UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Scalable Transactions for Scalable Distributed Database Systems

Permalink
https://escholarship.org/uc/item/5zf8p47g

Author
Pang, Gene

Publication Date
2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Scalable Transactions for Scalable Distributed Database Systems

by

Gene Pang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Michael J. Franklin, Chair
Professor Ion Stoica
Professor John Chuang

Summer 2015

Scalable Transactions for Scalable Distributed Database Systems

Copyright 2015
by
Gene Pang

Abstract

Scalable Transactions for Scalable Distributed Database Systems

by

Gene Pang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael J. Franklin, Chair

With the advent of the Internet and Internet-connected devices, modern applications can experience very rapid growth of users from all parts of the world. A growing user base leads to greater usage and large data sizes, so scalable database systems capable of handling the great demands are critical for applications. With the emergence of cloud computing, a major movement in the industry, modern applications depend on distributed data stores for their scalable data management solutions. Many large-scale applications utilize NoSQL systems, such as distributed key-value stores, for their scalability and availability properties over traditional relational database systems. By simplifying the design and interface, NoSQL systems can provide high scalability and performance for large data sets and high volume workloads. However, to provide such benefits, NoSQL systems sacrifice traditional consistency models and support for transactions typically available in database systems. Without transaction semantics, it is harder for developers to reason about the correctness of the interactions with the data. Therefore, it is important to support transactions for distributed database systems without sacrificing scalability.

In this thesis, I present new techniques for scalable transactions for scalable database systems. Distributed data stores need scalable transactions to take advantage of cloud computing, and to meet the demands of modern applications. Traditional techniques for transactions may not be appropriate in a large, distributed environment, so in this thesis, I describe new techniques for distributed transactions, without having to sacrifice traditional semantics or scalability.

I discuss three facets to improving transaction scalability and support in distributed database systems. First, I describe a new transaction commit protocol that reduces the response times for distributed transactions. Second, I propose a new transaction programming model that allows developers to better deal with the unexpected behavior of distributed transactions. Lastly, I present a new scalable view maintenance algorithm for convergent join views. Together, the new techniques in this thesis contribute to providing scalable transactions for modern, distributed database systems.

To my wife and children

Contents

Contents **ii**

List of Figures **iv**

List of Tables **vi**

1 Introduction **1**
1.1 Trends of Large-Scale Applications 1
1.2 Emergence of Cloud Computing 1
1.3 Scaling with NoSQL 2
1.4 Bringing 'SQL' Back to NoSQL 3
1.5 Scalable Transactions for Scalable Distributed Database Systems 4
1.6 Summary and Contributions 5
1.7 Dissertation Overview 6

2 Background **7**
2.1 Introduction 7
2.2 Traditional Database Systems 7
2.3 Database Transactions 9
2.4 Scaling Out Database Management Systems 14
2.5 Transactions in a Distributed Setting 19
2.6 Materialized Views 22
2.7 Summary 24

3 A New Transaction Commit Protocol **26**
3.1 Introduction 26
3.2 Architecture Overview 28
3.3 The MDCC Protocol 30
3.4 Consistency Guarantees 41
3.5 Evaluation 45
3.6 Related Work 54
3.7 Conclusion 56

4 A New Transaction Programming Model **57**
4.1 Introduction 57
4.2 The Past, The Dream, The Future 58
4.3 PLANET Simplified Transaction Programming Model 62
4.4 Advanced PLANET Features 68
4.5 Geo-Replication 71
4.6 Evaluation 78
4.7 Related Work 87
4.8 Conclusion 88

5 A New Scalable View Maintenance Algorithm **89**
5.1 Introduction 89
5.2 Motivation 90
5.3 Goals for Scalable View Maintenance 91
5.4 Existing Maintenance Methods 94
5.5 Possible Anomalies 95
5.6 Scalable View Maintenance 101
5.7 SCALAVIEW Algorithm 105
5.8 Proofs 106
5.9 Evaluation 111
5.10 Related Work 119
5.11 Conclusion 121

6 Conclusion **122**
6.1 Contributions 122
6.2 Future Work 123
6.3 Conclusion 124

Bibliography **126**

List of Figures

2.1 Typical model for a single-server database system . . . 9
2.2 Typical scalable architecture for distributed database systems . . . 19

3.1 MDCC architecture . . . 29
3.2 Possible message order in MDCC . . . 38
3.3 TPC-W write transaction response times CDF . . . 48
3.4 TPC-W throughput scalability . . . 49
3.5 Micro-benchmark response times CDF . . . 50
3.6 Commits/aborts for varying conflict rates . . . 52
3.7 Response times for varying master locality . . . 53
3.8 Time-series of response times during failure (failure simulated at 125 seconds) . 54

4.1 Round trip response times between various regions on Amazon's EC2 cluster. . . 58
4.2 Client view of PLANET transactions . . . 63
4.3 PLANET transaction state diagram . . . 66
4.4 Sequence diagram for the MDCC classic protocol . . . 72
4.5 Transaction outcomes, varying the timeout (20,000 items, 200 TPS) . . . 79
4.6 Commit & abort throughput, with variable hotspot (200,000 items, 200 TPS) . 81
4.7 Average response time, with variable hotspot (200,000 items, 200 TPS) . . . 81
4.8 Commit throughput, with variable client rate (50,000 items, 100 hotspot) . . . 82
4.9 Commit response time CDF (50,000 items, 100 hotspot) . . . 83
4.10 Transaction types, with variable data size (200 TPS, uniform access) . . . 84
4.11 Average commit latency, with variable data size (200 TPS, uniform access) . . . 84
4.12 Admission control, varying policies (100 TPS, 25,000 items, 50 hotspot) . . . 85
4.13 Admission control, varying policies (400 TPS, 25,000 items, 50 hotspot) . . . 86

5.1 History and corresponding conflict dependency graph for example anomaly . . . 98
5.2 Simple three table join for the micro-benchmark . . . 112
5.3 View staleness CDF . . . 113
5.4 View staleness CDF (across three availability zones) . . . 114
5.5 Data size amplification as a percentage of the base table data . . . 115
5.6 Band join query for micro-benchmark . . . 115

5.7 View staleness CDF for band joins 116
5.8 View staleness CDF scalability 117
5.9 Throughput scalability 117
5.10 Simple two table linear join query (Linear-2) 118
5.11 Simple three table linear join query (Linear-3) 119
5.12 Simple four table star join query (Star-4) 119
5.13 View staleness CDF for different join types 120

List of Tables

3.1 Definitions of symbols in MDCC pseudocode 40

Acknowledgments

I would like to express my sincere gratitude to my advisor, Michael Franklin, for his guidance throughout my research. He always provided me with great insight into my work, and really taught me to think critically about the core research problems. Also, as someone who had no prior research experience, I am extremely grateful for the opportunity he had given me to work with him in the AMPLab. Through this opportunity, I was able to collaborate with extremely bright professors and students, and this really shaped my graduate research career.

I am also grateful for the joint work and insight from my research collaborators. I worked very closely with Tim Kraska when he was a postdoc in the AMPLab, and he was instrumental in helping me cultivate the ideas in this thesis. I would like to thank Alan Fekete, for providing me with invaluable feedback on my research. I feel very fortunate to have the opportunity to work with Alan and to experience his expertise in database transactions. I am also thankful for Sam Madden, who helped me formulate the initial ideas for my work, when he was visiting the AMPLab. It is so incredible that I had been given the opportunity to interact with such great experts in the database community.

I would also like to express my appreciation for the rest of my dissertation committee, Ion Stoica and John Chuang. They contributed thoughtful comments to my research, and really helped me to improve this thesis. It is easy to be too close to my own work, so their perspectives as outside experts helped me to see the bigger picture.

Throughout my research, there have been many researchers who contributed to shaping my research, papers, and presentations. I would like to thank: Peter Alvaro, Michael Armbrust, Peter Bailis, Neil Conway, Dan Crankshaw, Ali Ghodsi, Daniel Haas, Sanjay Krishnan, Nick Lanham, Haoyuan Li, Evan Sparks, Liwen Sun, Beth Trushkowsky, Shivaram Venkataraman, Jiannan Wang, Reynold Xin.

My family was an enormous part of my journey through graduate studies, and I am deeply grateful for all of them. My parents were very supportive of my decision to leave a great job in industry, and to enter the PhD program at UC Berkeley. Their encouragement and also financial support were integral to my graduate studies. I also want to thank my in-laws, who were equally supportive of my career change, and also provided financial support. My graduate studies would have been impossible without the support and encouragement from my parents and in-laws.

My wife, Herie, and my children, Nathan and Luke, were so important to my graduate studies. Throughout the five years of my PhD, Herie's love, support, encouragement, and prayers were so consistent and were so critical to my daily work. She also had incredible patience with all the aspects of being married to a graduate student. She was my loving partner throughout all my studies, and I thank her so much for always being there for me. Nathan and Luke were both born during my PhD program, and I am very thankful for them. They bring me so much joy, and their excitement and energy are rejuvenating. They are a major source of motivation in my life.

I also want to thank God, and His overflowing love and mercy in my life. He has blessed me and answered my prayers in so many ways during my PhD studies. I am eternally thankful for His unending love, the redemptive power of the death and resurrection of His son, Jesus Christ, and the fellowship of His Holy Spirit. And last but not least, I would like to thank my family at Radiance Christian Church for their prayers and support during my graduate studies.

Chapter 1

Introduction

1.1 Trends of Large-Scale Applications

Modern applications are growing along various dimensions such as the number of users, complexity, and data size, as more and more people are connecting to the Internet through various devices. As Internet access becomes more pervasive, applications can experience exponential growth of users from all around the world. Because the user base can grow very large and very quickly, modern applications face new scalability challenges. Applications need database systems capable of handling the large size and growth of the demand.

To meet the demands of their users, companies such as Amazon, Facebook and Twitter need to scale their database systems. For example, the micro-blogging service Twitter saw its usage grow from 400 tweets per second in 2010, to 5,700 tweets per second in 2013, with a peak of 143,199 tweets per second [60]. Also, the social networking site Facebook ingests over 500 TB of data each day, which includes 2.7 billion likes [45]. Handling the rapid growth in usage, and managing the sheer size of the data are some of the challenges modern applications face. Therefore, many companies develop custom database systems in order to scale with the growth and large size of the data.

1.2 Emergence of Cloud Computing

Along with the growing demands of modern applications and workloads, cloud computing has also gained prominence throughout the industry [9]. Cloud computing refers to the services and applications running in a large cluster or data center of servers. A private cloud is cluster of servers operated for a single organization, and several large Internet companies such as Google and Facebook manage their own private cloud. A public cloud is cluster and related services available for public use, and some examples include Amazon Elastic Compute Cloud (EC2) and Rackspace Cloud Servers. Whether it is public or private, cloud computing has gained popularity because of the numerous benefits it can provide. A major benefit of cloud computing is the cost advantage from economies of scale. A very large cluster

of servers can enable a factor of 5 decrease in cost of electricity, networking bandwidth, and maintenance compared to a smaller data center [9].

Cloud computing can also benefit by utilizing heterogeneous, cheaper, commodity hardware for resources. Instead of running applications on only a single powerful machine, a cloud, or cluster, can distribute applications over many inexpensive commodity servers. This enables being able to incrementally increase the resources with incremental costs. Elasticity of resources and horizontal scalability are other significant benefits of cloud computing. Elasticity is the ability to dynamically adjust the allocated resources to adapt to the current demand. Horizontal scalability (or scaling out) is the property that when additional servers, or nodes, are added to a system, the performance increases proportionally to the added resources. Because of all of these benefits, many large companies like Google, Facebook and Microsoft deploy large clusters for internal computing use. There are also many clouds like Google Compute Engine, Amazon EC2, and Microsoft Azure that provides cloud infrastructure for public use.

1.3 Scaling with NoSQL

Because of the need for scalable data management solutions the emergence of and cloud computing, modern applications have turned to scalable distributed data stores on commodity hardware as their solution. Although the term *data store* can be general, in this thesis I use it interchangeably with *database system*, defined as a software system designed for maintaining and managing data. While there are various techniques for scaling data stores, a class of systems called NoSQL data stores have gained popularity for their scalability and availability properties. NoSQL systems mainly simplify the design and limit the interface in order to provide high scalability and performance for massive data sets. One such example is the distributed key-value store. A distributed key-value store is a system that manages a collection of key-value pairs, where the *value* is the data to be stored, and the *key* is an identifier for the data. Distributed key-value stores forgo complex queries and transactions and only provide limited access to a single key or a contiguous range of keys for scalability and performance benefits. Key-value stores behave more like a hash table than a fully-featured database system.

In order to provide easy horizontal scalability and fault tolerance, these NoSQL systems sacrifice traditional models of consistency and drop support for general transactions typically found in databases. For example, NoSQL systems tend to provide eventual consistency, which guarantees that if there are no new updates for a data item, the value of the data item will eventually converge to the latest update [81]. Supporting consistent transactions in a distributed system requires network communication (more details can be found in Section 2.5), and can hinder the system's ability to scale out to additional resources. Therefore, NoSQL systems abandon transactions in favor of scalability. For example, Amazon's Dynamo [33] is a massively scalable key-value data store that sacrifices transactions and strong consistency for high availability, and Google's Bigtable [24] is a scalable distributed data-

base system that limits transactions to contain access to a single key. These new examples of distributed data stores achieve the scalability required for massive data sets and large workloads by eliminating communication and coordination, by weakening consistency models or by not supporting general transactions. Horizontal scalability is very simple for these NoSQL systems, because coordination is not necessary between distributed servers, and each server operates independently from others. Therefore, NoSQL systems can increase performance and capacity by adding additional servers, or nodes.

While NoSQL storage systems have become very popular for their scalability, some applications need transactions and consistency. For example, Google designed Megastore [13] to support transactions, because the correctness of transactions are easier for developers to reason about, and many applications cannot tolerate the anomalies possible with eventual consistency. Therefore, even though the scalability of NoSQL systems is very desirable in the cloud computing age, giving up all the semantics that traditional databases have provided is not always worth it.

1.4 Bringing 'SQL' Back to NoSQL

NoSQL data stores have been popular for several years because of their high scalability, availability and fault tolerance, but there has been a recent trend towards increasing the support for transactions for these scalable systems. One common technique for supporting transactions in distributed database systems is to horizontally partition the data across the servers, and to execute local transactions within each partition. Horizontal partitioning involves splitting up a single collection (or a database table) and distributing the data items, or rows, into several collections, or tables. Restricting transactions to single partitions eliminates the need to coordinate with other partitions. However, to provide more general transactions across data in multiple partitions, additional communication and coordination are required between the distributed servers and this can affect the scalability of the system. Coordination can restrict scalability by decreased total throughput, increased latency, and possible unavailability due to failures. Common coordination techniques include the Paxos [54] consensus protocol, and the two-phase commit protocol (more details can be found in Section 2.5). For example, Google's Megastore [13] provides transactions by using the Paxos protocol within each partition to reliably store transaction operations for the data, and by using two-phase commit for transactions spanning multiple partitions. For distributed transactions across several partitions, using two-phase commit will incur additional latency and limit scalability, especially in a wide-area network environment. Two-phase commit uses two rounds of communication in order to fully commit a transaction, and during that time, locks are held, reducing the amount of concurrency possible.

This trend of introducing more traditional database concepts such as transactions and strong consistency is promising, but the cost is sometimes too great. While traditional database techniques can be used to provide transactions in distributed systems, they were designed when cloud computing and large distributed systems were not common. In the

cloud computing era, distributed systems are the norm, so algorithms need to consider the costs of network communication required for coordination. Novel techniques should be considered for the new types of distributed system deployments, and new application workloads. For example, centralized algorithms are not appropriate for large, distributed environments, because with heavy workloads, they can easily become the bottleneck and cripple the performance and scalability of the system.

1.5 Scalable Transactions for Scalable Distributed Database Systems

In this thesis, I propose new techniques to implement scalable transactions for scalable database systems. Since applications need data stores that can scale out with the high usage workloads and large data sizes, data stores need to be able to scale transaction processing and management as well. Since traditional transaction techniques and algorithms were developed before the emergence of cloud computing, they may not be appropriate for modern deployments where distributing data across large clusters is prevalent and not the exception. Before cloud computing, horizontal scalability was not a major concern, but now it is critical for meeting the demands of modern applications. I present new ways to improve the transaction performance and experience for scalable distributed databases, without sacrificing general transaction support or scalability. This thesis addresses three facets to improving transaction scalability and support in distributed databases.

1.5.1 New Transaction Commit Protocol

This thesis describes a new transaction commit protocol to reduce the response times of distributed transactions. Since distributed transactions have to communicate and coordinate with other servers in the system, reducing the number of round-trip messages is critical to reducing the response times of transactions. Since two-phase commit, the commonly used protocol, requires two round-trips and also has certain limitations in fault tolerance, it can negatively impact the response times and scalability of distributed transactions, especially in the wide-area network. I propose a new transaction commit protocol based on Generalized Paxos that executes with faster response times, provides durability for distributed transactions and enforces domain integrity constraints (developer-defined rules specifying the domain for data values).

1.5.2 New Transaction Programming Model

This thesis also proposes a new transaction programming model that exposes more details to application developers, providing them with better control over their distributed transactions, since failures and delays are possible with communication and coordination.

Traditional transaction programming models are very simple, but can be inflexible especially when unexpected events occur during transaction processing. In distributed database systems, there can be many sources of unpredictability, such as server failures or network delays, so developers should be able to adapt to those situations. I present a new transaction programming model that is more appropriate in distributed settings, and also supports optimizations to improve the performance of distributed transactions.

