.8. In this experiment, all transactions are single-partition, and the replication factor is 3 (i.e. 2-safety). However, since in our platform the replicas are forced to be placed on different machines, replication factor of 3 is not possible when the number of machines is 1 or 2. For these two configurations, the replication is 0-safety (i.e. no replication) and 1-safety, respectively.

When the cluster size is 1, all schemes perform similarly since there is no replication involved, except for
H-store, which achieves a higher throughput. This is because H-store eliminates locking overhead due to its serial transaction execution. With 2 machines (therefore 1-safety replication), the throughput of all schemes except Active-Memory drops, which is due to the overhead of replication in these protocols. In Calvin and H-Store, the same transaction has to be executed redundantly by all the replicas. For single-partition transactions, replaying the updates for 10 records in log shipping is not so much different than re-executing the transaction which is why its performance is very close to that of Calvin.

The throughput of Active-Memory, in contrast, increases and further goes up with 3 machines. The reason is that replication in Active-Memory does not involve the CPU of the backup nodes at all. So, by increasing the cluster size, the processing power to handle new transactions proportionally increases. The asynchronous nature of RDMA operations means that the primary replica can issue the RDMA operations for a transaction, proceed to the next transaction, then come back later to pick up completions for the former transaction and commit it.

This experiment shows that even for a workload with single node transactions, which is the sweet spot of H-Store, its replication overhead dominates the benefit of its efficient single node transaction execution. Calvin not only has this overhead, but also has the cost of sequencing transactions and scheduling their locks, thereby achieving a lower throughput than H-Store. In general, Active-Memory can achieve 1.5x to 2x higher throughput compared to the other three protocols as the cluster scales out.

The average latency in the four replication protocols for a cluster size of 5 with 2-safety is reported in Table 5.2. Even though that Active-Memory involves two network roundtrips, it has lower latency compared to log shipping, which involves only one network roundtrip. This is because Active-Memory bypasses the RPC sub-system and also does not involve replaying logs at the replicas. Calvin has an order of magnitude higher latency due to the batching nature of its sequencing step. H-Store has the lowest latency among the others, owing to its simple command replication protocol.

To understand where the performance gain in Active-Memory comes from, Figure 5.9 illustrates the breakdown of CPU time for log shipping (LS) and Active-Memory (AM). Here, we fixed the number of machines to 5 with 2-safety, and varied the percentage of write transactions from zero (i.e. only read-only transactions) to 100% (the same workload in Figure 5.8). When there is no write transaction, both schemes perform similarly since no replication is involved, with 80% of the CPU time spent on performing transaction logic, and 20% on other tasks (including receiving client requests, copying each request to the corresponding worker’s buffer, orchestrating the RPC messages sent to and received from the other servers, and polling acknowledgements). As the ratio of write transactions increases, however, receiving the log messages (in yellow) by the replicas and replaying them (in blue) incur significant overhead for log shipping; a cost that is non-existing in Active-Memory. Even though that relative to log shipping, Active-Memory spends more time on initializing the RDMA messages (in red) and sending them (in orange), all in all much less CPU time is

<table>
<thead>
<tr>
<th>H-Store</th>
<th>Calvin</th>
<th>Log shipping</th>
<th>Active-Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>85µs</td>
<td>1253µs</td>
<td>142µs</td>
<td>121µs</td>
</tr>
</tbody>
</table>
left for performing the transaction logic in log shipping.

5.7.3 Multi-Partition Transactions

We next compare the impact of multi-partition transactions on the throughput of different replication protocols. The cluster size is set to 5, and the replication factor is 2-safety. While each single-partition transaction chooses all of its 10 records from the same partition, a multi-partition transaction selects 2 out of 10 of its records randomly from different partitions other than the rest of the 8 records (we experimented with different numbers of remote records per transaction, and observed similar results. Therefore their plots are omitted due to space constraints).

