Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases

by

Evan Philip Charles Jones

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2012

Author . . . .
Department of Electrical Engineering and Computer Science
December 28, 2011

Certified by . . . .
Samuel Madden
Associate Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor
Chair, Department Committee on Graduate Students

Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases

by

Evan Philip Charles Jones

Submitted to the Department of Electrical Engineering and Computer Science
on December 28, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

This thesis presents Dtxn, a fault-tolerant distributed transaction system designed specifically for building online transaction processing (OLTP) databases. Databases have traditionally been designed as general purpose data processing tools. By being designed only for OLTP workloads, Dtxn can be more efficient. It is designed to support very large databases by partitioning data across a cluster of commodity servers in a data center. Combining multiple servers together allows systems built with Dtxn to be cost effective, highly available, scalable, and fault-tolerant.

Dtxn provides three novel features. First, it provides reusable infrastructure for building a distributed OLTP database out of single machine databases. This allows developers to take a specialized backend storage engine and use it across multiple machines, without needing to re-implement the distributed transaction infrastructure. We used Dtxn to build four different applications: a simple key/value store, a specialized TPC-C implementation, a main-memory OLTP database, and a traditional disk-based OLTP database.

Second, Dtxn provides a novel concurrency control mechanism called speculative concurrency control, designed for main memory OLTP workloads that are primarily composed of transactions with a single round of communication between the application and database. Speculative concurrency control executes one transaction at a time, with no concurrency control overhead. In cases where there may be stalls due to network communication, it speculates future transactions. Our results show that this provides significantly better throughput than traditional two-phase locking, outperforming it by a factor of two on the TPC-C benchmark.

Finally, Dtxn supports live migration, allowing part of the data on one server to be moved to another server while processing transactions. Our experiments show that our approach has nearly no visible impact on throughput or latency when moving data under moderate to high loads. It has significantly less impact than the best commercially available systems when the database is overloaded. The period of time where the throughput is reduced is less than half as long as failing over to another replica or using virtual machine migration.

Thesis Supervisor: Samuel Madden
Title: Associate Professor

Acknowledgments

My mother has always told me to "dream the impossible dream." Well, you can scratch this one off my list. I never thought that I would be accepted to MIT, and now, here I am, finishing my Ph.D. So first, I need to thank my parents for always encouraging me to aim high. As teachers themselves, they taught me that education is important and valuable. Without their encouragement, I might have taken the safe path to a career in industry, and I almost certainly would not have applied to MIT.

Second, I need to thank my wife, Eran, for letting me chase my dreams. We have been physically separated for much of the last four years, with me spending the bulk of my time in Boston and her dream job requiring her to be in New York. Despite that, whenever I expressed any doubt about what I was doing, she always reminded me of the reasons I wanted to study computer science.

Next, I need to thank my committee: Barbara Liskov and Robert Morris, who kindly agreed to read my thesis. They caught a number of stupid, simple mistakes, as well as asked insightful questions that made me think and rework some of the material. Even though this meant additional work for me, their feedback has made this work better. I also want to thank Barbara for being my adviser when I first arrived at MIT.

I ended up working with my adviser, Sam Madden, because I thought that the distributed transaction protocol described in the original H-Store paper might be wrong. I asked him about it, and he encouraged me to think about a solution. That initial serendipitous conversation led directly to the work that is presented in this thesis. Since then, Sam has allowed me to pursue my interests, while providing useful criticism and suggestions about what to work on next. He provided many opportunities to interact with people at other institutions and in industry, and encouraged me to contribute to other projects, such as teaching 6.033 recitations and running the SIGMOD programming contest. While some of these projects consumed time that could have been spent doing research, they also taught me about the broader academic community, and so I don't regret doing them. I also appreciated his assistance in writing introductions, abstracts, and conclusions just before paper deadlines, allowing me and our co-authors to finish those last few experiments. In short: he was a great adviser. Thanks to him, I'll always remember that lines on graphs should be thick and different in both color and style, since I had to fix nearly every graph I've ever showed him. But he was right: it did make them more readable.

My research begun its life as part of the H-Store project, which was a collaboration between people at MIT, Brown, Yale, and VoltDB, a company that is commercializing the concept. I want to thank everyone involved for providing valuable insights and spending many hours discussing the work. I won't name anyone to avoid leaving someone out. However, I do need to thank Daniel Abadi at Yale for collaborating on my concurrency control work, and Andy Pavlo at Brown for many, many helpful conversations and for actually attempting to use my software.

Carlo Curino started his postdoc after I had been at MIT for two years, and ended up sitting in my office. He managed to convince me that we should work together, and I managed to convince him that we should work on something related to databases as a service. I have to admit that I wasn't initially convinced that collaborating was a good idea, as I was trying to focus on my H-Store work. It took me a while to realize that the

many, many, hours that he and I spent discussing and debating were actually useful. They forced me to think about alternatives that I had not considered. I like to think that about half the time when Carlo questioned what I was doing, I was already doing the right thing. However, the other half of the time, I was wrong, and he made me realize that it would be better to do something else. Even when I was right, explaining it solidified and clarified my thinking. In the end, we managed to get a lot done on the Relational Cloud project in the two years he spent at MIT. He made my life more interesting and more productive. He also taught me the value of drinking single espressos: it allows you to split a double with someone else, and also to take more breaks. However, I still adhere to my strict limit of three double espressos a day.

The other faculty and fellow students at MIT are what make it such an interesting place to be. I don't think I would have ever finished a Ph.D. if I wasn't surrounded by so many smart people who are all very passionate about computer science. I don't want to even attempt to name them because the list is long and I will almost certainly leave someone out. One of my only regrets is that I didn't get to know more of these people better, and that I didn't actively collaborate with more of them.

There have been a number of teachers prior to MIT who have influenced my life, which led, indirectly or not, to me finishing a Ph.D. Two in particular who helped me were Paul Ward and Srinivasan Keshav, who I worked with during my Master's degree at the University of Waterloo. They encouraged me to apply to the Ph.D. program at my top choice school, even though I thought I wanted to work in industry. I also suspect that without their reference letters, I never would have been accepted.

Finally, I would like to thank the Lucky Star bus company for providing inexpensive transportation between Boston and New York. During the course of my studies, I have taken 174 trips between the two, for a total of 30 days, 23 hours. Without the inexpensive transportation provided by Lucky Star, Fung Wah, Megabus, and Bolt Bus, I never would have been able to afford this many trips, and my relationship with Eran would have suffered. Despite my appreciation, I look forward to never getting on that bus again.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Applications that require high throughput access to data are increasingly relying on main memory, as it provides much better and more predictable performance than magnetic disk. The decrease in cost provided by Moore's law and the growth of large data centers means that it is now cost effective to store significant amounts of data in RAM across a cluster of computers. Current commodity servers can fit up to 196 GB of RAM in a standard 1U rack unit, so a single rack of 40 servers can have an aggregate capacity of approximately 7.5 TB. This is more than sufficient for most online transaction processing (OLTP) applications, such as bank transaction processing, airline reservation systems, e-commerce, or web applications. The data volume of these applications is relatively modest because the recommended practice today is to offload historical data to a separate dedicated system. This allows one system to be optimized for interactive OLTP, while another data warehouse system is used for online analytical processing (OLAP). Despite these changes, traditional databases are designed for a wide variety of applications, including both OLTP and OLAP. As a result, they are very flexible, but also have conservative designs that limit their performance.

The traditional approach to managing high performance database workloads has been to use very large machines. However, these high-end systems are very expensive when compared to the cost of commodity servers on a per-CPU or per-gigabyte of RAM basis. Today, it is widely accepted that the most cost effective approach to building big systems is to combine the resources of a cluster of servers, connected with commodity networking. Large organizations are building huge data centers that host thousands of commodity servers. Thus, high performance databases must be designed to be distributed across many servers in a data center to meet the needs of modern applications.

Since interactive OLTP applications are critical for the operation of a business, the database must be highly available. Running on a cluster of computers provides the opportunity to use replication and fault isolation to protect against independent failures. However, replication cannot help when the application needs evolve. Successful systems need to be able to scale in response to changing load, and applications will change their schema as they are upgraded. Traditional systems make these changes in a maintenance window when the database is unavailable. In today's world, this is unacceptable. To avoid maintenance windows and further improve availability, a database needs to support live migration, allowing data to be moved between machines while processing transactions. This allows the system to be scaled dynamically in response to load, to reconfigure the data distribution, or to simply take a machine out of service for repairs.

While distributed databases have been well studied, this new environment provides the opportunity to re-examine old design decisions to adapt to the properties of OLTP workloads and clusters of commodity servers. In this thesis, we present Dtxn, a fault-tolerant distributed transaction system built for distributed databases running OLTP workloads. It allows backend database developers to build a large database system out of single machine storage engines, by partitioning and distributing data across many machines. It is optimized for memory-resident workloads, and can provide significantly better throughput than traditional transaction processing systems for in-memory storage engines. It provides the following novel features:

Reusable infrastructure for OLTP databases. Dtxn provides abstractions for building a distributed database out of single machine databases. It is designed for workloads that require serializable transactions that each access small amounts of data, which is an ideal fit for OLTP applications. The database is treated as a black box, so that Dtxn can be used for different data storage applications. In this thesis, we describe how it has been applied to four applications with very different workloads.

Speculative concurrency control. This is a new concurrency control technique for main-memory databases that provides serializability without tracking data accesses at a fine granularity. It is motivated by the observation that in main-memory databases, latching and locking consume a large fraction of the CPU [45, 40, 60, 84, 95]. Speculative concurrency control eliminates most of this overhead, improving throughput for transaction processing workloads that are mostly composed of transactions that have a single round of communication with the database. This is a good fit for OLTP which is mostly composed of simple data accesses (create/read/update/delete), auto-commit statements, or stored procedures.

Live data migration using a cached-based approach. Dtxn can move data from one machine to another while transactions are being processed. This improves the availability of the system by allowing it to be reconfigured without downtime. Existing approaches copy all the data to be moved before switching over to the destination machine. This adds additional load to the source machine, which can impact performance significantly. Instead, Dtxn switches to the destination immediately, then fetches referenced data on demand from the source. We show that this allows load to be shifted away from the source machine faster than existing techniques, allows partial migration and repartitioning, and has a minimal visible impact on application performance.

1.1 Application Requirements

Dtxn is able to provide better performance than existing transaction processing systems because it is designed for a specific application domain: high throughput online transaction processing. It attempts to meet the following application requirements:

Use commodity servers and networks: The most cost effective computing platform today is whatever is mass produced. As a result, commodity servers with provide the best performance per dollar. Today, that means a system with one or two CPUs, each with four or six cores (a total of four to twelve cores), anywhere from 36 to 144 GB of RAM, and one to six disks. These systems are typically interconnected via a single gigabit Ethernet port. In order to make the most efficient database system possible, Dtxn is designed to run on commodity servers inside large data centers.

OLTP workloads: Dtxn is designed specifically for online transaction processing (OLTP) applications. These workloads are composed of short transactions that only contain a few operations, with no user interaction. Each operation is typically a simple query or update that accesses a small number of records (less than 100). Updates are an important part of these workloads. OLTP workloads typically have a number of pre-defined high-level operations that are performed at high rates. This description fits most applications that manage interactive user requests such as banking, airline reservation systems, e-commerce, and web applications.

The opposite of OLTP is online analytical processing (OLAP), which typically involves processing queries that must access large amounts of data. These queries are frequently not known in advance, and are typically very read intensive with few updates. These workloads appear in data mining and reporting applications, where users want to discover interesting information from large data sets.

These two classes of applications are both large and important, but are very different. Databases have traditionally been designed to support both workloads, and hence are not optimized for either. However, the current industry practice is to have two separate database systems, one for the online data, and a second for analytic tasks. The analytic system is typically loaded with the new data from the OLTP system either periodically (e.g., at end of each business day), or continuously. This ensures that the business critical OLTP system is isolated from any analytic tasks, and that the two systems can be optimized separately. Importantly for Dtxn, this means that a system designed for only OLTP applications can be used.

High availability: The OLTP systems in most enterprises are critical to the business, as these are the systems that are required to process each user request, such as each purchase, every page view, or each financial transaction. If the system handling these tasks is down, the business loses money. As a result, the system must be highly available. This means that there should be no single points of failure, and that recovery must be considered carefully. The system should be able to incorporate changes without downtime, such as scaling by adding or removing machines.

High throughput and low latency: Traditional databases function very well for modest workloads that can be handled by a single commodity server. Today, a server that costs $3,000 USD running an open-source database server can easily process a few thousand transactions per second. However, a single machine database provides no easy way to scale the system to handle increased application load or data volume.

Table 1.1: Summary of application requirements and solutions

Requirement	Solutions
Commodity servers	Scale out via partitioning
OLTP workloads	Single round transactions, Small data accesses, Memory resident, Partitioning
High availability	Replication, Partial failure, Live migration
High throughput	Memory resident, Speculative concurrency control
Low latency	Memory resident, Replication instead of disk logging
Serializability	Speculative concurrency control
Reuse	Data agnostic

Dtxn is designed for applications that must process more transactions than a single machine can handle, while keeping latency low for interactive use.

Serializable transactions: Transactions with serializable consistency have proven to be a very useful abstraction, making it feasible for people to write concurrent programs without having to think about concurrency. Some systems choose to provide relaxed consistency models, either by only providing guarantees for single item accesses, or by eschewing any guarantees at all. However, there are many applications where a strict correctness guarantee is required (e.g., banking). Perhaps more importantly, these strong guarantees make the system easier to use, by making it easier to understand what happens when data is accessed and manipulated concurrently. Thus, Dtxn provides the traditional serializable transaction semantics.

Reusable for different backend storage engines: It has become popular for specialized storage engines to be built for different applications, as it is possible to obtain better efficiency or to provide features customized for the application. In order to use these specialized storage engines in a distributed system, backend developers must implement data distribution, concurrency control, replication, and live migration. Dtxn provides these features, allowing backend developers to focus on the aspects of their system that are novel.

1.2 Solutions

In order to meet the application requirements, Dtxn includes the following features. A summary showing which requirements lead to which features is shown in Table 1.1

Memory-resident workloads: RAM is the highest performance storage technology available. In order to provide extremely high throughput and low latency, Dtxn is optimized for database-based applications where the working set fits in main memory. As argued earlier, this is now feasible for OLTP applications. Dtxn can be used for databases where most requests require disk accesses, although speculative concurrency control should not be used for these applications.

Single round transactions: In order to avoid networking and distributed coordination overhead, Dtxn is optimized to execute a transaction with one round of communication, without round-trips between the database and the client application. This reduces overhead and introduces many opportunities for optimizations. In OLTP workloads, many operations are simple data accesses (e.g. single auto-commit SQL statements), which can trivially be implemented in this efficient, single-round form. Multi-operation transactions are typically defined in advance as part of the application code. As a result, it is possible for them to be written as stored procedures, to be executed by the database. Thus, it is feasible for most transactions in OLTP workloads to be executed in an efficient, single round of communication. However, Dtxn still supports traditional transactions that involve multiple rounds of communication.

Small data accesses: OLTP transactions typically manipulate small amounts of data, such as a single purchase order or one user's bank accounts, and thus Dtxn is designed for these workloads. As a result, Dtxn processes an entire operation at a time, and passes complete result sets as single network messages. This will not work well for applications that must retrieve multiple megabytes of data, as is typical for OLAP workloads. These workloads need to support streaming results and pipelined processing, which is not a good fit for Dtxn.

Replication: The primary technique for providing high availability is to replicate the system to protect against independent software and hardware failures. Dtxn can replicate all data across more than one physical machine for this purpose. Replication also provides data durability, replacing forced writes to stable storage. This provides lower latency than waiting for magnetic disk writes to complete. However, Dtxn also provides support for traditional write-ahead disk-based logging, if desired.

Scale out via explicit partitioning: In order to use multiple independent servers, the data and tasks must be partitioned across them in some way. In examining a variety of OLTP workloads, we have found that most of them can be partitioned such that most transactions only need to access a single partition. We assume that the user has carefully partitioned their application so that this is true. Additionally, Dtxn is designed so that there are no single bottlenecks for the majority of operations, allowing the aggregate performance of the system to be increased by adding more machines.

Fault isolation: Partitioning also improves availability by providing a unit of isolation and allowing for fault isolation. Dtxn is designed to continue functioning even if some partitions of the database are unavailable. Any operations that must access the unavailable partitions will fail, but operations that access the functioning partitions can continue.

Speculative concurrency control: Previous research has found that traditional lock-based concurrency control adds significant overhead to transaction processing in main-memory OLTP database systems. Dtxn has been designed to remove many of the reasons traditional systems need to execute concurrent transactions, such as waiting for disk, or commands from the application. As a result, single round transactions, such as single data operations (e.g. single SQL statements) or stored procedures can typically be executed sequentially, with no concurrency control overhead. However, this causes transactions that span multiple partitions, called *multi-partition transactions* to stall while waiting for two-phase commit to complete. As a result, Dtxn provides a low-overhead concurrency control mechanism called speculative concurrency control that can exploit the properties of stored procedures.

Live migration: Successful applications must be able to handle growth, and critical applications must be constantly available. As a result, the database must be capable of moving data in order to scale out to more machines, while continuing to process transactions. Dtxn supports live migration, permitting data to be redistributed without stopping. Our live migration system is designed to support partial migration, which allows an existing partitions to be split or recombined. This not only allows a system to be scaled, but also allows data to be re-partitioned in case the data access patterns change.

Data agnostic: In order to be applicable for different types of storage engines, Dtxn cannot make any assumptions about the structure of the data that is stored in the system. Dtxn's interfaces are chosen to be the minimum required to support replication, concurrency control and data migration while allowing the storage engine the most flexibility possible. Dtxn effectively treats the storage engine as a black box.

1.3 Outline

In the next chapter we describe the architecture of the Dtxn system, and the details about its protocols for distributed transactions and replication. We then describe how this design has been applied to four different applications. In Chapter 4 we describe Speculative Concurrency Control, our novel technique optimized for main-memory storage and stored procedures. Live migration is discussed in Chapter 5. Finally, we summarize the previous work on distributed transaction processing systems that are similar to Dtxn before presenting our conclusions and future work.

Chapter 2

Dtxn Design

Dtxn is a framework for building distributed storage systems that run on a cluster of computers in a single data center. It supports serializable transactions and replication for fault tolerance. Systems built with Dtxn can be scaled by adding and removing nodes while transactions are being processed. Dtxn is designed to be useful for a wide variety of systems, and so it does not directly provide an implementation for storing and processing data. Instead, it provides a set of interfaces that can be used to take a single-node storage engine and make it into a distributed system. We will explain how Dtxn works from the bottom up, by describing an example application: a simple key/value storage system. To begin, the backend developer builds a "local" storage engine, which is just a library to store data.

2.1 Local Storage Engines

A local storage engine is some custom code that stores data in some format that is useful for applications. As an example, a local key/value storage engine can be a library that supports a few operations on a collection of byte strings: get a key, update a key, and delete a key. If it stores data in memory, this is trivial to build. It only requires using an appropriate hash table or binary tree from an existing library. An application can use this engine by calling the appropriate functions.

In Dtxn, this component is called a *storage engine*, and is the primitive building block for a distributed storage system. The backend developer creates a storage engine by implementing the interface shown in Figure 2-1. The interface's most important function is executeFragment(), which executes an application-supplied operation and returns the result. Dtxn knows nothing about the internals of the storage engine. Instead, it treats it as an opaque state machine, passing operations from the application as byte strings and returning the results of those operations as byte strings. This allows Dtxn to be used for a wide variety of different storage systems, and makes it possible to send the operations and results across a network. The storage engine also needs to support operations that can be undone by recording an *undo record*. The undo record is an opaque structure returned by the storage engine that will later either be applied to undo the effects of the operation or freed if the operation is permanent.

This interface is sufficient to support transactions. In Dtxn, a transaction is composed

```
interface StorageEngine:
  // Returns the output produced by executing the operation represented by fragment
  // If undoRecord is not null, then it must be used to store information so the fragment
  // can be undone. In this case, either applyUndoRecord() or freeUndoRecord() will be
  // called after the transaction completes.
  bytes executeFragment(bytes fragment, UndoRecord undoRecord);

  // Applies undoRecord to undo the transaction. Called in LIFO order.
  void applyUndoRecord(UndoRecord undoRecord);

  // Frees resources associated with undoRecord. Called in FIFO order.
  void freeUndoRecord(UndoRecord undoRecord);
```

Figure 2-1: Pseudocode for the StorageEngine interface

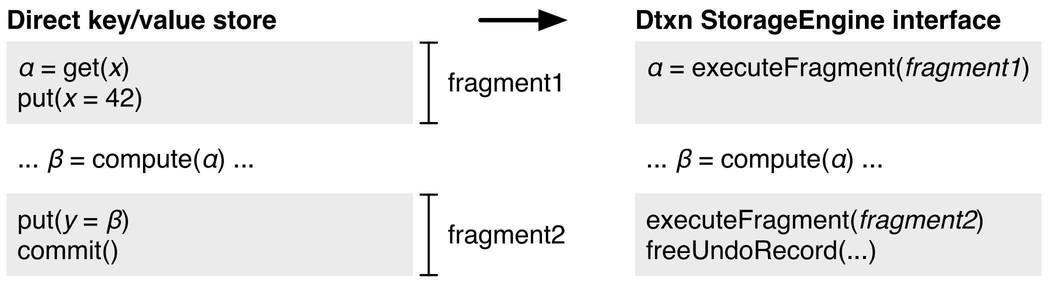


Figure 2-2: Example key/value store code transformed to use the StorageEngine interface

of multiple operations called *fragments*. A fragment is an application-supplied operation encoded as a byte string that will be passed to the storage engine. The storage engine itself does not need to be aware of transactions, as Dtxn manages concurrency by only executing one transaction at a time, discussed in detail in Chapter 4. An example of how an application transaction using a key/value interface would be translated to use the StorageEngine interface is shown in Figure 2-2.

This interface is not useful by itself, as it provides no benefits over using the storage engine directly, and is less convenient. However, it allows Dtxn to implement transactions, replication, and logging, without needing to understand anything about the operations. This makes it easier for backend developers to build new storage engines. Most importantly, this interface allows Dtxn to build a distributed storage system by combining many storage engine instances together. Each storage engine stores a logical subset of the entire system's data, called a *partition*. Each partition is independent, and must be small enough to be served by a single machine. To make a partition accessible over a network, support transactions and make changes durable, Dtxn embeds a storage engine in a *partition server*.

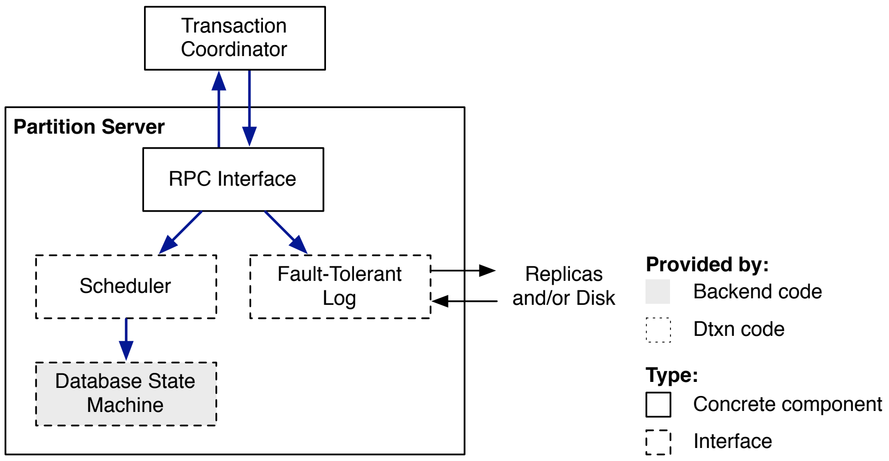


Figure 2-3: Partition server software components

2.2 Partition Servers

The *partition server* is the process that stores a partition of data by embedding a storage engine. Note that a partition in Dtxn is a logical part of the data stored in the system, while a partition server is a single process that stores a partition of data. Since Dtxn supports replication, a partition may actually be stored in multiple servers. A partition server provides three essential features: network communication, durability, and transactional concurrency control. To provide network communication, the partition server implements a protocol for passing fragments to the storage engine. This protocol is essentially an RPC interface for the storage engine interface shown in Figure 2-1. The partition server contains three components: the scheduler, the fault-tolerant log, and the storage engine, as shown in Figure 2-3. It passes requests and responses between these components and the network. The result is that backend developers only need to provide an implementation of the simple storage engine interface, and Dtxn makes it accessible to network clients, provides transactions and replicates it across multiple storage engines.

Although applications do not actually communicate directly with a partition server, for the sake of explaining the system, consider our key/value application that now wishes to communicate with a storage engine embedded in a partition server. Rather than simply calling the API directly, it takes the fragment of work it wishes to perform, and puts it in a generic Fragment message. This message is sent to the partition server, which then uses the storage engine interface to execute it. This generates results, which are then returned in a generic FragmentResponse message. The results are then extracted from the response and returned to the application. If there are multiple applications communicating with a given partition server, the server will receive and process the fragments in some arbitrary order.

This is sufficient for simple operations, but when an application wishes to be able to execute a transaction, more support is required. First, it must be possible to abort a transaction, for example if the application decides to abort it for some reason. Second, if a transaction is composed of multiple fragments of work, then interleaving fragments from other transactions may not be permitted. The partition server relies on a *scheduler* to en-

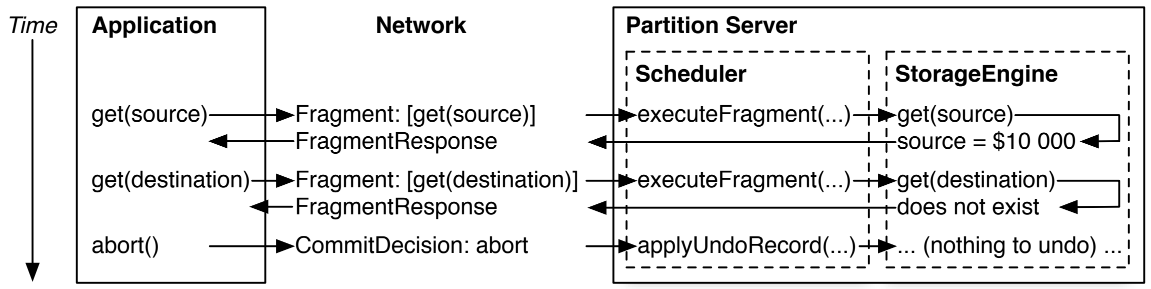


Figure 2-4: Example message flow for an aborted balance transfer

force serializable consistency. The scheduler is responsible for controlling how fragments from different transactions are interleaved by implementing the desired concurrency control scheme. The default scheduler uses speculative concurrency control, but Dtxn provides two other implementations, as described in detail in Chapter 4. Backend developers can also provide their own, if they have specialized needs.

In the case of our key/value storage system, consider an application that wishes to perform a classic balance transfer. It can do this by reading the source key, decrementing and writing the new balance, reading the destination key, and incrementing and writing the new balance. This can be executed as a transaction composed of four fragments (get(source), get(destination), put(source), put(destination)). Using a transaction ensures that we maintain a constant global balance. If at any time the application decides to abort, for example if the destination account does not exist, it sends an abort message, which causes the effects to be rolled back by invoking the applyUndoRecord() method of the StorageEngine interface. A sequence diagram showing this flow of messages between the application, the partition server, and the storage engine is shown in Figure 2-4. If the transaction wants to commit, it can either combine the commit message with the last put(), or send it as a separate message. This example shows a single application communicating with a partition server. When multiple applications are issuing transactions at the same time, the scheduler component is responsible for deciding how fragments from different transactions are interleaved, which is described in Chapter 4.

The partition server makes data durable by relying on a *fault-tolerant log*. This is an abstraction of a traditional write-ahead redo log. When a transaction commits, it must record a redo log record that can be used to recover the results of the transaction. In Dtxn, the redo log is a logical log that will always be applied to consistent checkpoints. By default, the redo record is the sequence of fragments that were executed for the transaction. This assumes that the operations that were performed are deterministic, meaning that re-executing them later on a checkpoint or a replica will always produce the same results. If the execution of the fragments for a given storage engine is not deterministic, a storage engine can optionally provide its own redo log records.

Before a transaction commits, its redo record is written to the log. After the log indicates that the fragments are durably stored, the transaction is considered to be committed. The assumption is that in case of failure, the fragments in the log can be re-executed, and the storage engine will end up in the same state as before the failure. This requires that storage engine operations are deterministic, which is the same requirement as a traditional

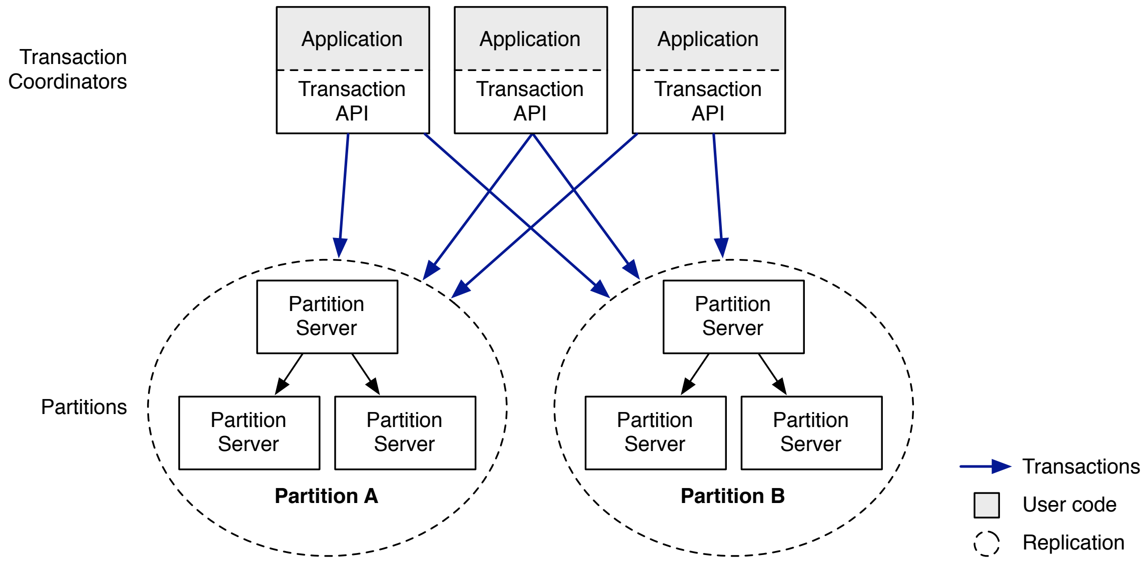


Figure 2-5: Transaction coordinators and partition servers

database redo log. However, they do not need to be idempotent, as the log implementation will enforce that they are only applied once. This is done by tagging them with sequence numbers and performing recovery from clean, consistent checkpoints. The default implementation uses this log to replicate the storage engine across multiple partition servers, described in Section 2.7. A traditional disk-based implementation and a null implementation that does nothing are also provided by Dtxn. Again, if the backend developers wish to use some alternative scheme, they are free to provide their own implementation.

A single partition is useful by itself as a standalone database server that is replicated for fault-tolerance. However, in order to store large amounts of data and to take advantage of multiple CPUs and machines, Dtxn provides access to a collection of partitions via the transaction API.

2.3 Transaction API

Applications that issue transactions to Dtxn are called *transaction coordinators*. Dtxn's transaction API allows applications to execute transactions that access multiple partitions. At the moment, we will assume that applications know which partitions store the data that they want to manipulate. The *metadata service*, described in Section 2.4, relaxes this restriction. The transaction coordinator library provided by Dtxn manages the distributed transaction, using the protocols described in Section 2.6. Applications call the transaction API, which communications with partition servers. A diagram showing the relationship between applications, the transaction API, and partition servers is shown in Figure 2-5.