1.5.3 New Scalable, Convergent View Maintenance Algorithm

In this thesis, I also describe a new scalable view maintenance algorithm for convergent views. Materialized views and derived data allow faster and more complex queries for simple distributed data stores like key-value stores. There are already existing techniques for maintaining materialized views in databases, but they depend on database transactions and centralized algorithms. A centralized algorithm can impose an undesired bottleneck on a system and is not scalable, because all processing is localized to one server. I propose new distributed techniques for maintaining views in a scalable way.

1.6 Summary and Contributions

In this thesis, I present new techniques for providing and using scalable transactions for modern, distributed database systems. There are three main components of scalable distributed transactions addressed in this thesis: a new transaction commit protocol for executing transactions, a new transaction programming model for interacting with distributed transactions, and a new view maintenance algorithm for scalably maintaining derived data. The key contributions of this thesis are:

- I propose MDCC, Multi-Data Center Consistency, a new optimistic commit protocol, which achieves wide-area transactional consistency while requiring only one message round-trip in the common case.
- I describe a new approach using quorum protocols to ensure that domain constraints are not violated.
- I present experimental results using the TPC-W benchmark showing that MDCC provides strong consistency with costs similar to eventually consistent protocols.
- I introduce PLANET, Predictive Latency-Aware NEtworked Transactions, a new transaction programming model that exposes details of the transaction state and allows callbacks, thereby providing developers with the ability to create applications that can dynamically adapt to unexpected events in the environment.
- I demonstrate how PLANET can predict transaction progress using a novel commit likelihood model for a Paxos-based geo-replicated commit protocol.

- I present optimizations for a strongly consistent database and an empirical evaluation of PLANET with a distributed database across five geographically diverse data centers.
- I describe an investigation of the anomalies that can arise when applications attempt to maintain views in a scalable way in partitioned stores.
- I propose SCALAVIEW, a new scalable algorithm for maintaining join views while preventing or correcting the possible anomalies.
- I present an evaluation of view staleness, overheads, and scalability of SCALAVIEW compared with other existing techniques.

1.7 Dissertation Overview

The rest of this dissertation is organized as follows. Chapter 2 presents background on database systems and distributed transactions. Chapter 3 introduces MDCC, a new transaction commit protocol for faster distributed transactions while maintaining consistency. Chapter 4 describes PLANET, a new transaction programming model for helping developers better handle distributed transactions. Chapter 5 discusses SCALAVIEW, a scalable algorithm for maintaining convergent join views for distributed databases. Chapter 6 concludes this thesis with a discussion of the results and outlines areas for future work.

Chapter 2

Background

2.1 Introduction

Before discussing how transactions can be made more scalable for large-scale distributed database systems, it is important to understand what transactions are and how current systems execute them. This chapter provides background information on transactions in single-server database systems and in distributed database systems. Section 2.2 presents background on traditional database systems. Section 2.3 discusses the semantics of database transactions. Section 2.4 describes how modern, distributed storage systems typically scale out to handle greater data size and transaction workload. Section 2.5 describes various techniques commonly used for distributed transactions, and Section 2.6 gives an overview on different algorithms for materialized views and how transactions are utilized by algorithms to maintain views. Examining the restrictions and scalability limitations of current distributed transactions and algorithms provides a better foundation for understanding how the rest of this thesis provides scalable solutions for those issues.

2.2 Traditional Database Systems

In this thesis, traditional database systems refer to single-server, relational [27] database systems. The term "traditional" does not imply antiquated or obsolete, since there are many traditional database systems being used today, like MySQL or PostgreSQL. These single-server systems support transactions using well-known techniques, many of which are discussed in this chapter.

2.2.1 The Relational Model

In this section, I briefly describe the relational model that is common to many of the single-server database systems today. In the relational model, a database is a collection of *relations* (also known as *tables*). Each relation consists of a set of *tuples*, also known as *rows* or *records*.

A tuple is an ordered collection of *attribute* values, or *column* values. All tuples for a given relation have the same set of attributes, defined by the *schema* of the relation. The schema of the relation is a collection of attribute names and their data types (Integer, String, etc.).

While the relational model defines how the data is logically organized and represented, *relational algebra* provides the foundation for querying the database. Relational algebra consists of operators that transform relations to produce new relations. By composing these relational operators, a wide variety of queries an be expressed. In what follows, I describe a subset of the relational operators that are relevant for the queries in the rest of this thesis.

Selection The selection operator is written as $\sigma_{\gamma}(R)$, where R is a relation, and γ is a predicate expression that evaluates to true or false. The operator will produce result relation that is a subset of relation R, where a tuple from R is in the result if and only if the predicate γ evaluates to true. For example, the expression $\sigma_{age<20}(Students)$ will return every *Student* who is younger than 20.

Projection The projection operator is written as $\pi_{a_1,...,a_n}(R)$, where R is a relation, and $a_1, ..., a_n$ is a set of attribute names from the schema of R. The operator will produce a result relation with all the tuples of R, but with a subset of the attributes, $a_1, ..., a_n$. Since the result must also be a relation, any duplicate tuples in the result are removed. For example, the expression $\pi_{name,age}(Students)$ will return only the *name* and *age* of every *Student*, even though there are other attributes of *Student*.

Cartesian Product The Cartesian product (or cross product) operator is written as $R \times S$, were R and S are relations. The operator will produce a relation where every tuple of R is concatenated together with every tuple of S, to construct new resulting tuples. The resulting schema is the union of attributes of the schemas from R and S. If the schemas of R and S have attributes in common, then those attribute names must be renamed to eliminate the naming conflict.

Join There are several different types of joins, but a common type of join is called the *natural join*. The natural join operator is written as $R \bowtie S$, where R and S are relations. The natural join is similar to the Cartesian product, because it produces tuples which are combined from the tuples from R and the tuples from S. However, a tuple is in the result of the natural join if and only if the R tuple and S tuple are equal for the attribute names common to both R and S.

A more general type of join is the θ-join, and it is written as $R \bowtie_{\theta} S$. θ is a join predicate which evaluates to true or false. A tuple is in the result set of the θ-join if and only if the predicate θ evaluates to true. The θ-join is equivalent to $\sigma_{\theta}(R \times S)$.

2.2.2 Single-Server Model

At a high level, the single-server paradigm is quite simple. The database system resides on a single server, and the applications and users access the server in order to interact with

the data. Figure 2.1 shows the high level overview of how applications access a single-server database system. To query or modify the data, all applications and users must communicate with the single instance of the database management system, which is usually on a separate server. Many elements make up a database system, but the details of each individual component are not critical to understanding how applications interact with a database in the single-server model. For comparison, Section 2.4.4 discusses the distributed database system model.

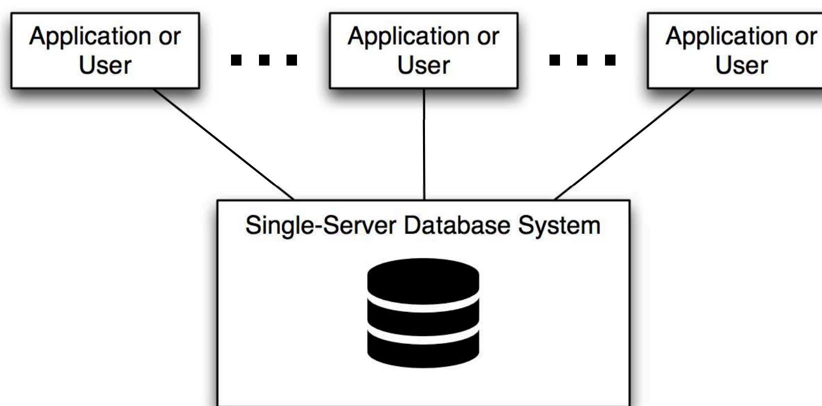


Figure 2.1: Typical model for a single-server database system

2.3 Database Transactions

A transaction is the basic unit of work for interacting with a database system. For the purposes of this thesis, a transaction is defined as a sequence of read and write operations of data items in the database, and can have one of two outcomes: commit or abort. When a transaction completes all of its operations successfully, the transaction *commits* and its effects are permanently reflected in the database. When a transaction stops before it completes all of its operations, the transaction *aborts* and any operations already executed are undone, or rolled back. A transaction may abort for various reasons such as a user request or an internal error in the database system.

This thesis is primarily focused on transactions that contain writes of one or more data items, so read-only workloads are not considered. The following is a simple example of a transaction with a sequence of operations to transfer funds ($100) from bank account A to bank account B.

$$
\begin{aligned}
&START \quad TRANSACTION \\
&balanceA \leftarrow READ(A) \\
&balanceB \leftarrow READ(B) \\
&WRITE(A, balanceA - 100) \\
&WRITE(B, balanceB + 100) \\
&COMMIT \quad WORK
\end{aligned}
$$

This transaction will read the balances from accounts A and B, and then remove $100 from account A and add it to account B. All of these operations belong to the same transaction, or unit of work, so they are logically related to each other. This means that all of the operations, or none of the operations should succeed. This property is called *atomicity*. However, it can be difficult to provide atomicity because database systems may abort a transaction because of an internal error, or may fail or crash. Therefore, to ensure atomicity, database systems must carefully manage transactions to remove the effects of partial execution of transactions. After the transaction commits, the results of the operations are made permanent in the database and will not be lost even through a system failure. This property is called *durability*. Along with *atomicity* and *durability*, database systems ensure two other properties for transactions, *consistency* and *isolation*. Together, the four fundamental properties of transactions are known as the *ACID properties*: Atomicity, Consistency, Isolation, and Durability. The following section further describes the *ACID properties*.

2.3.1 ACID Properties

There are four fundamental properties that database systems ensure for transactions.

2.3.1.1 Atomicity

Atomicity is the property that either all of the operations in a transaction are successful, or none of them are. With this property, developers do not have to worry that only some of the operations in a transaction will complete, for whatever reason. When a transaction cannot complete successfully for any reason such as internal errors, system failures, or a user decision to abort, the database system ensures that the partial execution of the incomplete transaction does persist. The database system achieves *atomicity* by *undoing* the operations of the partially executed transaction. By *undoing* those operations, the database system can return to the state before the transaction began, and developers do not have to be concerned with incomplete transactions. This can greatly ease application development.

2.3.1.2 Consistency

The database system ensures that every transaction transforms the state of the database from one consistent state to another consistent state, even when many transactions are executing

concurrently, or when there are failures in the system. The notion of "consistent" state is dependent on the developer and application, and it is assumed that each individual successful transaction would keep the database consistent if run by itself with no concurrency or failures. Maintaining domain integrity constraints is an example of keeping consistent database state. Domain integrity constraints are user-defined rules for the valid domain for an attribute. A rule defined to disallow product sale prices to be negative ($price >= 0$), is an example of an integrity constraint.

2.3.1.3 Isolation

Isolation is the property that when a transaction executes, it does not observe the effects of other concurrent operations of other transactions. Each transaction views the database as if it were the only one executing in the system, even though it may not be. The database system guarantees that the net result of concurrent transactions with interleaved operations is equivalent to executing the transactions individually, in some serial order. For example, if there are two concurrent transactions, $T1$ and $T2$, the database system will guarantee that the states of the database will be identical to running all of $T1$ and then $T2$, or running all of $T2$ and then $T1$. There are varying levels of isolation that database systems provide, and they are described further in Section 2.3.2.

2.3.1.4 Durability

Durability is the property that the effects of a transaction are never lost, after the transaction has committed. This means even if the database system or various components fail, the effects should still persist. Database systems usually achieve durability of transactions by using a *log*, which records the operations of the transactions in the system. The database log is a sequential, in-order history of database operations that is stored on storage that can survive system failures. The log must persist on a stable storage device such as a disk, because the contents in memory are lost when a system crashes. Systems use the Write-Ahead Logging (WAL) protocol for the log, which requires any update to the database must be recorded and durable in the log *before* that update is durable in the database. WAL also requires that log entries must be flushed to durable storage in log sequence order. Because all operations are recorded in the log, a transaction is considered committed after all of its operations (including the commit record) is in the log on durable storage. So, even when the database system restarts after a crash, it can reconstruct and recover the state of the committed transactions by using the information in the log. Therefore, even with failures, the effects of committed transactions are not lost and will continue to persist in the database.

2.3.2 Isolation Levels

Database systems guarantee the *isolation* property for transactions executing concurrently. Transactions are not exposed to the actions of other concurrent transactions in the system.

However, there are varying levels of isolation [15] that a database system can provide. In general, stronger levels of isolation lead to lower degrees of concurrent execution, and lower levels of isolation enable higher degrees of concurrency. This section discusses the common isolation levels available in database systems.

Some terminology is commonly used in discussing isolation levels. A *history* is an ordering of operations (reads and writes) of a set of transactions. Although transactions execute concurrently, the ordering of the interleaved operations can be modeled by a *history*. A *conflict* is any two operations on the same data item from distinct transactions where one of the operations is a write. A *dependency graph* of transactions can be generated from any given *history*. Every *conflict* generates an edge in the graph. If $op1$ of $T1$ happens before $op2$ of $T2$ in the conflict, then there is an edge from $T1$ to $T2$ in the graph. Two histories are equivalent if they both have the same dependency graph for the same set of transactions. If a history is equivalent to some *serial history* (a history from transactions executing without any concurrency), then it is considered *serializable*. For two isolation levels $L1$ and $L2$, $L1$ is *weaker* than $L2$ (or $L2$ is *stronger* than $L1$) if all non-serializable histories of $L2$ are possible in $L1$, and there is at least one non-serializable history that is possible in $L1$ but impossible in $L2$.

2.3.2.1 Serializable

Serializable isolation is the strongest level of isolation provided by ANSI SQL database systems [15]. This level of isolation is related to the concept of *serializability*. As stated above, serializability is the property that a history of interleaved execution of concurrent transactions is equivalent to a history of the same transactions running in some serial order (no concurrency). Each transaction executes *as if* it is fully isolated from others, even when its operations are potentially interleaved with operations of other transactions. Because concurrent execution is equivalent to a serial order, serializable isolation prevents any anomalies related to the interleaved execution.

Serializable isolation is typically achieved with *two-phase locking*. Two-phase locking (2PL) is a commonly used locking protocol for database systems to ensure serializability. 2PL has simple rules for locking: a transaction must acquire read-locks (shared locks) for any data item it reads, and write-locks (exclusive locks) for any data item it writes, and no additional locks may be acquired one any lock is released. It is called two-phase locking because every transaction goes through two phases. The transaction first acquires locks in the *growing* phase, and then releases its locks in the *shrinking* phase. By implementing 2PL, transactions are isolated from others, and this property results in serializable isolation. Since locks are held from the *growing* phase to the *shrinking* phase, this level of isolation sacrifices some concurrent execution for serializable isolation. Also, database systems must detect and resolve potential deadlocks when using a locking protocol.

Serializability is also possible with concurrency control mechanisms that do not use locking. *Optimistic concurrency control* (OCC) is a common approach that avoids the overhead of locks. OCC avoids using locks by using three phases. In the **Read** phase, the transaction

reads all the values it needs for its execution. During this phase, the transaction does not write into externally visible locations, but rather to private locations local to the transaction. In the **Validation** phase, the database system checks to see if the reads and writes of the transaction conflict with other concurrently executing transactions. If the validation passes, then all the private writes are written to the database in the **Write** phase. If the validation fails, then the transaction cannot commit and must be restarted. OCC is called *optimistic* because it allows all concurrent execution, and then later checks to see if there could be isolation issues. Because it is optimistic, OCC is typically appropriate for low-contention scenarios. However, in scenarios with high-contention, the OCC validation will fail more often which leads to more transaction restarts, thus reducing the performance of the database system.

2.3.2.2 Repeatable Read

Repeatable read is an isolation level with fewer guarantees compared to serializable isolation. Repeatable read guarantees that a transaction only reads data items and data item updates from committed transactions. It also ensures that any data item a transaction T reads or writes is not updated by another transaction until T completes. However, repeatable read isolation allows *phantom read* anomalies, which arise when a transaction reads additional data items on the second execution of the identical query. For example, query Q reads all users with $age > 20$, and transaction T executes Q twice. If another transaction inserts a new data item with $age > 20$ in between the two executions of query Q, then transaction T will see additional items in the result not found in the first result. With a lock-based implementation, repeatable read isolation allows phantom reads to occur because locks on predicates or ranges ($age > 20$ in the example) are not utilized.

2.3.2.3 Read Committed

Read committed is a level that ensures fewer guarantees than repeatable read isolation. With a lock-based implementation, read committed allows more concurrent execution of transactions by releasing read locks early. For any data item reads in the transaction, read locks are acquired, but they are released as soon as the query completes, and not held until the end of the transaction. If the transaction tries to read the same data items again, the values may have changed because another transaction may have modified those items and committed. This anomaly is called *fuzzy reads*. However, data items that a transaction reads are guaranteed to have been already committed by some earlier transaction. Currently, Microsoft SQL Server, Oracle Database, and PostgreSQL all use read committed isolation as default.

2.3.2.4 Read Uncommitted

Read uncommitted is an isolation level weaker than read committed. Read uncommitted isolation allows *dirty reads*, which is an anomaly where a transaction may read data values

which never existed in any consistent state of the database. For example, a transaction can read data items written by another transaction *still in progress*, so if the other later aborts, the previously read data items never really existed in the database. With a lock-based implementation, read uncommitted does not take acquire any read locks, so it allows higher levels of concurrency, while allowing transactions to read uncommitted values.