Figure 5.10 show the measured throughput of the four protocols with varying percentage of multi-partition transactions. H-Store performs better than log shipping and Calvin when there is no distributed transaction. However, with only 20% distributed transactions, the performance of H-Store drops to half of Calvin. This is because in H-Store, all participating nodes and their replicas are blocked for the entire duration of the transaction, which in this case takes one network roundtrip. The performance of all the other three replication protocols also drop with more multi-partition transactions. While the relative throughput of Active-Memory to that of log shipping remains the same with different percentage of distributed transactions, the relative throughput of Calvin to Active-Memory reaches from $\frac{1}{2}$ for single-partition transactions to $\frac{1}{3}$ for 100% distributed transactions.

5.7.4 Network Bandwidth

Our next experiment analyzes the impact of network bandwidth on throughput. We throttled the bandwidth by running background network data transfer jobs and made sure that during the course of the experiment, they consumed the requested portion of bandwidth.

Figure 5.11 shows the throughput of different protocols on 5 servers with 2-safety replication for single-partition transactions. When the available bandwidth is less than 10Gb/sec, the active-active schemes perform
Figure 5.11: **Network Bandwidth** – The measured throughput of the replication schemes on 5 servers, with replication factor set to 2-safety, and 10 read-modify records per transaction.

Table 5.3: The network traffic per transaction in each replication protocol in our unified platform.

<table>
<thead>
<tr>
<th></th>
<th>Single-partition Transactions</th>
<th>Multi-partition Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>H-Store</td>
<td>~ 0.3KB</td>
<td>~ 1.5KB</td>
</tr>
<tr>
<td>Calvin</td>
<td>~ 0.5KB</td>
<td>~ 0.6KB</td>
</tr>
<tr>
<td>Log shipping</td>
<td>~ 2.5KB</td>
<td>~ 3.7KB</td>
</tr>
<tr>
<td>Active-Memory</td>
<td>~ 4.5KB</td>
<td>~ 6KB</td>
</tr>
</tbody>
</table>

much better than the other two protocols. By enforcing a deterministic order of executing transactions at all replicas, these techniques can effectively minimize the network communication, which is key in delivering good performance in a network with very limited bandwidth. In such low bandwidth networks, Active-Memory is outperformed by log shipping as well by a factor of 2. To inspect this result in more depth, we also measured the network traffic per transaction in each of these replication schemes (Table 5.3). For a single-partition transaction, both H-Store and Calvin require relatively very little network traffic. Active-Memory, on the other hand, requires 15×, 9× and 1.8× more traffic than H-Store, Calvin and log shipping respectively. This explains the reason for this lower throughput compared to the other schemes when the network is slow: Active-Memory becomes network-bound when the bandwidth is limited. However, as the bandwidth exceeds 10Gb/sec, Active-Memory starts to outperform the other protocols. Increasing the available bandwidth does not result in any increase in the throughput of the other schemes, which is due to the important fact that these schemes are CPU-bound, thereby not exploiting high bandwidth of the modern networks. We performed the same experiment for the workload with multi-partition transactions (which we used for the experiment in Section 5.7.3), and observed similar results, except that H-Store was massively outperformed by Calvin, even when the bandwidth was limited. The measured network footprint for each multi-partition transaction is shown in the last column of Table 5.3. The plot for the distributed transactions is omitted due to space constraints.

In short, this experiment reveals that in the conventional Ethernet networks where the bandwidth was scarce, adding processing redundancy to avoid the network communication would be extremely beneficial.
However, in the modern networks with their abundant bandwidth, the bottleneck has shifted in the direction of CPU; the replication protocols that do not leverage CPU-efficient RDMA operations and high bandwidth of these networks are not optimal anymore to provide LAN-based replication.

5.7.5 Replication Factor

The previous three experiments fixed the replication factor to 2-safety. In this experiment, we examine the effect of different replication factors on throughput. Here, all transactions are single-partition, each reading and modifying 10 records.

The results in Figure 5.12 show the throughput of the four protocols as we varied the replication factor. With no replication (i.e. 0-safety), H-Store achieves the highest throughput, which is in agreement with the results of the scalability experiment. Active-Memory and log shipping exhibit the same performance as they use the same 2PL-based execution. The sequencing and scheduling overhead of Calvin accounts for its lower throughput.