Distributed transactions could potentially involve arbitrary data flow between participants, but Dtxn provides a limited request/response model composed of *rounds* of communication. Each round contains a set of fragments, where each fragment is sent to a different partition. Dtxn distributes the fragments and waits for the responses. After all the responses

```
structure Round:
  // Globally unique transaction id returned by Coordinator.GetTransactionId().
  integer transactionId;

  // Maps partition ids to the fragment for that partition.
  Map<integer, bytes> fragments;


interface Coordinator:
  // Starts or continues executing the transaction with fragments in round.
  // Returns a map of (partition id, response) pairs containing the responses.
  Map<integer, bytes> Execute(Round round);

  // Commits or aborts an active transaction.
  // shouldCommit is true if the transaction should commit, false if it should abort.
  void Finish(integer transactionId, boolean shouldCommit);

  // Returns a globally unique transaction id.
  integer GetTransactionId();
```

Figure 2-6: Simplified pseudocode for the Dtxn transaction API

have been received, they are passed back to the application. The application can then continue the transaction in any way it chooses. A simplified version of the transaction API is shown in Figure 2-6, and is described in Section 2.6. Applications create a Round structure to represent each round. To distinguish between rounds that belong to the same transaction and rounds that start a new transaction, the application must set Round.transactionId to the value returned by Coordinator.GetTransactionId(). The application adds fragments to Round.fragments, the submits it for execution using Coordinator.Execute(). It returns the results from each partition. To commit or abort the transaction, the application calls Coordinator.Finish().

In the balance transfer example from before, we described using four fragments to execute this transaction (get(source), get(destination), put(source), put(destination)). Assuming the application knows which partitions stores the source and destination accounts, we could use the transaction API to execute this transaction as four rounds, each one composed of the single fragment from before. However, it would be more efficient to execute this transaction in two rounds: the first containing both get() operations, to retrieve the source and destination balances. After the first round, the application can verify that the two accounts exist and have sufficient balances for the transfer. At this point, it can either abort the transaction, or can execute a second round containing the two put() requests. Figure 2-7 shows a visual grouping of these fragments into rounds. Note that if the source and destination accounts happen to reside on the same partition, the application will need to combine the get() and put() requests appropriately. In this case, the first round will contain one fragment containing a request to fetch both accounts (get(source, destination)),

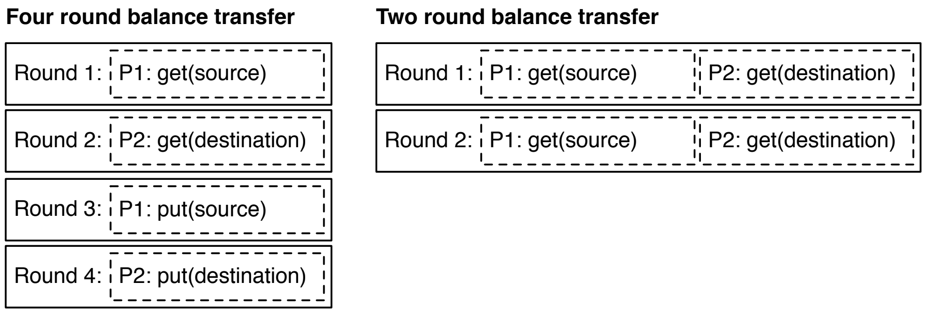

Four round balance transfer
Round 1: P1: get(source)
Round 2: P2: get(destination)
Round 3: P1: put(source)
Round 4: P2: put(destination)
Two round balance transfer
Round 1: P1: get(source) P2: get(destination)
Round 2: P1: get(source) P2: get(destination)

Figure 2-7: Four round and two round versions of a balance transfer transaction

and similarly the second round contains one fragment with both updates (put(source=*x*, destination=*y*)).

Limiting transactions to execute in rounds simplifies the implementation significantly, on both the coordinator and the partition server. Since a transaction can only execute one round at a time and each partition will only execute one fragment in a round, it naturally avoids the issue of how to manage concurrency inside a transaction if one participant is asked to perform two units of work at the same time. The limit of one fragment per partition per round forces applications to combine requests. This is somewhat inconvenient, as Dtxn could provide a generic "multi-operation" fragment. However, it also encourages applications to combine multiple requests into a single fragment, which is, in general, more efficient as it reduces the number of network messages. This is not a significant problem because most applications create a friendlier interface on top of the Dtxn interface. For example, for a key/value store, the programmer interface should be an API that hides the fact that the keys are divided across partitions. Thus, this complexity is exposed to the backend developer, but not to applications that use the database.

The request/response model makes it easy for all data to be collected at the coordinator, which could be a performance bottleneck for some types of operations. To avoid this, it is possible to use the transaction fragments as metadata to describe how data should be communicated between participants, then arranging for the participants to communicate directly. For example, a fragment could tell partition *x* to send data to partition *y*, while the fragment for partition *y* instructs it to join the data from *x* with local data, and return the response. This type of communication has not been required for the applications we have built, and so Dtxn does not include any support for this pattern.

The primary limitation imposed by the Dtxn round model is that partitions cannot decide to add participants to the distributed transaction, which is permitted by the "tree of processes" transaction model [62], typically assumed to be used by two-phase commit protocols and implementations, such as the X/Open XA interface [96]. In this model, participants in the distributed transaction can be arranged hierarchically in a tree, so that a processes can act as both a participant and a coordinator by delegating work to other processes. Instead, in Dtxn, only the coordinator can add or remove participants. A consequence of this is that a partition cannot easily be the coordinator for a transaction, since it cannot add participants to a transaction. While this does limit some optimization opportunities, this has not been significant for the transactional applications we have examined.

This issue is discussed in more detail in Section 3.3, as it was an issue when integrating Dtxn with VoltDB.

Applications send fragments to partitions, and not directly to individual partition servers. This abstraction permits Dtxn to replicate partition servers and transparently handle failures. It additionally allows Dtxn the freedom to place partitions wherever it decides it is appropriate, rather than hard coding the location of individual servers. This is the basis for Dtxn's support for migrating data between partitions, described in Chapter 5.

Transaction coordinators must record a two-phase commit log, to make distributed transactions durable. This means that if a coordinator fails, then the transactions that it is processing may be aborted. In rare cases, if the coordinator becomes unavailable when some transactions are in the "prepared" state, then the classic two-phase commit "blocking" problem occurs, and a transaction will be stuck until the coordinator comes back. Dtxn's API supports recovering transactions that are in this state. However, coordinators only ever communicate with partitions, and not with each other. This means that coordinators can be added as needed to handle the load, as there is no distributed state to be managed.

While developers using the transaction API must be aware that the Dtxn system involves accessing data across multiple partitions, it still manages some problems. Rather than addressing requests directly to physical servers, applications issue fragments to logical partitions. This allows Dtxn to manage the server location and handle failover in a transparent way. The API also implements distributed transactions, which provides serializable consistency, atomicity, and durability. Finally, it provides some basic partition membership and metadata support.

Up to now, we have assumed that the application knows all the partitions that exist, and what data is stored in each partition. Since Dtxn is unaware of the details of the underlying storage engines, applications must be responsible for determining how to direct a high-level application operation (e.g. a balance transfer) to the underlying partitions. However, the transaction API provides information about the available partitions to assist with this. It provides a list of the unique identifiers for each partition, as well as a small amount of opaque metadata (current limit of 1 MB) that can be used by the application as it sees fit. For example, in the key/value storage system, the metadata could store the range of keys stored in each partition. Then when an operation is requested, the application can consult this list of key ranges to determine which partition stores the requested key.

The partition metadata is stored in the *metadata service*, described in the next section. When a transaction coordinator starts, it must be provided with the addresses for the metadata servers. It then contacts one of these servers to obtain a list of the available partitions and their associated partition metadata. At this time, it also obtains a unique client id from the metadata server, which permits it to generate unique transaction ids of the form (*client id*, *sequence number*). The coordinator then establishes connections to each partition server. As part of the RPC protocol used to communicate between each component, the coordinator will transparently re-establish communication with any partition that fails. Since partitions are replicated, this may involve switching servers.

2.4 Metadata Service

The metadata service is a service provided by Dtxn that stores a list of each partition in the system plus some application-specific data, and is also used to store membership information for the replication protocol used by partition servers. While the details of the replication protocol are described in Section 2.7, two critical pieces of information must be stored: the membership of the replicated group and the identity of the current master. This information must be managed in a strongly consistent way, as the correctness of the replication protocol depends on it. Additionally, it must be highly available because if it goes down, the primary/backup protocol will halt on failure. Thus, this service runs a consensus algorithm to manage this information. We use Apache Zookeeper [50], an existing service which provides a consensus algorithm and the primitives we need.

The metadata service stores a list of all partitions, plus a small amount of application-specific metadata for each partition and a version number. This metadata is available to applications via the transaction API, and can be used to determine how to route operations to partitions. For example, it could contain the set of table ranges stored on each partition. The transaction coordinator sends the metadata version number along with each request to the partition. The partition server validates that the version number matches the current version. If it does not match, the transaction coordinator will need to abort the transaction, refresh the partition metadata and restart the operation.

From a correctness perspective, the partition metadata does not need to be maintained in a strongly consistent way because each partition stores the authoritative record about the data that it is responsible for, and the version numbers will ensure that transaction coordinators and partition servers agree on the metadata used to route queries. This has the advantage that redistributing data does not involve the metadata service. Thus, the metadata service really just stores a centralized cached copy, providing a convenient place for transactions coordinators to access this data for the entire system. Dtxn keeps it up to date as best as possible, by pushing changes from the partition servers.

In our key/value store example, consider a system that initially is composed of a single partition responsible for the entire key space of all possible byte strings, the interval ["", ∞). This partition has id x and version 1. A transaction coordinator starts and reads this information from the metadata service and caches it. This coordinator executes a number of transactions on the system. At some point, the system grows, so the partition is split in half. A new partition y is created, and half the data is moved to it using live migration, described in Chapter 5. Partition x is now responsible for the interval ["", "n") with version 2, and partition y is responsible for ["n", ∞) with version 1. As part of the migration, the partition servers send their new metadata and version numbers to the metadata service. When the transaction coordinator starts a transaction to fetch key "next", it sends the request to partition x with version 1. The partition server will notice that the version number does not match and reject it. This causes the transaction coordinator to reread the system metadata from the metadata service. It discovers that partition y is now responsible for the key "next", so it sends the request there where it is processed correctly.

For some applications the contents of the request itself is sufficient to validate that the query was sent correctly. This is true for requests that match the attributes used to partition the data, such as our key/value store example where the partition server can verify that it

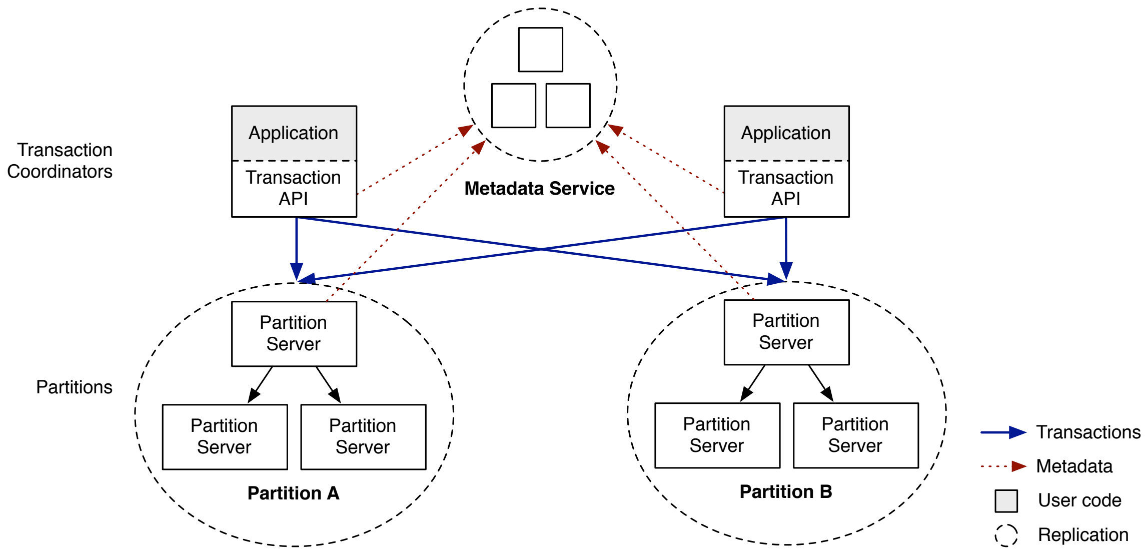


Figure 2-8: Dtxn system with two replicated partitions

is responsible for the keys it is being asked to read or write. However, for more advanced systems that can fetch data based on attributes that are not used to partition the data, this is not true. Consider extending the key/value store to be able to search for substrings inside the value. If the system wants to query for the substring "sub" it must send the query to all partitions. In the previous example, if the coordinator was out of date and only sent it to partition *x*, it might miss values stored on partition *y*. However, the version number will detect the case where the coordinator was out of date, causing it to discover the new partition *y*.

This metadata service is not involved in processing transactions, so it is not heavily loaded. It is only accessed when transaction coordinators start and when partition servers fail. However, it does need to be highly available. If it goes down, new coordinators cannot start, as they cannot get the list of partitions. Additionally, the replicated partitions cannot reconfigure their group membership, which means a partition will halt if any partition server fails when the metadata service is not available. A diagram showing all the processes in a complete Dtxn system is shown in Figure 2-8.

2.5 Optional Stored Procedure Client Interface

In order to reduce network round-trips and improve performance, Dtxn is optimized for executing single transaction stored procedures. Applications make a request to execute a named procedure with a set of arguments. This procedure is composed of arbitrary code interleaved with data accesses. The applications we have built using Dtxn have all provided similar application interfaces. Hence, it made sense to provide a generic stored procedure client interface that could be used for all of them.

The stored procedure interface provides a simpler RPC interface that can be easily implemented in other programming languages, rather than being required to call the somewhat complex C++ transaction API. It also isolates client applications from the database.

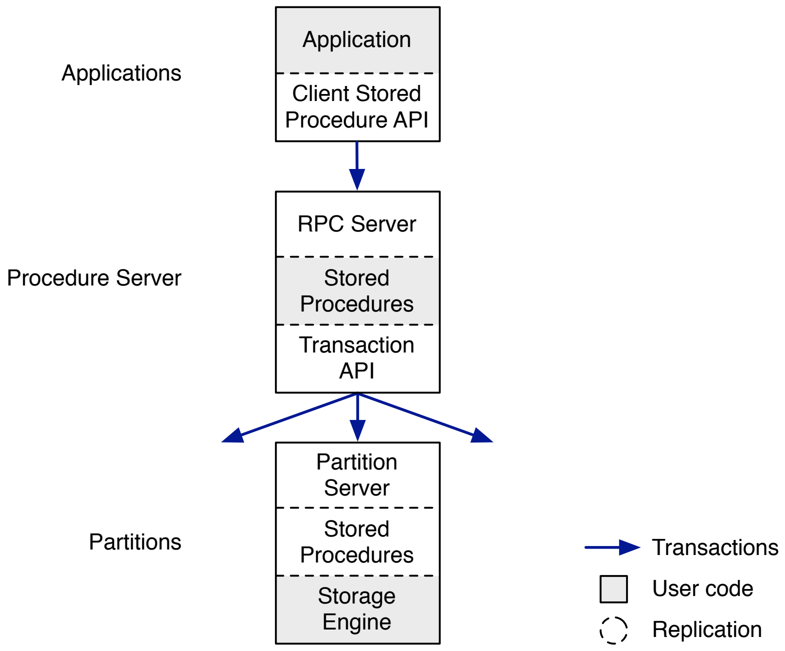


Figure 2-9: Layers of the Dtxn system with the stored procedure interface

It permits the stored procedures to be modified without changing the client interface. This also permits application code to be executed as closely to the data as possible. If the stored procedure accesses a single partition of data, then this interface provides a mechanism to execute the stored procedure directly on the partition server hosting the data. For these reasons, this is the preferred way to build a Dtxn system.

The Dtxn stored procedure interface is composed of a minimal client API that takes a stored procedure name and arguments. This request is first received by a new process: the *procedure server*. The procedure server first consults the stored procedure metadata to determine if this procedure accesses a single partition. If it only needs data on one partition, the request is forwarded to that partition to be executed close to the data. Instead, if the transaction must access multiple partitions, the partition server invokes the stored procedure provided by the backend developer. The stored procedure then calls the transaction API as before. At the partition servers, an implementation of the StorageEngine interface is provided that wraps the application's actual storage engine. This wrapper allows local stored procedures to be invoked directly on the partition's data, or fragments for distributed transactions to be processed as before. This means the Dtxn system is now composed of three layers: client applications, transaction coordinators, and partition servers, as shown in Figure 2-9. The transaction coordinators and partition servers can be considered part of the Dtxn system, while the client applications are independent of it.

The client application programming interface is shown in Figure 2-10. The client stored procedure implementation provides connection management and load balancing. Procedure servers register with the metadata service when they start. When the clients start, they get the list of available transaction coordinators from the metadata servers. This list is periodically refreshed in order to detect new coordinators. The client then randomly chooses a coordinator for each request, in order to approximately load-balance requests across all

```
interface StoredProcedureClient:
  // Calls a stored procedure on the Dtxn system.
  // procedure_name is the name of the stored procedure.
  // arguments is a list of arguments that can be serialized
  // Returns the serialized results of the stored procedure
  bytes Invoke(string procedure_name, List<Object> arguments)
```

Figure 2-10: Client API for Dtxn stored procedures

coordinators. Since the client stored procedure API forces callers to manually manage naming and serialization, most implementations write a thin user-friendly interface around this API that makes it look more like a local function call, by serializing and deserializing native types to and from byte strings.

2.6 Distributed Transaction Protocol

In this section, we describe the protocol used to execute transactions. We defer discussing concurrency control until Chapter 4. Here, we only consider a system that is processing a single transaction at a time.

When a transaction begins, Dtxn assumes that it may access any partition with any number of rounds of communication. We call this a *multi-partition transaction*. However, Dtxn is optimized for *single partition transactions* that access one partition using one round of communication, like a stored procedure invocation. If the application specifies that a transaction is a single partition transaction, Dtxn handles it differently from multi-partition transactions.

2.6.1 Single Partition Transactions

The single fragment in a single partition transaction is forwarded directly to the partition server responsible for the data. The partition server is responsible for all concurrency control and durability for this transaction. This ensures that if a workload contains only single partition transactions, there is no communication between partitions, and thus the system can scale by adding more machines. The partition server receives a transaction, executes it from start to finish, logs it using the fault-tolerant log interface and returns the results. This is very efficient. Execution with concurrent transactions is described in Chapter 4.

Requests to partition servers are sent as a Fragment message, defined in Figure 2-11. Single partition transactions send a request with state set to FINISHED_LOCAL_COMMIT.

2.6.2 Multi-Partition Transactions

The coordinator executes multi-partition transactions by sending out the fragments and collecting the responses for each round. The commit or abort decision is decided using two-

```
enumeration FragmentState:
  // The transaction may execute more fragments at this partition.
  ACTIVE
  // The transaction is done and this is the only participating partition. The commit
  // is performed locally.
  FINISHED_LOCAL_COMMIT
  // The transaction is done at this partition, but there may be other participants.
  // The partition should prepare for a two-phase commit.
  FINISHED_2PC

structure Fragment:
  // Globally unique transaction identifier.
  integer transactionId
  // Application-specific request to be processed as part of the transaction.
  bytes request
  // State of this partition after the fragment has been executed.
  FragmentState state
```

Figure 2-11: Fragment message definition

phase commit (2PC). In the general case, the state field of the Fragment message is set to ACTIVE, which indicates that more requests may be made as part of this transaction at this partition. Applications can optimize the commit protocol by specifying that partitions are finished as soon as they can determine this information. In this case, some fragments that specify state = FINISHED_2PC can be sent, even while other partitions are still ACTIVE. However, after a partition has been set as finished, it is not permitted to receive more work as part of the transaction. Piggybacking the prepare message along with the request in this way is the early prepare optimization [88]. If a transaction is finished with a partition, but has no work for it, it sends a fragment with an empty request field, which only prepares the transaction.

When a partition receives a prepare request, as indicated by state = FINISHED_2PC, it prepares to commit. It makes its part of the transaction durable using the fault-tolerant log before returning the response to the coordinator. We also support the two-phase commit read-only optimization [88]: if the partition was read-only for this transaction, it can immediately commit the transaction and the coordinator does not need to send the final commit or abort vote. This allows the partition to begin executing other transactions early, which reduces contention. It also reduces the number of network messages. However, this optimization must be applied carefully, as in some circumstances it can cause locking to violate serializability because locks are released early, violating the two-phase locking rule. However, it works correctly when used at the very end of a transaction, or with other concurrency control mechanisms, as discussed in Chapter 4.

When the transaction has finished and the coordinator has prepared and collected all the votes from each partition, it writes the result to its local log and sends the commit or abort decision to all partitions. This introduces the classic two-phase commit blocking problem, where if the coordinator crashes, all prepared partitions must wait for it to come back before they can finish the transactions. To solve this, we intend to replicate the coordinator log using our replication protocol. However, our current prototype does not implement this feature.

There are two special types of transactions that are worth discussing. The first type is single-round multi-partition transactions. These transactions access multiple partitions with a single round of communication, and are known to be finished with all partitions before the round begins. In this case, the application sends a single batch of fragments with state = FINISHED_2PC. This is the most efficient kind of multi-partition transaction, as it involves only two messages to each partition: one for the fragment, and and a final response for the commit/abort decision. The original H-Store paper called these one-shot transactions, and noted that they appear in many OLTP applications [95]. Speculative concurrency control can handle these transactions very efficiently, as described in Chapter 4.

The second type is multi-round, single partition transactions. In this case, the transaction involves multiple rounds of communication with a single partition. At the end of the transaction, the coordinator can recognize that only one partition was accessed. Thus, it commits the transaction by sending a final fragment with FINISHED_LOCAL_COMMIT.

2.7 Replication Protocol

While Dtxn provides the flexibility for backend developers to select the technique they wish to use to make transactions durable via the fault-tolerant log interface, most applications use the provided replication protocol. In this section, we briefly describe the protocol that we use, and how it interacts with the transaction protocol described in Section 2.6. This protocol takes a *group* of processes and ensures that each member sees the same sequence of messages. We use a primary/backup protocol, where one process is designated the *primary*, and all others are *backups*. The identity of the primary and the backups is stored in the metadata service. This structure is used in many distributed systems, like Bigtable [21]. A good formal description is provided by the authors of the Niobe protocol [66]. Our version is very similar, so our description here is brief and informal. We only briefly describe failure handling, as it is complex and the failure-free case is much more important for the performance of a system.

In normal operation, all requests are sent to the primary. The primary orders them in some arbitrary way (e.g. the order they are received), then forwards them to the backups. The primary begins processing the request immediately, overlapping the execution with the replication delay. As soon as the backups receive the request, they return an acknowledgment back to the primary. They do not yet execute the request. After the primary has received all acknowledgments from the backups, the request is considered to be reliably stored. The request will only be lost if all replicas crash. At this point, the primary can respond to the client with the results. Backups will execute the request once they receive the subsequent request, which indicates that the previous request was replicated successfully. This avoids the backups needing to undo an operation if the primary fails and the operation is lost. Primaries, on the other hand, may have executed a transaction that was not successfully replicated. In this case, they cause themselves to fail and perform recovery. This protocol replicates a message in a single round of communication.

We do not discuss failure handling in detail, as it is complicated and the Niobe protocol paper describes it formally [66]. However, it is possible for replicas to be added to a running system, or for replicas that have fallen behind to synchronize their state with the current group. This involves copying a checkpoint from an existing replica then replaying the log of operations.

The replication protocol is designed so the metadata service is only involved when changing the replica group membership, or when a failure is detected. The metadata service only stores the identity of the primary, and the identity of the backups. It does not store any data. Using a primary/backup protocol with a metadata service like this is a design trade-off between the types of failures that can be handled, the cost of replication, and complexity. One alternative would be for partitions to run a consensus protocol directly. However, this requires a minimum of three replicas, the number of replicas must be odd, and a majority must be available to process transactions. With a primary/backup protocol, any number of replicas can be used. Assuming the metadata service is available, the group is available as long as one replica is functioning. This decreases the cost of replication, compared to a consensus protocol. However, it relies on the metadata service to be available in order to handle failures, and so this service must be engineered to be highly available.

An alternative is to add *witnesses* that are part of the replicated group but are only used when recovering from a failure [64, 87]. This permits a group to have any number of "active" replicas, just like a primary/backup protocol. In case of a failure, the witnesses and active replicas are used to reconfigure the group membership as appropriate. This is theoretically superior to the master election service approach, since it depends on fewer processes to be functioning. However, the consensus protocol itself is more complicated, as it must support witnesses and reconfiguration. We chose the option that is simpler to implement, since these protocols are challenging to get right [20]. This allows us to leverage the existing consensus protocol implemented in Zookeeper. We leave modifying Dtxn to use witnesses to future work.

The partition server integrates the replication protocol with Dtxn's transactions by replicating each partition server log record. Once the log record is replicated, it is considered durable. For example, when a commit request starts being processed on a partition server, the partition server records a redo record using the logging interface. When replication is being used, the logging implementation replicates the redo record, so each backup will apply and stay synchronized with the primary. Once all replicas acknowledge the message, the commit completes. Similarly, for distributed transactions, the partition server writes both a prepare record and the final commit/abort record to the log. Both of these need to be replicated in order to survive failures.

Our replication protocol only permits one pending request at a time, in order to simplify the protocol and the failure recovery. Thus, using group commit is critical for getting good performance. We use a simple policy: when there are no replication requests pending, we immediately send a single log record as a single replication message, to minimize latency. While that request is pending, we queue any subsequent log records. When the replication request has completed, we group together all pending log records and replicate them. This has been sufficient to ensure that replication message processing is a small fraction of the CPU time spent processing requests.

Chapter 3

Applications

We have used Dtxn to build four different database systems. Each of these has different properties, highlighting Dtxn's flexibility. The four systems are a simple in-memory key/value store, a specialized TPC-C system, a main-memory database system based on the H-Store design and the VoltDB open source code, and Relational Cloud, a more traditional disk-based distributed database. In this chapter, we describe these implementations in detail.

3.1 Key/Value Store

Like many key/value storage systems, our key/value system stores byte string keys along with a byte string value. Thanks to Dtxn's support for transactions, we extended this simple model by adding support for multi-key transactional updates. The database supports a single operation based on the minitransactions introduced by Sinfonia [3]. An operation request is composed of a set of key/value pairs to compare, a set of keys to read, and a set of key/value pairs to write. The operation executes by comparing the current value of all keys in the compare set with the value provided in the request. If they all match, then the keys in the read set are read, and the keys in the write set are written. We chose this operation because it only requires one round of communication with the partitions, but it can be used to build more complex operations. For example, some keys can be used as locks, or the compare operation can be used to implement optimistic concurrency control.

In general, a minitransaction operation may need to access every partition in the system. However, the hope is that most requests will only access a few keys on a single partition. In order to get better locality, keys are sorted lexicographically and partitioned into ranges. Dtxn can process operations that access a single partition as efficiently as a key/value store that does not support transactions. These operations only require a single message to the partition, and are executed without concurrency control overhead. Operations that access multiple partitions, on the other hand, require one message to each participating partition to execute the request, then a final message to each partition with the result of the two-phase commit. This involves more messages, and some small additional overhead for concurrency control. Since these minitransactions cannot involve multiple rounds of communication, this application does not test all the possible types of distributed transactions

This application relies on Dtxn's concurrency control mechanisms. It supports speculative concurrency control as well as traditional locking and optimistic concurrency control techniques. We used this flexibility to compare the performance of these techniques in Chapter 4. Since minitransactions involve a single round of communication, even in the general case, this application is ideally suited for speculative concurrency control, so that is the default choice.

3.2 TPC-C

In order to test an application that has more complex transactional requirements than our simple key/value store, we created an implementation of the TPC-C transaction processing benchmark [27]. Our implementation began as a single node version [95], and we distributed it using Dtxn. This implementation is written in C++ and stores all the data in main-memory data structures that are part of the C++ standard library. Each table and index is represented as either a binary tree or hash table, as appropriate for the type of accesses required. All queries are manually implemented in C++. As a result, it has very high performance, but is difficult to modify.

The TPC-C benchmark models the OLTP workload of an order processing system. It is comprised of a mix of five transactions with different properties. The data size is scaled by adding warehouses, which adds a set of related records to the other tables. We partition the TPC-C database by warehouse [95]. We replicate the items table, which is read-only, to all partitions. We vertically partition the stock table, and replicate the read-only columns across all partitions, leaving the columns that are updated in a single partition. This partitioning means 89% of the transactions access a single partition, and the others are simple single round multi-partition transactions.

Our implementation tries to be faithful to the specification, but there are two differences. First, we reorder the operations in the new order transaction to avoid recording an undo record to handle user aborts. Second, we change how clients generate requests. The TPC-C specification assigns clients to a specific (warehouse, district) pair, and requires a think time between requests. Thus, there is a strict relationship between the number of warehouses and the number of requests per second. In order to vary this, we eliminate the think time and change the clients to generate requests for an assigned warehouse but a random district.

This application, like the key/value store, only uses single round distributed transactions. However, it has much more complex and expensive operations that involve multiple index traversals, reads and writes. As a result, each operation involves significantly more computation in the "application code" than the key value store. Additionally, each transaction acquires many locks in a complex order. As a result, the locking and optimistic concurrency control implementations have significantly different performance. In particular, there are local and distributed deadlocks when using locks.

3.3 H-Store/VoltDB

The design of the H-Store database system [95] is being commercialized as VoltDB [100]. This system executes stored procedures written in Java on top of a main-memory storage engine written in C++. The current VoltDB implementation includes a very simple distributed transaction protocol. When executing a distributed transaction, it stops processing transactions on all partitions, executes the distributed transaction, then resumes execution. As a result, distributed transactions have very poor performance. However, distributed transactions are also very rare in the applications that are currently using VoltDB. Therefore, this very simple "blocking" concurrency control scheme is sufficient, as the system rarely executes distributed transactions.

As part of the research on the H-Store design, we wanted to perform experiments using the full SQL support provided by VoltDB. However, we needed finer-grained distributed transactions and support for speculative execution. To do this, we replaced VoltDB's existing transaction protocol with Dtxn. Unfortunately, this was not trivial, despite the fact that Dtxn was designed for systems like VoltDB. One major challenge is that VoltDB is designed assuming that for distributed transactions, the coordinator executing the Java stored procedure will be one of the partitions that is participating in the transaction. This means the stored procedure can access data on the local partition without network round trips. Dtxn, on the other hand, assumes that the distributed transaction coordinator runs on a different node, and is not a participant in the transaction. This works well if the Java stored procedure runs on a coordinator node, but now every data access involves a network round-trip. This will be less efficient for certain types of distributed transactions, where there are multiple rounds of communication with a single partition, and occasional rounds of communication with other partitions. Dtxn's design is somewhat simpler and easier to implement, but is less efficient in this case.

We were able to work around this limitation by abusing the transaction system to "transfer" coordination to the required partition. The node that first receives the transaction request starts the transaction with a single fragment, directed to the partition that should be the coordinator. This partition immediately responds to this fragment with an empty response. Once this is completed, the partition can coordinate the transaction as normal. This adds some additional overhead and delay to beginning a transaction, but was able to get the system working without any code changes to Dtxn.

In the future, we plan to examine how much this actually matters for this application. There are two possible solutions. The first is to execute the stored procedure for a distributed transaction on a coordinator node, making remote accesses for every data item. This has the advantage that it matches Dtxn's design, but has the disadvantage that all data accesses are remote. For applications with a single round of data accesses, this will not matter. However, if the application involves many rounds of accesses to a "home" partition, with a few accesses to other partitions, it would significantly reduce network round-trips to execute the stored procedure on the home partition. The other argument is that distributed transactions are rare in most Dtxn applications, so again making these transactions faster will not have a significant impact on overall throughput. Additionally, this problem really only appears for certain types of distributed transactions, where there is a strong affinity for a single partition. It is unclear how often these kinds of transactions appear. The second alternative is to re-design Dtxn's mechanism to permit partitions to also coordinate transactions. This will require a significant redesign of the distributed transaction protocol, and requires careful thought about how it integrates with concurrency control.