2.3.2.5 Snapshot Isolation

Snapshot isolation is a level that is typically implemented with *Multi-Version Concurrency Control* (MVCC) rather than locking. In relation to the other levels, snapshot isolation is stronger than read committed, but weaker than serializable isolation. MVCC works by creating a new version of each data item each time it is modified, and allows different transactions to see different versions of the data. With MVCC, different transactions read different snapshots of the database, without having to use read locks. A snapshot is the state of the database at a particular time, which is usually the time when the transaction started. Avoiding locks increases the concurrency and can greatly improve performance. Transaction T can commit only when the updated data has not been changed by other concurrent transactions since the start of transaction T. This validation is checked using global start and commit timestamps. If a value was modified, it means another transaction already committed an update to that value, so transaction T is aborted.

2.4 Scaling Out Database Management Systems

Because workloads and data sizes are growing, database management systems must be able to scale to handle the load and storage demands. Scaling out, or adding additional servers, is a common technique to increase performance and capacity of a system, especially in the age of cloud computing. Being able to add capacity by quickly and incrementally adding servers to a cluster has been a great benefit of cloud computing. Many database systems scale out by horizontally partitioning their data, and distributing the partitions across many servers.

There are two main techniques to scale out database systems: sharded transactional database systems, and distributed NoSQL database systems. This section describes the techniques used in distributed database systems for scaling out.

2.4.1 Data Replication

Most distributed database systems replicate their data. Replication is when systems store multiple copies, or replicas, of the data at separate locations. Although replicas are allowed to store only a proper subset of the data, for the purposes of this thesis, full replication (replicas store a full copy of the data) is the main focus. There are several benefits for data replication:

Durability With replication, distributed systems can provide durability by avoiding a single point of failure for the data. Even if one replica fails, the data is still intact in the other replicas. Therefore, data replication can provide durability even with the existence of failures.

Availability Replication also provides availability of the data during failures. Because the data can be accessed from different replicas, even if there is a failure to one of the servers, the data can be retrieved from another copy.

Reduced Latency With replication, separate replicas can be placed closer to the users and applications accessing the data. As the data is placed closer to the access point, the network latency of interacting with the data is reduced.

Scalability Replication can also allow database systems to scale out to accommodate growth in the workload. Additional replicas can be added to the system to add capacity for adjusting to new workloads.

Because of these benefits, most database systems use data replication. However, there are many variations in how replicas are kept in-sync with each other, and the rest of this section discusses the different types of replication techniques relevant to this thesis.

2.4.1.1 Master-based vs. Distributed Replication

One aspect of data replication is *where* the replicas are updated. The two main alternatives are master-based replication and distributed replication. Master-based replication is a centralized technique that requires all updates to modify the master replica first, and then the master replica propagates the updates to the rest of the replicas. For partitioned database systems, there is usually a separate master for each partition. The master-based approach is advantageous because it is simple for the application to update all the data at the master, and the master has all the up-to-date data. However, since it is a centralized technique, the master can be a bottleneck, and if the master fails, the data will be unavailable until the master is restored.

In contrast to master-based replication, distributed replication does not require that all updates go through the master. Updates can be applied to different replicas in different orders. This technique is scalable since there is no designated master that may become the bottleneck, and a replica failure does not render the data unavailable. However, distributed replication can be more complicated because the data can be updated from multiple replicas. This means there is no single master that has the up-to-date data, and the replicas may not be in-sync with each other.

2.4.1.2 Synchronous vs. Asynchronous Replication

One aspect of data replication is *when* updates to the data are propagated to replicas. The two alternatives are synchronous replication and asynchronous replication. With syn-

chronous replication, all updates to replicas are part of the originating transaction, so the transaction cannot commit until all replicas are updated. With synchronous replication, all the replicas are up-to-date, and reading any of the copies will retrieve the latest data. However, there are drawbacks to synchronous replication. The transaction commit must wait for the replication to complete, so communicating with all the replicas will negatively impact the latency of transactions. Also, if a replica fails, the transaction will block, waiting the replica to be restored.

Asynchronous replication differs from synchronous replication, by separating the replicas updates from the originating transaction. A transaction can commit before any of the replicas are successfully updated. Transaction commits do not have to wait for the replicas, so replication does not affect the response times of commits. However, asynchronous updates means different replicas will not be exact copies of each other. This reduced level of consistency may be appropriate for some workloads, but is not suitable for all. Also, durability suffers when a replication failure occurs after the commit, but before any replica is successfully updated. In this scenario, the transaction will be considered committed without any of the replicas having the effects of that transaction, so the committed transaction would be lost.

2.4.1.3 Geo-Replication

Geo-replication is data replication that is performed across geographically diverse locations, which can be distributed all over the world. Geo-replication is not a specific protocol for replication, but it is a replication scheme for systems deployed across several different sites. Large Internet companies like Google and Facebook have made geo-replication popular, in order to reduce correlated failures like natural disasters, and to place replicas closer to users all around the world. However, the major drawback is the high network latency between the replica locations. Since sites may be geographically far from each other, the network latency between sites could be hundreds of milliseconds. Therefore, different replication techniques are required to optimize for this environment. Later in this thesis, I present new techniques for distributed transactions that address the challenges of geo-replication.

2.4.2 Sharded Transactional Database Systems

A popular way to scale relational database systems in the cloud is to *shard* the database. This technique involves horizontally partitioning the data, and storing each partition on separate database instances on separate servers. Sharding is a special case of horizontal partitioning, because the partitions do not have to be on the same database server. By partitioning the data onto multiple machines, the database system can scale out to handle more load, and store more data. Sharding is typically deployed with transactional relational database systems, such as MySQL or PostgreSQL, running separately for each individual partition. This technique enables ACID transactions within each shard of the database, but since each partition is a separate instance, there is no simple way to coordinate between all of them

for queries or updates. For queries, external tools must be developed in order to combine results from all the partitions. Join queries are particularly difficult because more of the query processing must be done in the client and not the database system. With sharding, transactions cannot span multiple shards, so this limits the kinds of updates possible in the system. Database schemas and applications must be designed carefully in order to prevent cross-partition transactions. If the system is not defined carefully, resharding may be required. Resharding is the re-partitioning of data across all the individual databases, and it may be quite complex and expensive to perform.

2.4.3 Distributed NoSQL Database Systems

NoSQL systems are a newer class of distributed database systems, that emphasize performance, scalability and availability. In focusing on simplicity and scalability, NoSQL data stores do not usually support relational data, or ACID transactions found in traditional database systems. There are several different types of NoSQL data stores, but distributed key-value stores are popular for addressing workloads similar to those of relational database systems.

A distributed key-value store manages a collection of key-value pairs, where the *value* is the data to be stored and managed, and the *key* is an identifier for the *value*. The key-value data model provides two ways to operate on the data: *put(key, value)* and *get(key)*. The interface *put(key, value)* is for inserting or updating the *value* for a specific *key*, and the interface *get(key)* is for reading the *value* for a specific *key*. An extension to the key-value data model is the ordered key-value data model, which enables access to a contiguous range of keys. The key-value data model allows applications to easily read and write data associated with keys. The simplified interface allows these systems to be very performant and scalable. However, because of the simple interface, distributed key-value stores do not support transactions across multiple keys. Some systems even provide lower levels of consistency for the value associated with a single key, in order to provide more availability during failures. In general, most of these data stores support durability of data and updates using replication. By duplicating the data on multiple servers, the data will not be lost even when a server fails.

Bigtable [24] is a key-value store from Google, which inspired other similar systems such as Apache HBase [7]. The architecture is organized as a sorted map partitioned across many servers. Every key-value insert or update goes to a master *tablet* responsible for the key, and it provides durable, single-key transactions. Amazon Dynamo [33] and Apache Cassandra [6] are different in that they are quorum-based key-value stores that provide eventual consistency. The quorum-based protocol means a group (or quorum) of servers is responsible for a particular key, as opposed to a single master. These systems provide eventual consistency per key, meaning eventually, if new updates cease, the data item will converge to the correct answer. However, special merge rules must be defined in order to handle conflicting updates. By adjusting the read and write quorum, Dynamo can provide a variety of consistency and performance levels. However, operations can only involve a single

key at a time. Dynamo and Cassandra give up transactions and stronger consistency in favor of availability, performance, and scalability. Yahoo's PNUTS [30] is another key-value store which provides single-key transactions for timeline consistency, by routing all writes to the master-per-key. However, asynchronous replication is used to replicate the key updates, so data loss may be possible during failures.

For providing transactions containing writes to multiple records, Google developed Megastore [13] and Spanner [31]. Both systems horizontally partition the data across many servers. Megastore provides ACID transactions for each partition, but the transactions must be serialized per partition. This could greatly reduce the throughput of transactions. Transactions that span multiple partitions are strong discouraged, because two-phase commit is required and greatly increases the latency. Spanner improves on Megastore by synchronizing time in the entire cluster of servers with atomic clocks, to provide global timestamp ordering for snapshot isolation. Both Megastore and Spanner use Paxos for storing transaction log positions across a quorum of servers, and both must serialize commit log records per partition. Because transactions spanning multiple partitions must use two-phase commit, the increased latency limits the scalability of the systems.

2.4.4 Distributed Architecture

While most distributed database systems have different designs and goals, their architectures do have similarities. A major commonality in the designs of these scalable database systems is that they are all *shared nothing architectures* [80]. In a shared nothing architecture, all the nodes or servers do not share memory or disk resources between each other. Each server has its own private memory and disk, and can be viewed as an independent unit in the system. Because each node operates independently without sharing memory or disk storage, shared nothing architectures are good at scaling out to a large number of machines. Since these architectures are easily scalable, they are widely used for modern, large-scale database systems.

Figure 2.2 shows an example of a shared nothing architecture for a distributed database system. In the figure, the database system is made up of several nodes (servers), represented by the colored squares. The database is distributed across different partitions, and in the figure, different partitions have different colors. Each partition is replicated, so each replica of a partition is represented by the same color in the figure.

Figure 2.2 also shows how the application interacts with the distributed database system. There may be many application servers, all independent from each other, and also the database system. The application runs with a client library which provides a lot of the functionality of the distributed database system, such as transactions or join queries. The separation of the application layer and the storage layer allows each layer to scale out independently. The architecture shown in Figure 2.2 is similar to existing systems, like Megastore, and is the main distributed architecture used for the rest of this thesis.

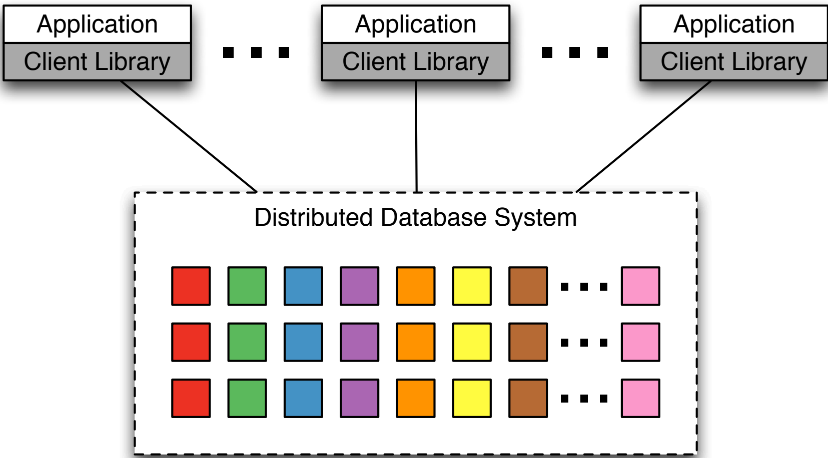


Figure 2.2: Typical scalable architecture for distributed database systems

2.5 Transactions in a Distributed Setting

Executing transactions in a distributed setting face additional challenges. Communication is required in order to coordinate between different participants, and the communication is typically executed over the network, contributing additional latency. This added network latency from coordination causes distributed transactions to be slower than transactions processed on a single server. Also, in a distributed setting, there are other possible sources of failures, such as individual server crashes and communication failures. Even partial failures at a site or in the network can cause serious issues with transactions. These challenges make providing transactions in a distributed setting more difficult. In this section, I discuss several existing techniques useful for distributed transactions.

2.5.1 Two-Phase Commit Protocol

The two-phase commit protocol [16, 64] is an atomic commit protocol that is widely used for committing transactions atomically over a distributed collection of participating processes. For a particular transaction using the two-phase commit protocol, there are various abstract roles; there is a single coordinator, and several participants. The transaction's coordinator is responsible for executing the two-phase commit protocol to durably commit the transaction among the other participants. In normal operation, there are two phases: the voting phase and the commit phase.

Voting Phase In the voting phase (or prepare phase), the transaction coordinator asks all the other participants to *prepare* to commit the transaction locally, and each participant

replies with a vote of "yes" or "no" (depending on success or failure). Each participant *prepares* to commit by executing the local transaction until right before it can commit. This involves acquiring all the required locks on data items and writing entries into the transaction log. After the coordinator collects all the responses, the coordinator must decide whether or not to commit the transaction. If any of the participants replies with a "no", then the transaction must be aborted. The coordinator logs the decision to its own durable log, and then proceeds to the commit phase.

Commit Phase In the commit phase, the coordinator sends a message to inform all the participants of the transaction result, and to complete the processing of the prepared transaction. The participants must wait until the receive the final transaction result from the coordinator to make progress.

Two-phase commit is the accepted protocol for distributed transactions, but it requires two round-trip message rounds, and is not completely fault tolerant. For example, if the coordinator fails before making the commit/abort decision, the rest of the processes will not know the outcome of the transaction. In fact, if the coordinator and a participant fail, the protocol will block, and not be able to recover and make progress. It is possible that the failed participant received a commit message before any of the other participants and already performed the local commit, while the other participants are still waiting for the decision. Because of this uncertainty, the state of the transaction is unknown and the protocol is blocked until the failed participant is restored. The next section describes three-phase commit, a solution for this scenario.

2.5.2 Three-Phase Commit Protocol

Three-phase commit [78] is a non-blocking solution, that adds an additional message round to 2PC. Thee three phases are the voting phase, pre-commit phase, and the commit phase. The voting phase and the commit phase are similar to the corresponding phases from two-phase commit.

Voting Phase This phase is similar to the voting phase for two-phase commit. The transaction coordinator asks all the other participants to vote to commit the transaction. Once the coordinator receives all the responses, it decides commit if and only if all responses are "yes".

Pre-Commit Phase After the coordinator decides to commit the transaction, it sends a pre-commit message to all the participants, and the participants transition to a *pre-commit* state and respond with an "ACK". Once the coordinator receives the "ACKs" from all the participants, it can move on to the commit phase.

Commit Phase The coordinator sends the commit message to all the participants, and the participants must wait for the commit message in order to finish executing the transaction and progress.

Three-phase commit eliminates some of the problems with failures in two-phase commit by introducing an additional pre-commit phase. In the pre-commit phase, the coordinator distributes the intent to commit, so when a participant is in the *pre-commit* state, it is already certain that all participants have voted “yes” in the voting phase. In order for the coordinator to move on to the commit phase, it must receive an “ACK” from every participant, so then it knows that every participant is in the *pre-commit* state. Therefore, even if a participant fails (either before or after receiving the commit message), the other participants are still certain of the outcome of the transaction, and the protocol does not have to block. However, introducing this additional phase imposes an additional round-trip of messages which will adversely affect the latency of commits.

2.5.3 Paxos Distributed Consensus Algorithms

Distributed consensus is the process of coordinating a set of participants to agree on a single value, especially with the possibility of server or network failures. Distributed consensus can be an effective component for executing transactions by providing durability in unreliable and distributed environments. For example, systems like Megastore or Spanner use a distributed consensus protocol to store durable logs that tolerate server failures. The Paxos [54] family of algorithms solves the distributed consensus problem. Many systems use the Paxos protocol for durably storing values among a set of participants, while tolerating failures of servers. There are two main roles in the classic Paxos algorithm: proposers and acceptors. A proposer is responsible for submitting a value to store, and acceptors are responsible for storing the value. In practice a single process can take on multiple roles. Paxos also requires quorums in the set of acceptors. A quorum is defined to be a subset of the acceptors, such that any two quorums has at least 1 acceptor in common. Any majority of the acceptors is typically used for Paxos quorums. The classic Paxos algorithm works in two phases to store a value among the acceptors.

Phase 1 In the first phase, a proposer sends a *Prepare* message to all acceptors, with a proposal number greater than any previously used proposal number. If an acceptor receives a *Prepare* message with a proposal number N greater than any previously received proposal number, then the acceptor replies with a *Promise*, promising that the acceptor will ignore all messages with proposal number less than N. If an acceptor already received a value to store from a proposer at an earlier time, the acceptor also sends that value with the *Promise* message.

Phase 2 Once the proposer receives a *Promise* from a quorum of acceptors (a majority, for classic Paxos), the second phase starts. The proposer examines the set of *Promise* responses, and determines if any of the messages contain a value. If at least one of the *Promise* messages contain a value, then the proposer must choose the value with the largest proposal number. If there were no previously proposed values, then the proposer can freely choose a value to propose. In typical scenarios, an application will

submit a value to the proposer to store, so if the proposer has the choice, it will choose the value supplied by the application. The proposer then sends an *Accept* message with the chosen value to the acceptors. When an acceptor receives an *Accept* message with a proposal number N, it will accept the value and respond with an *Accepted* message if and only if the acceptor did not make a promise for another *Prepare* message with a proposal number greater than N.