Increasing the replication factor entails more redundant processing for H-Store and Calvin. For example, the throughput of H-Store for 1-safety replication is almost half of its throughput for no-replication, since each transaction is now executed twice. The same applies to Calvin. In addition, since the transaction modifies all 10 records that it reads, log replay in log shipping incurs almost the same cost as redoing the transaction. As the replication factor increases, the gap between Active-Memory and the other schemes widens since higher replication factors translate to more redundancy for them. On the other hand, a higher replication factor only increases the number of RDMA write operations that Active-Memory needs to send to its replicas, which does not impact the throughput proportionally, since modern NICs can handle up to tens or hundreds of million RDMA operations per second without putting much overhead to the CPU.

5.7.6 Impact of Contention

The goal of this experiment is to measure the impact of data skew on the throughput of different replication schemes. In YCSB, the skew is controlled using the Zipfian constant $\theta$. When $\theta$ is 0, the records are...
uniformly selected from the entire cluster, and when it is 0.9, the accesses are highly skewed, resulting in a small set of contended records. Since each transactions touch 10 records, choosing a record randomly from the entire cluster makes all transactions multi-partition.

Figure 5.13 shows the measured throughput with different skew factors. For the values of theta up to 0.4, the performances of all the protocols remain unaffected. As we saw before in Section 5.7.3, H-Store performs poorly when the workload is dominated with distributed transactions, while Active-Memory maintains its relative better performance compared to the other replication schemes. As contention reaches 0.8, both Active-Memory and log shipping encounter high abort rates due to lock conflicts, and therefore their performances degrade significantly. The throughput of Calvin drops as well since the locks for each transaction have to be held during the network communication with the other servers. However, due to its deterministic lock scheduling and the fact that a transaction’s working set is known prior to execution, Calvin can tolerate high contention better than all the other schemes. In particular, at high contention, the performance of Calvin is up to 70% better than the second-best scheme, which is Active-Memory.

Note that Active-Memory is a replication protocol, and does not specify how concurrency should be managed. Currently, our implementation is based on 2PL for executing transactions. Moreover, in our protocol, a transaction is executed immediately upon arriving in the system with no sophisticated scheduling. By implementing Active-Memory on top of a different execution scheme than simple 2PL, which possibly employs a more sophisticated transaction re-ordering (such as [110] or [68]), one can expect that it will circumvent such cases and become more resilient to high data skew.

### 5.7.7 Read-Write Ratio

To analyze the effect of transactions with different read-write ratio on throughput, we varied the number of updates in transactions (out of 10 operations). Figure 5.14 shows the result for this experiment. H-Store has the highest throughput for read-only transactions (i.e. no write operations), as such a transaction is executed only in one node without the need to be replicated. Calvin, on the other hand, has the overhead of sequencing transactions and scheduling locks, even though that we applied the optimization that read-only transactions
are only executed in the first replica, and not in the others. Log shipping and Active-Memory perform similarly for read-only transactions due to their identical execution scheme when there is no replication involved.

However, as transactions start to have more write operations, the log would contain more records and thus more work for the replicas. Also, even with 1 write operation, the optimizations to H-Store and Calvin is no longer possible, which explains their drop. With 4 write operations, log shipping delivers the same performance as the deterministic approach of H-store, and with 10 writes, the same performance as Calvin. The throughput of both deterministic protocols, on the other hand, do not degrade significantly with more writes, as all operations are still handled locally without much extra redundancy compared to fewer writes. For Active-Memory, more writes in transactions indicate more undo log entries to replicate and more in-place updates to send, which explains the decrease in its throughput. However, it retains its better performance compared to the other schemes for all numbers of write count, and the more write there is, the wider the gap between Active-Memory’s primary-backup replication and log shipping.

5.8 Related Work

In this section, we discuss related works on high availability in conventional networks as well as RDMA-based OLTP systems.