While we do not discuss this application further in this thesis, this demonstrates that Dtxn can be used with complex transactions that involve multiple rounds of communication.

3.4 Relational Cloud

Our final application used Dtxn to build a traditional distributed disk-based database by distributing data across single node databases. This was the prototype for our "databases as a service" research project, Relational Cloud. We used MySQL instances as the individual storage node. Since MySQL provides durability via write-ahead logging and concurrency control via locking, the partition server in Relational Cloud does not use many Dtxn components. We implement the scheduler interface directly to avoid using Dtxn's concurrency control, and typically use the null logging implementation to avoid additional durability overhead. If replication is used, the replicas replay the update statements in commit order in a single thread. This is very similar to MySQL's own replication implementation, although in Dtxn replication is synchronous while typically with MySQL it is asynchronous.

The main components that Relational Cloud uses from Dtxn are a distributed transaction coordinator and the partitioning metadata service. Most importantly, it uses Dtxn's support for migrating data between partitions. In particular, the live migration experiments in Chapter 5 were performed using the Relational Cloud prototype.

Chapter 4

Speculative Concurrency Control

Databases rely on concurrency control to provide the illusion of sequential execution of transactions, while actually executing multiple transactions simultaneously. However, researchers have long observed that for main-memory databases, concurrency control is expensive relative to the cost of processing queries [37, 36, 101, 95]. One study estimated the cost of concurrency control to be approximately 30-40% of the CPU time, as shown in Figure 4-1.

Databases execute multiple transactions at a time in order to make use of resources that would otherwise be idle when a transaction stalls. In a traditional database, there are two primary sources of stalls: waiting to read data from disk and waiting for the next command from the application. Both of these can be avoided by storing data in main memory and executing transactions as stored procedures. In this case, the system can execute a single transaction from start to finish without any stalls. As a result, no concurrency control is needed as there is no concurrency: only one transaction executes at a time. This is very efficient, as the CPU is fully utilized, and thus executing more than one transaction at a time would not provide any benefit.

Databases also use concurrency to utilize multiple CPUs, by assigning transactions to different threads. However, recent work shows that this approach does not scale to large numbers of cores, due to the synchronization required to share data between threads [84]. Instead, Dtxn relies on the database to be divided into a number of partitions, then runs one thread per partition. This permits the system to use multiple CPUs or multiple independent computer systems, without any shared data and thus without needing any additional coordination. If a transaction is composed of a single stored procedure that only accesses data in a single partition, which we call a single partition transaction, then it can be executed without stalls. Across the entire system, each partition can independently execute single partition transactions in the order that they are received. This can fully utilize many CPUs across many machines, without the overhead of concurrency control. In this chapter, we show that using this approach is up to two times faster than a lightweight locking implementation.

However, this relies on an application being "perfectly partitionable," such that each transaction touches exactly one partition. Many real applications have some transactions that span multiple partitions. This means that that partitions can no longer execute independently, and some form of concurrency control is needed to ensure these multi-partition transactions execute correctly and make effective use of computing resources.

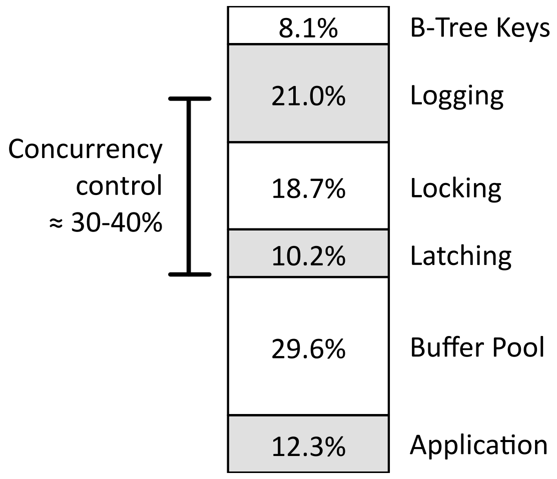


Figure 4-1: CPU consumption for TPC-C New Order transactions on Shore [45]

In this chapter, we propose a new form of concurrency control, called *Speculative Concurrency Control*, which is designed for these "imperfectly partitionable" applications. These workloads are composed mostly of single-partition transactions, but also include some multi-partition transactions that impose network stalls. The goal is to allow a processor to do something useful during a network stall, while not significantly hurting the performance of single-partition transactions.

Most applications we have examined are imperfectly partitionable. For example, the TPC-C transaction processing benchmark, even when carefully partitioned, is composed of 89% single-partition transactions. The remaining 11% touch multiple partitions. Other applications follow similar patterns. They can be carefully partitioned so the common operations access a single partition, but other less common operations still need to access multiple partitions. Many-to-many relationships cause this kind of partitioning. The relationship can be partitioned so that accessing the relationship in one direction accesses a single partition. For example, consider users who can belong to multiple groups. If the data is partitioned by group, then accessing a group and all of its users accesses a single partition. However, when this is done, accessing the relationship in the other direction will need to access all partitions. In the users and groups example, if we wish to access all the groups that a specific user belongs to, we will need to query all partitions. Thus, many-to-many relationships that are partitioned so the common operations are single-partition accesses tend to require a small fraction of the workload to access multiple partitions.

In the remainder of this chapter, we describe how Dtxn performs concurrency control for a naive blocking scheme, speculative concurrency control, and a traditional two-phase locking approach. Since Dtxn separates concurrency control from the storage engine, we present experiments comparing the performance of these schemes.

4.1 Concurrency Control Schemes

4.1.1 Blocking

The simplest scheme for handling multi-partition transactions is to block until they complete. When the partition receives the first fragment of a multi-partition transaction, it is executed and the results are returned. All other transactions are queued. When subsequent fragments of the active transaction are received, they are processed in order. The last fragment of the transaction is marked so the the partition server can prepare for two-phase commit by making the transaction durable (see Section 2.2). Ideally, this last fragment will include work, so the two-phase commit prepare message is piggybacked along with the last unit of work. After the transaction is committed or aborted, the queued transactions are processed. In effect, this system assumes that all transactions conflict, and thus can only execute one at a time.

One complication is that multi-partition transactions must be executed in the same relative order on different partitions. If this is not enforced the system will deadlock. Consider a system with two partitions, P_1 and P_2. If two coordinators submit transactions A and B that access both partitions at approximately the same time, then P_1 could start A first and P_2 could start B first. The coordinator for A is left waiting for P_2's response and the coordinator for B is waiting for P_1's response, causing a deadlock. Any traditional deadlock resolution strategy could be chosen here. However, in this system deadlocks are very common because the granularity of the locks is an entire partition, and many transactions start work on multiple partitions in parallel, causing race conditions. Thus, we chose to prevent deadlocks by forwarding multi-partition transactions through a designated *central coordinator*.

The central coordinator is elected via the metadata service (see Section 2.4). It assigns a global order to the distributed transactions, and ensures that partitions execute transactions in this order. Although this is a straightforward approach, the central coordinator can become a bottleneck for multi-partition transactions. We discuss this further in Section 4.1.2.

The blocking concurrency control scheme is efficient if the workload is composed only of single partition transactions. Figure 4-2 shows a partition executing two single partition transactions. In this case, the transactions are executed back-to-back without any unused resources. However, multi-partition transactions introduce stalls. Effectively, if a multi-partition transaction arrives, the partition queues all other transactions until the active transaction finishes by committing or aborting. The multi-partition transaction must record an undo record to permit the transaction to be undone if it aborts. Concurrency control will never cause single partition transactions to abort. Thus, unless the storage engine decides that the operation must abort (e.g. due to a user abort or constraint violation), undo information does not need to be recorded, which improves performance. Pseudocode describing how each partition server executes transactions is shown in Figure 4-3.

The network stall caused by multi-partition transactions can introduce a performance bottleneck, even if multi-partition transactions only comprise a small fraction of the workload. On our experimental systems, described in Section 4.2, the minimum network round-trip time between two machines connected to the same gigabit Ethernet switch was measured using `ping` to be approximately 40 μs. The average CPU time for a TPC-C trans-

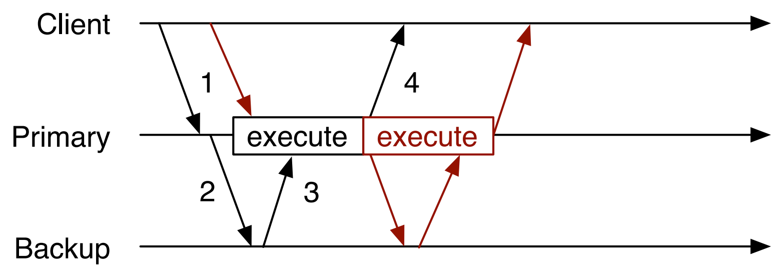


Figure 4-2: Timeline diagram of a partition executing two single partition transactions

```
Transaction Fragment Arrives
if no active transactions:
  if single partition:
    execute fragment without undo record
    commit
  else:
    execute fragment recording undo record
else if fragment continues the active multi-partition transaction:
  // another fragment for a transaction with multiple rounds of work
  continue transaction with fragment
else:
  queue fragment

Commit/Abort Decision Arrives
if abort:
  undo aborted transaction
execute queued transactions
```

Figure 4-3: Blocking pseudocode for the partition server

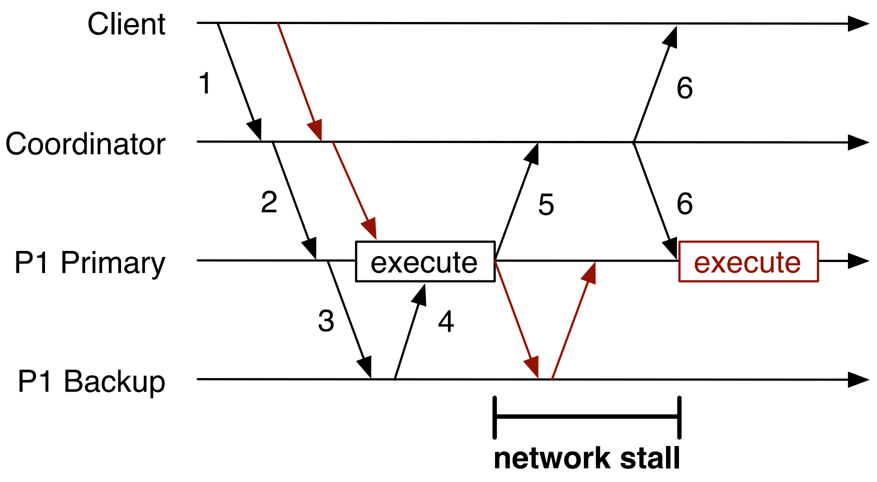


Figure 4-4: Blocking concurrency control executing two multi-partition transactions

action in our system is 26 μs. The simplest multi-partition transaction is composed of a single round of fragments sent to two partitions. In this case, each partition receives two messages: the transaction fragment and commit. Figure 4-4 shows one partition executing two single-round multi-partition transactions. In this case, there is a network stall between when the partition responds with the results (message 5) and the coordinator tells it to commit (message 6). The delay between the response and the commit message must be at least the network round trip time, which means that in that period the partition could execute two single-partition transactions. The challenge is to permit the storage engine to do useful work during this time without reducing the efficiency of single partition transactions.

4.1.2 Speculative Execution

Speculative concurrency control makes use of the network stall by speculating future transactions. More precisely, when the last fragment of a multi-partition transaction has been executed, the partition must wait to learn if the transaction commits or aborts. In the majority of cases, it will commit. Thus, we can execute queued transactions while waiting for two-phase commit to finish. The results of these speculatively executed transactions cannot be sent outside the database, since if the first transaction aborts, the results of the speculatively executed transactions may be incorrect. Undo information must be recorded for speculative transactions, so they can be rolled back if needed. In the typical case, the first transaction commits and all completed speculatively transactions are also immediately committed. Thus, speculation hides the latency of two-phase commit by performing useful work instead of blocking.

Aborts occur for three reasons. First, applications can include explicit "user" aborts. For example, when performing a balance transfer, the transaction may need to abort if the source account is overdrawn. In this case, the source account partition may decide to abort while the destination account partition is prepared to commit. The second source of aborts are failures of individual partitions. For example, if a distributed transaction involves two partitions, and one fails, Dtxn allows transaction processing to continue on the remaining partition by aborting the incomplete transaction. These failures can be due to hardware or software, such as a bug in the storage engine or transaction code. The third source is failures of transaction coordinators. Provided that they have not reached the prepare phase of two-phase commit, partitions can decide to abort transactions that are managed by failed coordinators. It is possible to avoid these sources of aborts, which can lead to a system that strictly orders transactions in advance [98]. However, this means the system cannot tolerate partial failure or bugs as gracefully as Dtxn.

Speculation produces serializable execution schedules because once a transaction t has finished all of its reads and writes and is simply waiting to learn if t commits or aborts, we can be guaranteed that any transaction t^* which makes use of t's results on a partition p will be serialized after t. However, t^* may have read data that t wrote. Thus, if t aborts, we must also abort t^* to avoid exposing t's dirty (rolled-back) data.

The simplest form of speculation is when the speculatively executed transactions are single partition transactions, so we consider that case first.

Speculating Single Partition Transactions

Each partition maintains a queue of unexecuted transactions and a queue of uncommitted transactions. When a transaction arrives, it is unexecuted, so it is placed on the unexecuted queue. The uncommitted queue is used for multi-partition transactions that are waiting for two-phase commit, or for speculated transactions. These transactions have been executed, but are waiting to learn if they can be committed or not. Thus, this queue is needed to allow them to be undone.

The head of the uncommitted transaction queue is always a non-speculative transaction. After a partition has executed the last fragment of a multi-partition transaction, it executes single partition transactions speculatively from the unexecuted queue, adding them to the end of the uncommitted transaction queue. An undo record is recorded for each transaction, which is extra overhead for single partition transactions. If the non-speculative transaction aborts, the uncommitted transactions must be undone in the reverse order (last in first out). Thus, each transaction is removed from the tail of the uncommitted transaction queue, undone, then pushed onto the head of the unexecuted transaction queue to be re-executed. If the non-speculative transaction commits, then the queue of uncommitted transactions commits. Each transaction is removed from the head of the queue and results are sent. When the uncommitted queue is empty, the system resumes executing transactions non-speculatively. For any subsequent single partition transactions that cannot abort, execution can proceed without the extra overhead of recording undo information.

Note that any time we have a queue of unexecuted transactions, the Dtxn scheduler can choose any transaction to execute next. This may execute transactions in a different order than the order that they were received, but this is a valid serializable order, as all these transactions were issued concurrently, so they can be executed in any order. This means that many policies could be used to select the next transaction. We have found that the best policy is to prioritize multi-partition transactions when there is no speculation in order to start them as soon as possible on all partitions. This avoids stalling other partitions when one partition has a large backlog of single partition transactions to execute, which maximizes the throughput for the entire system. When speculating, we choose to speculate single partition transactions where possible, since this allows them to be committed as soon as the multi-partition transaction commits.

As an example of how simple speculation works, consider a two-partition database (P_1 and P_2) with two integer records, x on P_1 and y on P_2. Initially, $x = 5$ and $y = 17$. Suppose the system executes three transactions, A, B_1, and B_2, in order. Transaction A is a multi-partition transaction that swaps the value of x and y. It executes in two rounds: the first reads the previous values of x and y, and the second writes the swapped values. Transactions B_1 and B_2 are single partition transactions on P_1 that increment x and return its value.

Both partitions begin by executing the first fragments of transaction A, which read x and y. P_1 and P_2 execute the fragments and return the values to the coordinator. Since A is not finished, partition P_1 cannot speculate B_1 or B_2. If it did, the result for transaction B_1 would be $x = 6$, which is incorrect for the sequence A, B_1, B_2. The coordinator for A eventually sends the final fragments, which write $x = 17$ on partition P_1, and $y = 5$ on partition P_2. After finishing these fragments, the partitions send their "ready to commit" acknowledgment to the coordinator, and wait for the decision. Partition P_1 can now begin speculative execution because A is finished. It executes transactions B_1 and B_2 with undo records and the results are queued. If transaction A were to abort at this point, both B_2 and B_1 would be undone and re-executed. Once P_1 receives the commit message for A, it sends the results for B_1 and B_2 to the clients and discards their undo records, assuming they have already completed. The final result is that $x = 19$ and $y = 5$, which corresponds to the order A, B_1, B_2.

This describes purely *local speculation*, where speculative results are buffered inside a partition and not exposed until they are known to be correct. Only single partition transactions and the first fragment of multi-partition transactions can be speculated in this way, since the results cannot be exposed in case they must be undone. However, we can speculate many multi-partition transactions if the coordinator is made aware of the speculation.

Speculating Multi-Partition Transactions

In order to speculate multi-partition transactions, the coordinator must be aware that a result is speculative. This is required so that coordinators do not return the final commit or abort result to the application until the speculation has been determined to be correct. Partitions add a *dependency* to result messages for speculated multi-partition transactions, indicating that this result is speculative and depends on a previous transaction committing. The key part is that this dependency means that the *coordinator* can make these decisions along with the partition server, whereas with local speculation the coordinator is unaware that speculation is happening, as speculation is confined to the partition server. We call this *distributed speculation*, as multiple processes in the Dtxn system are involved. Speculated single partition transactions are still buffered at the partition server until the previous multi-partition transactions commit. The pseudocode that formalizes the partition server logic for this scheme is shown in Figure 4-6.

To help explain how this works, consider the previous example where initially $x = 5$ and $y = 17$, except now the system executes A, B_1, then a new multi-partition transaction C, which multiplies both x and y by 2, then finally B_2. The system executes transaction A as before, and partition P_1 speculates B_1. The results for B_1 must be buffered locally as before.

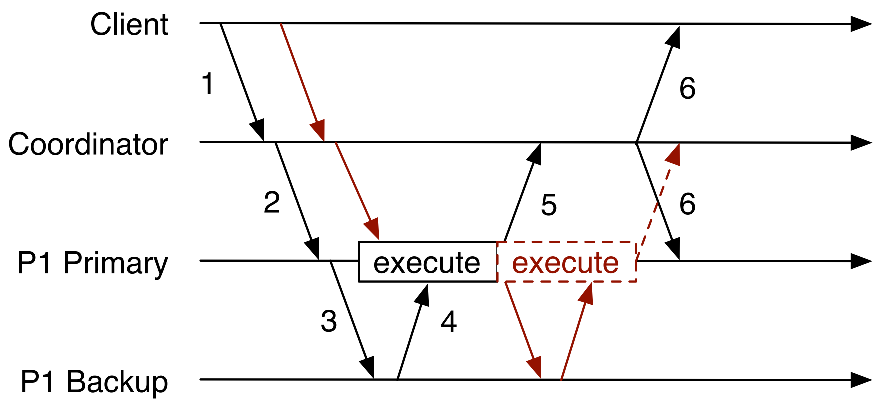


Figure 4-5: Speculative concurrency control executing two multi-partition transactions

Partition P_2 speculates its fragment of C, computing $y = 10$. With local speculation, it must wait for A to commit before returning this result to the coordinator, since if A aborts, the result will be incorrect. However, with distributed speculation, partition P_2 returns the result for its fragment of C, with a dependency on A. This informs the coordinator that it must wait for A to commit before these results are correct. Similarly, partition P_1 can speculate its fragment of C, computing and returning $x = 36$ with a dependency on A. Partition P_1 can also speculate B_2, computing $x = 37$. Again, it cannot result this result.

The coordinator for A and C could receive responses for C before it receives all the responses for A. The dependency from C on A will prevent it from returning the results to the application. However, after the coordinator commits A, it will examine all transactions that depend on it, which in this case is C. Since A committed, the speculative results are correct and C can commit, so the coordinator immediately sends the commit message for C. This saves a network round-trip and the time to execute C, improving throughput and decreasing latency. Figure 4-5 shows a timeline diagram of speculative concurrency control executing two multi-partition transactions, similar to this case. In the figure, after executing the first transaction the partition can immediately speculate the second transaction and return results to the coordinator.

If A had aborted, the coordinator would send an abort message for A to P_1 and P_2, then discard the incorrect results it received for C. As before, the abort message would cause the partitions to undo the transactions on the uncommitted transaction queues. Transaction A would be aborted, but the other transactions would be placed back on the unexecuted queue and re-executed. The partitions would then re-execute and resend results for C. The re-executed results would not be speculative, so the coordinator could handle it normally.

This scheme allows a sequence of multi-partition transactions that are composed of a single round of fragments to be executed without blocking. We call such transactions *simple multi-partition transactions*. Transactions of this form are quite common. For example, if there is a table that is mostly used for reads, it may be beneficial to replicate it across all partitions. Reads can then be performed locally, as part of a single partition transaction. Occasional updates of this table execute as a simple multi-partition transaction across all partitions. Another example is if a table is partitioned on column x, and records are accessed based on the value of column y. This can be executed by attempting the access on all partitions of the table, which is also a simple multi-partition transaction. As a third example, all distributed transactions in TPC-C are simple multi-partition transactions [95]. Thus, distributed speculation extends the types of workloads where speculation is beneficial.

There are a few limitations to speculation. First, speculation can only be used after executing the last fragment of a transaction. Although it works for transactions that require multiple rounds, it is less effective. These multi-round transactions require stalls between fragments, since it is not possible to speculate a transaction in the middle of another one.

Second, distributed speculation can only be used when the transactions come from the same coordinator. This is necessary so the coordinator is aware of the outcome of the earlier transactions and can cascade aborts as required. In the current implementation with a centralized coordinator for multi-partition transactions, this happens naturally. However, a single coordinator can become a bottleneck, as discussed in Section 4.1.1. If some other scheme were used to order distributed transactions when using multiple coordinators, each coordinator must dispatch transactions in batches to take advantage of this optimization. This requires delaying transactions, and tuning the number of coordinators to match the workload.

Speculation has the advantage that it does not require locks or read/write set tracking, and thus requires less overhead than traditional concurrency control. This means that storage engines only need to be capable of undoing transactions, and do not need to be aware of concurrency control or speculation. Dtxn handles all the logic for speculating, aborting, and re-executing transactions, as necessary. A disadvantage is that it assumes that all transactions conflict, and therefore occasionally unnecessarily aborts transactions.

Distributed Transaction Ordering

The central coordinator that we employ for both the blocking and speculative concurrency control schemes seems like it could potentially be a bottleneck. However, this should not be a problem for most real systems, since the coordinator only needs to handle a fraction of the messages, and there is a lot of room for optimization.

First, all single partition transactions are sent directly to the partitions, without going through the central coordinator. For the workloads we have examined, this accounts for 80-90% of transactions. This means the central coordinator only needs to handle a fraction of the workload. Second, our current implementation is single threaded, and thus only uses a single core. It currently can handle a maximum of around 10,000 transactions per second. A multi-threaded version should be able to easily use multiple cores, which should mean it should be able to handle somewhere around 40,000 to 80,000 transactions per second, depending on how parallelizable the ordering task turns out to be. At this level, this means the overall system would be processing somewhere around 800,000 transactions per second. This is sufficient for all but the very largest database systems.

Third, the central coordinator currently orders and forwards message requests to the backend partitions. This means that for each transaction, it must handle multiple messages. An alternative is that transaction coordinators could request timestamps from the central coordinator and handle the distributed transaction itself. This means the central coordinator would only need to handle one request and response per transaction. Additionally, transaction coordinators could batch these requests, so if the application using the transaction

```
Transaction Fragment Arrives
if uncommittedQueue.empty():
  // no active transactions
  if single partition:
    execute fragment without undo record
    commit
  else:
    execute fragment recording undo record
else if fragment continues active multi-partition transaction:
  // another fragment for a transaction with multiple rounds of work
  continue transaction by executing fragment
else:
    unexecutedQueue.queue(transaction)

// re-evaluate if we should speculate transactions: an active multi-partition
// transaction could have finished, or a new transaction could have arrived
while uncommittedQueue.tail() is finished locally:
  // policy can decide what transaction should be speculated next
  transaction = unexecutedQueue.dequeue()
  transaction.fragment.results = execute transaction.fragment recording undo record
  headTransaction = uncommittedQueue.head()

  if transaction.isMultiPartition() and
      transaction.coordinator == headTransaction.coordinator:
    // multi-partition with the same coordinator: use distributed speculation
    transaction.fragment.results.dependency = headTransaction.transactionId
    send transaction.fragment.results
  uncommitedQueue.queue(transaction)

Commit/Abort Decision Arrives
if abort:
  undo and re-queue all speculative transactions
  undo aborted transaction
else:
  while next speculative transaction is not multi-partition:
    // this stops committing transactions at the next multi-partition transaction:
    // the coordinator must send a commit message
    commit speculative transaction
    send results
execute/speculate queued transactions
```

Figure 4-6: Speculative concurrency control pseudocode for the partition server

API is multi-threaded, this could reduce the messages to fewer than one per transaction. Google's Percolator uses a central coordinator that batches requests in this way for their distributed snapshot isolation system, and they claim it can handle approximately 2 million timestamps per second on a single machine [85]. Thus, a Dtxn system using this approach could handle approximately 10-20 million transactions per second.

Finally, if this is not sufficient, then the transaction ordering task can be truly distributed by using loosely synchronized clocks [2]. Coordinators starting transactions would stamp them with the current time, t. Before starting a transaction, partition servers would wait until their local clocks are at least time $t+\varepsilon$, where ε is set to be greater than the largest possible clock skew plus one-way network delay. This period allows enough time so that with high probability, any transaction started at a time less than t will have been received, and thus will be started before t. In the rare case of an exceptional delay, and a transaction with timestamp t is received after the partition has started another transaction with a greater timestamp, it will return an error, causing the transaction to be restarted. This scheme has the advantage that it is truly distributed. However, it has the disadvantage that transaction coordinators must "reserve" their order at any partition that may be involved in a transaction when the transaction begins. Depending on the workload, this may involve extra empty "do nothing" messages.

Other schemes, such as using vector timestamps to causally order transactions, are also possible. We leave it to future work to evaluate these alternatives.

4.1.3 Locking

In the locking scheme, transactions acquire read and write locks on data items while executing, and are suspended if they make a conflicting lock request. Transactions must record undo information in order to rollback in case of deadlock. Locking permits a single partition to execute and commit non-conflicting transactions during network stalls for multi-partition transactions. The locks ensure that the results are equivalent to transaction execution in some serial order. The disadvantage is that transactions are executed with the additional overhead of acquiring locks and detecting deadlock.

We avoid this overhead where possible. When our locking system has no active transactions and receives a single partition transaction, the transaction can be executed without locks and undo information, in the same way as the blocking and speculation schemes. This works because there are no active transactions that could cause conflicts, and the transaction is guaranteed to execute to completion before the partition executes any other transaction. Thus, locks are only acquired when there are active multi-partition transactions.

Our locking scheme follows the strict two phase locking protocol. Since this is guaranteed to produce a serializable transaction order, clients send multi-partition transactions directly to the partitions, without going through the central coordinator. This is more efficient when there are no lock conflicts, as it reduces network latency and eliminates an extra process from the system. However, it introduces the possibility of distributed deadlock. Our implementation uses cycle detection to handle local deadlocks, and timeout to handle distributed deadlock. If a cycle is found, it will prefer to kill single partition transactions to break the cycle, as that will result in less wasted work.

Since our system is motivated by systems like H-Store [95] and DORA [84], each partition runs in single-threaded mode. Therefore, our locking scheme has much lower overhead than traditional locking schemes that have to latch a centralized lock manager before manipulating the lock data structures. Our system can simply lock a data item without having to worry about another thread trying to concurrently lock the same item. The only type of concurrency we are trying to enable is *logical* concurrency where a new transaction can make progress only when the previous transaction is blocked waiting for a network stall — *physical* concurrency cannot occur.

When a transaction is finished and ready to commit, the fragments of the transaction are sent to the backups. This includes any data received from other partitions, so the backups do not participate in distributed transactions. The backups execute the transactions in the sequential order received from the primary. This will produce the same result, as we assume transactions are deterministic. Locks are not acquired while executing the fragments at the backups, since they are not needed. Unlike typical statement-based replication, applying transactions sequentially is not a performance bottleneck because the primary is also single threaded. As with the previous schemes, once the primary has received acknowledgments from all backups, it considers the transaction to be durable and can return results to the client.

4.2 Experimental Evaluation

In this section we explore the trade-offs between the above concurrency control schemes by comparing their performance on some microbenchmarks and a benchmark based on TPC-C. The microbenchmarks are designed to discover the important differences between the approaches, and are not necessarily presented as being representative of any particular application. Our TPC-C benchmark is intended to represent the performance of a more complete and realistic OLTP application.

Our prototype is written in C++ and runs on Linux. We used six servers with two Xeon 3.20 GHz CPUs and 2 GB of RAM. The machines are connected to a single gigabit Ethernet switch. The clients run as multiple threads on a single machine. For each test, we use 15 seconds of warm-up, followed by 60 seconds of measurement (longer tests did not yield different results). We measure the number of transactions that are completed by all clients within the measurement period. Each measurement is repeated three times. We show only the averages, as the confidence intervals are within a few percent and needlessly clutter the figures.

For the microbenchmarks, we use our simple key/value store application described in Section 3.1. The workload has one transaction which reads and updates a set of values. We use small 3 byte keys and 4 byte values to avoid complications caused by data transfer time. For the TPC-C benchmarks, we use our custom in-memory execution engine described in Section 3.2. On each benchmark we compare the three concurrency control schemes described in Section 4.1.

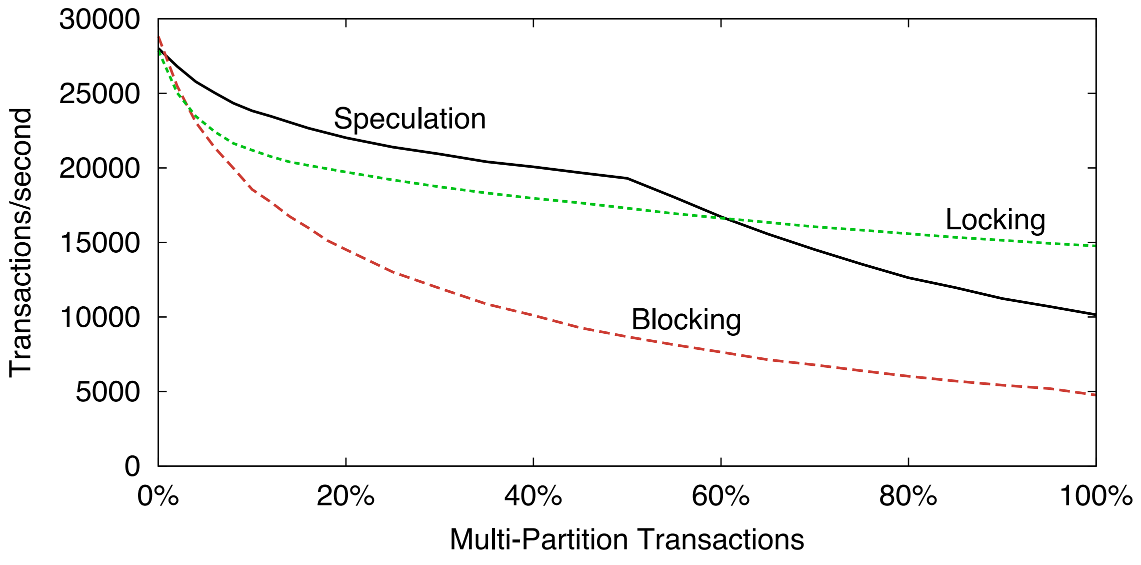


Figure 4-7: Microbenchmark throughput without conflicts

4.2.1 Microbenchmark

Our microbenchmark implements a simple mix of single partition and multi-partition transactions, in order to understand the impact of distributed transactions on throughput. We create a database composed of two partitions, each of which resides on a separate machine. The partitions each store half the keys. Each client issues a read/write transaction which reads and writes the value associated with 12 keys. For this test, there is no sharing (i.e., potential conflict across clients): each client writes its own set of keys. We experiment with shared data in the next section. To create a single partition transaction, a clients selects a partition at random, then accesses 12 keys on that partition. To create a multi-partition transaction, the keys are divided evenly by accessing 6 keys on each partition. To fully utilize the CPU on both partitions, we use 40 simultaneous clients. Each client issues one request, waits for the response, then issues another request.