When the the proposer, or a separate learner process, receives the *Accepted* message from a quorum of acceptors, that means the value has been safely stored in the set of acceptors. Since Paxos depends on receiving responses from quorums and not every single acceptor, it has the property to be able to tolerate failures and to continue to make progress during failures. So, as long as a quorum of the acceptors is still operating, Paxos can still make progress. Since Paxos uses two phases, to successfully save a value requires two round-trip message rounds.

While classic Paxos is the core of the consensus algorithm, most systems use Multi-Paxos as a common optimization. Multi-Paxos is like the classic algorithm, but the proposer can "reserve" the leadership for multiple instances of Paxos. This allows for the proposer to continue to propose and store values in Paxos instances without having to execute **Phase 1** each time. This greatly reduces the latency for durably storing values over several distributed acceptors.

2.5.4 Consensus on Commit

Consensus on commit [39] is a distributed commit algorithm which uses both two-phase commit and the Paxos algorithm. Consensus on commit solves the blocking problem of two-phase commit, by using the Paxos algorithm to durably store decisions among several participants. Therefore, individual failures are not fatal, since decisions are stored in multiple locations. With consensus on commit, each participant of the two-phase commit protocol stores its *prepare* vote among acceptors in an instance of Paxos, before informing the coordinator. Therefore, if the coordinator fails, the correct state can be recovered from the Paxos state.

2.6 Materialized Views

The previous sections present background on transactions in single-server and distributed database systems. However, it is also important to address how to read the transactional data. Read queries are part of many use cases such as interactive workloads or analytic workloads, so improvements to read queries can positively impact performance for a variety of applications. This section describes the background on database views and how they can benefit read queries, and some of the challenges when implemented in distribute database systems.

In relational databases, a view is a virtual relation that represents the results of a query in the database. A view is defined as a query over database tables or views. Users can

query views just like any other table, but the contents of views are not materialized in the database. Views enable *logical data independence*, because the base table schemas can be modified underneath the views without having to change application and user queries. Since the contents of views are not stored in the database, accessing views is always computed at runtime.

In contrast to logical views, materialized views store the contents of the view in the database. A materialized view stores the precomputed results of its view query, so it uses additional storage for the cached results. Materializing the contents can improve read performance because querying the view does not require recomputing the view for every access. Since materialized views store precomputed results, database systems must handle how to update or maintain the views when the base data is updated. The simplest method of maintaining materialized views is to recompute the entire view every time the base data is updated. However, this can be quite expensive, especially when the data is updated frequently and the updates are small relative to the entire view. Therefore, previous work focused on *incremental view maintenance* of views. View maintenance is inherently relevant to transactions because base table updates and view updates that depend on those base table updates are semantically related and should occur as an atomic unit. This thesis focuses on the challenges of scalable maintenance for join views. In the rest of this section, I present prior work on materialized view maintenance.

2.6.1 Incremental View Maintenance

There has been significant previous work on incrementally updating materialized views in database systems. Early work in incremental view maintenance from Blakely et al. [17], and Ceri and Widom [23] investigated new algorithms and rules to update materialized views with smaller updates, instead of re-computing the entire view. Ceri and Widom introduced a method of maintaining materialized views with automatic generation of *production rules*. Production rules are a collection of operations executed, or triggered, for every insert, update, or delete of the base tables tuples related to the view. These incremental production rules are triggered to run within the same base table transactions, in order to update materialized views correctly.

Automatic generation of production rules depends on the properties of *safe table references* and *no duplicates*. A view definition has *safe table references* if base table references (in the projection or equality predicates) contain a key of the base tables. Views should also not contain duplicate records for efficient incremental production rules. A common way guaranteeing a view does not contain duplicate records is to include the keys of the base tables in the view definition. This thesis also focuses on views without duplicate records, by including the key columns of the base tables in view definitions.

While many of the incremental view maintenance techniques update views synchronously with the base transactions, other work has been done for asynchronous, or deferred, view maintenance [69, 88, 86, 71]. These deferred techniques opt for some staleness in the views for faster base transactions and potentially improved system utilization. Because of the de-

ferred update to views, it is possible to update views incorrectly and leave inconsistencies. A common technique to correctly update views is to use compensation queries to adjust the results of the incremental updates. In order to determine how to generate the compensation queries, compensation algorithms must examine the sequential sequence of committed transactions.

In addition to deferred maintenance, other related work have investigated updating materialized views from many different distributed sources [87, 2, 84, 26]. These algorithms are particularly effective for data warehouses. A data warehouse is a system designed for analytical workloads, instead of transactional workloads, and is typically separate from the transactional system. In this setting, distributed data sources are typically separate from the data warehouse, and the views in the data warehouse are updated incrementally as updates from the sources arrive. These new techniques use the sequential sequence of transaction updates at the data warehouse, and issue compensation queries to the various data sources to update the views correctly.

2.6.2 Distributed View Maintenance

Most of the prior work on incremental view maintenance have been focused on single-server database or data warehouse systems. Even in scenarios with multiple data sources in a data warehouse environment, the maintenance algorithms are centralized and need to examine the global sequence of committed transactions. This reliance on centralized algorithms and global sequence of transactions is not scalable for large distributed database systems. When view maintenance requires a centralized algorithm, it has to process all of the updates in the system, so it cannot scale out and may become a bottleneck. Also, if an algorithm needs the sequential order of all operations in the system, a centralized sequencer is required, which can be a bottleneck. Centralized designs make these algorithms less suitable for scalable distributed database systems.

For scalable systems, it is important to be able to distribute, or scale out, algorithms. Asynchronous and distributed view maintenance [3, 85] has been investigated for modern scalable distributed database systems. While previous techniques have focused on incremental view maintenance with a single or a few view updaters, work on distributed view maintenance investigate the challenges when scaling out to many view updaters. Distributed view maintenance involves asynchronously maintaining secondary indexes, and co-locating indexes on join keys. By co-locating related records in the indexes, local joins can be performed for the materialized views. In this thesis, I explore new techniques for scalable algorithms for maintenance of join views.

2.7 Summary

There has been significant previous work investigating transactions for traditional and modern distributed database systems. However the various techniques depend on centralized

algorithms or costly coordination and communication. These centralized or costly techniques will not be able to scale with database systems scaling out to many machines. In the rest of this thesis, I present new algorithms and techniques for scalable transactions in distributed database systems.

Chapter 3

A New Transaction Commit Protocol

3.1 Introduction

Modern applications demand more performance and capacity from their data management systems, so large-scale distributed database systems are commonly used. As applications become more popular, they can experience exponential growth from users all over the world. Therefore, many applications use several different data centers in order to allow users to access the closest data center, to reduce their response times. So, tolerance to the outage of a single data center is now considered essential for many online services. Achieving this for a database-backed application requires replicating data across multiple data centers, and making efforts to keep those replicas reasonably synchronized and consistent. For example, Google's e-mail service Gmail is reported to use Megastore [13], synchronously replicating across five data centers to tolerate two data center outages: one planned, one unplanned.

Replication across geographically diverse data centers (called geo-replication) is qualitatively different from replication within a cluster, data center or region, because inter-data center network delays are in the hundreds of milliseconds and vary significantly (differing between pairs of locations, and also over time). These delays are near the limit of total latency that users will tolerate, so it becomes crucial to reduce the number of message round-trips taken between data centers, and desirable to avoid waiting for the slowest data center to respond.

For database-backed applications, it is very valuable and useful when the underlying database system supports transactions: multiple operations (such as individual reads and writes) grouped together. Transactions are valuable to developers because of their ACID properties: Atomicity, Consistency, Isolation, and Durability. More details on transactions are described in Section 2.3. When the underlying database system ensures the ACID properties, it is easier for developers to reason about the behavior of database interactions. The traditional mechanism for transactions that are distributed across multiple servers is called two-phase commit (2PC), but this has serious drawbacks, especially in a geo-replicated system. 2PC depends on a reliable coordinator to determine the outcome of a transaction,

so it will block for the duration of a coordinator failure, and (even worse) the blocked transaction will be holding locks that prevent other transactions from making progress until the recovery is completed. Therefore, this can greatly limit the scalability of distributed transactions. Three-phase commit, described in the background section 2.5.2, is a variation of 2PC that replicates the log entries at the cost of an additional message round-trip, thus increasing the latency.

To avoid the long latencies of coordinating transactions with 2PC, many distributed systems sacrifice some of the guarantees of transactions. Some systems achieve only eventual consistency by allowing updates to be run first at any site (preferably local to the client) and then propagate asynchronously with some form of conflict resolution so replicas will converge later to a common state. Others restrict each transaction so it can be decided at one site, by only allowing updates to co-located data such as a single record or partition. In the event of a failure, these diverse approaches may lose committed transactions, become unavailable, or violate consistency.

Various projects [39, 13, 57, 31] proposed to coordinate transaction outcome based on Paxos [54]. The earliest design, *Consensus on Transaction Commit* [39], shows how to use Paxos to reliably store the abort or commit decision of a resource manager for recovery. However, it treats data replication as an orthogonal issue. Newer proposals focus on using Paxos to agree on a log-position similar to state-machine replication. For example, Google's Megastore [13] uses Paxos to agree on a log-position for every commit in a data shard called *entity group* imposing a total order of transactions per shard. Unfortunately, this design makes the system inherently unscalable as it only allows executing one transaction at a time per shard; this was observed [48] in Google's App Engine, which uses Megastore. Google's system Spanner [31] enhances the Megastore approach, automatically resharding the data and adding snapshot isolation, but does not remove the scalability bottleneck as Paxos is still used to agree on a commit log position per shard. Paxos-CP [65] improves Megastore's replication protocol by combining non-conflicting transactions into one log-position, significantly increasing the fraction of committed transactions. However, the same system bottleneck remains, and the experimental results of Paxos-CP are not encouraging with a throughput of only four transactions per second.

Surprisingly, all these new protocols still rely on two-phase commit, with all its disadvantages, to coordinate any transactions that access data across shards. They also rely on a single master, requiring two round-trips from any client that is not local to the master, which can often result in several hundred milliseconds of additional latency. Such additional latency can negatively impact the usability of websites; for example, an additional 200 milliseconds of latency, the typical time of one message round-trip between geographically remote locations, can result in a significant drop in user satisfaction and abandonment of websites [73].

In this chapter, I introduce MDCC (short for "Multi-Data Center Consistency"), an optimistic commit protocol for transactions with a cost similar to eventually consistent protocols. MDCC requires only a single wide-area message round-trip to commit a transaction in the common case, and is "master-bypassing", meaning it can read or update from any node in

any data center. MDCC replicates data synchronously, so the data is still available even if an entire data center fails and is inaccessible. Like 2PC, the MDCC commit protocol can be combined with different isolation levels that ensure varying properties for the recency and mutual consistency of read operations. In its default configuration, it guarantees "read-committed isolation" without lost updates [15] by detecting and preventing all write-write conflicts. That is, either all updates of a transaction eventually persist or none *(atomic durability)*, updates from uncommitted transactions are never visible to other transactions *(read-committed)*, concurrent updates to the same record are either resolved if commutative, or prevented (no lost updates), but some updates from successful committed transactions might be visible before all updates become visible *(no atomic visibility)*. It should be noted, that this isolation level is stronger than the default, read-committed isolation, in most commercial and open-source database platforms. On the TPC-W benchmark deployed across five Amazon data centers, MDCC reduces per transaction latencies by at least 50% as compared to 2PC or Megastore, with orders of magnitude higher transaction throughput compared to Megastore.

MDCC is not the only system that addresses wide-area replication, but it is the only one that provides the combination of low latency (through one round-trip commits) and strong consistency for transactions, without requiring a master or significant limitations on the application design (static data partitions minimizing cross-partition transactions). MDCC is the first protocol to use Generalized Paxos [52] as a commit protocol on a per record basis, combining it with adapted techniques from the database community, such as escrow transactions [63] and demarcation [14]. The key idea is to achieve single round-trip commits by 1) executing parallel Generalized Paxos on each record, 2) ensuring every prepare has been received by a *fast* quorum of replicas, 3) disallowing aborts for successfully prepared records, and 4) piggybacking notification of commit state on subsequent transactions. A number of subtleties need to be addressed to create a "master-bypassing" approach, including support for commutative updates with value constraints, and for handling conflicts that occur between concurrent transactions.

The remainder of this chapter is organized as follows. In Section 3.2 I show the overall architecture of MDCC. Section 3.3 presents MDCC, a new optimistic commit protocol for the wide area network, that achieves wide-area transactional consistency while requiring only one network round trip in the common case. Section 3.4 discusses the read consistency guarantees of MDCC. Experimental results of MDCC and other systems across five data centers are in Section 3.5. In Section 3.6 I relate MDCC to other existing work, and I conclude the chapter in Section 3.7.

3.2 Architecture Overview

Background Section 2.4.4 showed an overview of the typical design of scalable database systems. In this section, I describe MDCC-specific variations to that distributed architecture. MDCC uses a library-centric approach similar to the architectures of DBS3 [19],

Megastore [13] or Spanner [31], and is shown in Figure 3.1. This architecture separates the stateful component of a database system as a distributed record manager. All higher-level functionality (such as query processing and transaction management) is provided through a stateless DB library, which can be deployed at the application server.

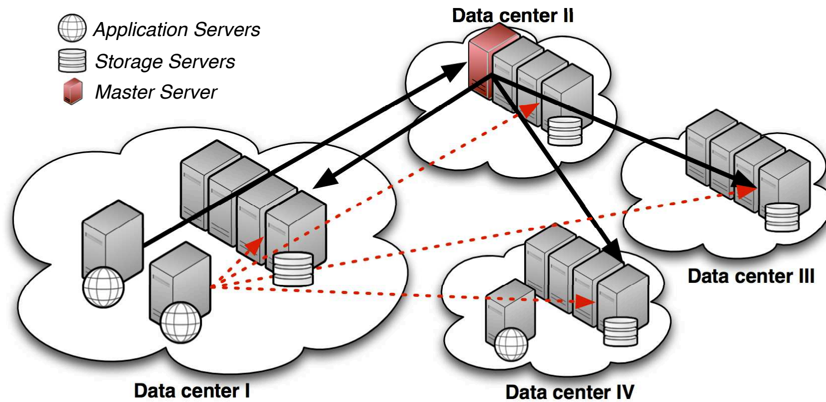


Figure 3.1: MDCC architecture

As a result, the only stateful component of the architecture, the storage node, is significantly simplified and scalable through standard techniques such as range partitioning, whereas all higher layers of the database system can be replicated freely with the application tier because they are stateless. Every storage node is responsible for one or more horizontal partitions of the data and partitions are completely transparent to the application. MDCC places storage nodes in geographically distributed data centers, with every node being responsible for one or more horizontal partitions. Although not required, every data center contains a full replica of the data, and the data within a single data center is partitioned across multiple machines.

The DB library provides a programming model for transactions, and is mainly responsible for coordinating the replication and consistency of the data by using the MDCC commit protocol. The DB library also acts as a transaction manager and is responsible to determine the outcome of a transaction. In contrast to many other systems, MDCC supports an individual master per record, which can either be storage nodes or app-server and is responsible to coordinate the updates to a record. This allows the transaction manager to either take over the mastership for a single record and to coordinate the update directly, or to choose a storage node (the current master) to act on its behalf (black arrows in Figure 3.1). Furthermore, often it is possible to avoid the master altogether, allowing the transaction manager to coordinate the update, without acquiring any mastership (red arrows in Figure 3.1). This leads to a very flexible architecture in which storage nodes or application servers can act as coordinators, depending on the specific situation.

In the remaining sections of this chapter, I present the MDCC transaction commit protocol.

3.3 The MDCC Protocol

In this section, I describe MDCC, a new optimistic commit protocol for transactions operating on cross-partition, synchronously replicated data in the wide-area network. Intra-data center latencies are largely ignored because they are only a few milliseconds compared to hundreds of milliseconds for inter-data center latencies. The target is a fault-tolerant atomic commit protocol with reduced latency from fewer message rounds by avoiding contacting a master, and high parallelism. MDCC makes the trade-off of reducing latency by using more CPU resources to make sophisticated decisions at each site. MDCC exploits a key observation of real workloads; either conflicts are rare, or conflicting updates commute until a domain integrity constraint (e.g., add/subtract with a value constraint that the attribute should be greater than 0).

At its core, the protocol is based on known extensions of Paxos, such as Multi-Paxos [54] and Generalized Paxos [52]. Innovations I introduce enhance these consensus algorithms in order to support transactions on multiple data items without requiring static partitioning. In this section, I present a sequence of optimizations, refining from an initial design to the full MDCC protocol. Section 3.3.2 describes methods to allow multi-record transactions with read-committed isolation without lost updates (see Section 3.4.1) using Multi-Paxos, with two round-trips of messaging. Section 3.3.3 incorporates Fast Paxos, so one round-trip is often possible even without a local master. Then Section 3.3.4 describes how Generalized Paxos is used to combine commit decisions for transactions that are known to be commutative, and this relies on database techniques that determine state-based commutativity for operations like increment and decrement. While the component ideas for consensus and for deciding transaction commutativity exist in the database community, how MDCC uses them for transaction and the combination of them is novel. In contrast to pessimistic commit protocols such as two-phase commit, the protocol does not require a *prepare phase* and can commit transactions in a single message round-trip across data centers if no conflicts are detected.