5.8.1 Replication in Conventional Networks

In shared-nothing databases, the primary copy eager replication, also known as active-passive, is implemented through log shipping, and has been the most widely used technique to provide fault tolerance and high availability in strongly consistent databases [104] (such as Oracle TimesTen [51] and Microsoft SQL Server Always On [65]). In conventional networks, the coordination between replicas in log shipping, however, incurs significant cost, motivating much research effort in this area. Qin et al., identified two problems with using log shipping in high performance DBMSs [79]. First, logs can be so large in size that the network
bandwidth becomes the bottleneck. second, sequentially replaying the log at the backups can make them constantly fall behind the primary replica. Eris [59] is a recent transactional system that proposes a network co-design which moves the task of ordering transactions from the replication protocol to the datacenter network, and eliminates the coordination overhead for a specific class of transactions, namely independent transactions (the same class that H-Store [44] and Granola [20] also optimize for). RAMCloud [70, 72] is a distributed key-value store that keeps all the data in the collective DRAM of the cluster, and only supports single-key operations. RAMCloud also takes a primary-backup approach to replication. However, unlike our work, RAMCloud does not assume non-volatile memory, and only keeps one copy of each object in memory. Instead, the durability is guaranteed by storing the changes to remote disks using a distributed log. Therefore, a failure of a node may suspend the operation of the system for the records on that node until the data is recovered from the backup disks.

RemusDB is another active-passive system that proposes to push replication functionality outside of the databases to the virtualization layer itself [64]. In general, even though that many of the optimizations for removing coordination between replicas in the aforementioned research projects indeed improve the performance in specific cases or environments, but the main problem, which is overhead of replaying logs, still persists.

For deterministic active-active replication, two well-known examples, namely H-Store [44] and Calvin [95], were studied in detail in Section 5.3.2. In short, they both rely on a deterministic execution and replication paradigm to ascertain that all replicas go through the same serial history of transactions. This way, replicas never diverge from each other without having to coordinate during execution. H-Store is mainly optimized for single-partition transactions, as it also eschews record locking due to its single-thread-per-partition serial execution. On the other hand, Calvin is more efficient for multi-partition transactions or workloads with high data contention, as it employs a novel distributed sequencer and deterministic locking scheme. Calvin requires that the read-set and write-set of transactions are known apriori, while H-Store requires to know all the transactions upfront.

As we already saw in Section 5.3, both active-passive and active-active introduce computation overhead, each in a different way. Active-passive involves exchanging multiple rounds of messages, and replaying the logs. Active-active results in processing the same transaction redundantly multiple times. Active-Memory uses RDMA Write to forgo both of these overheads.

5.8.2 RDMA-based OLTP

RDMA, combined with the emerging NVM technologies as the main permanent storage, have been changing the landscape of distributed data store designs, especially in the area of distributed transaction processing. In addition to proposals to use RDMA for state machine replication through Paxos [99, 77], an array of key-value stores (e.g. [66, 42, 55]) and full transactional databases (e.g. [26, 108, 110, 43, 69]) have been proposed, each with a different way to leverage RDMA for executing transactions. Among these systems, some provide high availability using different variants of RDMA and implement active-passive replication [26, 93, 113, 43]. For example, FaRM [26] implements log shipping using one-sided RDMA Write operations to transfer logs to the non-volatile memory of the backups. To reduce transaction latency and enable group-replication,
the coordinator in the FaRM protocol considers a transaction replicated once the RDMA Writes are sent to backups. Tailwind [93] is an RDMA-based log replication protocol built on top of RAMCloud, that proposes a mechanism to detect incomplete RDMA operations if a failure happens to the primary while it is replicating its log. In any case, no matter how the logs are shipped (using one-sided or two-sided RDMA), they have to be processed by the backups synchronously or asynchronously to provide high availability, which as we have already discussed in previous sections, imposes processing overhead on backups. Active-Memory is the first fault-tolerant replication scheme that fully leverages RDMA Write to consistently modify the state of the backup nodes in a failure atomic way.