We vary the fraction of multi-partition transactions and measure the transaction throughput. The results are shown in Figure 4-7. From the application's perspective, the multi-partition and single partition transactions do the same amount of work, so ideally the throughput should stay constant. However, concurrency control overhead means this is not the case. The performance for locking is linear in the range between 16% and 100% multi-partition transactions. The reason is that none of these transactions conflict, so they can all be executed simultaneously. The slight downward slope is due to the fact that multi-partition transactions have additional communication overhead, and thus the performance degrades slightly as their fraction of the workload increases.

The most interesting part of the locking results is between 0% and 16% multi-partition transactions. As expected, the performance of locking is very close to the other schemes at 0% multi-partition transactions, due to our optimization where we do not set locks when there are no multi-partition transactions running. The throughput matches speculation and blocking at 0%, then decreases rapidly until 16% multi-partition transactions, when there

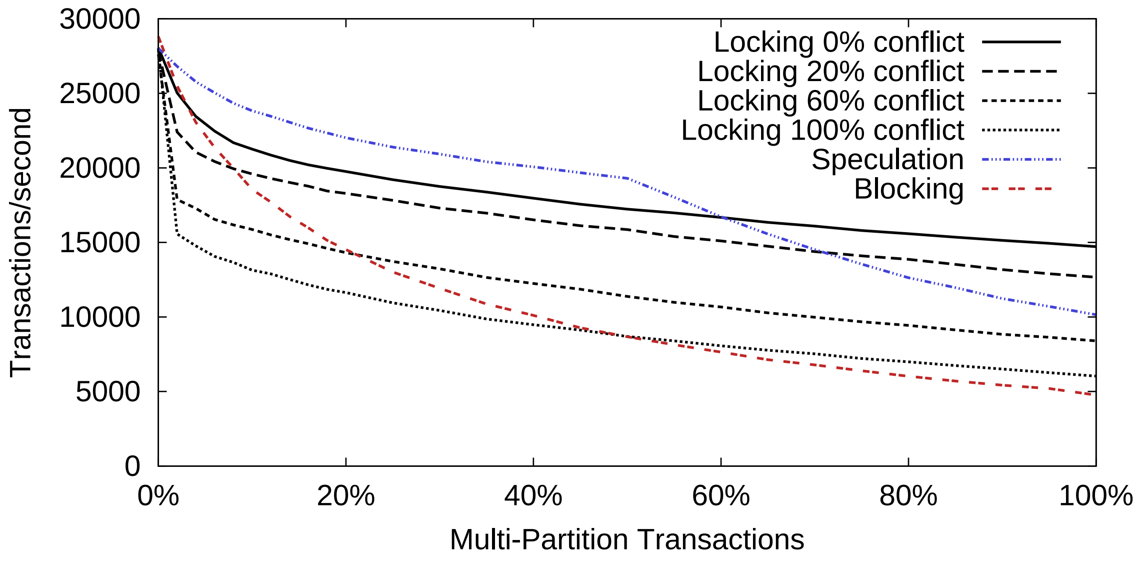


Figure 4-8: Microbenchmark throughput with conflicts

is usually at least one multi-partition transaction in progress, and therefore nearly all transactions are executed with locks.

The performance of blocking in this microbenchmark degrades steeply, so that it never outperforms locking on this low-contention workload. The reason is that the advantage of executing transactions without acquiring locks is outweighed by the idle time caused by waiting for two-phase commit to complete. Our locking implementation executes many transactions without locks when there are few multi-partition transactions, so there is no advantage to blocking. If we force locks to always be acquired, blocking does outperform locking from 0% to 6% multi-partition transactions.

With fewer than 50% multi-partition transactions, the throughput of speculation parallels locking, except with approximately 10% higher throughput. Since this workload is composed of single-partition and simple multi-partition transactions, the single coordinator can speculate all of them. This results in concurrent execution, like locking, but without the overhead of tracking locks. Past 50%, speculation's performance begins to drop. This is the point where the central coordinator uses 100% of the CPU and cannot handle more messages. To scale past this point, we would need to implement distributed transaction ordering, as described in Section 4.1.2.

For this particular experiment, blocking is always worse than speculation and locking. Speculation outperforms locking by up to 13%, before the central coordinator becomes a bottleneck. With the bottleneck of the central coordinator, locking outperforms speculation by 45%.

4.2.2 Conflicts

The performance of locking depends on conflicts between transactions. When there are no conflicts, transactions execute concurrently. However, when there are conflicts, there is additional overhead to suspend and resume execution. To investigate the impact of conflicts,

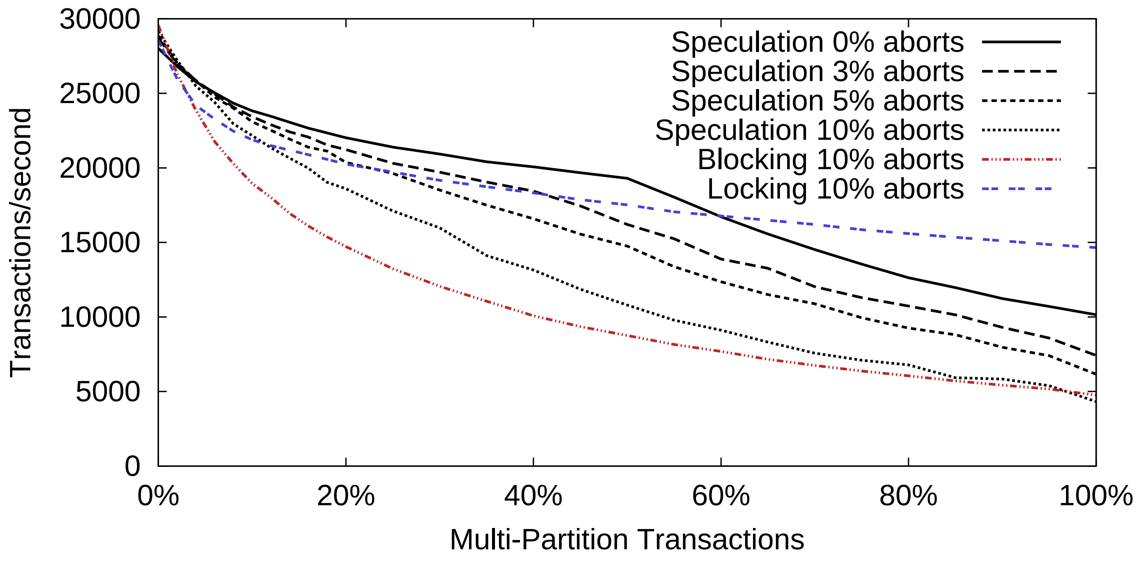


Figure 4-9: Microbenchmark throughput with aborts

we change the pattern of keys that clients access. When issuing single partition transactions, the first client only issues transactions to the first partition, and the second client only issues transactions to the second partition, rather than selecting the destination partition at random. This means the first two clients' keys on their respective partitions are nearly always being written. To cause conflicts, the other clients write one of these "conflict" keys with probability p, or write their own private keys with probability $1-p$. Such transactions will have a very high probability of attempting to update the key at a same time as the first two clients. Increasing p results in more conflicts. Deadlocks are not possible in this workload, allowing us to avoid the performance impact of implementation dependent deadlock resolution policies.

The results in Figure 4-8 show a single line for speculation and blocking, as their throughput does not change with the conflict probability. This is because they assume that all transactions conflict. The performance of locking, on the other hand, degrades as conflict rate increases. Rather than the nearly straight line as before, with conflicts the throughput falls off steeply as the percentage of multi-partition transactions increases. This is because as the conflict rate increases, locking behaves more like blocking. Locking still outperforms blocking when there are many multi-partition transactions because in this workload, each transaction only conflicts at one of the partitions, so it still performs some work concurrently. However, these results do suggest that if conflicts between transactions are common, the advantage of avoiding concurrency control is larger. In this experiment, speculation is up to 2.5 times faster than locking.

4.2.3 Aborts

Speculation assumes that transactions will commit. When a transaction is aborted, the speculatively executed transactions must be undone and re-executed, wasting CPU time. To understand the effects of re-execution, we select transactions to be aborted at random with

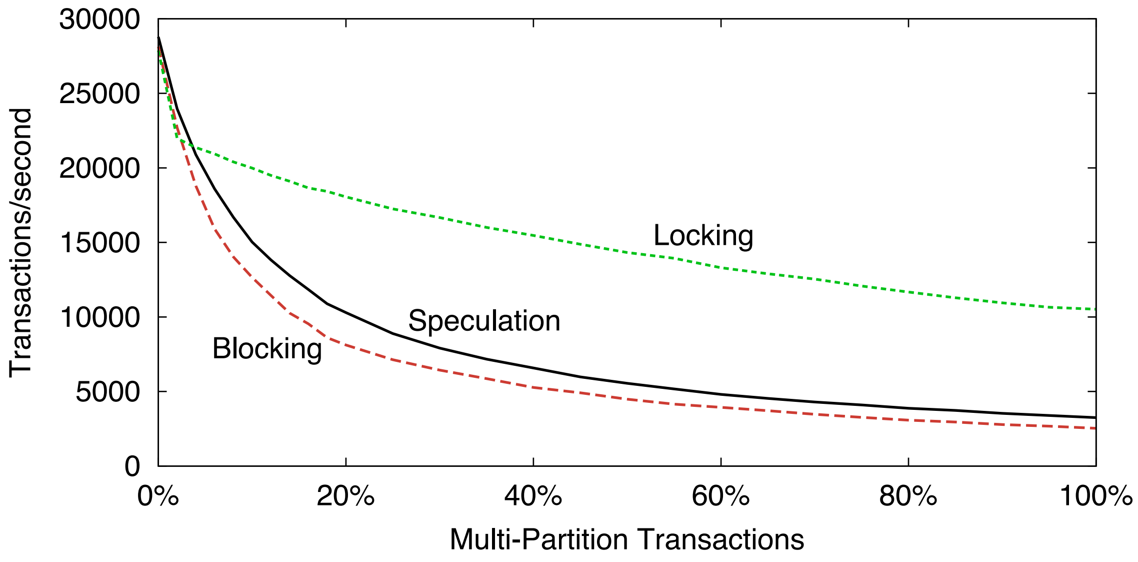


Figure 4-10: Microbenchmark throughput with two round multi-partition transactions

probability p. When a multi-partition transaction is selected, only one partition will abort locally. The other partition will be aborted during two-phase commit. Aborted transactions are somewhat cheaper to execute than normal transactions, since the abort happens at the beginning of execution. They are identical in all other respects, (e.g., network message length).

The results for this experiment are shown in Figure 4-9. The cost of an abort is variable, depending on how many speculatively executed transactions need to be re-executed. Thus, the 95% confidence intervals are wider for this experiment, but they are still within 5%, so we omit them for clarity. Since blocking and locking do not have cascading aborts, the abort rate does not have a significant impact, so we only show the 10% abort probability results. This has slightly higher throughput than the 0% case, since abort transactions require less CPU time.

As expected, aborts decrease the throughput of speculative execution, due to the cost of re-executing transactions. They also increase the number of messages that the central coordinator handles, causing it to saturate sooner. However, speculation still outperforms locking for up to 5% aborts, ignoring the limits of the central coordinator. With 10% aborts, speculation is nearly as bad as blocking, since some transactions are executed many times. These results suggest that if a transaction has a very high abort probability, it may be better to limit to the amount of speculation to avoid wasted work.

4.2.4 General Multi-Partition Transactions

When executing a multi-partition transaction that involves multiple rounds of communication, speculation can only begin when the transaction is known to have completed all its work at a given partition. This means that there must be a stall between the individual fragments of transaction. To examine the performance impact of these multi-round transactions, we changed our microbenchmark to issue a multi-partition transaction that requires two rounds of communication, instead of the simple multi-partition transaction in the original benchmark. The first round of each transaction performs the reads and returns the results to the coordinator, which then issues the writes as a second round. This performs the same amount of work as the original benchmark, but has twice as many messages.

The results are shown in Figure 4-10. The blocking throughput follows the same trend as before, only lower because the two round transactions take nearly twice as much time as the multi-partition transactions in the original benchmark. Speculation performs only slightly better, since it can only speculate the first fragment of the next multi-partition transaction once the previous one has finished. Locking is relatively unaffected by the additional round of network communication. Even though locking is generally superior for this workload, speculation does still outperform locking as long as fewer than 4% of the workload is composed of general multi-partition transactions.

4.2.5 TPC-C

We ran TPC-C with the warehouses divided evenly across two partitions. In this workload, the fraction of multi-partition transactions ranges from 5.7% with 20 warehouses to 10.7% with 2 warehouses. The throughput for varying numbers of warehouses are shown in Figure 4-11. With this workload, blocking and speculation have relatively constant performance as the number of warehouses is increased. The performance is lowest with 2 partitions because the probability of a multi-partition transaction is highest (10.7%, versus 7.2% for 4 warehouses, and 5.7% for 20 warehouses), due to the way TPC-C new order transaction requests are generated. After 4 warehouses, the performance for blocking and speculation decrease slightly. This is due to the larger working set size and the corresponding increase in CPU cache and TLB misses. The performance for locking increases as the number of warehouses is increased because the number of conflicting transactions decreases. This is because there are fewer clients per TPC-C warehouse, and nearly every transaction modifies the warehouse and district records. This workload also has deadlocks, which leads to overhead due to deadlock detection and distributed deadlock timeouts, decreasing the performance for locking. Speculation performs the best of the three schemes because the workload's fraction of multi-partition transactions is within the region where it is the best choice. With 20 warehouses, speculation provides 9.7% higher throughput than blocking, and 63% higher throughput than locking.

4.2.6 TPC-C Multi-Partition Scaling

In order to examine the impact that multi-partition transactions have on a more complex workload, we scale the fraction of TPC-C transactions that span multiple partitions. We execute a workload that is composed of 100% new order transactions on 6 warehouses. We then adjust the probability that an item in the order comes from a "remote" warehouse, which is a multi-partition transaction. With TPC-C's default parameters, this probability is 0.01 (1%), which produces a multi-partition transaction 9.5% of the time. We adjust this parameter and compute the probability that a transaction is a multi-partition transaction. The throughput with this workload is shown in Figure 4-12.

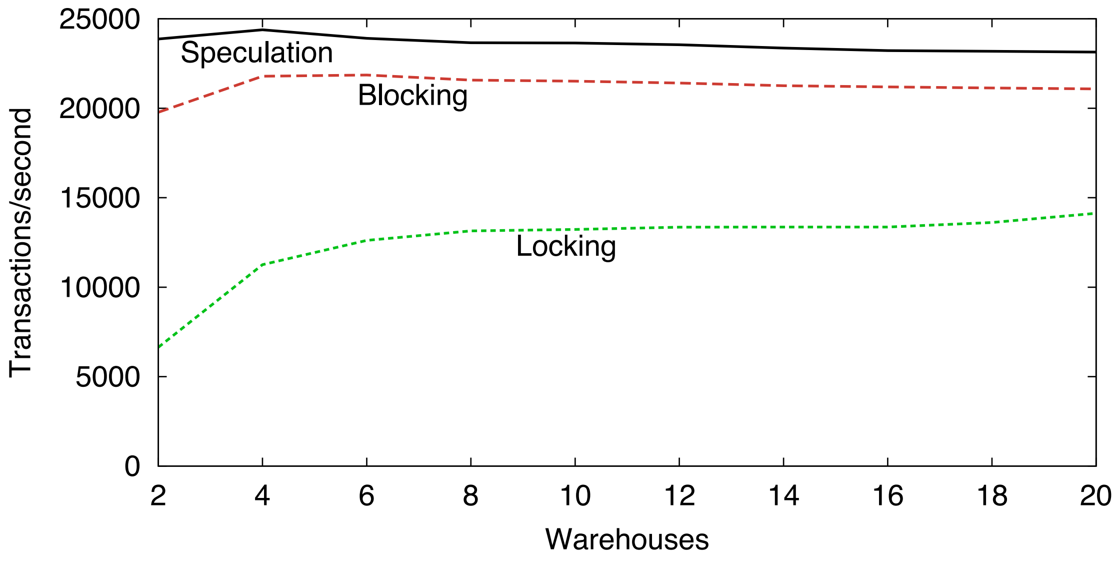


Figure 4-11: TPC-C throughput with varying warehouses

The results for blocking and speculation are very similar to the results for the microbenchmark in Figure 4-7. In this experiment, the performance for locking degrades very rapidly. At 0% multi-partition transactions, it runs efficiently without acquiring locks, but with multi-partition transactions it must acquire locks. The locking overhead is higher for TPC-C than our microbenchmark for three reasons: more locks are acquired for each transaction, the lock manager is more complex, and there are many conflicts. In particular, this workload exhibits local and distributed deadlocks, hurting throughput significantly. Again, this shows that conflicts make traditional concurrency control more expensive, increasing the benefits of simpler schemes.

Examining the output of a sampling profiler while running with a 10% multi-partition probability shows that 34% of the execution time is spent in the lock implementation. Approximately 12% of the time is spent managing the lock table, 14% is spent acquiring locks, and 6% is spent releasing locks. While our locking implementation certainly has room for optimization, this is similar to what was previously measured for Shore, where 16% of the CPU instructions could be attributed to locking [45].

4.2.7 Summary

Our results show that the properties of the workload determine the best concurrency control mechanism. Speculation performs substantially better than locking or blocking for multi-partition transactions that require only a single round of communication and when a low percentage of transactions abort. Our low overhead locking technique is best when there are many transactions with multiple rounds of communication. Table 4.1 shows which scheme is best, depending on the workload; we imagine that a query executor might record statistics at runtime and use a model like that presented in Section 4.3 below to make the best choice.

Optimistic concurrency control (OCC) is another "standard" concurrency control algo-

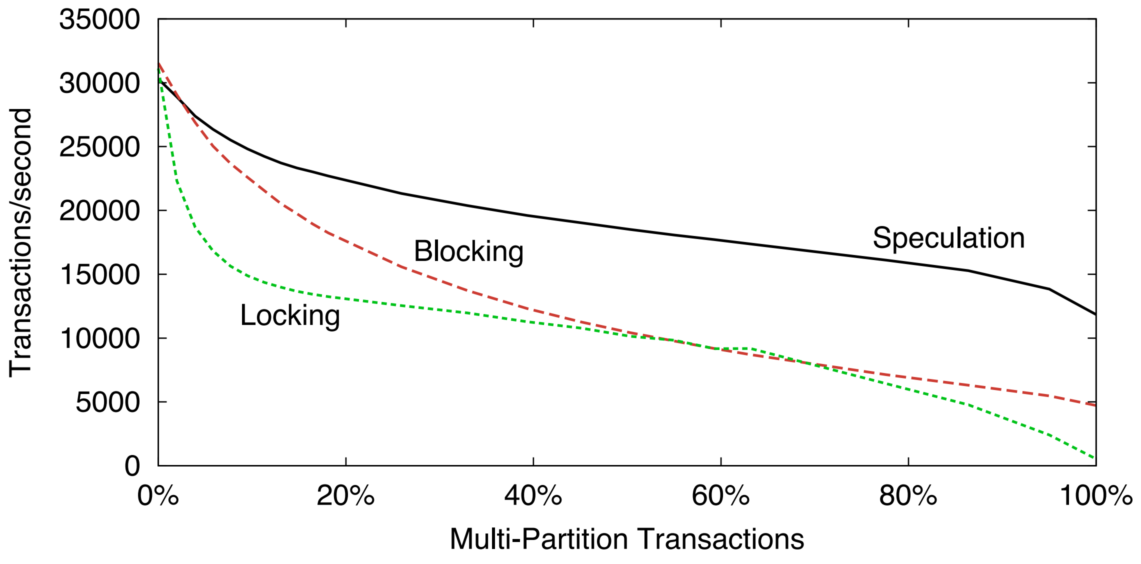


Figure 4-12: TPC-C throughput with 100% New Order transactions

		Few Aborts		*Many Aborts*	
		Few Conflicts	*Many Conflicts*	*Few Conflicts*	*Many Conflicts*
Few multi-round xactions	*Many multi-partition xactions*	Speculation	Speculation	Locking	Locking or Speculation
	Few multi-partition xactions	Speculation	Speculation	Blocking or Locking	Blocking
Many multi-round xactions		Locking	Locking	Locking	Locking

Table 4.1: Summary of the best concurrency control scheme for different situations. Speculation is preferred when there are few multi-round (general) transactions and few aborts.

rithm. It requires tracking each item that is read and written, and aborts transactions during a validation phase if there were conflicts. Intuitively, we expect the performance for OCC to be similar to that of locking. This is because, unlike traditional locking implementations that need complex lock managers and careful latching to avoid problems inherent in physical concurrency, our locking scheme can be much lighter-weight, since each partition runs single-threaded (i.e., we only have to worry about the logical concurrency). Hence, our locking implementation involves little more than keeping track of the read/write sets of a transaction — which OCC also must do. Consequently, OCC's primary advantage over locking is eliminated. We have run some initial results that verify this hypothesis, and plan to explore the trade-offs between OCC and other concurrency control methods and our speculation schemes as future work.

4.3 Analytical Model

To improve our understanding of the concurrency control schemes, we analyze the expected performance for the multi-partition scaling experiment from Section 4.2.1. This model predicts the performance of the three schemes in terms of just a few parameters (which would be useful in a query planner, for example), and allows us to explore the sensitivity to workload characteristics (such as the CPU cost per transaction or the network latency). To simplify the analysis, we ignore replication.

Consider a database divided into two partitions, P_1 and P_2. The workload consists of two transactions. The first is a single partition transaction that accesses only P_1 or P_2, chosen uniformly at random. The second is a multi-partition transaction that accesses both partitions. There are no data dependencies, and therefore only a single round of communication is required. In other words, the coordinator simply sends two fragments out, one to each partition, waits for the response, then sends the commit or abort decision. Each multi-partition transaction accesses half as much data in each partition, but the total amount of data accessed is equal to one single partition transaction. To provide the best case performance for locking, none of the transactions conflict. We are interested in the throughput as the fraction of multi-partition transactions, f, in the workload is increased.

4.3.1 Blocking

We begin by analyzing the blocking scheme. Here, a single partition transaction executes for t_{sp} seconds on one partition. A multi-partition transaction executes for t_{mp} on both partitions, including the time to complete the two-phase commit. If there are N transactions to execute, then there are Nf multi-partition transactions and $\frac{1}{2}N(1-f)$ single partition transactions to execute at each partition. The factor of $\frac{1}{2}$ arises because the single partition transactions are distributed evenly between the two partitions. Therefore the time it takes to execute the transactions and the system throughput are given by the following equations:

$$\text{time} = Nft_{mp} + \frac{N(1-f)}{2}t_{sp}$$
$$\text{throughput} = \frac{N}{\text{time}} = \frac{2}{2ft_{mp} + (1-f)t_{sp}}$$

Effectively, the time to execute N transactions is a weighted average between the times for a pure single partition workload and a pure multi-partition workload. As f increases, the throughput will decrease from $\frac{2}{t_{sp}}$ to $\frac{1}{t_{mp}}$. Since $t_{mp} > t_{sp}$, the throughput will decrease rapidly with even a small fraction of multi-partition transactions.

4.3.2 Speculation

We first consider the local speculation scheme described in Section 4.1.2. For speculation, we need to know the amount of time that each partition is idle during a multi-partition transaction. If the CPU time consumed by a multi-partition transaction at one partition is t_{mpC}, then the network stall time is $t_{mpN} = t_{mp} - t_{mpC}$. Since we can overlap the execution of the next multi-partition transaction with the stall, the limiting time when executing a pure multi-partition transaction workload is $t_{mpL} = \max\left(t_{mpN}, t_{mpC}\right)$, and the time that the CPU is idle is $t_{mpI} = \max\left(t_{mpN}, t_{mpC}\right) - t_{mpC}$.

Assume that the time to speculatively execute a single partition transaction is t_{spS}. During a multi-partition transaction's idle time, each partition can execute a maximum of $\frac{t_{mpI}}{t_{spS}}$ single partition transactions. When the system executes Nf multi-partition transactions, each partition executes $\frac{N(1-f)}{2}$ single partition transactions. Thus, on average each multi-partition transaction is separated by $\frac{(1-f)}{2f}$ single partition transactions. Therefore, for each multi-partition transaction, the number of single partition transactions that each partition can speculate is given by:

$$N_{hidden} = \min\left(\frac{1-f}{2f}, \frac{t_{mpI}}{t_{spS}}\right)$$

Therefore, the time to execute N transactions and the resulting throughput are:

$$\text{time} = Nft_{mpL} + (N(1-f) - 2NfN_{hidden})\frac{t_{sp}}{2}$$
$$\text{throughput} = \frac{2}{2ft_{mpL} + ((1-f) - 2fN_{hidden})t_{sp}}$$

In our specific scenario and system, $t_{mpN} > t_{mpC}$, so we can simplify the equations:

$$
\begin{aligned}
N_{hidden} &= \min\left(\frac{1-f}{2f}, \frac{t_{mp}-2t_{mpC}}{t_{spS}}\right) \\
\text{throughput} &= \frac{2}{2f(t_{mp}-t_{mpC})+((1-f)-2fN_{hidden})t_{sp}}
\end{aligned}
$$

The minimum function produces two different behaviors. If $\frac{1-f}{2f} \geq \frac{t_{mpI}}{t_{spS}}$, then the idle time can be completely utilized. Otherwise, the system does not have enough single partition transactions to completely hide the network stall, and the throughput will drop rapidly as f increases past $\frac{t_{spS}}{2t_{mpI}+t_{spS}}$.

Speculating Multi-Partition Transactions

This model can be extended to include speculating multi-partition transactions, as described in Section 4.1.2. The previous derivation for execution time assumes that one multi-partition transaction completes every t_{mpL} seconds, which includes the network stall time. When multi-partition transactions can be speculated, this restriction is removed. Instead, we must compute the CPU time to execute multi-partition transactions, and speculative and non-speculative single partition transactions. The previous model computed the number of speculative single partition transactions per multi-partition transaction, N_{hidden}. We can compute the time for multi-partition transactions and speculative single partition transactions as $t_{period} = t_{mpC} + N_{hidden}t_{spS}$. This time replaces t_{mpL} in the previous model, and thus the throughput becomes:

$$
\text{throughput} = \frac{2}{2ft_{period}+((1-f)-2fN_{hidden})t_{sp}}
$$

4.3.3 Locking

Since the workload is composed of non-conflicting transactions, locking has no stall time. However, we must account for the overhead of tracking locks. We define l to be the fraction of additional time that a transaction takes to execute when locking is enabled. Since locking always requires undo records, we use t_{spS} to account for the single partition execution time. Furthermore, the overhead of two-phase commit must be added, so for multi-partition transactions we use t_{mpC}. The time to execute N transactions, and the resulting throughput are given by:

$$
\begin{aligned}
\text{time} &= Nflt_{mpC} + \frac{N(1-f)}{2}lt_{spS} \\
\text{throughput} &= \frac{N}{\text{time}} = \frac{2}{2flt_{mpC}+(1-f)lt_{spS}}
\end{aligned}
$$

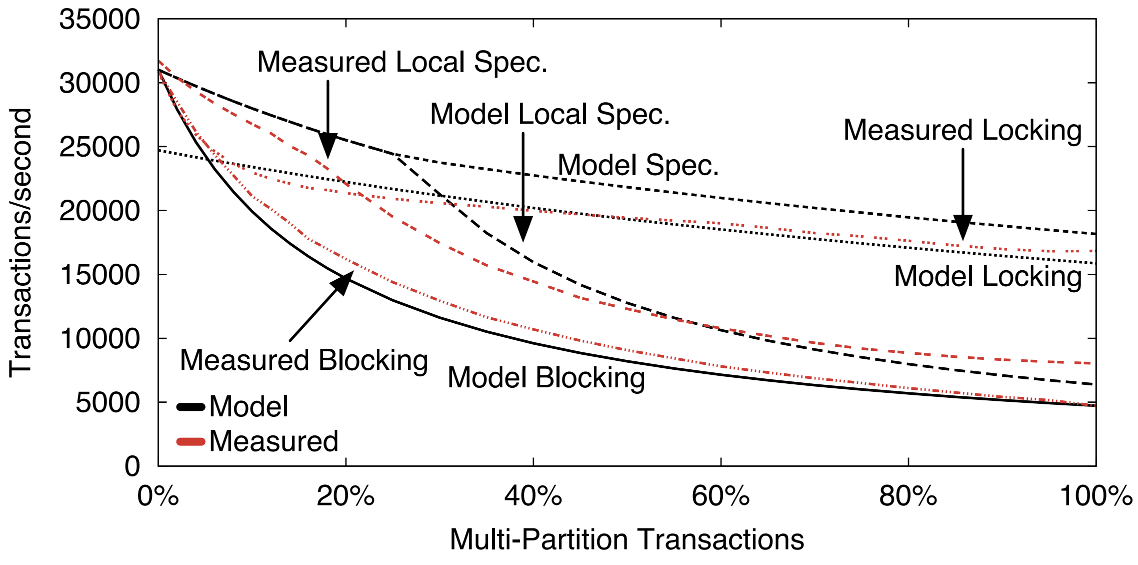


Figure 4-13: Model throughput compared to measured throughput without replication

Variable	Measured	Description
t_{sp}	64 μs	Time to execute a single partition transaction non-speculatively.
t_{spS}	73 μs	Time to execute a single partition transaction speculatively.
t_{mp}	211 μs	Time to execute a multi-partition transaction, including resolving the two-phase commit.
t_{mpC}	55 μs	CPU time used to execute a multi-partition transaction.
t_{mpN}	40 μs	Network stall time while executing a multi-partition transaction.
l	13.2%	Locking overhead. Fraction of additional execution time.

Table 4.2: Analytical model variables

4.3.4 Experimental Validation

We measured the model parameters for our implementation. The values are shown in Table 4.2. Figure 4-13 shows the analytical model using these parameters, along with the measured throughput for our system without replication. As can be observed, the two are a relatively close match. This suggests that the model is a reasonable approximation for the behavior of the real system. These results also show that speculating multi-partition transactions leads to a substantial improvement when they comprise a large fraction of the workload.

Chapter 5

Live Migration

Large database systems must be able to adapt to changing workload demands, such as increases in load that can be caused by sudden spikes in popularity or by slow, steady growth. Distributed systems running in large data centers can, in theory, add and remove resources dynamically, as needs change. However, it is difficult to elastically scale databases. Traditionally, it requires upgrading hardware or moving data during maintenance windows, taking the database out of service. This downtime is unacceptable in today's world where services are expected to be continuously available.

Providing elasticity for stateless applications like web servers is straightforward, but is much harder for stateful applications like databases. This is because large quantities of data must be moved, or *migrated*, from one machine to another. This transfer of data can incur significant load on the source machine and take significant time. Worse, this additional load is incurred exactly when the source needs to offload work. Anecdotally, providing this kind of efficient migration is both important and difficult in current systems. For example, in October 2010, Foursquare (a social, location-based application) experienced 11-hours of downtime due to skew in data growth followed by an attempted migration of part of the data to a new node [34].

The key requirements for elasticity in a data storage system are:

- *Low-overhead*, to facilitate movement of data off of highly loaded servers in response to load spikes without increasing load on the source, and
- *Partial migration*, in order to move just a portion of the data to another server, allowing the data distribution to be re-optimized on the fly, and to split a single machine's work across multiple machines.

Unfortunately, existing approaches to database migration do not meet these requirements. Replication-based solutions, where a replica of the databases is created from a dirty copy of the data and the transaction log, followed by a failover from the master to this new replica, impose significant extra load on the source node as data is copied off of it. Virtual machine (VM) migration can efficiently migrate entire VMs (which may host databases) [65, 24], but requires expensive shared storage. Neither of these approaches allow partial migration, since data must be moved at the granularity of an entire database or machine. Recently proposed academic solutions, such as Zephyr [31] also impose significant load on the source and do not readily support partial migration. We discuss the differences between our work and these systems in more detail in Section 6.6 and in our experimental evaluation (Section 5.5).