3.3.1 Background: Paxos

In this section, I provide additional details to Section 2.5.3 on the principles of Paxos and how MDCC adapts Paxos to update a single record.

3.3.1.1 Classic Paxos

Paxos is a family of quorum-based protocols for achieving consensus on a single value among a group of replicas. It tolerates a variety of failures including lost, duplicated or reordered messages, as well as failure and recovery of nodes. Paxos distinguishes between *clients*, *proposers*, *acceptors* and *learners*. These can be directly mapped to the scenario where clients are app-servers, proposers are masters, acceptors are storage nodes and any node can be a learner. In the remainder of this chapter I use the terminology of clients, masters and

storage nodes. In the implementation, MDCC places masters on storage nodes, but that is not necessary for correctness.

The basic idea in Classic Paxos [54], as applied for replicating a transaction's updates to data, is as follows: Every record has a master responsible for coordinating updates to the record. At the end of a transaction, the app-server sends the update requests to the masters of each the record, as shown by the solid lines in Figure 3.1. The master informs all storage nodes responsible for the record that it is the master for the next update. It is possible that multiple masters exist for a record, but to make progress, eventually only one master is allowed. The master processes the client request by attempting to coordinate the storage nodes to *a*gree on the update. A storage node accepts an update if and only if it comes from the most recent master the node knows of, and it has not already accepted a more recent update for the record.

In more detail, the Classic Paxos algorithm operates in two phases. **Phase 1** tries to establish the mastership for an update for a specific record r. A master P, selects a proposal number m, also referred to as a ballot number, higher than any known proposal number and sends a *Phase1a* request with m, to at least a majority of storage nodes responsible for r. The proposal numbers must be unique for each master because they are used to determine the latest request. To ensure uniqueness the requestor's ip-address is concatenated. If a storage node receives a *Phase1a* request greater than any proposal number it has already responded to, it responds with a *Phase1b* message containing m, the highest-numbered update (if any) including its proposal number n, and promises not to accept any future requests less than or equal to m. If P receives responses containing its proposal number m from a majority Q_C of storage nodes, it has been chosen as a master. Now, only P will be able to store a value among the storage nodes for proposal number m.

Phase 2 tries to write a value. P sends an accept request *Phase2a* to all the storage nodes of Phase 1 with the proposal number m and value v. v is either the update of the highest-numbered proposal among the *Phase1b* responses, or the requested update from the client if no *Phase1b* responses contained a value. P must re-send a previously accepted update to avoid losing the saved value. If a storage node receives a *Phase2a* request for a proposal numbered m, it accepts the proposal, unless it has already responded to a *Phase1a* request having a number greater than m, and sends a *Phase2b* message containing m and the value back to P. If the master receives a *Phase2b* message from the majority Q_C of storage nodes for the same proposal number, consensus is reached and the value is considered learned by the master. Reaching consensus means that no master will not be able to save a different value for that Paxos instance. Afterwards, the master informs all other components, app-servers and responsible storage nodes, about the success of the update. It is possible to avoid this delay by sending *Phase2b* messages directly to all involved nodes. This significantly increases the number of messages, so MDCC currently does not use this optimization.

Note, that Classic Paxos is only able to learn a single value per single instance, which may consist of multiple ballots. Thus MDCC uses one separate Paxos instance per version of a record, with the requirement that the previous version has already reached consensus successfully.

3.3.1.2 Multi-Paxos

The Classic Paxos algorithm requires two message rounds to agree on a value, one in Phase 1 and one in Phase 2. If the master is reasonably stable, using Multi-Paxos (multi-decree Synod protocol) makes it possible to avoid Phase 1 by reserving the mastership for several instances [54]. Multi-Paxos is an optimization for Classic Paxos, and in practice, Multi-Paxos is implemented instead of Classic Paxos, to take advantage of fewer message rounds.

MDCC explores this by allowing the proposers to suggest the following meta-data *[StartInstance, EndInstance, Ballot]*. Thus, the storage nodes can vote on the mastership for all instances from *StartInstance* to *EndInstance* with a single ballot number at once. The meta-data also allows for different masters for different instances. This supports custom master policies like round-robin, where *serverA* is the master for instance 1, *serverB* is the master for instance 2, and so on. Storage nodes react to these requests by applying the same semantics for each individual instance as defined in *Phase1b*, but they answer with a single message. The database system stores this meta-data including the current version number as part of the record, which enables a separate Paxos instance per record. To support meta-data for inserts, each table stores a default meta-data value for any non-existent records.

Therefore, the default configuration assigns a single master per table to coordinate inserts of new records. Although a potential bottleneck, the master is normally not in the critical path and can be bypassed, as explained in Section 3.3.3.

3.3.2 Transaction Support

The first contribution of MDCC is the extension of Multi-Paxos to support multi-record transactions with read-committed isolation and without the lost-update problem. MDCC ensures atomic durability (all or none of the updates will persist), prevents all write-write conflicts (if two transactions try to update the same record concurrently at most one will succeed), and guarantees that only updates from committed transactions are visible. Guaranteeing higher read consistencies, such as atomic visibility and snapshot isolation, is an orthogonal issue and discussed in Section 3.4.

MDCC guarantees this consistency level by using a Paxos instance per record to accept an *option* to execute the update, instead of writing the value directly. After the app-server learns the options for all the records in a transaction, it commits the transaction and asynchronously notifies the storage nodes to execute the options. If an option is not yet executed, it is called an *outstanding option*.

3.3.2.1 The Protocol

As in all optimistic concurrency control techniques, MDCC assumes that transactions collect a write-set of records at the end of the transaction, which the protocol then tries to commit. Updates to records create new versions, and are represented in the form $v_{read} \rightarrow v_{write}$, where v_{read} is the version of the record read by the transaction and v_{write} is the new version of the

record. This allows MDCC to detect write-write conflicts by comparing the current version of a record with v_{read}. If they are not equal, the record was modified between the read and write and a write-write conflict was encountered. For inserts, the update has a missing v_{read}, indicating that an insert should only succeed if the record doesn't already exist. Deletes work by marking the item as deleted and are handled as normal updates. MDCC only allows one outstanding option per record and requires that the update is not visible until the option is executed.

The app-server coordinates the transaction by attempting to store the options (reach consensus among the storage nodes) for all the updates in the transaction. It proposes the options to the Paxos instances running for each record, with the participants being the replicas of the record. Every storage node responds to the app-server with an accept or reject of the option, depending on if v_{read} is valid, similar to validated Byzantine agreement [21]. Hence, the storage nodes make an active decision to *accept* or *reject* the option. This is fundamentally different than existing uses of Paxos (e.g., Consensus on Transaction Commit [39] or Megastore), which sends a fixed value (e.g., the "final" accept or commit decision) and only considers the ballot number to decide if the value should be accepted. The reason why this change does not violate the Paxos assumptions is because at the end of Section 3.3.1.1 a new record version can only be chosen if the previous version was successfully determined. Thus, all storage nodes will always make the same abort or commit decision. This scheme serializes updates to records, but this is not as limiting as serializing updates to an entire partition as in Megastore. I describe how to relax this requirement in Section 3.3.4.

Just as in 2PC, the app-server commits a transaction when it learns all options as accepted, and aborts a transaction when it learns any option as rejected. The app-server learns an option if and only if a majority of storage nodes agrees on the option. In contrast to 2PC, MDCC makes another important change. MDCC does not allow clients or app-servers to abort a transaction once it has been proposed. Decisions are determined and stored by the distributed storage nodes within MDCC, instead of being decided by a single coordinator with 2PC. This ensures that the commit status of a transaction depends only on the status of the learned options and hence is always deterministic even with failures. Otherwise, the decision of the app-server/client after the prepare has to be reliably stored, which either influences the availability (the reason why 2PC is blocking) or requires an additional round as done by three-phase commit or Consensus on Transaction Commit [39].

If the app-server determines that the transaction is aborted or committed, it informs involved storage nodes through a *Learned* message about the decision. The storage nodes in turn execute the option (make visible) or mark it as rejected. Learning an option is the result of each Paxos instance and thus generates new version of the record, whether the option is learned as accepted or rejected. Note, that so far only one option per record can be outstanding at a time as MDCC requires the previous instance (version) to be decided.

As a result, it is possible to commit the transaction (commit or abort) in a single round-trip across the data centers if the masters of all the records in the transaction are in the local data center. This is possible because the commit/abort decision of a transaction depends entirely on the learned values and the application server is not allowed to prematurely abort

a transaction (in contrast to 2PC or Consensus on Transaction Commit). The *Learned* message to notify the storage nodes about the commit/abort can be asynchronous, but does not influence the correctness, and only affects the possibility of aborts caused by stale reads. By adding transaction support, this design is able to achieve 1 round-trip commits if the master is local, but when the master is not local, MDCC requires 2 round-trips, due to the additional communication with the remote master. Communication with a local master is ignored because the latency is negligible (few milliseconds) compared to geographically remote master communication (hundreds of milliseconds).

3.3.2.2 Avoiding Deadlocks

The described protocol is able to atomically commit multi-record transactions. Without further effort, concurrent transactions may cause deadlocks by waiting on options of other transactions. For example, if two transactions t_1 and t_2 try to learn an option for the same two records r_1 and r_2, t_1 might successfully learn the option for r_1, and t_2 for r_2. Since transactions do not abort without learning at least one of the options as aborted, both transactions are now deadlocked because each transaction waits for the other to finish. MDCC applies a simple pessimistic strategy to avoid deadlocks. The core idea is to relax the requirement that MDCC can only learn a new version if the previous instance is committed. For example, if t_1 learns the option $v_0 \rightarrow v_1$ for record r_1 in one instance as accepted, and t_2 tries to acquire an option $v_0 \rightarrow v_2$ for r_1, t_1 learns the option $v_0 \rightarrow v_1$ as accepted and t_2 learns the option $v_0 \rightarrow v_2$ as rejected in the next Paxos instance. This simple technique causes transaction t_1 to commit and t_2 to abort or in the case of the deadlock as described before, both transactions to abort. The Paxos safety property is still maintained because all storage nodes will make the same decision based on the policy, and the master totally orders the record versions.

3.3.2.3 Failure Scenarios

Multi-Paxos allows the MDCC transaction commit protocol to recover from various failures. For example, a failure of a storage node can be masked by the use of quorums. A master failure can be recovered from by selecting a new master (after some timeout) and triggering Phase 1 and 2 as described previously. Handling app-server failures is trickier, because an app-server failure can cause a transaction to be pending forever as a "dangling transaction". MDCC avoids dangling transactions by including in all of its options a unique transaction-id (e.g., UUIDs) as well as all primary keys of the write-set, and by additionally keeping a log of all learned options at the storage node. Therefore, every option includes all necessary information to reconstruct the state of the corresponding transactions. Whenever an app-server failure is detected by simple timeouts, the state is reconstructed by reading from a quorum of storage nodes for every key in the transaction, so any node can recover the transaction. A quorum is required to determine what was decided by the Paxos instance. Finally, a data center failure is treated simply as each of the nodes in the data center failing.

Although not implemented, MDCC can adapt bulk-copy techniques to bring the data up-to-date more efficiently without involving the Paxos protocol (also see [13]).

3.3.3 Transactions Bypassing the Master

The previous section showed how MDCC achieves transactions with multiple updates in one single round-trip, if the masters for all transaction records are in the same data center as the app-server. However, two round-trips are required when the masters are remote, or mastership needs to be acquired.

3.3.3.1 Protocol

Fast Paxos [51] avoids the master by distinguishing between *c*lassic and *f*ast ballots. Classic ballots operate like the classic Paxos algorithm described above and are always the fall-back option. Fast ballots normally use a bigger quorum than classic ballots, but allow bypassing the master. This saves one message round to the master, which may be in a different data center. However, since updates are not serialized by the master, collisions may occur, which can only be resolved by a master using classic ballots.

I use this approach of fast ballots for MDCC. All versions start as an implicitly *fast* ballot number, unless a master changed the ballot number through a *Phase1a* message. This default ballot number informs the storage nodes to accept the next options from any proposer.

Afterwards, any app-server can propose an option directly to the storage nodes without going through a master, which in turn promise only to accept the first proposed option. Simple majority quorums, however, are no longer sufficient to learn a value and ensure safeness of the protocol. Instead, learning an option without the master requires a *fast* quorum [51]. Fast and classic quorums, are defined by the following requirements: (i) any two quorums must have a non-empty intersection, and (ii) there is a non-empty intersection of any three quorums consisting of two fast quorums Q_F^1 and Q_F^2 and a classic quorum Q_C. A typical setting for a replication factor of 5 (N) is a classic quorum size of 3 ($\lfloor \frac{N}{2} \rfloor + 1$) and a fast quorum size of 4 ($\lceil \frac{3N}{4} \rceil$). If a proposer receives an acknowledgment from a fast quorum, the value is safe and guaranteed to be committed. However, if a fast quorum cannot be achieved, collision recovery is necessary. Note, that a Paxos collision is different from a transaction conflict; collisions occur when nodes cannot agree on an option, conflicts are caused by conflicting updates to the same record.

To resolve the collision, a new classic ballot must be started with *Phase 1*. After receiving responses from a classic quorum, all potential intersections with a fast quorum must be computed from the responses. If the intersection consists of all the members having the highest ballot number, and all agree with some option v, then v must be proposed next. Otherwise, no option was previously agreed upon, so any new option can be proposed. For example, assume the following messages were received as part of a collision resolution from 4 out of 5 servers with the previously mentioned quorums (notation: *(server-id, ballot number,*

update)): (1,3,$v_0 \rightarrow v_1$), (2,4,$v_1 \rightarrow v_2$), (3,4,$v_1 \rightarrow v_3$), (5,4, $v_1 \rightarrow v_2$). Here, the intersection size is 2 and the highest ballot number is 4, so the protocol compares the following intersections:

$$[(2, 4, v_1 \rightarrow v_2), (3, 4, v_1 \rightarrow v_3)]$$
$$[(3, 4, v_1 \rightarrow v_3), (5, 4, v_1 \rightarrow v_2)]$$
$$[(2, 4, v_1 \rightarrow v_2), (5, 4, v_1 \rightarrow v_2)]$$

Only the last intersection has an option in common and all other intersections are empty. Hence, the option $v_1 \rightarrow v_2$ has to be proposed next. More details and the correctness proofs of Fast Paxos can be found in [51].

MDCC uses Fast Paxos to bypass the master for accepting an option, which reduces the number of required message rounds. Only one option can be learned per fast ballot. However, by combining the idea of Fast Paxos with Multi-Paxos and using the following adjusted ballot-range definitions from Section 3.3.1.2, *[StartInstance, EndInstance, Fast, Ballot]*, it is possible to reserve several instances as fast rounds. Whenever a collision is detected, the instance is changed to classic, the collision is resolved and the protocol moves on to the next instance, which can start as either classic or fast. It is important that classic ballot numbers are always higher ranked than fast ballot numbers to resolve collisions and save the correct value. Combined with the earlier observation that a new Paxos instance is started only if the previous instance is stable and learned, this allows the protocol to execute several consecutive fast instances without involving a master.

Without the option concept of Section 3.3.2 fast ballots would be impractical to use. Without options it would be impossible to make an abort/commit decision without requiring a lock first in a separate message round on the storage servers or some master (e.g., as done by Spanner). This is also the main reason why other existing Paxos commit protocols cannot leverage fast ballots. Using fast Paxos with options and the deadlock avoidance policy still produces a total order of operations, but with only one message round.

3.3.3.2 Fast-Policy

There exists a non-trivial trade-off between fast and classic instances. With fast instances, two concurrent updates might cause a collision requiring another two message rounds for the resolution, whereas classic instances usually require two message rounds, one to either contact the master or acquire the mastership, and one for Phase 2. Hence, fast instances should only be used if conflicts and collisions are rare.

Currently, MDCC uses a very simple strategy. The default meta-data for all instances and all records are pre-defined to be fast rounds with *[0,∞,fast=true,ballot=0]*. As the default meta-data for all records is the same, it does not need to be stored per record. A record's meta-data is managed separately, only when collision resolution is triggered. If MDCC detects a collision, it sets the next γ instances (default is 100) to classic. After γ transactions, fast instances are automatically tried again. This simple strategy stays in fast instances if

possible and in classic instances when necessary, while probing to go back to fast Paxos every γ instances.

3.3.4 Commutative Updates

The design based on Fast Paxos allows many transactions to commit with a single round-trip between data centers. However, whenever there are concurrent updates to a given data item, conflicts will arise and extra messages are needed to resolve the conflict. MDCC efficiently exploits cases where the updates are commutative, to avoid extra messages for conflict resolution, by using Generalized Paxos [52], which is an extension of Fast Paxos. In this section, I show how the novel option concept and the idea to use Paxos on a record instead of a database log-level as described in the previous sections enable the use of Generalized Paxos. Furthermore, in order to support the common case of operations on data that are subject to value constraints (e.g. value should be greater than 0), I present a new demarcation technique for quorums.

3.3.4.1 The Protocol

Generalized Paxos [52] uses the same ideas as Fast Paxos but relaxes the constraint that every acceptor must agree on the same exact sequence of values. Since some updates may commute with each other, the acceptors only need to agree on sets of commands which are compatible with each other. MDCC utilizes the notion of compatibility to support commutative updates.