5.9 Main Takeaways

In this chapter, we presented Active-Memory, an RDMA-based mechanism to provide high availability using strongly consistent primary-backup replication. First, we identified that existing active-passive and active-active replication protocols were optimized for reducing network traffic at the cost of increased computation overhead. While such a design decision makes sense for conventional networks, it is no longer the best design choice for new-generation networks that offers orders of magnitude higher bandwidth and lower latency. Therefore, the main objective of Active-Memory is to minimize the CPU processing overhead by relying on the new features and properties of RDMA-enabled networks. Active-Memory achieves this goal by 1) using a novel RDMA-compatible undo logging mechanism, and 2) updating data records in the replicas directly using one-sided RDMA write operations. Evaluation shows that Active-Memory can achieve $2 \times$ performance improvement compared to the second-best protocol that we evaluated.
Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

In this dissertation, we made the case that the emerging modern RDMA-enabled networks require that existing distributed OLTP systems are re-designed. This is because at their core, these systems were designed with the assumption that the network communication is the dominant bottleneck, where in fact this assumption is no longer relevant in modern networks.

In particular, we have made the following contributions in this thesis:

- **Network-Attached Memory (NAM) – an RDMA-aware architecture for distributed systems**: We proposed NAM as an alternative to the shared-nothing architecture for distributed systems built on top of RDMA. Compared to shared-nothing, a NAM architecture logically decouples compute and memory, enabling independent scale-out of either of them. Compute nodes are state-less, while memory nodes are computation-less. The database is partitioned and stored in the collective main memory of memory nodes, and is accessed by the compute nodes using RDMA operations.

- **NAM-DB – a scalable transaction processing system**: Based on the above-mentioned architecture, we have built a new OLTP system which exhibits almost linear scalability even when all transactions are multi-partition. Our system offers Snapshot Isolation, which is the most common isolation level in database applications. NAM-DB relies on a novel scalable timestamp oracle that, unlike existing timestamp generation techniques, allow remote compute nodes to directly co-operate in building the timestamp.

- **Chiller – contention-centric data partitioning**: Having eliminated the distributed transactions as the main source of poor scalability of transactional systems, we then designed a new partitioning technique that aims to remove the next source of bottleneck, which is data contention. In contrast to existing partitioning techniques whose objective is to minimize the number of distributed transactions, Chiller aims to minimize contention by co-locating contended records that are often accessed together, and minimize the duration of locks on these records. Compared to existing partitioning techniques, Chiller
significantly improves performance and scalability when the workload contains frequently accessed records.

- **Active-Memory – RDMA-based high availability:** We proposed a novel replication technique, which leverages RDMA efficiently to completely eliminate the processing redundancy in existing replication protocols. We proved its correctness, and presented its straight-forward recovery protocol. Active-Memory delivers significantly better performance compared to the existing network-optimized replication schemes.

### 6.2 Suggested Directions for Future Research

Based on the ideas presented in this dissertation, there are several interesting future research avenues.

**OLTP systems on programmable NICs:** One major limitation of today’s typical RNICs is the basic functionality and lack of expressiveness in operations they support. An RDMA operation can only operate on a contiguous region of remote memory. Accessing data records in OLTP systems, however, often require following pointers (such as looking up data in an index), demanding multiple RDMA operations that possibly take networks round-trips. The emerging programmable smart NICs, such as FPGA-based NICs, open new possibilities to design much simpler and more performant protocols [56].

**Leveraging the opportunity of transaction batching:** Database systems utilize request batching to enhance their performance [24]. We briefly explored a variant of batching in the design of the timestamp oracle in NAM-DB. However, its full potentials need an in-depth study. For example, Chiller’s contention-centric execution model (i.e. the two-region model) can largely benefit from batching. Also, Active-Memory can be made more efficient if transactions are replicated in batches. Applying request batching on an RDMA-based system, however, is not trivial at all, due to the primitive API of RDMA which allows only one contiguous memory access. Special data structures and/or different data layouts are likely to be needed. Smart NICs can play an important role here, too. On the requester side, multiple requests can be batched, and transmitted using one RDMA call. On the receiver side, a batch of requests can be processed directly by the programmable NIC with no/little interruption for CPU.

**General contention-aware execution:** Our proposed idea for handling contention was based on two-phase locking. Extending Chiller to work on other concurrency control techniques, such as optimistic and multi-version schemes, is an interesting and challenging future work. The main challenge lies in making the early commit concept of Chiller work on these schemes, and prove their correctness and recoverability.
Bibliography