In this chapter, we describe Dtxn's live migration support, and a novel migration technique we call Wildebeest[1]. Because moving data requires close integration with the storage engine, Dtxn only handles updating the metadata and redirecting clients when appropriate. The actual data transfer must be handled by the storage engine. We used Dtxn's migration support to implement two existing techniques (stop and copy and replica failover, described in detail in Section 5.5), as well as Wildebeest.

Wildebeest itself is a novel migration technique designed to elastically migrate a database from one server to another while incurring minimal additional load on the source. Like the rest of Dtxn, it is designed specifically for OLTP and web applications, and thus is optimized for workloads composed of large numbers of small transactions. Wildebeest's key idea is to use a logical (tuple-id based) "copy on demand" policy, where data is migrated only when it is needed. Immediately after migration is initiated, new transactions are sent to the migration destination, even though it has not yet received any data from the source. The destination then sends requests to the source for only the logical data ranges it needs to answer the query and adds them to its copy of the state.

This kind of fine-grained data movement requires tight integration with the database. Thus, in this chapter, we discuss our implementation of migration for a single application: the Relational Cloud SQL database built with Dtxn, described in Section 3.4. Our prototype is built using MySQL as the storage engine, and Dtxn to manage partitions and distributed transactions. It is designed for workloads where the working set fits in memory. While our implementation of Wildebeest is tuned for use with traditional disk-based SQL databases, we believe that its design is applicable to any transactional OLTP system. The only requirement is that transactions only access "small" amounts of data, such as under a megabyte. The rest of this chapter discusses Wildebeest specifically.

The key features of Wildebeest include:

- *Transactionally consistent migration* of all or part of a database that *immediately reduces load* on the source database when migration is initiated.

- *Logical copying.* Our approach moves parts of the database by copying data at a logical level. This enables repartitioning a single database across multiple machines as needed, rather than being forced to move data based on the physical data layout (e.g., the existing clustered indices or partitions), as existing approaches do.

- *Efficiency.* Our approach effectively turns local database reads into remote reads for missing data and local database writes. If not done carefully, when using a traditional disk-based storage engine, this could have a disastrous effect on transaction throughput during migration. We add a main-memory cache that allows us to avoid synchronous disk writes as data is migrated and a pre-fetching strategy to reduce the number of remote reads that are required.

[1]Wildebeest, also known as gnu, are large mammals that live on the African plains, noted for their migratory habits.

- *Fault tolerance.* If a crash occurs during migration, some new data may be on the destination, while old state may still be on the source. We introduce a recovery protocol that ensures that migration is able to restart despite such crashes.

Our experimental results (Section 5.5) show that Wildebeest has less visible performance impact than existing techniques. Most importantly, when using it to scale from one to two machines, Wildebeest improves the system throughput $7\times$ faster than the next best technique, taking 4 seconds while virtual machine migration takes 28 seconds. This is despite the fact that the VM approach uses shared disk, so it does not perform any disk I/O, while Wildebeest writes the data to disk on the destination machine. In low load situations, our system has a small visible impact at the moment migration is triggered, but can resume processing within less than one second, while virtual machine migration has significantly longer visible pauses, and replica failover increases request latency while the data is being migrated (over two minutes in our experiments).

5.1 Migration Procedure

Wildebeest migration involves moving data between any two partitions. Moving data is a primitive operation that supports many kinds of data migration, such as splitting an existing partition into two pieces, recombining two partitions into one, or rebalancing two partitions. Thus, it is a flexible tool that can be used in multiple ways to make changes to a running system.

Migration begins when the application indicates that migration should occur from the *source* partition to the *destination*. Wildebeest first quiesces the source, allowing outstanding transactions to finish. It then begins sending new transactions to the destination. Initially the destination has no data, so to answer queries it uses *query rewriting* to fetch ranges of data it needs from the source. As it copies data, the destination keeps a map that tells it which tuples have already been copied. Inserts are performed on the destination; updates and deletes require more care to ensure they are properly applied while some of the data resides on both the source and the destination.

In the remainder of this section, we discuss these steps in more detail, deferring the technical details of query rewriting, recovery, and performance optimizations to the following sections. Migration is a four-step process, as shown in Figure 5-1.

1. **Phase 1 – Prepare the destination**: When migration is initiated, a `SETUP` message is sent to the destination. The new partition is created on the destination host, initializing the tables and indices specified by the schema. The destination migration agent then sends the source the `PREPARE` message, but does not begin fetching any data. At this point, the destination will accept new transactions for this partition, but will queue them until it receives the `SWITCH` message from the source. During this phase, all queries to the partition are processed at the source (e.g., Q1 in Figure 5-1).

2. **Phase 2 – Drain the source**: When the source receives the `PREPARE` message it stops accepting new transactions. It informs routers that attempt to initiate new transactions

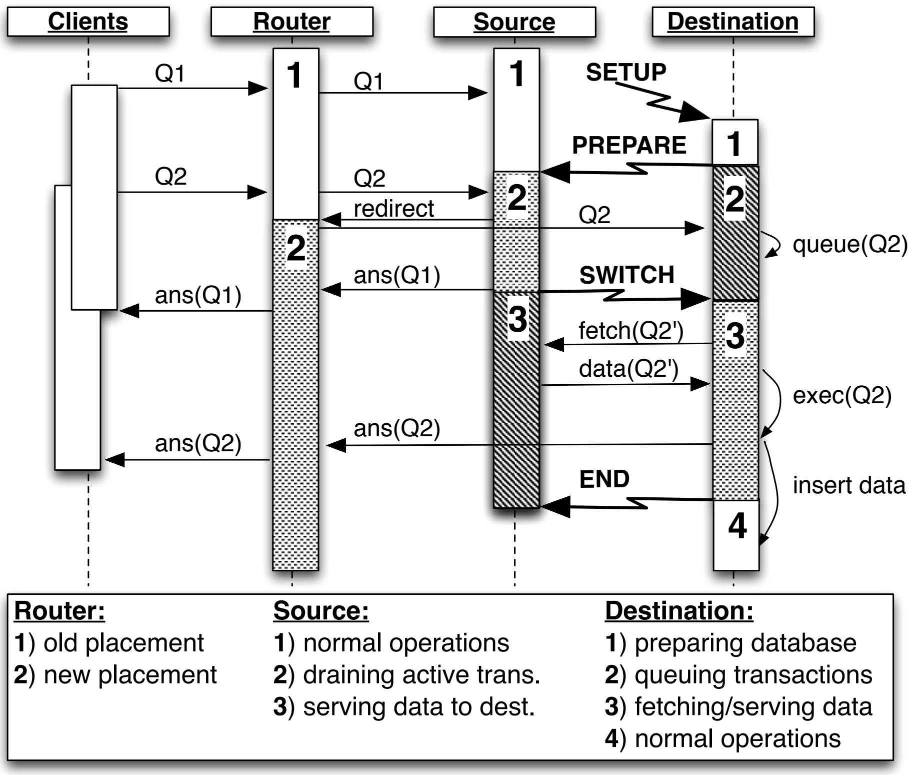


Figure 5-1: Sequence diagram of migration.

that the partition has moved to the destination. Routers receiving this message forward new transactions to the destination (e.g., Q2 in Figure 5-1). During this time, transactions that were already active are allowed to finish execution on the source. This phase will last until all active transactions have completed. Since Dtxn and Wildebeest are designed for OLTP and web applications that are composed of short transactions with a small number of statements, this will be a very short period of time, typically a few hundred milliseconds at most. During this time, transactions are queued at destination and will thus observe higher latency.

3. **Phase 3 – Destination fetching**: When all active transactions at the source have completed, the source notifies the destination by sending it the `SWITCH` message. When this is received, the destination begins executing client requests. For each query, the destination determines what data a query will access. If that data has not already been fetched, it retrieves the data from the source and inserts it into the local database. For high performance, data is inserted into main-memory and written lazily in the background, as described in Section 5.4.2. The destination finally executes the user's query locally. We describe a how the destination determines if it can answer a query in Section 5.2.1 and precisely what queries are issued to the source in Section 5.2.2.

   All tuples must eventually be fetched from the source to the destination and written to disk in order to complete the migration. To accomplish this, a background task is started at the destination which periodically fetches "missing" tuples from the source and another that durably writes tuples from the main memory table. Running this background task at a high rate will cause migration to complete more quickly, but will introduce more load on both servers. The disadvantage of migration taking a long time is that there is a higher probability of the partition being unavailable, as the partition will be down if either the source or the destination is inaccessible. By default we choose to run perform these background fetches slowly, as this provides the best runtime performance, with the cost of needing the source to remain up for longer. The administrator can request a more aggressive migration if this migration period must be minimized, such as when the source server must be immediately taken offline for maintenance.

4. **Phase 4 – Cleanup**: Finally, once all tuples have been copied to the destination and have been durably written to disk, the destination informs the source that the migration is complete with the `END` message. At this point, the source can "drop" the partition, reclaiming disk space. The destination then resumes normal query processing, without involving the source.

Our approach copies the data tuples on an as-needed basis from the source to the destination. All indexes are rebuilt on the destination host. This is more costly than a physical copy in some ways, as the destination repeats work that the primary has already performed. However, it provides two benefits. First, it reduces the amount of work on the source server, which we assume is overloaded, as we initially copy only the referenced data. This is approximately the same load that it would have served without migration. The extra work of rebuilding indexes occurs on the destination, which should have excess capacity. Secondly, it enables partial migration.

Partial migration is difficult with physical copy approaches, as we now discuss. We consider two physical-level partial-copy approaches: copy the entire partition then delete the portion of the data that has not migrated, or copy part of the primary clustered index. The "delete after copy" approach must copy significant data which will just be thrown away. It must also rebalance the primary index and perform many "random" secondary index deletes, which is effectively the same as rebuilding the index. The partial physical copy approach can work if the table is being partitioning by the primary clustered index key, and will likely be more efficient for that index. However, the primary index still needs to be rebalanced, and this approach will need to rebuild the secondary indexes. Thus, while a physical copy may be faster when moving an entire partition, a logical copy gives Wildebeest more flexibility in a partial-move situation.

The recoverability and fault-tolerance characteristics of this migration are discussed in Section 5.3.

5.2 Migration Techniques

In this section, we describe the details of our basic migration strategy. We begin by talking about queries, and then generalize to updates later. When a new query Q is received at the destination node, the migration agent parses it and checks whether the data it needs to answer the query has already been migrated. If some of the data is missing, the migration agent generates a query Q' (a rewriting of Q) that when executed on the source node will fetch the data required to process the query. The result is inserted locally at the destination. At this point, Q can be executed locally. For this process to work, we introduce two additional software components: a range-tracker, and a rewriting engine, which we discuss next.

5.2.1 Range Tracking and Query Analysis

To avoid redundantly fetching data from the source, the migration agent on the destination must carefully track which data has been moved from the source to the destination. For the tables we are migrating, the migration agent maintains a list of the ranges of tuples that have already been moved—we call this a *range set*. This allows the query analysis component to compare the tables and predicates in the `WHERE` clause of a query Q with what is available locally[2].

The analysis we perform is conservative, in the sense that it may sometimes falsely conclude that the data needed to answer a query Q is not available on the destination when it is, but it will never incorrectly decide all data is available when it is not. The basic approach is to test to see if predicates used in a query Q are contained in the set of ranges of cached values. While this *query containment* problem is in general hard [19], we are able to make it tractable via several heuristics covering the common query types in OLTP/web workloads. Complex input queries for which we cannot prove containment are handled conservatively, by fetching more data than is needed. In the worst case, this may lead to us retrieving all the input tables mentioned by the query before executing it. This ensures correctness at the price of performance. When inserting the data in the destination database we rely on the existence of primary keys in the data to avoid duplicates.

Due to storage overheads, maintaining range sets for each column in each migrated table is impractical. However, queries with predicates only on unindexed columns require a full table scan even when not migrating. This observation allows us to only maintain range sets for indexed columns, which, in the case of OLTP/web workloads, are typically predicated in every query.

Initially, each range set is empty. As tuples are fetched from the source via a predicate on a specific index, ranges are added to the corresponding range set at the destination. Ranges may be singletons if the index was accessed using an equality (e.g. $id = 0$ maps to [0, 0]), finite for bounded ranges (e.g. $0 \leq id \leq 7$ maps to [0, 7]), or infinite for unbounded ranges (e.g. $0 \leq id$ maps to $[0, \infty)$). In our implementation, we store the ranges in a binary tree, which can efficiently determine if a specific key or range of keys are contained in the range set. If ranges are countable (e.g., integers), then adjacent ranges can be combined to reduce the storage requirements. For uncountable ranges (e.g. string or multiple integer keys), ranges can only be combined by explicit range queries on the source.

[2]Simply storing the data in the local database is not sufficient because it can not differentiate between a tuple that does not exist and a tuple that has not been migrated. If we assumed that a tuple needs to be fetched from the source every time it is missing locally, the destination would end up repeating many queries for tuples that do not exist.

Each time a tuple is migrated, entries are added to the range sets for all indexes with unique keys, no matter what index was used to retrieve the tuple. This is correct because there is only a single possible tuple matching each key, so we can be certain we have fetched all possible tuples matching that value. For indices with non-unique keys (typically secondary indexes), range sets are only updated when accessing tuples via a predicate on that index. This ensures that when a range exists in a range set, all possible tuples in that range have been migrated.

There are several other considerations when performing this fetching:

- For `LIMIT` clauses on non-unique indexes, we can only update range sets for the keys where it is guaranteed that all values have been fetched. This means either fewer results than the limit were returned, or we must exclude the last key in the result set. For example, for the query with predicate $0 \leq id$ `LIMIT 4` and result set (1, 1, 1, 2), we can only add the range [0, 1] to the range set, since there may be more tuples with $id = 2$.

- For queries with conjunctive (`AND`) predicates over a set of indexed attributes A, data is only fetched if *all* ranges for the attributes in A indicate the data has not already been fetched. This works because if at least one attribute's range has been fetched, it contains the result set. The other attributes are only used to restrict the results. For these conjunctive queries, we cannot update range sets, except for the normal case of adding values to unique indexes for tuples that were retrieved.

- For disjunctive (`OR`) predicates, data must be fetched if *any* of the predicates are non-indexed or if any of the range sets indicates the predicated data has not yet been retrieved. In this case, however, we can update range sets in all indexes.

Range sets are also used to determine when migration is complete. The "background copy" task iterates over the primary index range set, issuing queries to fill the holes and ensure that all data has been migrated. This background copying process is described in more detail in Section 5.2.4.

We describe how these range sets are used to determine exactly what queries are sent from the destination to the source in the next section.

5.2.2 Query Rewriting

When the migration agent on the destination determines some of the data needed to answer a query has not yet been migrated, it issues a query to fetch the needed tuples. Since Wildebeest is implemented as middleware, this is done by means of query rewriting.

As noted in the previous section, we take a conservative approach to guarantee soundness. Rewritten queries are designed to always fetch all of the needed data, without performing excessive amounts of repeated fetching. We explore more opportunities for optimization in Section 5.4, where we discuss a technique that pre-fetches extra tuples by relaxing the query predicates.

In the following we briefly discuss rewriting of different classes of queries.

1. **Single table queries:** The query is rewritten by substituting the target list with a `SELECT *` statement. This guarantees that all the attributes of the records are migrated. If the `WHERE` clause has predicates over a column for which we maintain range sets, we determine the difference between the query ranges and the range sets. Specifically, we modify the `WHERE` clause with predicates that fetch only the missing tuples (i.e., tuples in the difference). If the tuple has no predicates, or single or disjunctive predicates on non-indexed fields, we fetch the entire table, since as noted above, these queries require a table-scan on the source anyway. Aggregate and `GROUP BY` clauses are simply removed. `LIMIT` clauses are left untouched by the basic rewriting strategy.

2. **Multi table queries:** Our general strategy is to determine the range of tuples that are accessed for each of the tables mentioned in the query, fetch all the required tuples with independent queries, then perform the join at the destination. When no predicates other than the join predicates are defined for a table, we perform a multi-step execution akin to a semi-join. Specifically:

   - Tuples from smaller or predicated tables are fetched first.
   - These tuples are used to bind the join attribute values in queries that are issued to the non-predicated tables.

   While this basic strategy is efficient for key-foreign key joins typically found in OLTP/web workloads, it is not ideal for OLAP-like queries that access large amounts of data. For these queries our strategy degenerates to fetching all the data from the source immediately, rather than fetching data lazily.

5.2.3 Inserts, Updates, and Deletes

Statements that modify the database also require some additional modification, because the data that is modified may be on the source, the destination, or both. For recovery of writes it is important to note that the logical "state" of the database is the union of the source and destination databases, with the destination as the authoritative copy in case of disagreement. This allows us to perform inserts and updates only on the destination, although deletes need to be run on both. Recovery and fault-tolerance are discussed in the next section.

1. **Inserts:** Inserts can be performed at destination provided that integrity constraints are respected. We limit our discussion here to unique/primary key constraints[3]. In order to enforce uniqueness constraints we check that all the potentially conflicting tuples are available at the destination by checking the appropriate range sets. If some tuples are missing, before running the `INSERT` statement, we execute a query on the same table with a `WHERE` clause containing an `OR` of all uniquely constrained fields. Fetching this data and storing it locally guarantees that every possible conflict will be

[3]Foreign key constraints can be supported through a similar mechanism, though we have not done so because they are not frequently used in OLTP/web workloads for performance reasons, and doing so would introduce significant overhead. Most shared nothing distributed databases, like DB2 Parallel Edition, do not support them for similar reasons.

detected at the destination. In order to handle auto-increment keys, the source sends the destination the next auto-increment value as part of the `SWITCH` message.

2. **Deletes:** Deletes need to be propagated both to the source and destination, since during recovery we will union the source and destination databases. Thus, we execute deletes as a distributed transaction on both the source and destination databases. This ensures that during recovery when merging the state of the source and destination databases, a deleted tuple stays deleted (we call this the "resurrection" problem, where a dead tuple comes back to life). This operation adds entries to the range set for the associated indexes, as with a regular query, which avoids querying of the source if later queries try to access the deleted tuples.

3. **Updates:** Most updates are handled by first ensuring that the target tuples have been migrated, following the same rewriting as for queries. Then the update can be processed locally. We currently do not support updates that modify primary keys because our example applications have not needed this functionality and so we have not implemented the correct rewriting. These updates need to be converted to a `SELECT-DELETE-INSERT` sequence, in order to correctly handle failure during the migration. This is required so that the original primary key is marked as no longer existing on the source, in order to handle failures that may occur during migration. This is to avoid the same potential "resurrection" issue described for deletes.

5.2.4 Background Fetching

Our approach lazily fetches tuples only when required to answer user queries. However, in order to complete migration we need to eventually also fetch those tuples that are never accessed by the workload, but are part of the partition/database being migrated. We introduce a background task on the destination that slowly fetches the missing tuples. The goal of this task is to fill the missing gaps in the primary index range sets, while minimally affecting performance. There are many potential policies that could be applied here, with various performance trade-offs. In our current prototype, we only implemented a single very simple policy. Every second, the background task checks how many tuples were fetched from the source. The background task then fetches enough tuples to ensure that at least T tuples are fetched every second (we use $T = 1000$ in our implementation). This policy adds no extra pressure during the early part of migration, when many misses cause a lot of data to be migrated, but it maintains a constant transfer rate and thus ensures that the migration will eventually complete. Investigating other potential policies is left as future work.

5.3 Recovery and Fault Tolerance

During migration, the logical state of the database is spread across both the source and the destination servers. As a result, failure of either server can cause the database to become unavailable. The source cannot serve queries without the destination because the destination may contain updates, and because any propagated deletes may cause the source's snapshot to no longer be consistent. If the source in unavailable, the destination can continue to serve any data that it has migrated, but any operation which must migrate data from the source will fail. This is different from the existing commercial systems, which have a well defined single point of time where they switch from the source to the destination. However, the migration procedure should be short, so this should not significantly affect the overall system availability.

In the rest of this section, we describe the details of how our recovery protocol works to handle individual component failures. The assumption is that the individual databases use traditional durability techniques, such as an ARIES-style undo and redo logging [72], or replication. Additional mechanisms are required to recover the state of the migration.

5.3.1 Router Failures

Routing information is stored persistently in a metadata database, and replicated locally at each backend node. Each backend node only maintains the portion of the metadata database that describes the partitions stored locally. The front-end router nodes only maintain a cache of the routing information. Inconsistencies in router caches are detected by the backend nodes, when transactions targeting a partition that has already been migrated are incorrectly sent to the source rather than destination. This is shown in Figure 5-1, when the source node in phase 2 returns a *redirect* response to the router for query Q2, causing it to be reissued to the correct destination. Routers thus contain only soft-state. In the event of a router failure, in-flight transactions are aborted, and routing tables are reloaded from the metadata database when the router recovers from the crash. This also allows us to spawn router nodes on demand, based on load.

5.3.2 Source and Destination Failures

To recover from a failure, the source and destination database servers first perform their standard independent recovery procedures. The migration message deliveries and the current migration state are critical for recovery correctness, and thus these messages are sent using standard reliable delivery techniques. Each message is resent until it is acknowledged by the destination. The messages are idempotent, so it does not matter if the destination receives them multiple times. Migration agents on both source and destination maintain a write-ahead persistent log. Records are force-written before messages are acknowledged by the receiver, and after acknowledgments are received by the sender. In case of failure, the migration agents on the source and destination read the log and can determine which migration phase they were in at the time of the crash.

Simple recovery strategy: A simple strategy for recovery that ensures correctness would be to simply restart migration with empty range sets on the destination. This may cause data to be re-fetched from the source. However, duplicate migrated data that is fetched from the source is discarded when it is inserted, so this is correct.

Efficient recovery strategy: To improve efficiency, we propose a variant of this protocol. In this variant, the destination, after basic database recovery, creates empty range sets for each migrated table and then scans each migrated table, adding range set entries for every unique index. For non-countable types (e.g., floats, strings) range set merging cannot be performed, causing these regenerated sets to contain a large number of entries. For this reason, we do not rebuild range sets for non-countable types in our current prototype. At the end of this procedure, migration can be resumed as usual. This significantly reduces the re-fetching of tuples.

5.3.3 Failure Analysis

To illustrate the correctness of our protocol, in this section we describe what happens after a source and destination failure in the various phases of migration shown in the sequence diagram in Figure 5-1.

Destination Failures In the event of a destination failure:

- *Before setup:* nothing to be done as migration has not begun.
- *Between setup and prepare:* abort migration by rolling back any setup.
- *Between prepare and switch:* nothing to be done. Continue to queue transactions and wait for the `SWITCH` message.
- *Between switch and end:* Optionally load the range sets by scanning the local database, and restart processing user transactions in migration mode. Duplicates are ignored by regular processing.
- *After end:* standard database recovery is sufficient.

Source Failures In the event of a source failure:

- *Before prepare:* standard database recovery is sufficient;
- *Between prepare and switch:* do not accept new transactions, and send the `SWITCH` message to the destination.
- *Between switch and end:* do not accept new transactions, and keep serving data requests from the destination.
- *After end:* Standard database recovery is sufficient, repeating the cleanup if needed;

This procedure will recover any application modifications that committed at any point during the migration. There are two cases; either:

1. The transaction committed while the source was executing transactions, in which case the source will ensure that it was durable, and the migration procedure will eventually migrate the changes to the destination, or
2. The transaction will have committed while the destination was executing transactions. In this case, the destination's local recovery will recover the modifications, and even if an older version of those tuples are migrated again, these older tuples will be discarded.

5.4 Optimizations

The previous sections discussed the basic components of Wildebeest and the recovery protocols. We now focus on a series of optimizations that provide significant performance improvement. The final system performance is validated in Section 5.5.

5.4.1 Batching

In Section 5.2 we described the basic policy in which we fetch the "bare minimum" data required to answer the user queries. However, due to per-query overhead, it is often more efficient to process larger groups of tuples at a time. This amortizes the per query costs of network messages, source data index lookups, and the destination writes across a larger number of tuples at a time.

We extended our query rewriting, described in Section 5.2.2, to fetch a configurable batch of tuples of size B_{size}. It uses the range sets to determine lower and upper bounds for the missing range of tuples, and adds a `LIMIT` clause. For example, if we have fetched ranges [0, 20], [30, 40] and we get a query for $id = 22$, we will rewrite the query to $id \geq 22$ `LIMIT B`.

This rewriting guarantees that we fetch all tuples required to answer the user queries, plus additional tuples up to either the B_{size} result set size limit B_{size}, or the end of the gap, whichever comes first. This avoids performing duplicate work on both the source and the destination, but leverages batching where possible. Increasing the batch size more aggressively fetches tuples from the source, which causes migration to complete faster, but also means that each migration query is more expensive for both the source and the destination. We explore the performance impact of this parameter in Section 5.5.

We choose to always scan forward from the requested key. The primary reason is that scanning in one direction (the increasing direction, in our case) avoids potential deadlocks when two scans in opposite directions collide. A second reason is that is a simple technique that works for uncountable ranges. For countable ranges, it is possible to use a more intelligent implementation that attempts to always fetch a range of size B_{size}, which may require starting at a value below the key. This would avoid some fetches from the source server, which would likely improve performance, at the cost of having some additional logic to determine the appropriate rewriting.

5.4.2 Write-Behind Caching

Migrating data from one database to another involves a significant amount of writing data to disk at the destination. Migrating at the logical level also requires secondary indexes to be rebuilt, which increases the cost of these inserts. The basic implementation we discussed earlier directly inserts migrated tuples into the destination database, as part of the same transaction as the query that needs to fetch them. While this is correct, it seriously impacts client latency and the system throughput. The additional inserts transform read-only transactions into read-write transactions, meaning that they must now be synchronously logged to disk. Moreover, this can decrease concurrency by transforming shared locks into exclusive locks, creating additional read-on-write conflicts.

To ameliorate this issue, we use write-behind caching. We insert migrated tuples into a main-memory transactional table, as an operation outside the transaction. This table implements two-phase locking, and aborts, but is not recoverable (e.g., does not use write-ahead logging.) This avoids additional logging for reads, and allows migration writes to be performed lazily in the background. Updates, inserts, and deletes are still performed as described earlier, except that deletes and updates must acquire exclusive locks on the main memory table, and both must remove any records they touch from the main memory table before releasing those locks. We rely on the fact that the data is already persistently stored on the source database. For this reason, this is crash-safe, even if the contents of this table are lost. As discussed in the previous section, in case of failure the recovery procedure will re-fetch any missing tuples from the source, merging them with the tuples that have been migrated and persisted at the destination.

Queries must now search two tables. Equality queries first search the main-memory table, then the traditional database table if the key was not found. Range queries must search both tables. This is similar to how log-structured merge trees[83], where the state of the table is contained in multiple trees.

In order to eventually complete migration, a background task runs whenever the main-memory table is not empty, and continuously reads ranges of tuples and inserts them into the final table. This insert task can be tuned to adjust the rate and batch size of transfer operations. More frequent transfers cause the migration to complete faster, but increases the performance impact on the destination, which could affect application queries. A larger batch size is generally more efficient, but similarly is more expensive to process.

5.5 Experimental Evaluation

We experimented with several scenarios designed to isolate different performance characteristics of our design. We used our Relational Cloud system, which combines uses Dtxn with MySQL as the storage engine. In each scenario, we run a workload against the source server for five minutes, then migrate it to an identical destination server. During the entire experiment we monitor the throughput and latency of transactions at the clients. Since we are interested in OLTP and web applications, we size our databases so that the working set fits in RAM, as this is the typical configuration for modern systems.

We test the performance of Wildebeest on the Yahoo Cloud Serving Benchmark (YCSB-B) [26], which is designed to model web applications. This workload uses a single table. Each tuple has an integer primary key and ten fields each containing 100-byte ASCII text values. Each client selects a tuple from a single table by primary key. The value is chosen using a Zipfian distribution, such that 50% of the accesses are to 6.5%. of the keys. The operations are 95% reads of the entire tuple and 5% updates of one of the values. For most experiments we used a database composed of 5 million rows, which occupied 6.5 GB of space on disk. For the scale out scenario, we used a smaller database of 1 million rows.

While our system is designed for general purpose multi-statement transactions, this simplified workload permits us to avoid some of the complexities of analyzing the performance of a more sophisticated transactions. For example, our throughput is CPU limited in these experiments. Despite its simplicity, the workload is representative of a significant fraction of the queries and updates in OLTP and web applications that retrieve records via indexed fields.

We consider three scenarios:

1. Complete database migration with clients generating a fixed load that is less than peak. This represents the case where we can predict that a database needs to be moved before it is heavily loaded, perhaps for load balancing purposes or hardware maintenance. This is discussed in Section 5.5.1.

2. Complete database migration with clients generating peak load. This case represents a database that needs to be moved because the physical node it is running on is over-committed/saturated. This is discussed in in Section 5.5.2.

3. Partial migration from a machine running at peak capacity to a second machine. In this test we move half the database and load. This demonstrates Wildebeest's ability to *elastically scale* a database. This is discussed in detail in Section 5.5.3.

We compare Wildebeest to stop and copy, replica failover, and virtual machine migration. These three techniques are supported by MySQL, with additional tools and software. They only support complete migration of a partition, and thus cannot directly support the partial migration scenario like Wildebeest can. For these technologies, we pre-partitioned the system using multiple instances of MySQL.

Stop and copy shuts down the source database, copies all the data files, then starts the database at the destination. Before shutting down the source, we allow all in progress transactions to finish. Stop and copy is a simple technique, but the partition is unavailable for the duration of the migration. For MySQL, we found that doing a "dirty" shutdown by killing the source database and performing recovery on the destination had shorter periods of unavailability than using a clean shutdown, so we use that version here.

Replica failover brings up a new replica from a checkpoint, catches it up to the original, then fails clients over from one to the other. In the case of MySQL, we create a hot checkpoint from the running database using Percona XtraBackup [86]. The new destination replica then synchronizes with the source using MySQL's built-in replication support. After the destination in nearly caught up, we stop accepting transactions at the source. Once the destination is completely caught up with the source, it becomes the master and we redirect clients. Replica failover only has a brief pause when allowing the destination to catch up completely with the source. However, the migration period is much longer than with stop and copy, because not only does it need to copy the data, but it also needs the destination to catch up with the in flight transactions. Additional resources are used during the copy period, so while the partition remains available, its performance is impacted. It is also possible that the destination cannot catch up to the source. In particular with MySQL, the destination executes replication writes in a single thread, while the source executes them on many threads. In YCSB-B with the default ratio of writes (5%), this is not a problem. However, when the write rate increases to approximately 30% on our machines, the destination can no longer catch up. If this happens, the pause time will be much longer, as the number of operations that must be replayed will be large.

Virtual machine migration runs the database in a virtual machine and uses its built-in migration support to migrate the database instance from one machine to another. In this case, the database is stored on a shared storage device. However, since we focus on memory-resident workloads, this does not affect the performance. VM migration copies the in-memory image while the virtual machine is running. There is a small pause while copying the last remaining state, but the migration is designed to minimize this period.

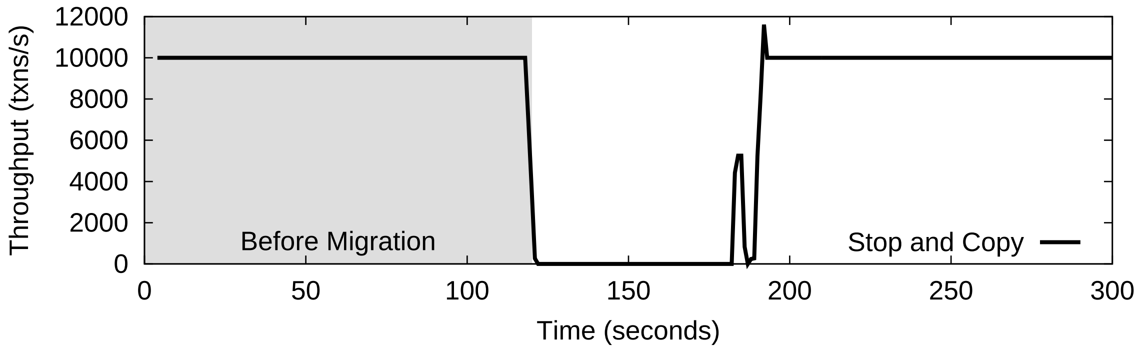
Our experimental test bed involves four identical Dell PowerEdge R710. Each has two quad core Intel Xeon E5530 2.4 GHz processors, for a total of 8 cores, although we limit our tests to 2 cores on each processor, for a total of 4 cores, in order to avoid scalability limits in our prototype code. Each system has 24 GB of RAM, and a 6 2TB 7200 RPM SAS disks (Seagate Constellation ST32000444SS; 150MB/s sequential IO; 8.5 ms seek), configured as a single RAID 5, formatted with ext4. These systems are interconnected to a single gigabit Ethernet switch. We used MySQL version 5.5.7 running on Ubuntu 10.10 with kernel version 2.6.35. We configured MySQL to use a 7 GB buffer pool, which is enough to hold the 6.5 GB database table.