Fast commutative ballots are always started by a message from the master. The master sets the record *base value*, which is the latest committed value. Afterwards, any client can propose commutative updates to all storage nodes directly using options, as described previously. In contrast to the previous section, an option now contains commutative updates, which consist of one or more attributes and their respective delta changes (e.g., $decrement(stock, 1)$). If a fast quorum Q_F out of N storage nodes accepts the option, the update is committed. When the updates involved are commutative, the acceptors can accept multiple proposals in the same ballot and the orderings do not have to be identical on all storage nodes. This allows MDCC to stay in the fast ballot for longer periods of time, bypassing the master and allowing the commit to happen in one message round. More details on Generalized Paxos are given in [52].

3.3.4.2 Global Constraints

Generalized Paxos is based on commutative operations like increment and decrement. However, many database applications must enforce integrity constraints, for example the stock of an item must be greater than zero. Under a constraint like this, decrements do not always commute. However, in this example, if database contains ample stock, updates can commute. Thus MDCC allows concurrent processing of decrements and increments while

ensuring domain integrity constraints, by requiring storage nodes to only accept an option if the option would not violate the constraint under all permutations of commit/abort outcomes for outstanding options. For example, given 5 transactions $t_{1 \ldots 5}$ (arriving in order), each generating an option $[stock = stock - 1]$ with the constraint that $stock \geq 0$ and current *stock* level of 4, a storage node s will reject t_5 even though the first four options may abort. This definition is analogous to Escrow [63] and guarantees correctness even in the presence of aborts and failures.

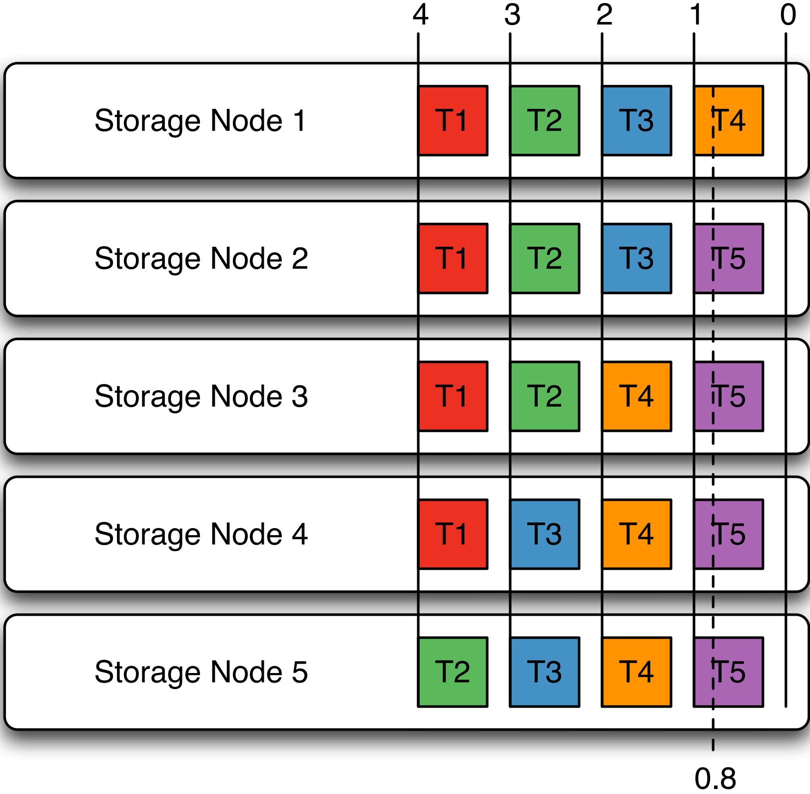


Figure 3.2: Possible message order in MDCC

Unfortunately, this still does not guarantee integrity constraints, as storage nodes base decisions on local, but not global, knowledge. Figure 3.2 shows a possible message ordering for the above example with five storage nodes. Here, clients wait for Q_F responses (4), and each storage node makes a decision based on its local state. There is an integrity constraint that the value should be greater than or equal to zero ($value \geq 0$), and the value is currently at 4. This means, at most 4 transactions is allowed to commit in order to satisfy the constraint. However, through different message arrival orders, it is possible for all 5 transactions to commit, even though committing them all violates the constraint. This occurs because each storage node only makes decisions based on its local state.

Therefore, I present a new *demarcation* based strategy for quorum based protocols. This new demarcation technique is similar to existing techniques [14] in that it uses local limits, but is used in different scenarios. Original demarcation uses limits to safely update distributed values, where MDCC uses limits for quorum replicated values.

Without loss of generality, assume a constraint of $value \geq 0$, and that all updates are decrements. Let N be the replication factor (number storage nodes), X be the base value for some attribute and δ_i be the decrement amount of transaction t_i for the attribute. Consider every replicated base value X as a *resource*, so the total number of *resources* in the system is $N \cdot X$. In order to commit an update, Q_F storage nodes must accept the update, so every successful transaction t_i reduces the resources in the system by at least $Q_F \cdot \delta_i$. If there are m successful transactions where $\sum_{i=1}^{m} \delta_i = X$, this means the attribute value reached 0, and the total amount of resources would reduce by at least $Q_F \cdot \sum_{i=1}^{m} \delta_i = Q_F \cdot X$. Even though the integrity constraint forbids any more transactions, it is still possible that the system still has $(N - Q_F) \cdot X$ resources remaining due to failures, lost, or out-of-order messages.

The worst case is where the remaining resources are equally distributed across all the storage nodes, otherwise, at least one of the storage nodes would start to reject options earlier. The remaining resources $(N - Q_F) \cdot X$ are divided evenly among the N storage nodes to derive a lower limit to guarantee the value constraint. Each storage node must reject an option if it would cause the value to fall below:

$$L = \frac{N - Q_F}{N} \cdot X$$

This limit L is calculated with every new base value of an attribute. When options in fast ballots are rejected because of this limit, the protocol handles it as a collision, resolves it by switching to classic ballots, and writes a new base value and limit L. In the example in Figure 3.2, the quorum demarcation lower limit would be:

$$\begin{aligned} L &= \frac{N - Q_F}{N} \cdot X \\ &= \frac{5 - 4}{5} \cdot 4 \\ &= 0.8 \end{aligned}$$

Since the lower limit is 0.8, the storage nodes would have rejected options from $T4$ and $T5$, causing those two transactions to abort. Since only $T1$, $T2$, and $T3$ commit, the integrity constraint is not violated.

3.3.4.3 MDCC Pseudocode

The complete MDCC protocol as pseudocode is listed in algorithms 1, 2, 3, 4, and 5, while table 3.1 defines the symbols and variables used in the pseudocode. The remainder of this

Symbols	Definitions
a	an acceptor
l	a leader
up	an update
$\omega(up, _)$	an option for an update, with ✓or ✗
✓/ ✗	acceptance / rejection
m	ballot number
$val_a[i]$	cstruct at ballot i at acceptor a
bal_a	$\max\{k \mid val_a[k] \neq none\}$
val_a	cstruct at bal_a at acceptor a
$mbal_a$	current ballot number at acceptor a
$ldrBal_l$	ballot number at leader l
$maxTried_l$	cstructs proposed by leader l
Q	a quorum of acceptors
$Quorum(k)$	all possible quorums for ballot k
$learned$	cstruct learned by a learner
$\sqcap$	greatest lower bound operator
$\sqcup$	least upper bound operator
$\sqsubseteq$	partial order operator for cstructs
$val\bullet\ \omega(up, _)$	appends option $\omega(up, _)$ to cstruct val

Table 3.1: Definitions of symbols in MDCC pseudocode

section sketches the algorithm by focusing on how the different pieces from the previous sections work together.

The app-server or client starts the transaction by sending proposals for every update on line 3. After learning the status of options of all the updates (lines 19-36), the app-server sends visibility messages to "execute" the options on lines 6-9, as described in Section 3.3.2.1. While attempting to learn options, if the app-server does not learn the status of an option (line 24), it will initiate a recovery. Also, if the app-server learns a commutative option as rejected during a fast ballot (line 32), it will notify the master to start recovery. Learning a rejected option for commutative updates during fast ballots is an indicator of violating the quorum demarcation limit, so a classic ballot is required to update the base value and limit.

When accepting new options, the storage nodes must evaluate the compatibility of the options and then accept or reject it. The compatibility validation is shown in lines 109-129. If the new update is not commutative, the storage node compares the read version of the update to the current value to determine the compatibility, as shown in lines 112-119. For new commutative updates, the storage node computes the quorum demarcation limits as described in Section 3.3.4.2, and determines if any combination of the pending commutative options violate the limits (lines 120-127). When a storage node receives a visibility message for an option, it executes the option in order to make the update visible, on line 133.

Algorithm 1 Pseudocode for MDCC

```
1: procedure TransactionStart                                   ▷ Client
2:     for all up ∈ tx do
3:         run SendProposal(up)
4:     end for
5:     wait to learn all update options
6:     if ∀up ∈ tx : learned ω(up,✓) then
7:         send Visibility[up,✓] to Acceptors
8:     else
9:         send Visibility[up,✗] to Acceptors
10:    end if
11: end procedure
12: procedure SendProposal(up)                                  ▷ Proposer
13:    if classic ballot then
14:        send Propose[up] to Leader
15:    else
16:        send Propose[up] to Acceptors
17:    end if
18: end procedure
19: procedure Learn(up)                                         ▷ Learner
20:    collect Phase2b[m, val_a] messages from Q
21:    if ∀a ∈ Q : v ⊑ val_a then
22:        learned ← learned ⊔ v
23:    end if
24:    if ω(up, _) ∉ learned then
25:        send StartRecovery[] to Leader
26:        return
27:    end if
28:    if classic ballot then
29:        move on to next instance
30:    else
31:        isComm ← up is CommutativeUpdate[delta]
32:        if ω(up,✗) ∈ learned ∧ isComm then
33:            send StartRecovery[] to Leader
34:        end if
35:    end if
36: end procedure
```

3.4 Consistency Guarantees

MDCC ensures atomicity (either all updates in a transaction persist or none) and ensures that two concurrent write-conflicting update transactions do not both commit. This sections describes the consistency guarantees of MDCC.

Algorithm 2 Pseudocode for MDCC - Leader l (part 1 of 2)

```
37: procedure ReceiveLeaderMessage(msg)
38:     switch msg do
39:         case Propose[up] :
40:             run Phase2aClassic(up)
41:         case Phase1b[m, bal, val] :
42:             if received messages from Q then
43:                 run Phase2Start(m, Q)
44:             end if
45:         case StartRecovery[] :
46:             m ← new unique ballot number greater than m
47:             run Phase1a(m)
48: end procedure
```

3.4.1 Read Committed without Lost Updates

The default consistency level of MDCC is *read committed*, but without the lost update problem [15]. Read committed isolation prevents dirty reads, so no transactions will read any other transaction's uncommitted changes. The lost update problem occurs when transaction t_1 first reads a data item X, then one or more other transactions write to the same data item X, and finally t_1 writes to data item X. The updates between the read and write of item X by t_1 are "lost" because the write by t_1 overwrites the value and loses the previous updates. MDCC guarantees read committed isolation by only reading committed values and not returning the value of uncommitted options. Lost updates are prevented by detecting every write-write conflict between transactions.

Currently, Microsoft SQL Server, Oracle Database, IBM DB2 and PostgreSQL all use read committed isolation by default. Therefore, the default consistency level for MDCC is sufficient for a wide range of applications.

3.4.2 Staleness & Monotonicity

Reads can be done from any storage node and are guaranteed to return only committed data. However, by just reading from a single node, the read might be stale. For example, if a storage node missed updates due to a network problem, reads might return older data. Reading the latest value requires reading a majority of storage nodes to determine the latest stable version, making it an expensive operation.

In order to allow up-to-date reads with classic rounds, MDCC can leverage techniques from Megastore [13]. A simple strategy for up-to-date reads with fast rounds is to ensure that a special pseudo-master storage node is always part of the quorum of Phases 1 and 2 and to switch to classic whenever the pseudo-master cannot be contacted. The techniques from Megastore can apply for the pseudo-master to guarantee up-to-date reads in all data centers. A similar strategy can guarantee monotonic reads such as *repeatable reads* or *read your writes*, by requiring the local storage node to always participate in the quorum.

Algorithm 3 Pseudocode for MDCC - Leader l (part 2 of 2)

49: **procedure** PHASE1A(m)
50:     **if** $m > ldrBal_l$ **then**
51:         $ldrBal_l \leftarrow m$
52:         $maxTried_l \leftarrow none$
53:         **send** $Phase1a[m]$ to $Acceptors$
54:     **end if**
55: **end procedure**
56: **procedure** PHASE2START(m, Q)
57:     $maxTried_l \leftarrow$ PROVEDSAFE(Q, m)
58:     **if** new update to propose exists **then**
59:         **run** PHASE2ACLASSIC(up)
60:     **end if**
61: **end procedure**
62: **procedure** PHASE2ACLASSIC(up)
63:     $maxTried_l \leftarrow maxTried_l \bullet \ \omega(up, _)$
64:     **send** $Phase2a[ldrBal_l, maxTried_l]$ to $Acceptors$
65: **end procedure**
66: **procedure** PROVEDSAFE(Q, m)
67:     $k \equiv \max\{i \mid (i < m) \land (\exists a \in Q : val_a[i] \neq none)\}$
68:     $\mathcal{R} \equiv \{R \in Quorum(k) \mid$
            $\forall a \in Q \cap R : val_a[k] \neq none\}$
69:     $\gamma(R) \equiv \sqcap\{val_a[k] \mid a \in Q \cap R\}$, for all $R \in \mathcal{R}$
70:     $\Gamma \equiv \{\gamma(R) \mid R \in \mathcal{R}\}$
71:     **if** $R = \emptyset$ **then**
72:         **return** $\{val_a[k] \mid (a \in Q) \land (val_a]k \ \neq none)\}$
73:     **else**
74:         **return** $\{\sqcup\Gamma\}$
75:     **end if**
76: **end procedure**

3.4.3 Atomic Visibility

MDCC provides atomic durability, meaning either all or none of the operations of the transaction are durable, but it does not support atomic visibility. The visibility point follows after the commit/durability point, when the asynchronous visibility message is sent to the acceptors. Therefore, some of the updates of a committed transaction might be visible whereas other are not. For example, if transaction *t˙1* inserts new data items A and B, and them commits, both inserts will be persistent in the system. However, it is possible that a subsequent transaction *t˙2* reads only one of A or B. Two-phase commit also only provides atomic durability, not visibility unless it is combined with other techniques such as two-phase locking or snapshot isolation. The same is true for MDCC. For example, MDCC could use a read/write locking service per data center or snapshot isolation as done in Spanner [31] to achieve atomic visibility. Another technique for atomic visibility without locking can be achieved by including the full write-set of the transaction with every update. It is then possible to use standard optimistic concurrency control techniques to detect if a com-

Algorithm 4 Pseudocode for MDCC - Acceptor a (part 1 of 2)

```
77: procedure ReceiveAcceptorMessage(msg)
78:     switch msg do
79:         case Phase1a[m] :
80:             run Phase1b(m)
81:         case Phase2a[m, v] :
82:             run Phase2bClassic(m, v)
83:         case Propose[up] :
84:             run Phase2bFast(up)
85:         case Visibility[up, status] :
86:             run ApplyVisibility(up, status)
87: end procedure
88: procedure Phase1b(m)
89:     if mbal_a < m then
90:         mbal_a ← m
91:         send Phase1b[m, bal_a, val_a] to Leader
92:     end if
93: end procedure
94: procedure Phase2bClassic(m, v)
95:     if bal_a ≤ m then
96:         bal_a ← m
97:         val_a ← v
98:         SetCompatible(val_a)
99:         send Phase2b[m, val_a] to Learners
100:     end if
101: end procedure
102: procedure Phase2bFast(up)
103:     if bal_a = mbal_a then
104:         val_a ← val_a • ω(up, _)
105:         SetCompatible(val_a)
106:         send Phase2b[m, val_a] to Learners
107:     end if
108: end procedure
```

mitted transaction was only partially read. During a read, the write-set is checked against previously performed reads and if a missing update is detected, the missing values may be retrieved, or the transaction may abort. Nevertheless, this strategy for atomic visibility can be expensive as it implies storing all write-sets with every update on all records.

3.4.4 Other Isolation Levels

Finally, MDCC can support higher levels of isolation. In particular, Non-monotonic Snapshot Isolation (NMSI) [70] or Spanner's [31] snapshot isolation through synchronized clocks are natural fits for MDCC. Both would still allow fast commits while providing consistent snapshots. Furthermore, as MDCC already checks the write-set for transactions, the protocol could be extended to also consider read-sets, allowing MDCC to leverage optimistic

Algorithm 5 Pseudocode for MDCC - Acceptor a (part 2 of 2)

```
109: procedure SetCompatible(v)
110:     for all new options ω(up, _) in v do
111:         switch up do
112:             case PhysicalUpdate[v_read, v_write] :
113:                 validRead ← v_read matches current value
114:                 validSingle ← no other pending options exist
115:                 if validRead ∧ validSingle then
116:                     set option to ω(up, ✓)
117:                 else
118:                     set option to ω(up, ✗)
119:                 end if
120:             case CommutativeUpdate[delta] :
121:                 𝒰 ← upper quorum demarcation limit
122:                 ℒ ← lower quorum demarcation limit
123:                 if any option combinations violate 𝒰 or ℒ then
124:                     set option to ω(up, ✗)
125:                 else
126:                     set option to ω(up, ✓)
127:                 end if
128:     end for
129: end procedure
130: procedure ApplyVisibility(up, status)
131:     update ω(up, _) in val_a to ω(up, status)
132:     if status = ✓ then
133:         apply up to make update visible
134:     end if
135: end procedure
```

concurrency control techniques and ultimately provide full serializability, at the expense of more transaction aborts due to conflicts, and more network messages.