For the virtual machine experiments, we used a commercially available virtual machine monitor (VMM). One server was used as a network attached disk server to store the virtual machine image and database (to use the VMM's migration facilities, we had to store data on a shared disk). The virtual machine migration used a separate network port from the application traffic, as recommended by the vendor. This provides it an advantage over Wildebeest and the other migration techniques, which were only configured to use a single port. We give each virtual machine 8 GB of RAM, which provides 7 GB of memory of the buffer pool, and 1 GB of memory for the database server and operating system. Although the performance of local and shared disk is quite different, we minimize the impact of these differences by using a database whose working set fits into main memory and using a benchmark that is CPU bound.

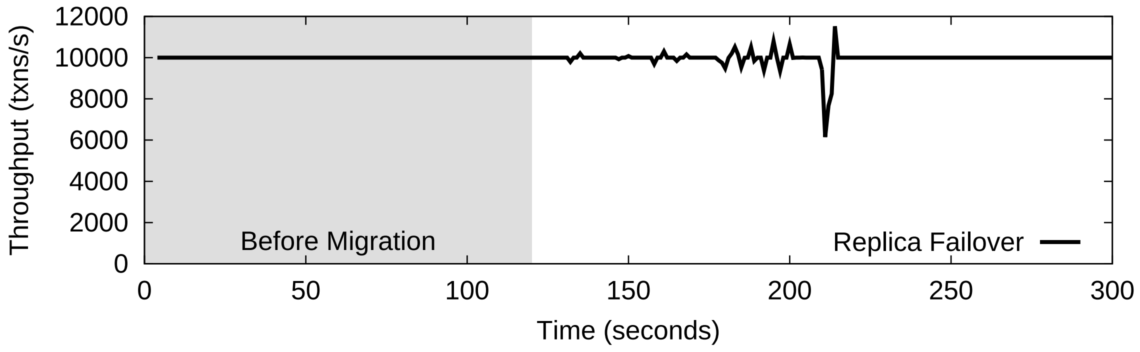
For each experiment, we measure the latency for each individual transaction. We bucket the results using a 3 second sliding window, computing the throughput and latency distribution in this window. We consider three measures of performance: i) the "visible" migration time, i.e., the portion of migration during which time the user perceives lower throughput; ii) the 95th percentile of latency during migration; iii) and the aggregate number number of "unserved" transactions, i.e., the integral of the difference between the transactions requested by the users and the transactions served by the system during migration. While we only show the results of a single experiment here, we did repeat each experiment 3 times, and the results are repeatable.

5.5.1 Complete Migration With Rate Limit

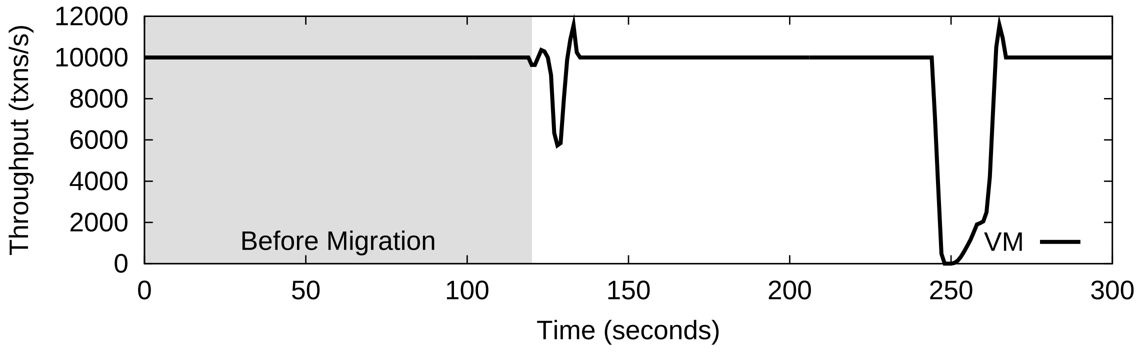
In the first scenario, the load is fixed at a rate of 10,000 or 20,000 client transactions per second. At a rate of 10,000 transactions per second, Wildebeest, replica failover and VM migration are able to handle the load during the migration, without a decrease in throughput, as shown in Figure 5-2. As expected, stop and copy shows a pause in Figure 5-2a as it takes 60 seconds to copy the database and 4 seconds to perform recovery, for a total downtime of approximately 64 seconds. Replica failover, in Figure 5-2b, shows only a small throughput impact during the copy. For just one 1 second interval, the throughput



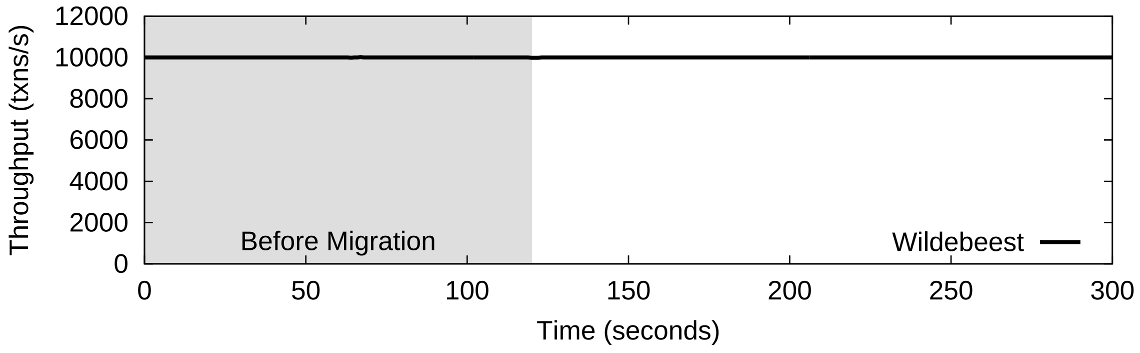
(a) Stop and copy



(b) Replica failover

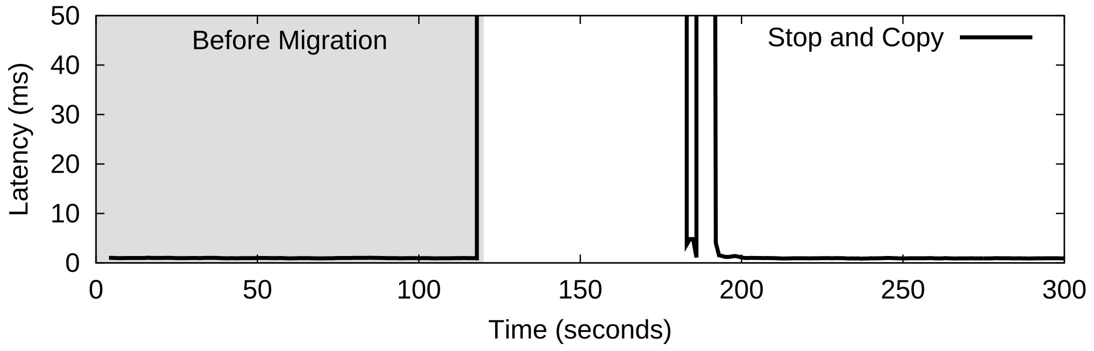


(c) Virtual machine migration

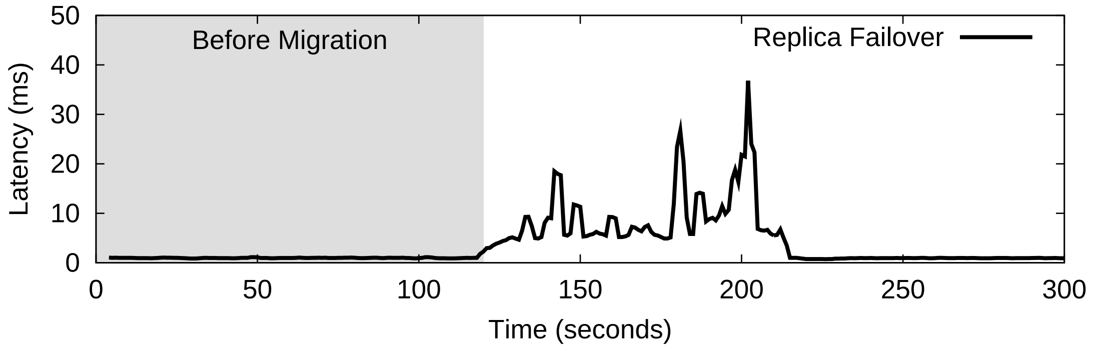


(d) Wildebeest

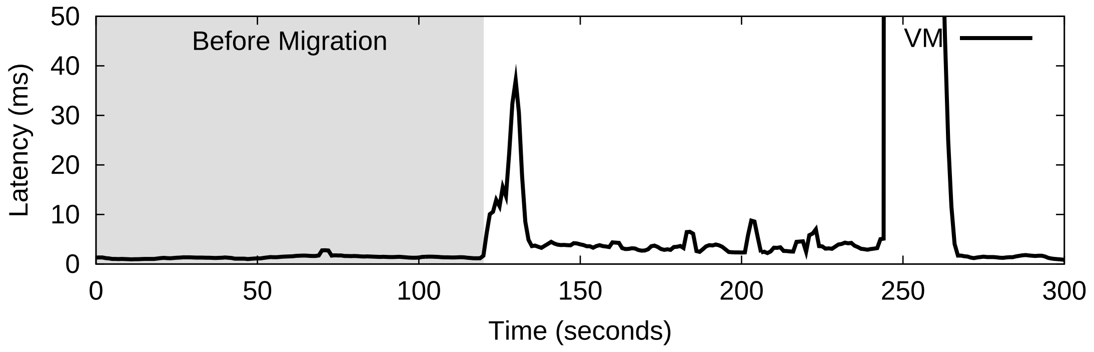
Figure 5-2: Throughput at a desired rate of 10,000 transactions per second



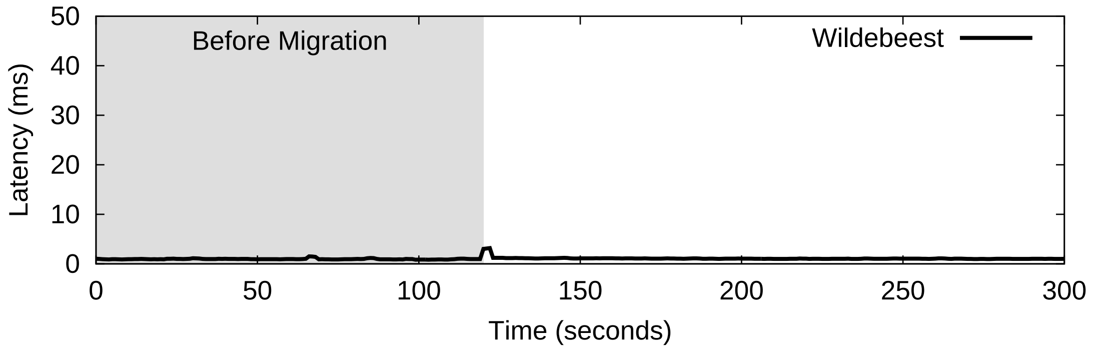
(a) Stop and copy



(b) Replica failover



(c) Virtual machine migration



(d) Wildebeest

Figure 5-3: Latency at a desired rate of 10,000 transactions per second

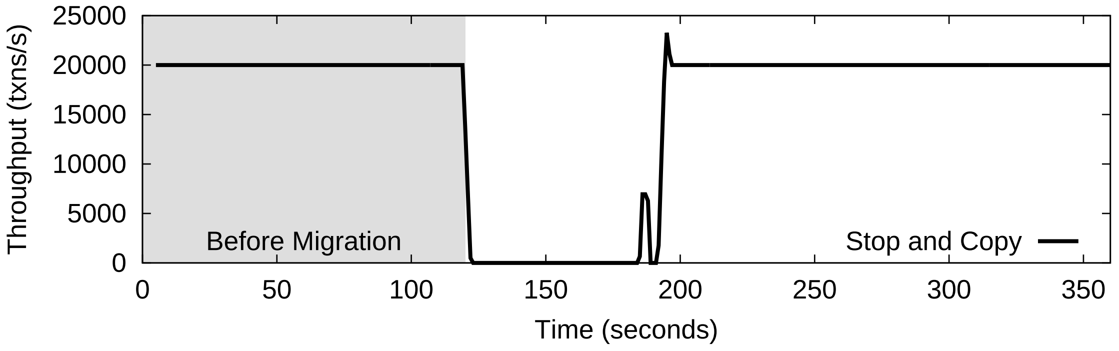
falls to 6,148 transactions per second. The VM approach (Figure 5-2c) requires a small amount of time to take a snapshot at the beginning of migration, decreasing throughput to 8,910 transactions per second for one second after migration. It also has a pause during the handoff at the end of migration, which in this case results in zero throughput for six seconds. Wildebeest shows nearly no impact on throughput, with only a tiny decrease down to 9,939 transactions per second during the one second window immediately after migration begins. Thus, it appears as a straight line in Figure 5-2d. The latency results in Figure 5-3 show similar trends as the throughput results. Wildebeest has a short period with increased latency, while the others have much larger increases.

At a rate of 20,000 transactions per second, shown in Figure 5-4, only Wildebeest is able to sustain a desired load. Stop and copy has a complete outage, while replica failover and VM migration decrease to around 15,000 transactions per second for around 100 seconds. They are unable to keep up with the request rate because of the additional resources required to copy the state during migration. The 95th percentile latency for Wildebeest at a rate of 20,000 transactions per second, in Figure 5-5d, shows that Wildebeest experiences a short pause at the instant migration is triggered. The latency increases from 2 milliseconds to around 8 milliseconds. During migration itself, latency is more than doubled than before migration time, hovering around 5 milliseconds. The reason is that the first queries must perform a "cache fetch" query at the source. However, latency falls back down as more of the workload is "cached" at the destination, settling to a value that matches the original latency after approximately 170 seconds. Replica failover and VM migration both have worse latency during the migration, with latencies around 20 ms and 15 ms, respectively. These results show that Wildebeest has minimal impact on the application when there are sufficient resources for the migration process. Other techniques will have more of a visible impact in this scenario.

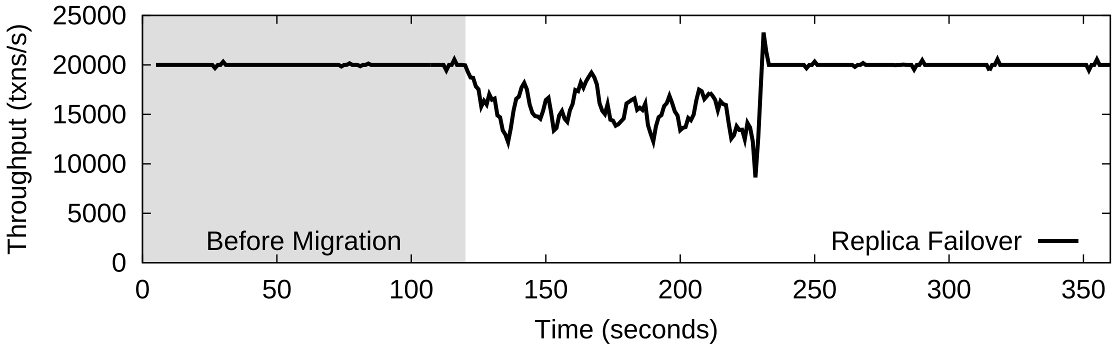
5.5.2 Complete Migration at Maximum Load

In this scenario, a database that is running at full load is migrated from one server to another. The throughput for replica failover and VM migration again falls to approximately 15,000 transactions per second during the period that state is being copied, as shown in Figure 5-6b and Figure 5-6c. Replica failover takes 103 seconds to copy the database state, during which the throughput is reduced. It then takes an additional 125 seconds for the destination to perform recovery on the checkpoint, then 81 seconds for replication to catch up to the source. In this case, replica migration takes a total of 310 seconds to actually transfer the data from the source to the destination and resume with full throughput. VM migration takes a total of 140 seconds, and again has a zero throughput period of approximately 11 seconds at the end.

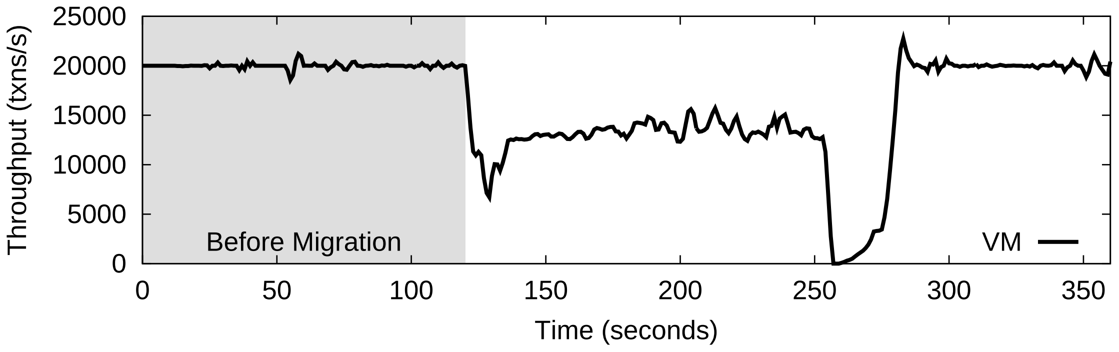
In this scenario, Wildebeest's performance is decreased for approximately 70 seconds after migration, shown in Figure 5-6d. Initially the performance is poor, falling as low as 7,000 transactions per second. However, it gradually increases as fewer and fewer requests require fetching data from the source. Before and after migration, the virtual machine performance is worse than the other techniques, with the virtual machines achieving 25,000 transactions per second while MySQL running directly on Linux achieves nearly 33,000 transactions per second. This difference appears to be due to the overhead of running in



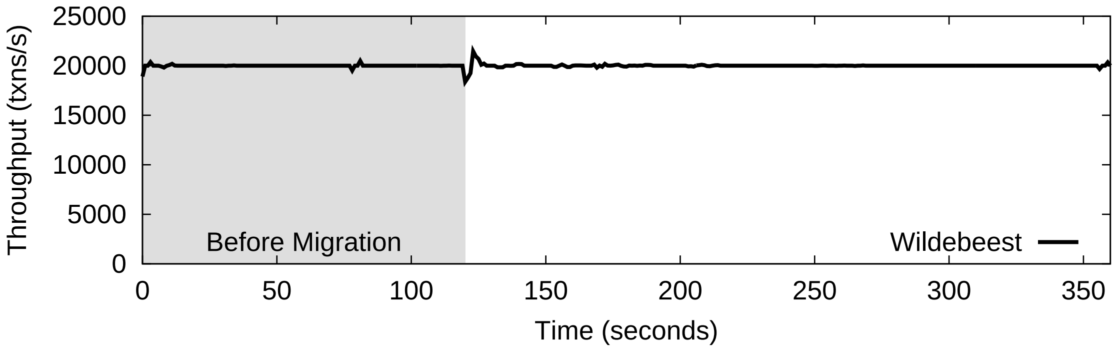
(a) Stop and copy



(b) Replica failover

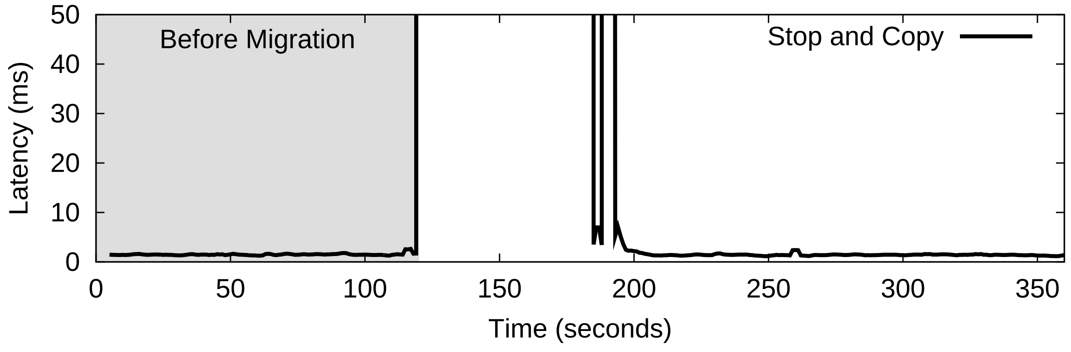


(c) Virtual machine migration

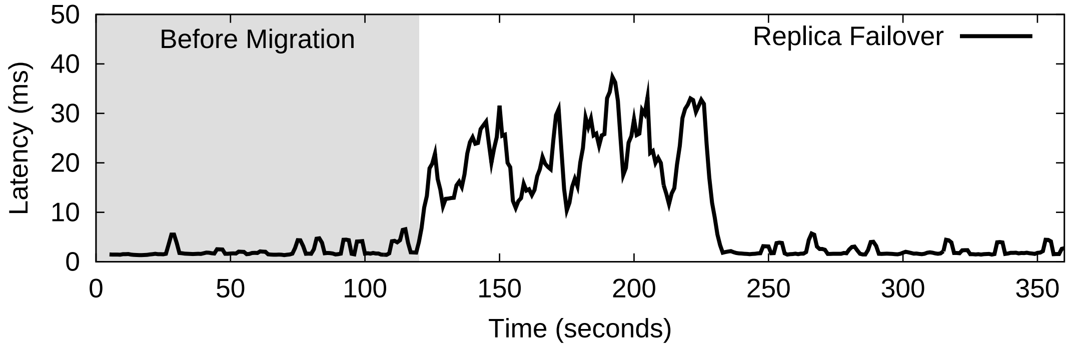


(d) Wildebeest

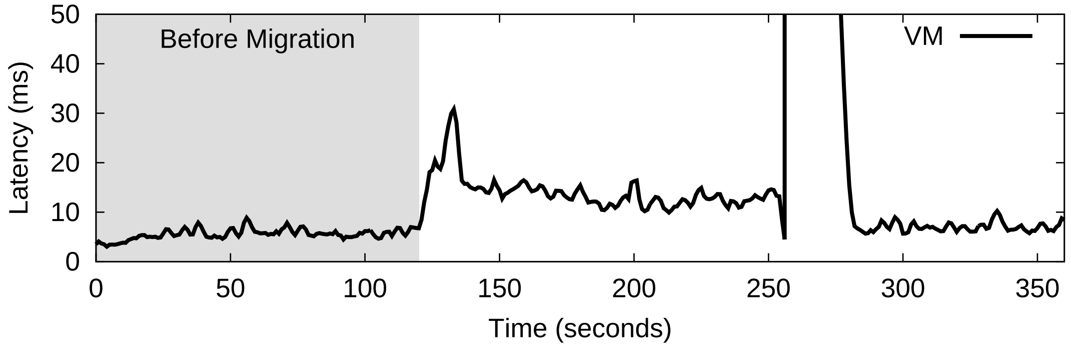
Figure 5-4: Throughput at a desired rate of 20,000 transactions per second



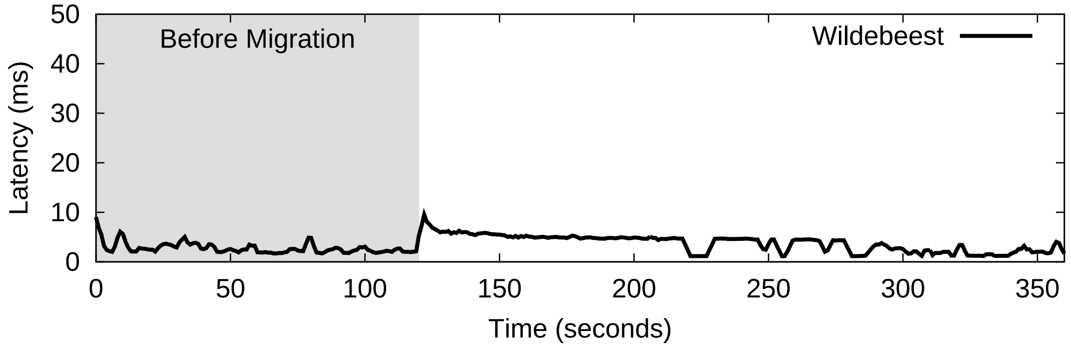
(a) Stop and copy



(b) Replica failover

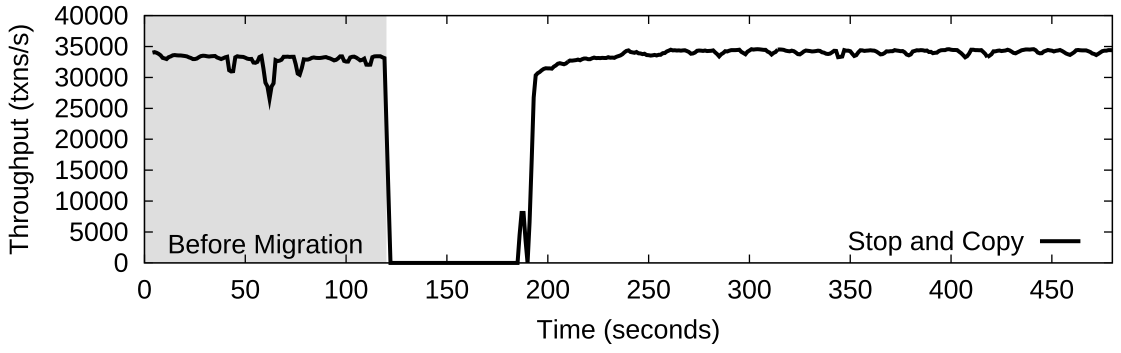


(c) Virtual machine migration

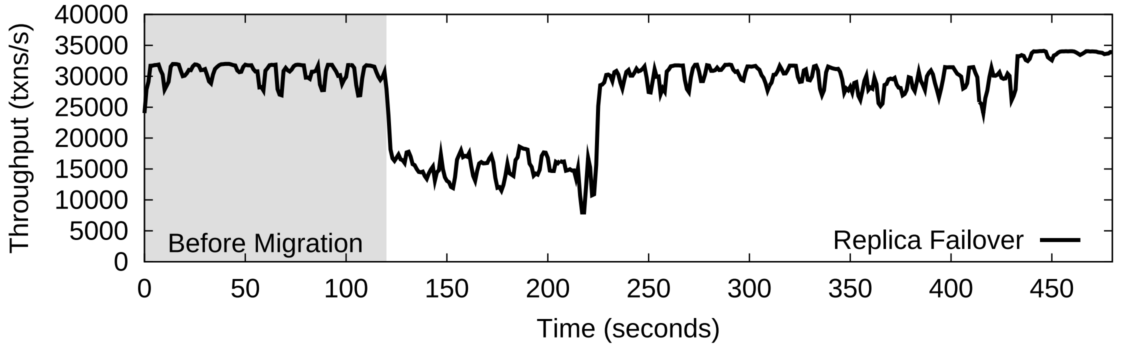


(d) Wildebeest

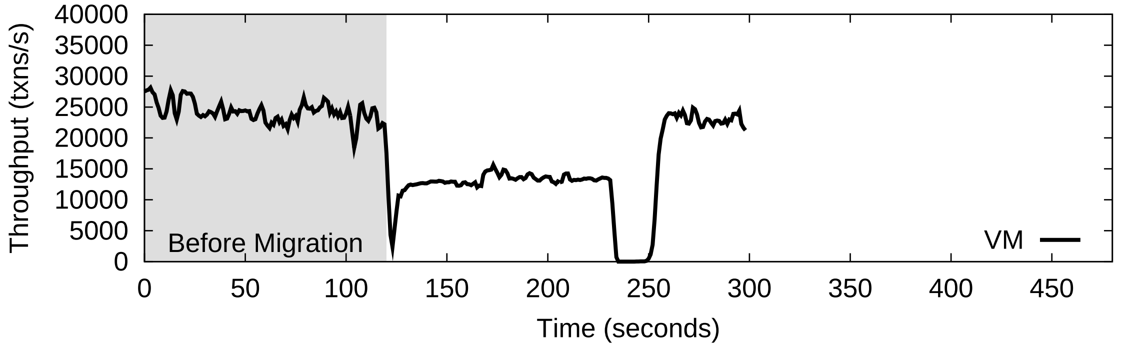
Figure 5-5: Latency at a desired rate of 20,000 transactions per second



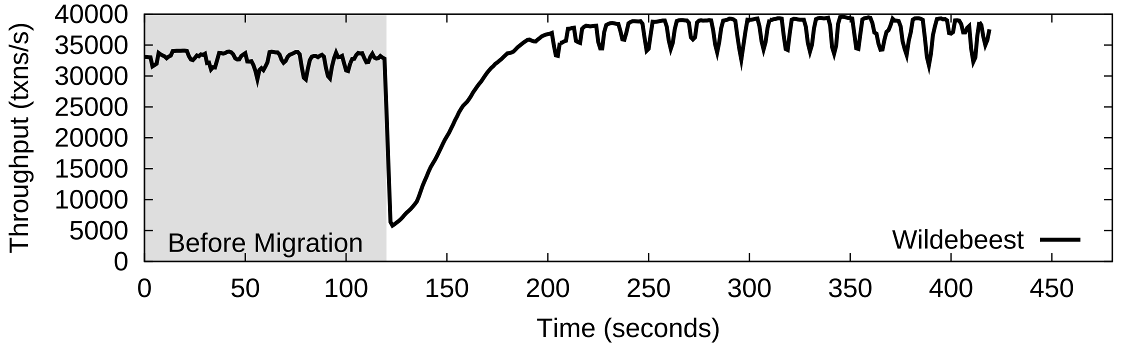
(a) Stop and copy



(b) Replica failover

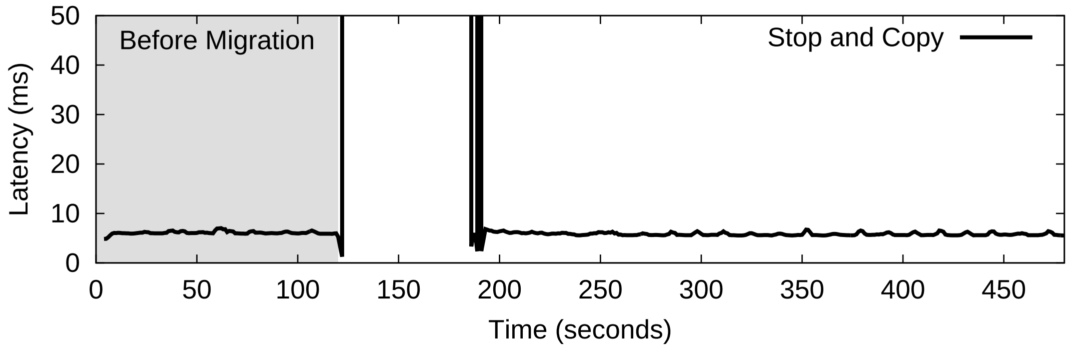


(c) Virtual machine migration

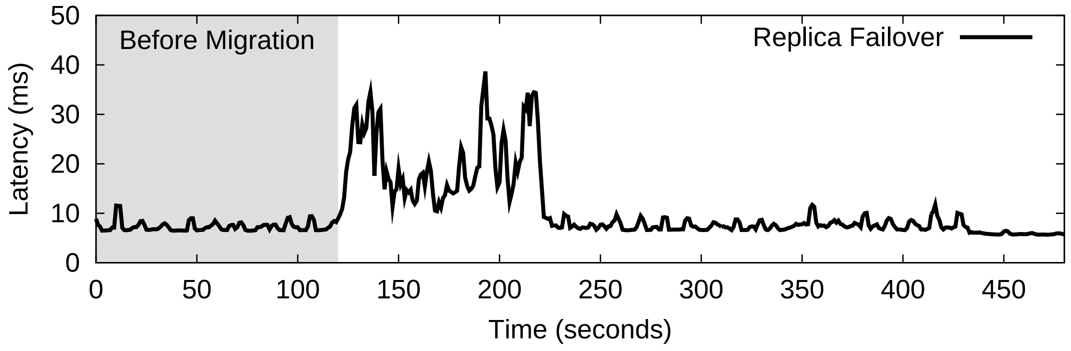


(d) Wildebeest

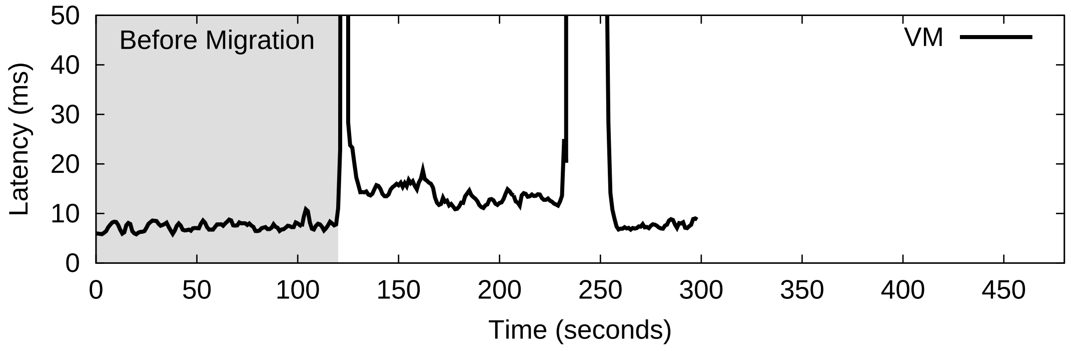
Figure 5-6: Throughput during complete migration at maximum load



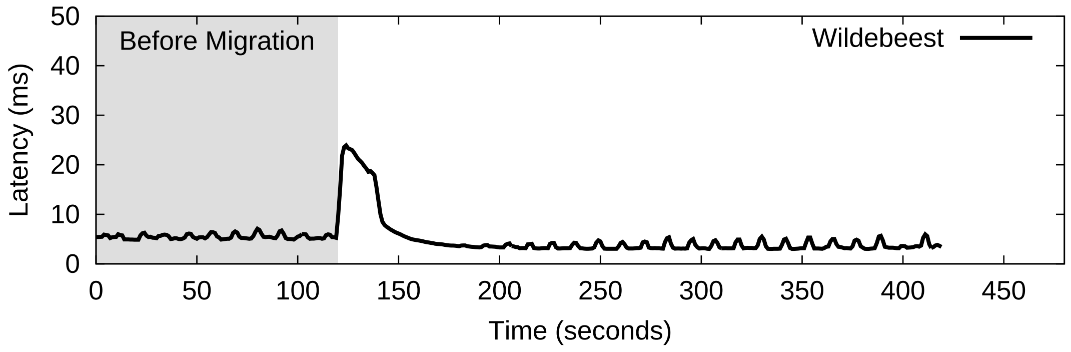
(a) Stop and copy



(b) Replica failover



(c) Virtual machine migration



(d) Wildebeest

Figure 5-7: Latency during complete migration at maximum load

a virtual machine. Also note that Wildebeest's performance after migration is better than before migration. This is because our prototype main-memory cache is specialized for integer primary keys, and is not a general purpose relational database table. Thus while data is served from the cache it is actually faster than when server MySQL. Wildebeest's throughput eventually decays back to MySQL's throughput once all the requests are being served from MySQL directly. For completeness, the latency results for this experiment are shown in Figure 5-7.

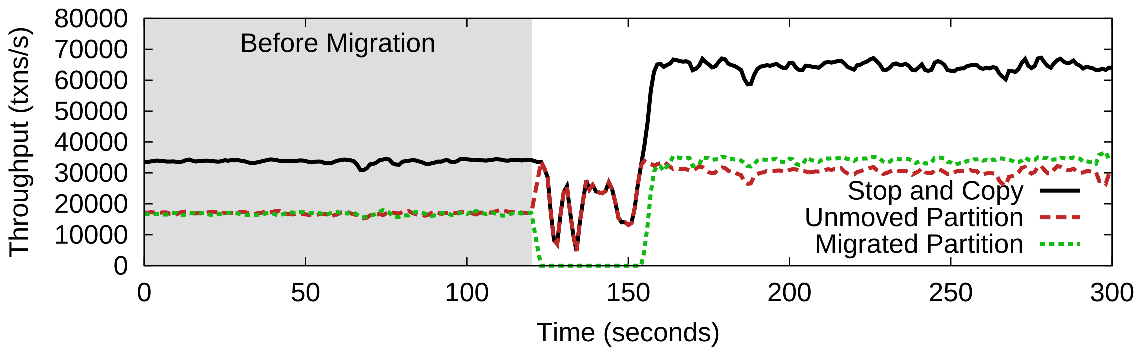
5.5.3 Scale Out Scenario

In the third scenario, we have a database that is overloaded. To increase the throughput, we move half of it to another server by range partitioning the table into two equal sized pieces. Our migration technique is able to re-partition the database while running, and can directly copy just half the data. For this scenario, we use a smaller 1 million row table which occupies approximately 3 GB of disk space (1.5 GB per partition). Requests are made with a uniform request distribution, in order to get perfect scale out. Wildebeest is able to split the single database into two, due to its support for partial migration. For the other techniques, we partition the database in advance, running two instances of MySQL with 2 GB buffer pools. For virtual machine migration, each virtual machine has 3 GB total memory. The combined throughput and the throughput for each partition is shown in Figure 5-8. The 95th percentile latency is shown in Figure 5-9.

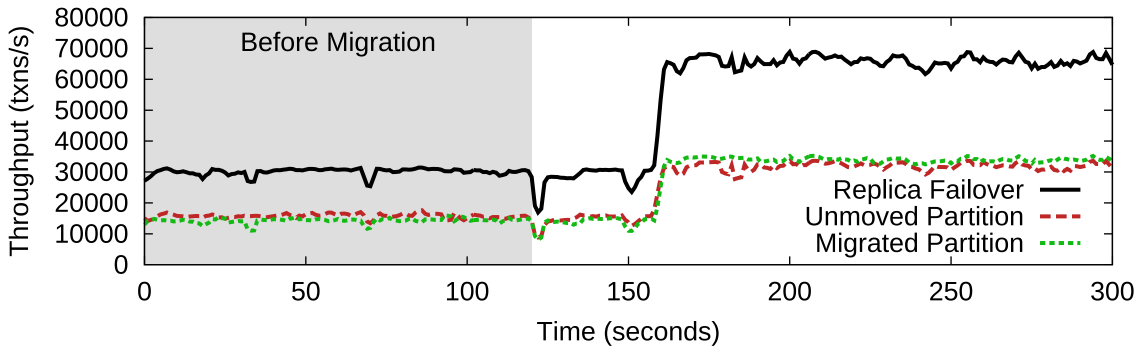
Looking at the combined throughput in this scenario, stop and copy appears to continue processing requests during the migration with only a modest throughput decrease, when in previous scenarios it served no transactions. This is because the partition that is not migrated continues to serve transactions in this case. Thus, the aggregate throughput is not far from the throughput before migration, since when one partition is stopped, the other partition can consume the extra resources. As shown in Figure 5-8a, the throughput for the migrating partition falls to zero, but the throughput for the remaining partition nearly doubles. Thus, while it looks like stop and copy has limited impact, in fact half the database is unavailable for 35 seconds.

Replica failover, on the other hand, has a small throughput decrease while the data is being copied, which takes about 15 seconds. However, the overall migration takes longer, for a total of 39 seconds before the second machine is utilized. Virtual machine migration is faster, taking 24 seconds to move the 2 GB database. However during this time, its throughput drops from approximately 30,000 transactions per second to 20,000 transactions per second, a decrease of 33%. This shows that there is a trade-off between how aggressive the system should copy state and the impact on the throughput. In all cases, after migration has completed, the system nearly doubles its throughput because it can use both machines.

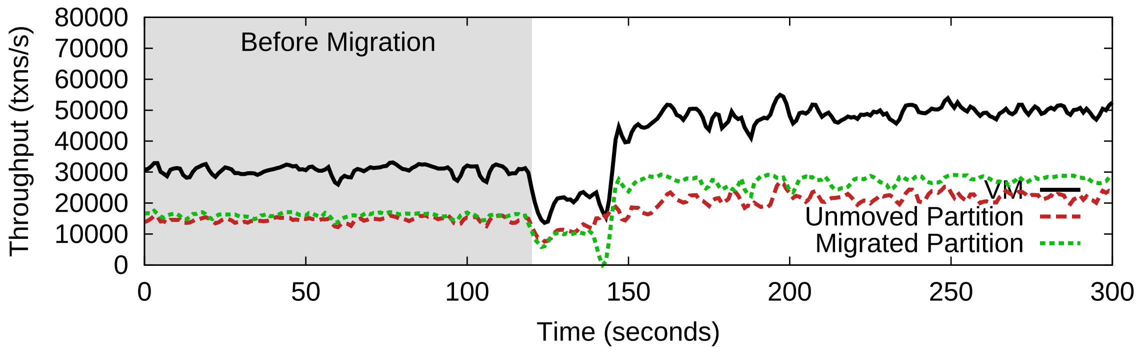
Wildebeest is able to double the throughput in significantly less time than the other approaches. It has a period of approximately 5 seconds where it experiences reduced throughput, after which it jumps up. This happens quickly because it immediately offloads work to the destination machine. As before, the throughput ends up more than double because our prototype's in-memory table is specialized, so reading from it is actually faster than reading from MySQL's buffer pool. The important part here is that if this were a real production system that were overloaded, Wildebeest would provide much better quality of service by



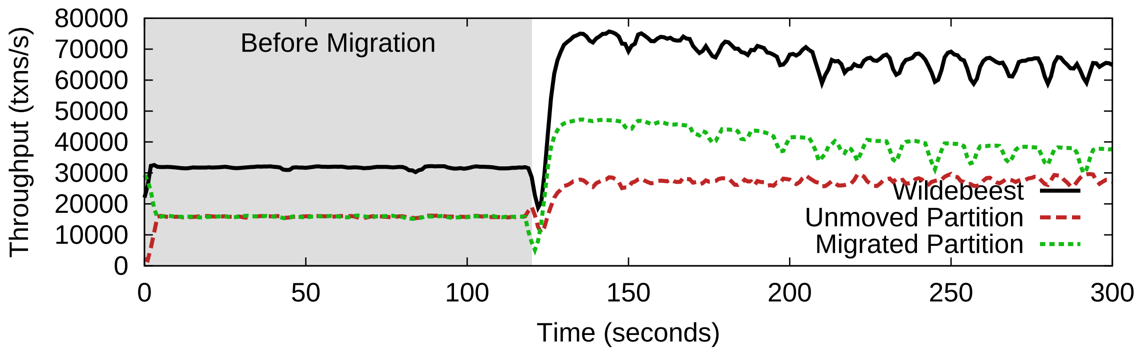
(a) Stop and copy



(b) Replica failover

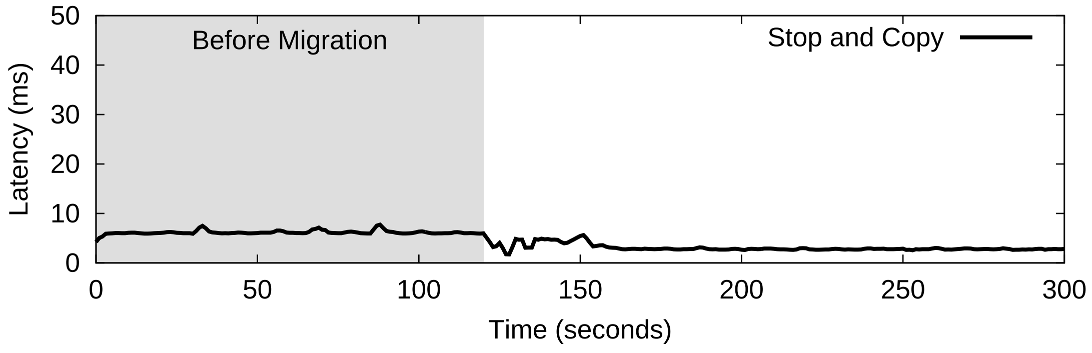


(c) Virtual machine migration

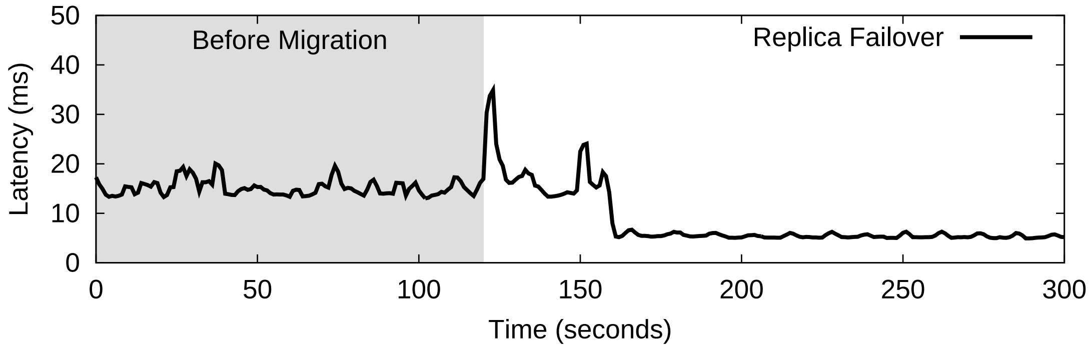


(d) Wildebeest

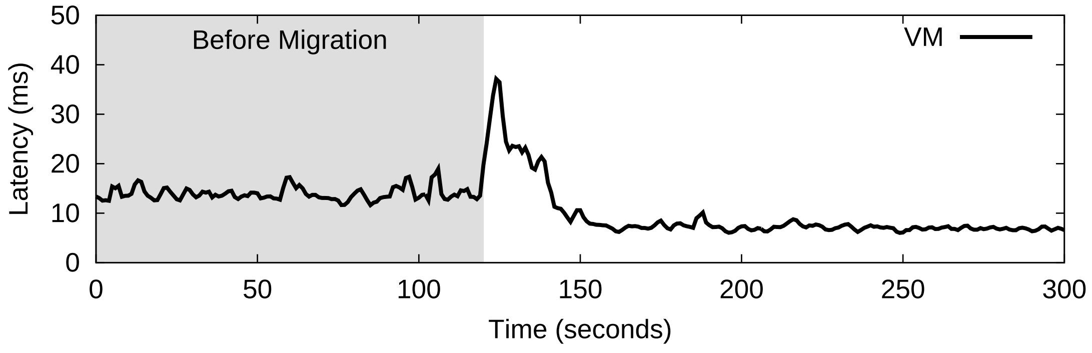
Figure 5-8: Throughput when scaling from one to two machines



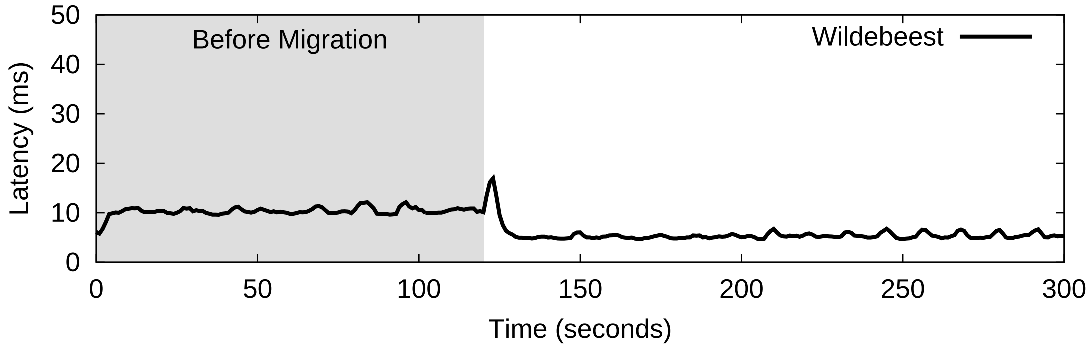
(a) Stop and copy



(b) Replica failover



(c) Virtual machine migration



(d) Wildebeest

Figure 5-9: Latency when scaling from one to two machines

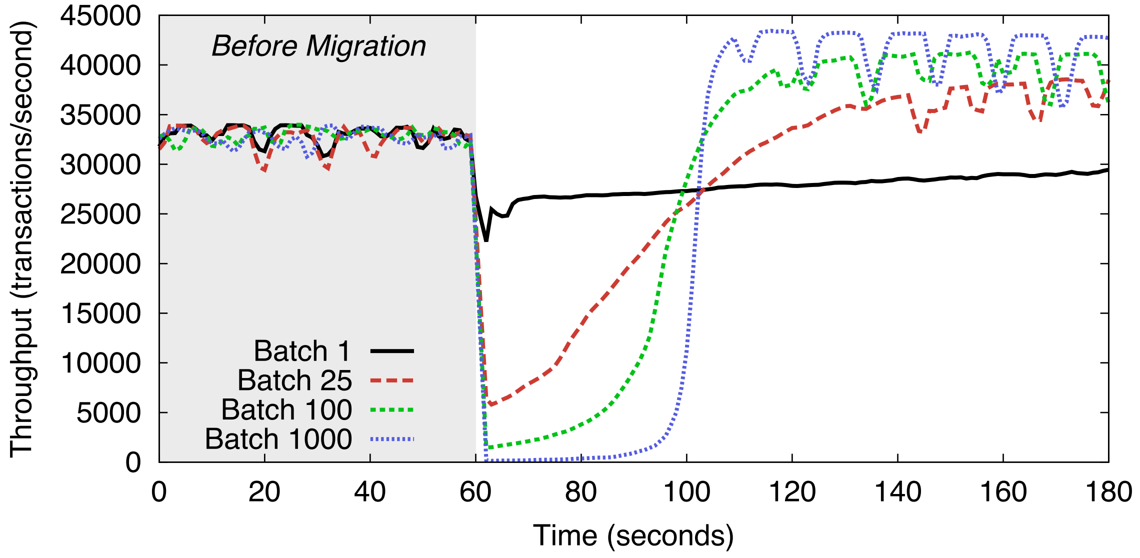


Figure 5-10: Throughput with Wildebeest when changing the batching size (B_{size}).

being able to react quickly to changes in load.

5.5.4 Impact of Batching Size

As a final experiment we show the impact of batching on the migration performance. We use the skewed full database example of Section 5.5.2, and present the impact on throughput of various batch sizes. As we show in Figure 5-10, a larger batch size (B_{size}) causes more aggressive data fetching at the beginning of migration. The effect is a more pronounced initial impact on performance for few seconds, followed by a quicker transition to almost full speed.

On the contrary, a smaller batch size spreads the migration costs over a longer period of time. It produces as a smaller impact on throughput, but it lasts longer as more individual requests must be sent to the source database. In the extreme case of a $B_{size} = 1$, Figure 5-10 shows that the throughput "settles," but at a lower level, which is then very slowly increasing. This is because nearly every request requires a fetch from the source server, and so the overhead of all these extra fetches decreases the systetm performance, and the throughput slowly increases as the hit rate improves. This shows that avoiding excess query overhead is important for good overall system performance.

This batch size has a similar impact on the source server. The larger batches cause high load as soon as migration is triggered, but it decreases rapidly, whereas small batches cause the load on the source server to be spread out over a longer period of time. For fully loaded scenarios like the one we show in Figure 5-10, different utilization scenarios might favor more or less aggressive batching.

As discussed before, in rate-limited scenarios our migration strategy is nearly completely invisible. In such scenarios, the choice of batch size should be the largest compatible with little perceivable performance degradation. This will minimize the overall migration time by more aggressively fetching tuples while maintaining performance. As part of our

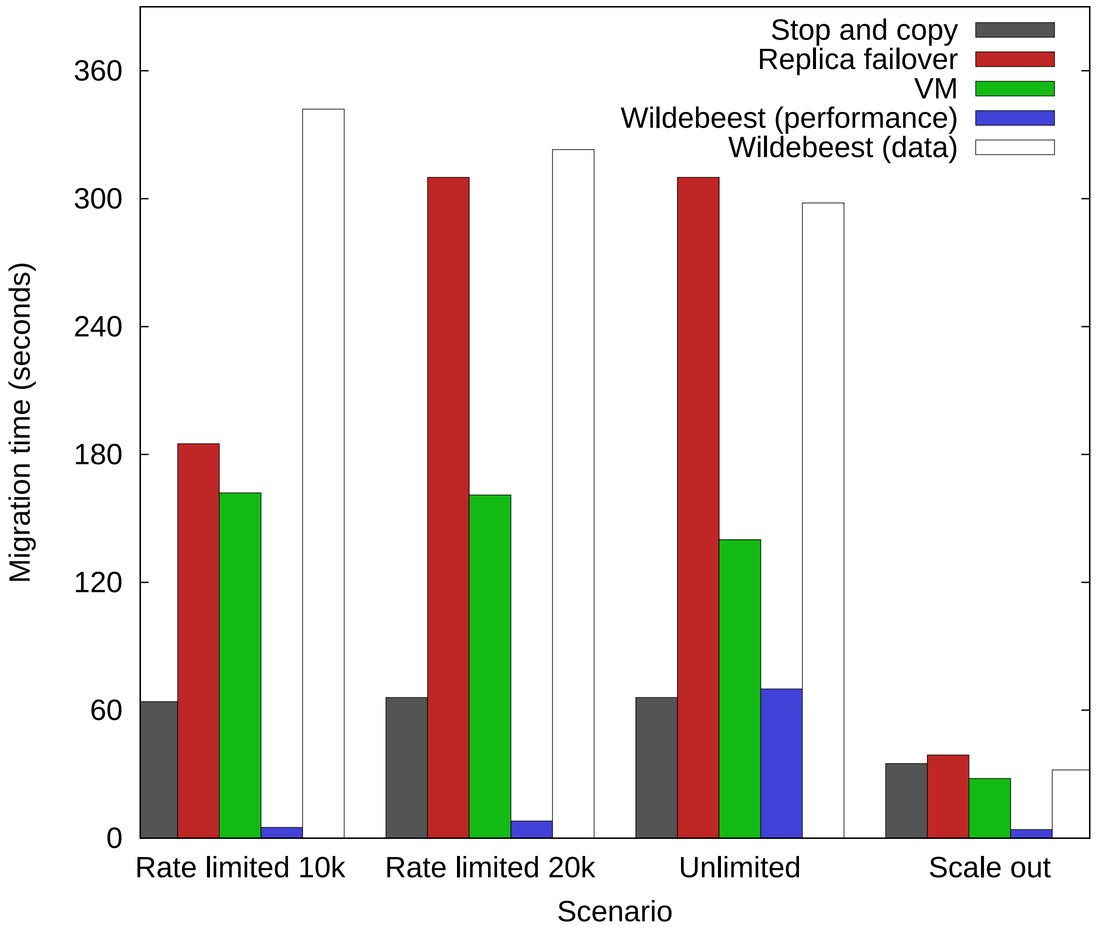


Figure 5-11: Time to migrate the complete database.

future work we plan to investigate algorithms to automatically adjust batching size on-line.

5.5.5 Migration Times

An important metric is how long the migration actually takes to complete for each technique in each scenario. The times are plotted in Figure 5-11. For Wildebeest, we included two times: the time that the performance returns to normal, and the time that all the data is actually migrated. Since Wildebeest moves data on demand, the time until the performance returns to normal is the fastest of all the techniques, while the time to move all data is the longest. For these experiments, we used a policy where we fetched a minimum of 10,000 rows per second. This places an upper bound on the length of time of the migration, while controlling the number of additional requests on the source and destination.

For the commercially available techniques, stop and copy takes the least time to complete. However, the database is unavailable during this time. Virtual machine migration is faster than replica failover, but also has more performance impact. In general, there is a trade-off between the performance impact on the running workload and the time it takes to migrate. A faster migration requires more resources.

5.5.6 Summary of Results

Dtxn, with its Wildebeest live migration system, is able to migrate data from one partition to another, with minimal visible performance impact. Our results show that it has shorter periods of time with reduced performance, when compared to stop and copy, replica failover, and virtual machine migration. Its use of logical data migration means that it is easily able to support partial migration. The combination of that with the fetch on demand policy means that it quickly offloads the source server, in case of heavy load. This means that it can be used in a reactive way to scale a system that is overloaded. This allows Dtxn-based systems to be scaled in an elastic way.

Chapter 6

Related Work

Distributed database systems and main-memory databases have been widely studied, and others have written excellent surveys of these subjects [30, 36]. The key design decisions that differentiate Dtxn from this previous work is that it relies on speculative concurrency control rather than locks, uses replication instead of write-ahead logging, provides a transaction protocol optimized for single partition transactions, and can migrate data at a logical level through a cache-based approach. In this chapter, we review the work that is relevant to these aspects of the design. We begin by reviewing the work on some of the related fundamental building blocks: concurrency control for main memory databases, distributed commit protocols and database replication. We then discuss previous work on database migration, and conclude by reviewing similar academic and industrial systems.

6.1 Concurrency Control

Databases rely on concurrency control to provide the illusion of sequential execution of transactions, while actually executing multiple transactions simultaneously. It is typically assumed that transactions provide serializable consistency, where the results of a set of transactions are equivalent to executing those transactions in some serial order. This provides a powerful abstraction: the transaction effectively sees a database that is unchanging, except by the transaction's own updates. Dtxn provides serializable transactions because it makes writing applications easier.

The most common technique to provide serializable consistency is strict two-phase locking, where a transaction acquires read and write locks on data items as they are accessed, then releases all the locks at the end of the transaction [7, 43]. However, locking adds overhead to each data access due to the bookkeeping required to acquire and release locks, it limits concurrency in cases of conflicting accesses, and adds overhead due to deadlock detection. Many researchers have noted that in main-memory systems, manipulating a lock can cost approximately the same amount as simply accessing the data [45, 40, 60, 84, 95].

A radical alternative for avoiding the costs of concurrency control is to not execute any transactions concurrently. This is the approach taken by Dtxn, where transactions are executed sequentially in each partition, and multiple partitions execute independently to take advantage of parallelism. The proposal that main-memory databases could eliminate concurrency control completely appears to have first been published by Garcia-Molina, Lipton and Valdes [37, 36] in 1984, and some early main-memory databases used this technique [61], or permitted concurrency control to be optionally disabled [53, 10]. Whitney et al. [101] presented a database execution engine that processes transactions in this way. Like Dtxn, they rely on partitioning to take advantage of multiple CPUs, although they also provide a system for users to manually specify which transactions types conflict. Kemme et al. presented a similar system that relied on each stored procedure being assigned to a single "conflict class" that never conflicts with other conflict classes [55]. In both cases, this information is used to allow some transactions to be processed concurrently without needing locks.

Thomson and Abadi presented a system that orders all transactions in advance, and transforms transactions with multiple rounds of communication into simpler operations that always involve a single round of communication [98]. As a result, they can avoid concurrency control entirely, and not even need to speculate transactions. However, their system has different failure handling properties. In particular, if one partition in the system fails, the system either must stop, or transactions that involve that partition are left in an unusual partially applied state. Dtxn chooses to use traditional two-phase commit to handle these types of partial system failures.

As an alternative to eliminating locks, many researchers have instead investigated how to reduce the cost of locking. Lehman and Carey presented a technique for dynamically adjusting the granularity of locks to balance the cost of acquiring and releasing locks against the additional concurrency obtained from finer grained locks [60]. The Starburst database used a similar approach, and additionally reduced the cost of obtaining locks and latches by redesigning the lock table [40]. The Dtxn approach of sequential execution inside a partition can be considered as a static policy with very coarse-grained locks on entire partitions.

DORA is a single machine database that changes the traditional assignment of work to threads [84]. Rather than assigning a transaction to a thread, it assigns partitions of data to specific threads, which is similar to Dtxn. This permits many locks to be kept local to a single thread, which can then be acquired and released much more cheaply than locks that must be maintained in global shared memory. Unlike Dtxn, DORA can migrate transactions to the appropriate thread based on data accesses.

Our speculative concurrency control mechanism is similar to an early proposal for pre-committed transactions [30]. A transaction can be "pre-committed" by releasing its locks after the transaction has committed, but before the log records have actually finished being written to disk. This permits other transactions to read the dirty data that has not yet been durably recorded. This is safe, provided that any dependent transactions that read dirty data cannot commit until the earlier transaction has also committed, which requires some additional bookkeeping. Additionally, clients must not be notified about commits until they are on disk. Speculative concurrency control must do the same bookkeeping, although it is simplified because we track this at the granularity of the entire partition, rather than individual data items.

The SP algorithm (for serial protocol) is inspired by this pre-commit idea, but is designed for main memory databases that execute transactions sequentially like Dtxn [9]. This version assigns each data item a write timestamp, which is used to determine when read-only transactions can commit. This means that read-only transactions that do not read dirty data can commit immediately. Speculative concurrency control avoids this additional overhead, but instead it must always assume that transactions accessed dirty data and so they must be delayed until previous transactions have committed. Because transactions in main-memory are so fast, we believe this is not a significant disadvantage. They also found they could use up to 3 CPUs concurrently by executing multiple read-only transactions at once, which is an optimization that could also be applied to Dtxn since we know if a transaction is read-only in advance. They also argued that partitioning is an effective approach for scaling up their system for their target telecommunications algorithms, which is an argument that applies to Dtxn as well.

The OPT [44] protocol applies pre-committed transactions to distributed databases. They make the same observation that we do: because the delay in finishing two-phase commit is long, it makes sense to speculate future transactions. They also only speculate one transaction, while speculative concurrency control will speculate many. This means that speculative concurrency control can overlap the two-phase commit for many transactions from the same coordinator, at the cost of potentially needing to do a cascading abort.

Reddy and Kitsuregawa proposed an extension of pre-commited transactions called speculative locking, where the database keeps both "before" and "after" versions for modified data items [57]. Transactions always process *both* versions. At commit, the correct execution is selected and applied by tracking data dependencies between transactions. This approach assumes that there are ample computational resources. In our environment, CPU cycles are limited, and thus our speculative concurrency control always acts on the "after" version, and does not track dependencies at a fine-granularity.

6.2 Atomic Commit Protocols

One of the important properties of transactions is that they are atomic: either all the changes are committed, or none are. An atomic commit protocol enforces this property in a distributed system, ensuring that either all participants commit or all participants abort a given transaction. The simplest and most widely known protocol is the basic two-phase commit protocol [41]. The basic version has two forced disk writes: the participants force their commit/abort vote, and the coordinator forces its final commit/abort vote. Extra disk writes record the final decision at each participant and acknowledgments at the coordinator, which are used to terminate the protocol and garbage collect old transaction records. These writes do not need to be forced to disk, but the acknowledgment that the transaction has been completed cannot be sent until the log record has actually been written to disk. This early paper actually gets this wrong. The pseudocode shows the participants sending the acknowledgment without writing the decision to disk. This could result in a transaction getting stuck in the prepared state on the participant. The coordinator can receive the acknowledgement, and so it can forget the outcome of the transaction. However, if the participant crashes after sending the acknowledgement, it can forget the outcome and be stuck.

The basic version of two-phase commit has three forced disk writes: prepare at the participant, commit at the coordinator, and commit at the participant. This last disk write does not need to be forced, but the acknowledgment that the transaction has been completed cannot be sent until the log record has actually been written.

Many tweaks and optimizations of this basic protocol have been presented. The presumed abort protocol [70] is a widely used variant, and is included in many commercial systems via the XA transaction model and interface [97, 96]. The optimization reduces disk writes for aborts, but more importantly it eliminates all disk writes and one network message on read-only participants. This is also sometimes called the two-phase commit read-only optimization. Another common optimization is the "early prepare" or "unsolicited vote" optimizations, first named by Mohan [88], but credited to Stonebraker [94]. While Stonebraker does not use the terms early prepare or unsolicited vote, his description of how INGRES handles distributed updates does describe that slaves prepare to commit transactions before responding to the coordinator, and without requiring an explicit prepare message.