3.5 Evaluation

I implemented a prototype of MDCC on top of a distributed key/value store across five different data centers using the Amazon EC2 cloud. To demonstrate the benefits of MDCC, I use the TPC-W and micro-benchmarks to compare the performance characteristics of MDCC to other transactional and other non-transactional, eventually consistent protocols. This section describes the benchmarks, experimental setup, and my findings.

3.5.1 Experimental Setup

I implemented the MDCC protocol in Scala, on top of a distributed key/value store, which used Oracle BDB Java Edition as a persistent storage engine. I deployed the system across five geographically diverse data centers on Amazon EC2: US West (N. California), US East

(Virginia), EU (Ireland), Asia Pacific (Singapore), and Asia Pacific (Tokyo). Each data center has a full replica of the data, and within a data center, each table is range partitioned by key, and distributed across several storage nodes as m1.large instances (4 cores, 7.5GB memory). Therefore, every horizontal partition, or shard, of the data is replicated five times, with one copy in each data center. Unless noted otherwise, all clients issuing transactions are evenly distributed across all five data centers, on separate m1.large instances.

3.5.2 Comparison with other Protocols

To compare the overall performance of MDCC with alternative designs, I used TPC-W, a transactional benchmark that simulates the workload experienced by an e-commerce web server. TPC-W defines a total of 14 web interactions (WI), each of which are web page requests that issue several database queries. In TPC-W, the only transaction which is able to benefit from commutative operations is the product-buy request, which decreases the stock for each item in the shopping cart while ensuring that the stock never drops below 0 (otherwise, the transaction should abort). I implemented all the web interactions using a SQL-like language but forego the HTML rendering part of the benchmark to focus on the database interactions. TPC-W defines that these WI are requested by emulated browsers, or clients, with a wait-time between requests and varying browse-to-buy ratios. In my experiments, I forego the wait-time between requests and only use the most write-heavy profile to stress the system. It should also be noted that read-committed is sufficient for TPC-W to never violate its data consistency. MDCC is a commit protocol, so the main metrics collected are the write transaction response times and throughput.

In these experiments, the MDCC prototype uses fast ballots with commutativity where possible (reverting to classic after too many collisions have occurred as described in Section 3.3.3.2). For comparison, I also implemented other forms of protocols protocols in Scala, using the same distributed data store, and accessed by the same clients. Those other protocols are described below:

Quorum Writes (QW) The quorum writes protocol (QW) is the standard for most eventually consistent systems and is implemented by simply sending all updates to all involved storage nodes then waiting for responses from a *quorum* of nodes. Typically, a quorum must contain a majority of the storage nodes, so that any two quorums will have at least one overlapping storage node. I used two different configurations for the write quorum: quorum of size 3 out of 5 replicas for each record (QW-3), and quorum of size 4 out of 5 (QW-4). I used a read-quorum of 1 to access only the local replica (this is the fastest read configuration). It is important to note that the quorum writes protocol provides no isolation, atomicity, or transactional guarantees.

Two-Phase Commit (2PC) Two-phase commit (2PC) is still considered the standard protocol for distributed transactions. 2PC operates in two phases. In the first phase, a transaction manager tries to prepare all involved storage nodes to commit the updates.

If all relevant nodes prepare successfully, then in the second phase the transaction manager sends a commit to all storage nodes involved; otherwise it sends an abort. Note, that 2PC requires all involved storage nodes to respond and is not resilient to single node failures.

Megastore* Megastore [13] is a strongly consistent, globally distributed database, using Paxos to fully serialize transactions within an entity group (a partition of data). I was not able to compare MDCC directly against the Megastore system because it is not publicly available. Google App Engine uses Megastore, but the data centers and configuration are unknown and out of my control. Instead, I simulated the underlying protocol to compare it with MDCC; I do this as a special configuration of the system, referred to as Megastore*. With Megastore, the protocol is described mainly for transactions within a partition. The authors state that 2PC is used across partitions with looser consistency semantics but omits details on the implementation and the authors discourage of using the feature because of its high latency. Megastore's primary use case is not for a general purpose transactional workload (such as TPC-W), but for statically partitioned, independent entity groups. Therefore, for experiments with Megastore*, all the data was placed into a single entity group to avoid transactions which span multiple entity groups. Furthermore, Megastore only allows that one write transaction is executed at any time (all other competing transactions will abort). As this results in unusable throughput for TPC-W, I included an improvement from Paxos-CP [65] and allowed non-conflicting transactions to commit using a subsequent Paxos instance. I also relaxed the read consistency to read-committed enabling a fair comparison between Megastore* and MDCC. Finally, Megastore* was given an advantage by placing all clients and masters local to each other in one data center (US-West), to allow all transactions to commit with a single round-trip.

3.5.2.1 TPC-W Write Response Times

To evaluate the main goal of MDCC to reduce the latency, the TPC-W workload was run with each protocol. A TPC-W scale factor of 10,000 items was used, with the data being evenly ranged partitioned to four storage nodes per data center. 100 evenly geo-distributed clients (on separate machines) each ran the TPC-W benchmark for 2 minutes, after a 1 minute warm-up period.

Figure 3.3 shows the cumulative distribution functions (CDF) of the response times of committed write transactions for the different protocols. Note that the horizontal (time) axis is a log-scale. Only the response times for write transactions are reported, as read transactions were always local for all configurations and protocols. The two dashed lines (QW-3, QW-4) are non-transactional, eventually consistent protocols, and the three solid lines (MDCC, 2PC, Megastore*) are transactional, strongly consistent protocols.

Figure 3.3 shows that the non-transactional protocol QW-3 has the fastest response times, followed by QW-4, then the transactional systems, of which MDCC is fastest, then 2PC, and

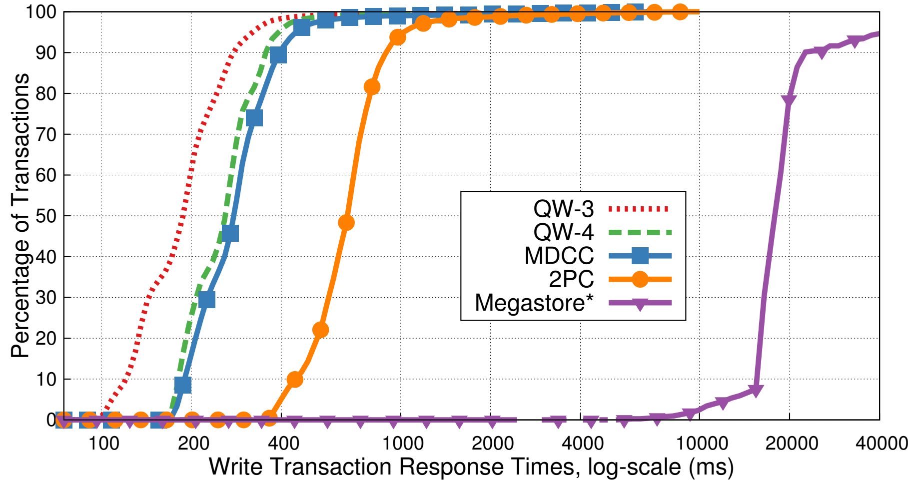


Figure 3.3: TPC-W write transaction response times CDF

finally Megastore* with slowest times. The median response times are: 188ms for QW-3, 260ms for QW-4, 278ms for MDCC, 668ms for 2PC, and 17,810ms for Megastore*.

Since MDCC uses fast ballots whenever possible, MDCC often commits transactions from any data center with a single round-trip to a quorum size of 4. This explains why the performance of MDCC is similar to QW-4. The difference between QW-3 and QW-4 arises from the need to wait longer for the 4th response, instead of returning after the 3rd response. There is an impact from the non-uniform latencies between different data centers, so the 4th is on average farther away than the 3rd, and there is more variance when waiting for more responses. Hence, an administrator might choose to configure a MDCC-like system to use classic instances with a local master, if it is known that the workload has most requests being issued from the same data center (see Section 3.5.3.3 for an evaluation).

MDCC reduces per transaction latencies by 50% compared to 2PC because it commits in one round-trip instead of two. Most surprisingly, however, is the orders of magnitude improvement over Megastore*. This can be explained since Megastore* must serialize all transactions with Paxos (it executes one transaction at a time) and heavy queuing effects occur. This queuing effect happens because of the moderate load, but it is possible to avoid the effect by reducing the load or allowing multiple transactions to commit for one commit log record. If so, performance would be similar to the classical Paxos configuration discussed in Section 3.5.3.1. Even without queuing effects, Megastore* would require an additional round-trip to the master for non-local transactions. Since Google's Spanner [31] uses 2PC across Paxos groups, and each Paxos group requires a master, Spanner should behave similarly to the 2PC data in figure 3.3.

I conclude from this experiment that MDCC achieves the main goal: it supports strongly consistent transactions with latencies similar to non-transactional protocols which provide weaker consistency, and is significantly faster than other strongly consistent protocols (2PC,

Megastore*).

3.5.2.2 TPC-W Throughput and Scalability

One of the intended advantages of cloud-based storage systems is the ability to scale out without affecting performance. I performed a scale-out experiment using the same setting as in the previous section, except that I varied the scale to (50 clients, 5,000 items), (100 clients, 10,000 items), and (200 clients, 20,000 items). For each configuration, the amount of data per storage node was fixed to a TPC-W scale-factor of 2,500 items and the number of nodes was scaled accordingly (keeping the ratio of clients to storage nodes constant). For the same arguments as before, a single partition was used for Megastore* to avoid cross-partition transactions.

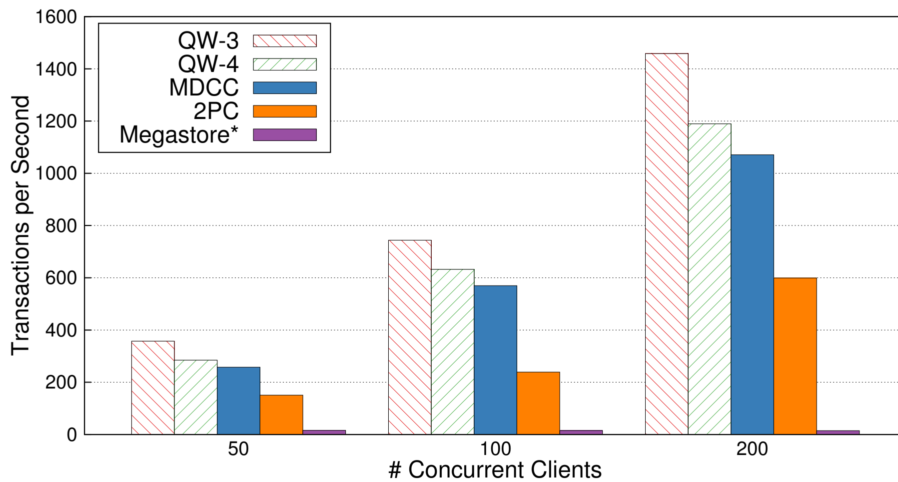


Figure 3.4: TPC-W throughput scalability

Figure 3.4 shows the results of the throughput measurements of the various protocols. The plot shows that the QW protocols have the lowest message and CPU overhead and therefore the highest throughput, with the MDCC throughput not far behind. For 200 concurrent clients, the MDCC throughput was within 10% of the throughput of QW-4. The experiments also demonstrate that MDCC has higher throughput compared to the other strongly consistent protocols, 2PC and Megastore*. The throughput for 2PC is significantly lower, mainly due to the additional waiting for the second round.

As expected, the QW protocols scale almost linearly; there is similar scaling for MDCC and 2PC. The Megastore* throughput is very low and does not scale out well, because all transactions are serialized for the single partition. This low throughput and poor scaling matches the results in [48] for Google App Engine, a public service using Megastore. In summary, Figure 3.4 shows that MDCC provides strongly consistent cross data center transactions with throughput and scalability similar to eventually consistent protocols.

3.5.3 Exploring the Design Space

I use a micro-benchmark to study the different optimizations within the MDCC protocol, and how it is sensitive to workload features. The data for the micro-benchmark is a single table of items, with randomly chosen stock values and a constraint on the stock attribute that it has to be at least 0. The benchmark defines a simple buy transaction, that chooses 3 random items uniformly, and for each item, decrements the stock value by an amount between 1 and 3 (a commutative operation). Unless stated otherwise, there are 100 geo-distributed clients, and a pre-populated product table with 10,000 items sharded on 2 storage nodes per data center.

3.5.3.1 Response Times

To study the effects of the different design choices in MDCC, the micro-benchmark is run with 2PC and various MDCC configurations. The MDCC configurations are:

- **MDCC.** the fully featured protocol
- **Fast.** the MDCC protocol without the commutative update support
- **Multi.** the MDCC protocol, but with all instances being Multi-Paxos (a stable master can skip Phase 1)

The experiment ran for 3 minutes after a 1 minute warm-up. Figure 3.5 shows the cumulative distribution functions (CDF) of response times of the successful transactions.

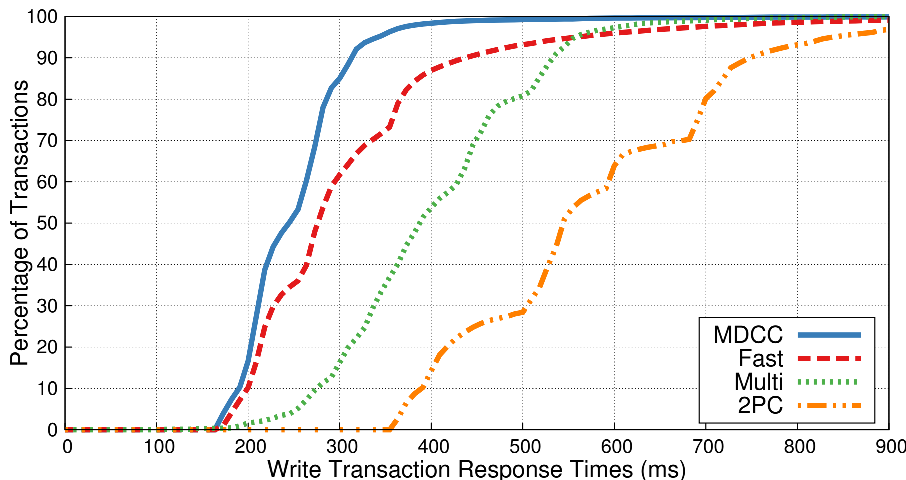


Figure 3.5: Micro-benchmark response times CDF

The median response times are: 245ms for MDCC, 276ms for Fast, 388ms for Multi, and 543ms for 2PC. 2PC is the slowest because it must use two round-trips across data centers

and has to wait for responses from all 5 data centers. For Multi, with the masters being uniformly distributed across all the data centers, most of the transactions (about 4/5 of them) require a message round-trip to contact the master, so two round-trips across data centers are required, similar to 2PC. In contrast to 2PC, Multi needs responses from only 3 of 5 data centers, so the response times are improved. The response times for Multi would also be observed for Megastore* if no queuing effects are experienced. Megastore* experiences heavier queuing effects because all transactions in the single entity group are serialized, but with Multi, only updates per record are serialized.

The main reason for the low latencies for MDCC is the use of fast ballots. Both MDCC and Fast return earlier than the other protocols, because they often require one round-trip across data centers and do not need to contact a master, like Multi. The improvement from Fast to MDCC is because commutative updates reduce conflicts and thus collisions, so MDCC can continue to use fast ballots and avoid resolving collisions as described in Section 3.3.3.2.

3.5.3.2 Varying Conflict Rates

MDCC attempts to take advantage of situations when conflicts are rare, so I explore how the MDCC commit performance is affected by different conflict rates. I defined a *hot-spot* area and modified the micro-benchmark to access items in the *hot-spot* area with 90% probability, and to access the *cold-spot* portion of the data with the remaining 10% probability. By adjusting the size of the *hot-spot* as a percentage of the data, the conflict rate in the access patterns can be altered. For example, when the *hot-spot* is 90% of the data, the access pattern is equivalent to uniformly at random, since 90% of the accesses will go to 90% of the data.

The Multi system uses masters to serialize transactions, so Paxos conflicts occur when there are multiple potential masters, which should be rare. Multi will simply abort transactions when the read version is not the same as the version in the storage node (indicating a write-write transaction conflict), to keep the data consistent, making Paxos collisions independent of transaction conflicts. On the other hand, for Fast Paxos, collisions are related to transaction conflicts, as a collision/conflict occurs whenever a quorum size of 4 does not agree on the same decision. When this happens, collision resolution must be triggered, which eventually switches to a classic ballot, which will take at least 2 more message rounds. MDCC is able to improve on it by exploring commutativity, but still might cause an expensive collision resolution whenever the quorum demarcation integrity constraint is reached, as described in Section 3.3.4.2.