To eliminate as many disk writes as possible, so-called "one-phase commit" protocols are possible, such as Implicit Yes Vote [4] and Coordinator Log [93]. They take the early prepare optimization further by sending log records to the coordinator. This means the coordinator has the only record of the transaction, so it must be contacted during recovery, or else participants cannot process new transactions.

Two-phase commit introduces a potential single point of failure in a distributed system: the coordinator. If the coordinator becomes unavailable after it informs all participants to prepare, then all participants must wait for it to come back to determine if the transaction should commit or abort. This is called the two-phase commit blocking problem. A number of three-phase commit protocols have been created to fix this problem, but Gray and Lamport claim that there is "none that provides a complete algorithm proven to satisfy a clearly stated correctness condition" [42]. Thus, there are no known implementations of three-phase commit in commercial systems.

A solution that does work is to replicate the coordinator decision. The Paxos Commit by Gray and Lamport [42] does exactly this. It ensures that the commit makes progress provided that a majority of the processes (called *acceptors*) are available. They also show that two-phase commit is a degenerate version of Paxos Commit with a single acceptor. The Scalaris key/value store [90] supports transactions using a variant of Paxos Commit [89]. The current implementation of Dtxn does not replicate the coordinator, but our design is to replicate the commit decision using the same primary/backup protocol used to replicate partitions, as described in Section 2.6.

6.3 Distributed Transaction Systems

The utility of distributed transactions has been widely understood, and so they appear in a number of commercial systems. A two-phase commit protocol for distributed transactions across different systems has been standardized as XA transactions [96], which is included in the Java programming language and supported by many applications as the Java Transaction API (JTA) [23]. Transactions were also added as part of the CORBA distributed object system [76]. Transaction monitors are middleware systems used to implement distributed transactions, typically by communicating with multiple XA or JTA participants. Commercial examples include Oracle Tuxedo [81] and JBoss Transactions [46]. Dtxn can be used as a transaction monitor, although that would not take advantage of its unique features like speculative concurrency control or live migration.

Many distributed transaction systems and protocols are described as supporting the "tree of processes" transaction model introduced by R* [62]. In this model, any participant can also be a coordinator by delegating tasks to other processes, forming a tree. Dtxn instead chooses to support a flat structure with dedicated coordinators and participants. While any commit protocol, include Dtxn's, could be used in a tree, by hiding the commit protocol inside the participant, this would require the backend developer to do significant additional work. Others have proposed nested transactions to be a useful abstraction for distributed systems, where subtransactions can be committed and aborted independently of the parent transaction [28]. We have found that Dtxn's flat structure is sufficient for building distributed storage systems, which can usually be easily divided into two parts: processes requesting work and processes storing data.

6.4 Replication Protocols

Replication is the standard technique to provide protection against independent failures. Replication protocols are used to build fault-tolerant systems by ensuring that more than one process receives the same sequence of operations. Consensus protocols such as Viewstamped Replication [78] and Paxos [59] are at the heart of any replication system. While there are many variants, the basic principle of all these protocols is the same: use a majority of independent tasks to decide the state of the system. They permit a system with $2f+1$ nodes to survive f failures.

These protocols need an odd number of replicas, and have proven to be tricky to implement [20]. To address these issues, many systems choose to implement the replication protocol in a special service that is then used for master elections and group membership [16, 50, 69]. The master then uses a primary/backup replication protocol with the group. This reduces the number of replicas, permitting a group of $f+1$ nodes to survive f failures, assuming that the master replication service is available. This is the approach used by Dtxn, as well as many other systems like the Google File System [38], Bigtable [21], and HBase [35]. The Niobe replication protocol provides a formal description of an algorithm to do this [66].

An different approach to reducing the number of replicas is to use *witnesses* that only need to participate when nodes fail [64, 87]. This means that a replicated group only needs $f+1$ nodes with a copy of the data, plus an additional f nodes in case of failures. This approach seems strictly superior than the master election service approach in terms of fault-tolerance. However, it is more difficult to understand and makes the protocol even more complicated, so it does not appear to be used outside of academic systems.

6.5 Database Replication

Replication protocols provide a primitive tool that can be applied in many different ways for building replicated databases [18, 54, 102]. Weismann and Schiper categorize the various approaches into three main groups: active, certification based, and weak voting [102]. Dtxn's primary/backup replication protocol uses active replication, which is sometimes called state machine replication. In this type of replication, the replicated operations represent deterministic operations. Each replica is a state machine that begins in a given state, and executes these operations in identical orders. As a result, they end up in the same state. In the case of Dtxn, the operations are entire stored procedures, or fragments of stored procedures. This is a very high level and compact representation for the operations, but requires that stored procedures be deterministic.

The other approaches, certification-based and weak voting, both rely on only a single replica to execute the operations. However, they then must replicate at least the set of updated database records, so that the updates can be applied on all replicas. Weismann and Schiper's simulation results show that active replication does not scale as well as the other approaches because all replicas execute all operations. Thus, the replicated system cannot possibly be faster than the non-replicated system. This does not apply to Dtxn for two reasons: first, it is designed for lightweight main-memory transactions, so there is little performance difference between executing a transaction and applying a write set, and recording the write set will impose additional overhead. Second, we can permit replicas to execute read-only transactions, provided that we are willing to accept slightly stale results.

6.6 Live Migration

Making changes to a running database is generally referred to as online reorganization, and has been widely studied, particularly in the context of single machine databases [92]. In this section, we review previous work on systems that transfer the execution of queries from one machine to another, as that is the problem we have focused on.

Commercially deployed systems support three techniques: stop and copy, replica failover, and virtual machine migration [65, 24, 75, 12]. Stop and copy means that the source database is paused, the data is copied to the destination, then the transactions are resumed. This implies a long period of unavailability while the data is copied, so this does not really count as "live" migration. Copying a smaller logical partition can reduce the unavailability window [15], but it still can be unacceptably long. Replica failover begins by making a hot backup copy of the database while it is running. Then this backup is brought up to date by replaying operations, just like replication does. Finally, once the backup has caught up, the source database is paused, the final operations are applied to the backup, and transactions then resume. The final approach is to use virtual machine migration, where the database is running in a virtual machine, and the in-memory image of the virtual machine is migrated from one machine to another.

Migration is an important feature for virtual machine monitors. The "pre-copy" technique is the most popular technique, used by VMware [75], Xen [24], and kvm [56]. This works by making an initial copy of memory while the virtual machine is running, then performing subsequent passes that copy the pages that were dirtied. This is repeated until only a few pages remain, or the migration process decides that this will never converge. Finally execution is paused, the remaining state is copied, and execution resumes on the destination. This is similar to replica failover for databases. Fetch on demand strategies have also been investigated [47, 58, 49], and are also referred to as "post-copy" techniques. They allow execution to be migrated very quickly, but incurs the cost of network page faults when data is first accessed. Hirofuchi et al. observe that this can be useful for migration in response to overload, which is similar motivation to our design.

Virtual machine migration usually requires that the virtual machine storage is network accessible by both the source and destination. However, migrating the storage used by a virtual machine has been proposed [48, 13], and is supported by commercial systems [67]. However, for a database that resides primarily in RAM, disk is typically not a bottleneck, so the technique used to migrate the storage does not impact performance.

Process migration is similar to virtual machine migration, but rather than migrating an entire operating system and all its applications, it only migrates a single process. Stop-and-copy, iterative copy, and copy on demand approaches have all been investigated for process migration [68]. While it is a popular research topic, process migration is not widely used. It is also possible to use techniques like process mig

Academic researchers have also investigated similar systems. The Clustra distributed database supported migration in order to scale up and down [14]. They used a replica failover approach, by taking a dirty copy of the data and replaying the redo log to apply any changes. To support partial replication, they use logical logging for the database and perform a logical copy of the data, similar to Wildebeest. They found that during migration, throughput was reduced, due to resource contention, with a direct trade-off between the agressiveness of the copying and the impact on observed throughput. Our experimental results show a similar relationship.

An alternative for scaling a system is to add or remove replicas. This works when the workload is composed mostly of read operations, as reads only need to be performed on one replica while writes need to be performed on all. Most replication systems include a mechanism to add and remove replicas while running. Researchers have used that to automatically decide when to add or remove replicas [22]. Wildebeest could be used with these decision systems, in order to automatically decide how to migrate data. However, Wildebeest is more flexible, as it supports partial migration while adding a replica must make a complete copy.

Albatross is a system for live database migration in a shared storage architecture, where the cache is copied to the destination to start with a hot cache. Our design is for a shared nothing architecture instead, and chooses to "pull" data into the cache on demand. The Zephyr system is a recent academic project that also partially uses a "fetch on demand" policy [31]. However, that system pushes all internal index nodes from the source to the destination before switching over, and then fetches the physical leaf nodes on demand. Thus, it still adds some additional load to the source server. It also cannot directly migrate part of a database due to copying at the physical level.

6.7 Similar Systems

The need to scale a database by distributing it across multiple machines was recognized long ago, and thus there is a large body of work on distributed databases, such as the pioneering work on R* [71], Gamma [29], and Bubba [11]. Most of the initial work focused on disk-based systems, and how to execute complex joins. More recently, to build extremely large systems developers are choosing distributed key/value stores, such as Google Bigtable [21], Yahoo PNUTS [25], HBase [35], or MongoDB [1]. These systems typically only provide operations over a single table key space (i.e. no joins), and only provide consistency on single keys. This makes it easier to distribute operations and data across multiple machines than general purpose relational databases. Two exceptions are Scalaris, which uses the Paxos Commit protocol and provides multi-key transactions [90], and Percolator [85], which extends Bigtable with snapshot isolation.

Dtxn aims to obtain similar scalability as simple key/value stores by requiring users to manually partition their data, but supporting complex operations inside a partition. Google Megastore [6] and Microsoft SQL Azure [8] similarly force users to only access a single partition, although Megastore also supports weakly consistent indices across all partitions. A Siemens project called Netbase [74] also relied on the schema being annotated to describe which attributes can be joined locally.

Other systems have also combined both replication and distributed transactions together. Argus [63] is a programming language and runtime system with support for nested transactions, while Dtxn is just an API. Arjuna [91] provides an API for replicated state machines, similar to Dtxn. Both these systems support nested transactions in a tree of processes, while Dtxn provides a flat namespace with transactions composed of scatter/gather rounds. Argust and Arjuna's approach is more flexible and powerful, but also more complex. The CORBA distributed object specification has an extension for transactions [76], and FT-CORBA [77, 73] is an extension that adds replication, but they are not widely adopted due to their complexity [32]. Dtxn has attempted to avoid this death by complexity by focusing specifically on main-memory data storage system

A number of distributed main-memory databases have also been proposed, each with slightly different design decisions. One of the earliest projects is PRISMA/DB [5], which used traditional concurrency control and two-phase commit with main-memory storage, and thus is more similar to traditional distributed databases than to Dtxn. The ClustRa Telecom Database [51] was replicated and designed for telecommunications applications. Similar to Dtxn, it uses primary/backup replication for each partition of a table and considers a log record to be durable when stored in memory of more than one system. It supported both disk and memory tables, and used traditional locking protocols. Sprint [17] is a middleware layer that partitions statements across commercial main-memory database instances, relying on disk logging but without support for replication. Similar to Dtxn, it optimizes for single partition transactions, but it classifies transactions at run time instead of ahead of time. They begin each transaction assuming it is a single-partition transaction, then upgrade from to a distributed database when needed. dsmDB [33] is a main memory database for a cluster of machines, but it uses traditional database algorithms and relies on distributed shared memory to fetch pages as needed.

A category of commercial products that are sometimes called in-memory data grids have emerged recently. These products are characterized by storing objects in memory across a cluster of machines, with replication and optionally strong consistency. Examples include Oracle Coherence [80], VMware GemFire [99] and Gigaspaces [39]. Typically these are used to cache data outside the database. This is the same application domain where Memcached [52], an in-memory cache server, is used. The difference is that while Memcached does not provide any consistency guarantees, in-memory data grids typically maintain transactional semantics and provide high availability and durability. RAMCloud [82] fits in this category, although they argue that replication is too expensive for main-memory systems, and instead rely on disk logging with very fast recovery [79].

Chapter 7

Conclusion and Future Work

This thesis presented the design of Dtxn, a system for building distributed OLTP databases. Dtxn provides excellent performance by distributing work across multiple machines, using a low-overhead concurrency control mechanism, and providing a distributed transaction protocol optimized for transactions with a single round of communication between the application and the database. It provides high availability through built-in replication. To support dynamic scaling, it can re-distribute data while continuing to process transactions. The result is a system that allows system developers to easily build high performance storage systems.

Speculative concurrency control is a novel concurrency control scheme specifically for partitioned, main memory databases. It is designed for workloads where most transactions execute as a stored procedure at a single partition, but some may need to span multiple partitions. It does not require tracking data accesses at a fine granularity. Instead, it uses speculation to hide stalls related to network access. This provides up to a factor of two better throughput than traditional two-phase locking. The biggest weakness of the current implementation is its reliance on a single coordinator to order distributed transactions. In order to scale the system to extremely large rates of distributed transactions, it must be possible to distribute this work across multiple coordinators. A timestamp-based approach could be feasible, where each coordinator uses a local real time clock to order transactions. This would improve the scalability of the system, as well as making it more robust to failures.

Live migration allows additional machines to be added to a running system. Our cache-based approach allows this migration to quickly offload the original source server. Traditional approaches add load by copying a significant amount of state before switching over. However, this only provides a mechanism for redistributing data. In a real system, there is also a planning problem of determining when and where to move partitions. This is a particularly hard problem when the application has been updated in a way that forces nearly all data to be redistributed. For example, this can occur when it is determined that the original partitioning is not optimal for the workload. In this case, the data must be redistributed through a series of smaller moves. Determining the best way to achieve this while continuing to serve queries is an unsolved problem.

Dtxn has proven to be a useful tool, by supporting multiple applications that have been used for two different research projects. This suggests that its design is re-usable, and can be useful for other data storage systems. Dtxn can be particularly useful for applications that have specialized storage needs that aren't well-met by existing database products. It provides system builders the ability to make their own distributed storage system, without sacrificing features like serializable transactions, fault tolerance, and elastic scalability.

While Dtxn is very useful in its current state, there are two reliability weaknesses that could be easily improved.First, when a transaction coordinator performs two-phase commit, it exposes the system to the traditional two-phase commit "blocking" problem because there is only a single commit log. To solve this, we need to replicate this log, either by adding the replication protocol to the transaction coordinator library, or by adding a replicated commit log service to explicitly record the outcome of two-phase commit. The second weakness is that if the metadata service is unavailable, partitions cannot recover from failures. Augmenting our replication protocol to support witnesses would allow each partition to have any number of replicas, while not relying on a single metadata service. These improvements would make Dtxn even more resilient than it currently is.

Bibliography

[1] 10gen, Inc. MongoDB, 2011. URL `http://www.mongodb.org/`.

[2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *SIGMOD*, pages 23–34, San José, CA, USA, 1995. URL `http://dx.doi.org/10.1145/223784.223787`.

[3] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, October 2007. URL `http://www.sosp2007.org/papers/sosp064-aguilera.pdf`.

[4] Yousef J. Al-houmaily and Panos K. Chrysanthis. Two-phase commit in gigabit-networked distributed databases. In *Parallel and Distributed Computing Systems (PDCS)*, pages 554–560, 1995. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.1725`.

[5] Peter M.G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W.P.J. Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA/DB: a parallel, main memory relational DBMS. *Knowledge and Data Engineering (TKDE)*, 4(6):541–554, December 1992. URL `http://doc.utwente.nl/55724/`.

[6] Jason Baker, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Monterey, CA, USA, January 2011. URL `http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf`.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5. URL `http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx`.

[8] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL server for cloud computing. In *International Conference on Data Engineering (ICDE)*, pages 1255–1263, Hannover, Germany, April 2011. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5767935`.

[9] Stephen Blott and Henry F. Korth. An almost-serial protocol for transaction execution in main-memory database systems. In *VLDB*, pages 706–717, 2002. URL `http://www.vldb.org/conf/2002/S20P02.pdf`.

[10] Philip Bohannon, Daniel Lieuwen, Rajeev Rastogi, Avi Silberschatz, S. Seshadri, and S. Sudarshan. The architecture of the Dalí main-memory storage manager. *Multimedia Tools and Applications*, 4:115–151, 1997. URL `http://dx.doi.org/10.1023/A:1009662214514`.

[11] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *Knowledge and Data Engineering (TKDE)*, 2(1):4–24, 1990. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=50903`.

[12] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Virtual Execution Environments (VEE)*, pages 169–179, San Diego, CA, USA, 2007. URL `http://doi.acm.org/10.1145/1254810.1254834`.

[13] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Virtual Execution Environments (VEE)*, pages 169–179, June 2007. URL `http://doi.acm.org/10.1145/1254810.1254834`.

[14] Svein Erik Bratsberg and Rune Humborstad. Online scaling in a highly available database. In *VLDB*, pages 451–460, Rome, Italy, September 2001. URL `http://www.vldb.org/conf/2001/P451.pdf`.

[15] Anna Brunstrom, Scott T. Leutenegger, and Rahul Simha. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *Conference on Information and Knowledge Management (CKIM)*, pages 395–402, Baltimore, MD, USA, 1995. URL `http://doi.acm.org/10.1145/221270.221652`.

[16] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350, Seattle, WA, USA, November 2006. URL `http://labs.google.com/papers/chubby-osdi06.pdf`.

[17] Lásaro Camargos, Fernando Pedone, and Marcin Wieloch. Sprint: a middleware for high-performance transaction processing. In *EuroSys*, pages 385–398, Lisbon, Portugal, June 2007. URL `http://dx.doi.org/10.1145/1272998.1273036`.

[18] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *SIGMOD*, pages 739–752, Vancouver, Canada, 2008. URL `http://infoscience.epfl.ch/record/118488`.

[19] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *STOC*, pages 77–90, Boulder, CO, USA, May 1977. URL `http://doi.acm.org/10.1145/800105.803397`.

[20] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Principles of Distributed Computing (PODC)*, pages 398–407, Portland, OR, USA, 2007. ACM. URL `http://labs.google.com/papers/paxos_made_live.pdf`.

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, Seattle, WA, USA, November 2006. URL `http://labs.google.com/papers/bigtable-osdi06.pdf`.

[22] Jin Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *International Conference on Autonomic Computing (ICAC)*, pages 231–242, June 2006. URL `http://www.eecg.toronto.edu/~amza/papers/ICAC06.pdf`.

[23] Susan Cheung and Vlada Matena. *Java Transaction API (JTA)*. Sun Microsystems, Inc., Santa Clara, CA, USA, February 2007. URL `http://oracle.com/technetwork/java/javaee/jta/`.

[24] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI*, pages 273–286, Boston, MA, USA, May 2005. URL `http://www.usenix.org/events/nsdi05/tech/full_papers/clark/clark.pdf`.

[25] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *VLDB*, Auckland, New Zealand, 2008. URL `http://www.vldb.org/pvldb/1/1454167.pdf`.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Symposium on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, IN, USA, 2010. URL `http://doi.acm.org/10.1145/1807128.1807152`.

[27] Transaction Processing Performance Council. TPC benchmark C. Technical report, February 2010. URL `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`. Revision 5.11.

[28] Charles T. Davies, Jr. Recovery semantics for a DB/DC system. In *Proceedings of the ACM annual conference*, pages 136–141, Atlanta, GA, United States, 1973. URL `http://doi.acm.org/10.1145/800192.805694`.

[29] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma database machine project. *Knowledge and Data Engineering (TKDE)*, 2(1):44–62, 1990. URL `http://dx.doi.org/10.1109/69.50905`.

[30] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, Boston, MA, USA, 1984. URL `http://doi.acm.org/10.1145/602259.602261`.

[31] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, June 2011. URL `http://www.cs.ucsb.edu/~sudipto/papers/zephyr.pdf`.

[32] P. Felber and P. Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *Transactions on Computers*, 53(5):497–511, May 2004. URL `http://dx.doi.org/10.1109/TC.2004.1275293`.

[33] Nelson D. Filho and Fernando Pedone. dsmDB: a distributed shared memory approach for building replicated database systems. In *Workshop on Dependable Distributed Data Management (WDDDM)*, pages 11–14, Glasgow, Scotland, March 2008. URL `http://dx.doi.org/10.1145/1435523.1435525`.

[34] Nathan Folkman. So, that was a bummer. Foursquare Blog, October 2010. URL `http://blog.foursquare.com/2010/10/05/so-that-was-a-bummer/`.

[35] Apache Software Foundation. Apache HBase, 2011. URL `http://hbase.apache.org/`.

[36] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *Knowledge and Data Engineering (TKDE)*, 4(6):509–516, 1992. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=180602`.

[37] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *Transaction on Computers*, C-33(5):391–399, 1984. URL `http://dx.doi.org/10.1109/TC.1984.1676454`.

[38] Sanjay Ghemawat, Howard Gobioff, and Shun T. Leung. The Google file system. In *SOSP*, pages 29–43, Lake George, NY, USA, October 2003. URL `http://labs.google.com/papers/gfs-sosp2003.pdf`.

[39] Gigaspaces. Gigaspaces XAP, 2011. URL `http://www.gigaspaces.com/`.

[40] Vibby Gottemukkala and Tobin J. Lehman. Locking and latching in a memory-resident database system. In *VLDB*, pages 533–544, Vancouver, Canada, August 1992. URL `http://www.vldb.org/conf/1992/P533.PDF`.

[41] Jim Gray. Notes on data base operating systems. In *Operating Systems*, Lecture Notes in Computer Science, pages 393–481. Springer Berlin / Heidelberg, 1978. URL `http://dx.doi.org/10.1007/3-540-08755-9_9`.

[42] Jim Gray and Leslie Lamport. Consensus on transaction commit. *Transactions on Database Systems (TODS)*, 31(1):133–160, March 2006. URL `http://research.microsoft.com/pubs/64636/tr-2003-96.pdf`.

[43] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992. ISBN 978-1558601901. URL `http://www.elsevierdirect.com/ISBN/9781558601901/Transaction-Processing`.

[44] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. Revisiting commit processing in distributed database systems. In *SIGMOD*, pages 486–497, 1997. URL `http://dx.doi.org/10.1145/253260.253366`.

[45] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, Vancouver, Canada, 2008. URL `http://dx.doi.org/10.1145/1376616.1376713`.

[46] Red Hat. JBoss transactions, 2011. URL `http://www.jboss.org/jbosstm`.

[47] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Virtual Execution Environments (VEE)*, pages 51–60, Washington, DC, USA, March 2009. URL `http://doi.acm.org/10.1145/1508293.1508301`.

[48] T. Hirofuchi, H. Ogawa, H. Nakada, S. Itoh, and S. Sekiguchi. A live storage migration mechanism over WAN for relocatable virtual machine services on clouds. In *Cluster Computing and the Grid (CCGrid)*, pages 460–465, Shanghai, China, May 2009. URL `http://dx.doi.org/10.1109/CCGRID.2009.44`.

[49] Takahiro Hirofuchi, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. Enabling instantaneous relocation of virtual machines with a lightweight vmm extension. In *Cluster, Cloud and Grid Computing (CCGrid)*, pages 73 –83, Melbourne, Australia, May 2010. URL `http://grivon.googlecode.com/svn/pub/docs/ccgrid2010-hirofuchi-paper.pdf`.

[50] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX*, pages 1–11, Boston, MA, USA, 2010. URL `http://www.usenix.org/event/atc10/tech/full_papers/Hunt.pdf`.

[51] Svein O. Hvasshovd, Oystein Torbjornsen, Svein E. Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *VLDB*, pages 469–477, San Francisco, CA, USA, 1995. URL `http://www.vldb.org/conf/1995/P469.PDF`.

[52] Danga Interactive. Memcached, 2011. URL http://memcached.org/.

[53] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *VLDB*, pages 48–59, Santiago de Chile, Chile, September 1994. URL http://www.vldb.org/conf/1994/P048.PDF.

[54] Alfrânio C. Júnior, António Sousa, Emmanuel Cecchet, Fernando Pedone, José Pereira, Luís Rodrigues, Nuno M. Carvalho, Ricardo Vilaça, Rui Oliveira, Sara Bouchenak, and Vaide Zuikeviciute. State of the art in database replication. Technical report, Universidade do Minho, September 2005. URL http://gorda.di.uminho.pt/deliverables/D1.1.

[55] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Distributed Computing Systems*, pages 424 –431, Austin, TX, USA, May 1999.

[56] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: The Linux virtual machine monitor. In *Ottawa Linux Symposium*, Ottawa, Canada, June 2007. URL http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf.

[57] P. Krishna Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *Knowledge and Data Engineering (TKDE)*, 16(2):154–169, March 2004. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1269595.

[58] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *EuroSys*, pages 1–12, Nuremberg, Germany, March 2009. URL http://doi.acm.org/10.1145/1519065.1519067.

[59] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, December 2001. URL http://dx.doi.org/10.1145/568425.568433.

[60] Tobin Lehman and Michael Carey. A concurrency control algorithm for memory-resident database systems. In *Foundations of Data Organization and Algorithms*, volume 367 of *Lecture Notes in Computer Science*, pages 489–504. Springer Berlin / Heidelberg, 1989. URL http://dx.doi.org/10.1007/3-540-51295-0_150.

[61] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *Databases in Parallel and Distributed Systems (DPDS)*, pages 177–187, 1988. URL http://portal.acm.org/citation.cfm?id=62617.

[62] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: a distributed database manager. *Transactions on Computer Systems (TOCS)*, 2:24–38, February 1984. URL http://doi.acm.org/10.1145/2080.357390.

[63] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM (CACM)*, 31(3):300–312, March 1988. URL `http://dx.doi.org/10.1145/42392.42399`.

[64] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the harp file system. In *SOSP*, pages 226–238, Pacific Grove, CA, USA, October 1991. URL `http://pmg.csail.mit.edu/papers/harp.pdf`.

[65] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *High Performance Distributed Computing (HPDC)*, pages 101–110, Garching, Germany, 2009. URL `http://doi.acm.org/10.1145/1551609.1551630`.

[66] John Maccormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. *Transactions on Storage*, 3:1:1–1:43, February 2008. URL `http://research.microsoft.com/pubs/64661/tr-2007-112.pdf`.

[67] Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. The design and evolution of live storage migration in VMware ESX. In *USENIX*, June 2011. URL `http://www.usenix.org/events/atc11/tech/final_files/Mashtizadeh.pdf`.

[68] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *Computing Surveys*, 32:241–299, September 2000. URL `http://doi.acm.org/10.1145/367701.367728`.

[69] Blake Mizerany and Keith Rarick. Doozer, 2011. URL `https://github.com/ha/doozerd`.

[70] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Principles of Distributed Computing (PODC)*, pages 76–88, Montreal, Canada, 1983. URL `http://dx.doi.org/10.1145/800221.806711`.

[71] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *Transactions on Database Systems*, 11(4): 378–396, 1986. URL `http://doi.acm.org/10.1145/7239.7266`.

[72] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *Transactions on Database Systems*, 17(1): 94–162, March 1992. URL `http://doi.acm.org/10.1145/128765.128770`.

[73] B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt. DOORS: towards high-performance fault tolerant CORBA. In *Distributed Objects and Applications (DOA)*, pages 39–48, 2000. URL `http://www.cs.wustl.edu/~schmidt/PDF/DOA-2000.pdf`.

[74] Julio C. Navas and Michael Wynblatt. The network is the database: data management for highly distributed systems. In *SIGMOD*, pages 544–551, Santa Barbara, CA, USA, 2001. URL `http://doi.acm.org/10.1145/375663.375737`.

[75] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *USENIX*, pages 391–394, Anaheim, CA, USA, April 2005. URL `http://www.usenix.org/events/usenix05/tech/general/full_papers/short_papers/nelson/nelson.pdf`.

[76] Object Management Group. *Transaction Service Specification*. Object Management Group, Needham, MA, September 2003. URL `http://www.omg.org/spec/TRANS/`. Version formal/2010-05-07.

[77] Object Management Group, Inc. *Fault Tolerant CORBA*. Object Management Group, Inc., Needham, MA, USA, 2010. URL `http://www.omg.org/spec/FT/`. Version formal/2010-05-07.

[78] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Principles of Distributed Computing (PODC)*, pages 8–17, 1988. URL `http://dx.doi.org/10.1145/62546.62549`.

[79] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, Cascais, Portugal, October 2011. URL `http://www.stanford.edu/~ouster/cgi-bin/papers/ramcloud-recovery.pdf`.

[80] Oracle. Coherence, 2011. URL `http://www.oracle.com/technetwork/middleware/coherence/`.

[81] Oracle. Tuxedo, 2011. URL `http://oracle.com/products/middleware/tuxedo/`.

[82] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *Operating Systems Review*, 43:92–105, January 2010. URL `http://www.stanford.edu/~ouster/cgi-bin/papers/ramcloud.pdf`.

[83] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996. URL `http://dx.doi.org/10.1007/s002360050048`.

[84] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. In *VLDB*, September 2010. URL `http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R83.pdf`.

[85] Daniel Peng and Frank Dabek. Large-scale incremental processing with distributed transactions and notifications. In *OSDI*, Vancouver, Canada, October 2010. URL `http://research.google.com/pubs/archive/36726.pdf`.

[86] Percona Inc. Percona XtraBackup, 2011. URL `http://www.percona.com/software/percona-xtrabackup/`.

[87] Jehan-François Pâris. Voting with witnesses: A consistency scheme for replicated files. In *IEEE International Conference on Distributed Computing Systems*, pages 606–612, Cambridge, MA, USA, 1986. URL `http://www.cs.uh.edu/~paris/MYPAPERS/Icdcs86.pdf`.

[88] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *International Conference on Data Engineering (ICDE)*, pages 520 –529, April 1993.

[89] Florian Schintke, Alexander Reinefeld, Seif Haridi, and Thorsten Schütt. Enhanced paxos commit for transactions on dhts. In *Cluster, Cloud and Grid Computing (CCGrid)*, pages 448–454, May 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5493454`.

[90] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: reliable transactional p2p key/value store. In *Workshop on Erlang*, pages 41–48, Victoria, BC, Canada, September 2008. URL `http://dx.doi.org/10.1145/1411273.1411280`.

[91] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, 8(1):66–73, January 1991. URL `http://dx.doi.org/10.1109/52.62934`.

[92] Gary H. Sockut and Balakrishna R. Iyer. Online reorganization of databases. *Computing Surveys*, 41:14:1–14:136, July 2009. URL `http://doi.acm.org/10.1145/1541880.1541881`.

[93] J.W. Stamos and F. Cristian. A low-cost atomic commit protocol. In *Reliable Distributed Systems*, pages 66–75, Huntsvilla, AL, USA, October 1990. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=93952`.

[94] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *Software Engineering*, SE-5(3):188–194, May 1979. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1702618`.

[95] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, Vienna, Austria, 2007. URL `http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf`.

[96] The Open Group. *Distributed Transaction Processing: The XA Specification*. X/Open Company Ltd., Reading, UK, July 1994. ISBN 1-85912-046-6. URL `http://www.opengroup.org/pubs/catalog/s423.htm`. Version 2.

[97] The Open Group. *Distributed Transaction Processing: Reference Model*. X/Open Company Ltd., Reading, UK, February 1996. ISBN 1-85912-170-5. URL `http://www.opengroup.org/pubs/catalog/g504.htm`. Version 3.

[98] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. In *VLDB*, Singapore, September 2010. URL `http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R06.pdf`.

[99] VMware. vFabric GemFire, 2011. URL `http://www.vmware.com/products/vfabric-gemfire/`.

[100] VoltDB Inc. VoltDB, 2011. URL `http://voltdb.com/`.

[101] Arthur Whitney, Dennis Shasha, and Stevan Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *High Performance Transaction Systems (HPTS)*, 1997. URL `http://www.cs.nyu.edu/shasha/papers/hpts.ps`.

[102] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *Knowledge and Data Engineering (TKDE)*, 17(4):551–566, 2005. URL `http://dx.doi.org/10.1109/TKDE.2005.54`.