Figure 3.6 shows the number of commits and aborts for different designs, for various *hot-spot* sizes. When the *hot-spot* size is large, the conflict rate is low, so all configurations do not experience many aborts. MDCC commits the most transactions because it does not abort any transactions. Fast commits slightly fewer, because it has to resolve the collisions which occur when different storage nodes see updates in different orders. Multi commits far

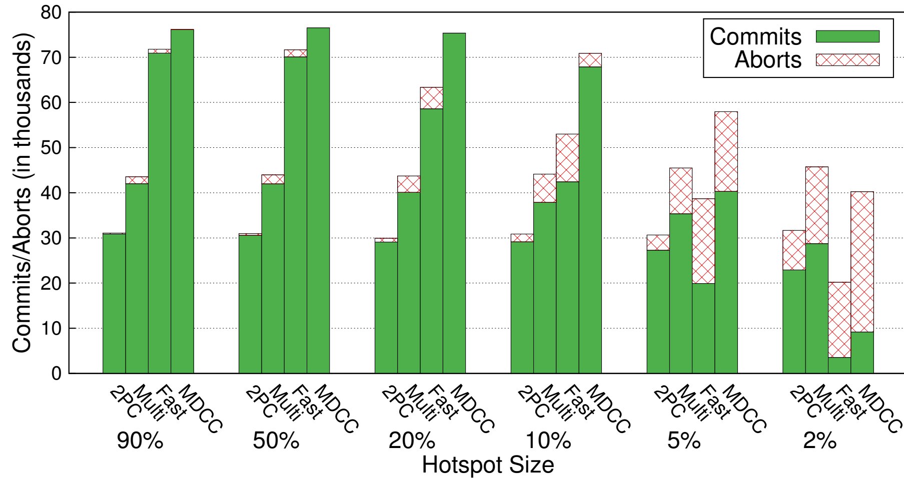


Figure 3.6: Commits/aborts for varying conflict rates

fewer transactions because most updates have to be sent to a remote master, which increases the response times and decreases the throughput.

As the *hot-spot* decreases in size, the conflict rate increases because more of the transactions access smaller portions of the data. Therefore, more transactions abort as the *hot-spot* size decreases. When the *hot-spot* is at 5%, the Fast commits fewer transactions than Multi. This can be explained by the fact that Fast needs 3 round-trips to ultimately resolve conflicting transactions, whereas Multi usually uses 2 rounds. When the *hot-spot* is at 2%, the conflict rate is very high, so both Fast and MDCC perform very poorly compared to Multi. The costly collision resolution for fast ballots is triggered so often, that many transactions are not able to commit. Therefore, fast ballots can take advantage of master-less operation as long as the conflict rate is not very high. When the conflict rate is too high, a master-based approach is more beneficial and MDCC should be configured as Multi. Exploring policies to automatically determine the best strategy remains as future work.

3.5.3.3 Data Access Locality

Classic ballots can save message trips in the situation when the client requests have affinity for data with a local master. To explore the trade-off between fast and classic ballots, the micro-benchmark was modified to vary the choice of data items within each transaction, so a configurable percentage of transactions will access records with local masters. At one extreme, 100% of transactions choose their items only from those with a local master; at the other extreme, 20% of the transactions choose items with a local master (items are chosen uniformly at random).

Figure 3.7 shows the box plots of the latencies of Multi and MDCC for different master localities. When all the masters are local to the clients, then Multi will have lower response times than MDCC, as shown in the graph for 100%. However, as updates access more

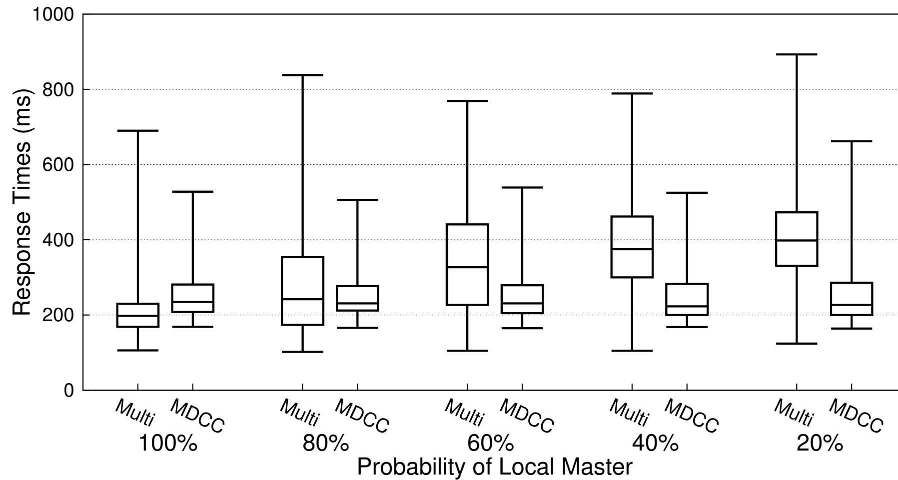


Figure 3.7: Response times for varying master locality

remote masters, response times for Multi get slower and also increase in variance, but MDCC still maintains the same profile. Even when 80% of the updates are local, the median Multi response time (242ms) is slower than the median MDCC response time (231ms). The MDCC design is targeted at situations without particular access locality, and Multi only out-performs MDCC when the locality is near 100%. Interesting to note is, that the max latency of the Multi configuration is higher than for full MDCC. This can be explained by the fact that some transactions have to queue until the previous transaction finishes, whereas MDCC normally operates in fast ballots and everything is done in parallel.

3.5.3.4 Data Center Fault Tolerance

How MDCC behaves with data center failures was also studied. With this experiment 100 clients were started issuing write transactions from the US-West data center. About two minutes into the experiment, a failure of the US-East data center was simulated, which is the data center closest to US-West. The failed data center was simulated by preventing the data center from receiving any messages. Since US-East is closest to US-West, "killing" US-East forces the protocol to actively deal with the failure. All the committed transaction response times were recorded and plotted the time series graph, in Figure 3.8.

Figure 3.8 shows the transaction response times before and after failing the data center, which occurred at around 125 seconds into the experiment (solid vertical line). The average response time of transactions before the data center failure was 173.5 ms and the average response time of transactions after the data center failure was 211.7 ms (dashed horizontal lines). The MDCC system clearly continues to commit transactions seamlessly across the data center failure. The average transaction latencies increase after the data center failure, but that is expected behavior, since the MDCC commit protocol uses quorums and must wait for responses from another data center, potentially farther away. The same argument

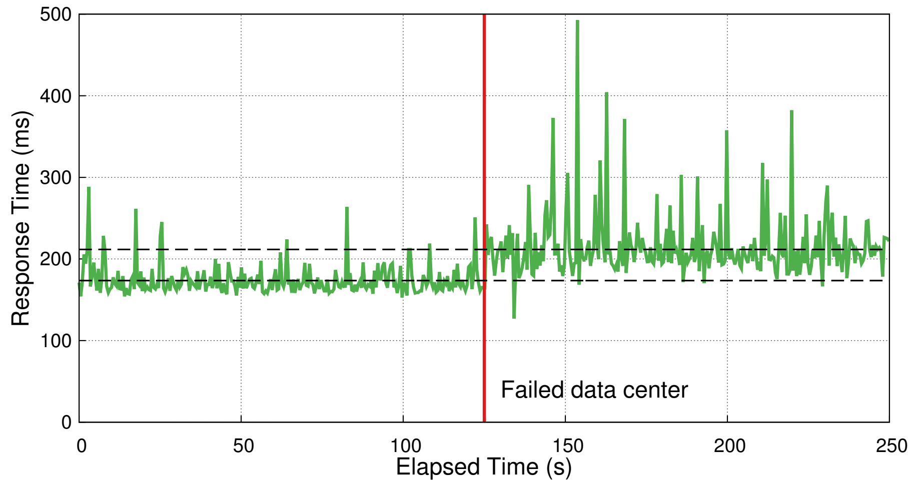


Figure 3.8: Time-series of response times during failure (failure simulated at 125 seconds)

also explains the increase in variance. If the data center comes up again (not shown in the figure), only records which have been updated during the failure, would still be impacted by the increased latency until the next update or a background process brought them up-to-date. Since fewer servers respond, it is more likely that the protocol is forced to wait for a delayed message. These results show the resilience of the MDCC protocol against data center failures.

3.6 Related Work

This section expands on the general background of Section 2. There has been recent interest in scalable geo-replicated data stores. Several recent proposals use Paxos to agree on log-positions similar to state-machine replication. For example, Megastore [13] uses Multi-Paxos to agree on log-positions to synchronously replicate data across multiple data centers (typically five data centers). Google Spanner [31] is similar, but uses synchronized timestamps to provide snapshot isolation. Furthermore, other state-machine techniques for WANs such as Mencius [57] or HP/CoreFP [35] could also be used for wide-area database log replication. All these systems have in common that they significantly limit the throughput by serializing all commit log records and thus, implicitly executing only one transaction at a time. As a consequence, they must partition the data in small shards to get reasonable performance. Furthermore, these protocols rely on an additional protocol (usually 2PC, with all its disadvantages) to coordinate any transactions that access data across shards. Spanner and Megastore are both master-based approaches, and introduce an additional message delay for remote clients. The authors of Spanner describe that they tried a more fine-grained use of Paxos by running multiple instances per shard, but that they eventually gave up because of the complexity. In this chapter, I showed it is possible and presented a system that uses

multiple Paxos instances to execute transactions. Mencius [57] uses a clever token passing scheme to not rely on a single static master. This scheme however is only useful on large partitions and is not easy applicable on finer grained replication (e.g., on a record level). Like MDCC, HP/CoreFP avoids a master, and improves on the cost of Paxos collisions by executing classic and fast rounds concurrently. Their hybrid approach could easily be integrated into MDCC but requires significant more messages, which is worrisome for real world applications. In summary, MDCC improves over these approaches, by not requiring partitioning, natively supporting transactions as a single protocol, and/or avoiding a master when possible.

Paxos-CP [65] improves Megastore's replication protocol by allowing non-conflicting transactions to move on to subsequent commit log-positions and combining commits into one log-position, significantly increasing the fraction of committed transactions. The ideas are very interesting but their performance evaluation does not show that it removes the log-position bottleneck (Paxos-CP only executes 4 transactions per second). Compared to MDCC, Paxos-CP requires an additional master-based single node transaction conflict detection, but is able to provide stronger serializability guarantees.

A more fine-grained use of Paxos was explored in Consensus on Commit [39], to reliably store the resource manager decision (commit/abort) to make it resilient to failures. In theory, there could be a resource manager per record. However, they treat data replication as an orthogonal issue and require that a single resource manager makes the decision (commit/abort), whereas MDCC assumes this decision is made by a quorum of storage nodes. Scalaris [74] applied consensus on commit to DHTs, but cannot leverage Fast Paxos as MDCC does. The use of record versioning with Paxos in MDCC has some commonalities with multi-OHS [1], a protocol to construct Byzantine fault-tolerant services, which also supports atomic updates to objects. However, multi-OHS only guarantees atomic durability for a single server (not across shards) and it is not obvious how to use the protocol for distributed transactions or commutative operations.

Other geo-replicated data stores include PNUTS [30], Amazon's Dynamo [33], Walter [79] and COPS [56]. These use asynchronous replication, with the risk of violating consistency and losing data updates in the event of major data center failures. Walter [79] also supports a second mode with stronger consistency guarantees between data centers, but this relies on 2PC and always requires two round-trip times.

Use of optimistic atomic broadcast protocols for transaction commit were proposed in [47, 66]. That technique does not explore commutativity and often has considerably longer response-times in the wide-area network because of the wait-time for a second verify message before the commit is final.

Finally, the MDCC demarcation strategy for quorums was inspired by classic demarcation [14], which first proposed using extra limits to ensure value constraints.

3.7 Conclusion

Scalable transactions for distributed database systems are important for the needs of modern applications. With geo-replicated database systems, the long and fluctuating latencies between data centers make it hard to support highly available applications that can scale out for massive workloads and can survive data center failures. Reducing the latency and improving the scalability for transactional commit protocols are the main goals of this research.

I proposed MDCC as a new approach for scalable transactions with synchronous replication in the wide-area network. The MDCC commit protocol is able to tolerate data center failures without compromising consistency, at a similar cost to eventually consistent protocols. It requires only one message round-trip across data centers in the common case. In contrast to 2PC, MDCC is an optimistic commit protocol and takes advantage of situations when conflicts are rare and/or when updates commute. It is the first protocol applying the ideas of Generalized Paxos to transactions that may access records spanning partitions. I also presented a new technique to guarantee domain integrity constraints in a quorum-based system.

MDCC provides a scalable transactional commit protocol for the wide-area network which achieves strong consistency at a similar cost to eventually consistent protocols.

Chapter 4

A New Transaction Programming Model

4.1 Introduction

In chapter 3, I proposed a new commit protocol for more scalable transactions for distributed database systems. Lower latency transactions are very beneficial for developing demanding applications, but there are still challenges in interacting with distributed transactions. Modern database environments can significantly increase the uncertainty in transaction response times and make developing user facing applications more difficult than ever before. Unexpected workload spikes [18], as well as recent trends of multi-tenancy [32, 36], and hosting databases in the cloud [48, 72, 29] are all possible causes for transaction response times to experience higher variance and latency. With modern distributed, geo-replicated database systems, the situation is only worse. Geo-replication is now considered essential for many online services to tolerate entire data center outages [38, 30, 5, 4]. However, geo-replication drastically influences latency, because the network delays can be 100's of milliseconds and can vary widely. Figure 4.1 exhibits the higher latency (~$100ms$ average latency) and variability (latency spikes exceeding $800ms$) of RPC message response times between geographically diverse Amazon EC2 regions.

With these modern database environments of higher latency and variance, there are currently only two possible ways developers can deal with transactions in user facing applications such as web-shops or social web-applications. Developers are forced to either wait longer for the transaction result, or be uncertain of the transaction result. These two options cause an undesirable situation, as clients interacting with database-backed applications are frequently frustrated when transactions take too long to complete, or fail unexpectedly.

To help developers cope with the uncertainty and higher latency, I introduce Predictive Latency-Aware NEtworked Transactions (PLANET), a new transaction programming model. PLANET provides staged feedback from the system for faster responses, and greater visibility of transaction state and the commit likelihood for minimizing uncertainty. This

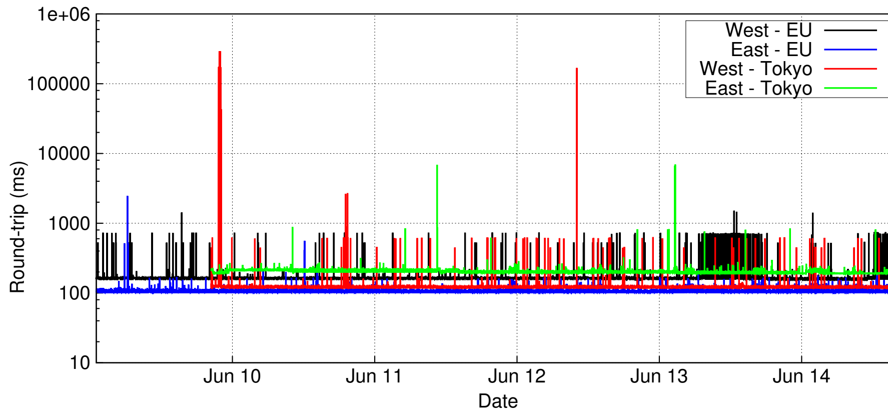


Figure 4.1: Round trip response times between various regions on Amazon's EC2 cluster.

enables building user facing applications in unpredictable environments, without sacrificing the user experience in unexpected periods of high latency. Also, PLANET is the first transaction programming model that allows developers to build applications using the *guesses and apologies* paradigm as suggested by Helland and Campbell [43]. By exposing more transaction state to the developer, PLANET also enables applications to speculatively initiate some processing, thereby trading the expense of an occasional apology or revocation for faster response in the typical case that the transaction eventually succeeds. Furthermore, in cases where delays or data access patterns suggest that success is unlikely, PLANET can quickly reject transactions rather than spending user time and system resources on a doomed transaction. With the flexibility and insight that PLANET provides, application developers can regain control over their transactions, instead of losing them in unpredictable states after a timeout.

The remainder of this chapter describes the details and features of PLANET. Section 4.2 describes how current models fall short, and outlines how PLANET satisfies my vision for a transaction programming model for unreliable environments. In Sections 4.3 and 4.4, I describe the details of PLANET, and in Section 4.5, I discuss the commit likelihood models and implementation details for MDCC, a geo-replicated database system. Finally, various features of the PLANET transaction programming model are evaluated in Section 4.6, followed by related work and conclusion in Sections 4.7 and 4.8.

4.2 The Past, The Dream, The Future

In a high variance distributed environment, unexpected periods of high latency can cause database transactions to either take a long time to complete, or experience unusually high failure rates. In this section, I first outline the pitfalls of current techniques when dealing with high latency and variance, describe the requirements for a latency-aware transaction

```java
Session sess = factory.openSession();
Transaction tx = null;
try {
  tx = sess.beginTransaction();
  tx.setTimeout(1); // 1 second
  // The transaction operations
  boolean success = tx.commit();
} catch (RuntimeException e) {
  // Handle the exception
} finally {
  sess.close();
}
```

Listing 4.1: Typical Hibernate Transaction

programming model, and finally provide an overview of my solution, PLANET.

4.2.1 The Past: Simple Timeouts

Current, state-of-the-art transaction programming models such as JDBC or Hibernate, provide little or no support for achieving responsive transactions. Existing transaction programming models only offer a time-out and ultimately implement a “fire-and-hope” paradigm, where once the transaction is started, the user can only hope that it will finish within the desired time frame. If the transaction does not return before the application's response-time limit, its outcome is entirely unknown. In such cases, most applications choose to display a vague error message.

Listing 4.1 shows a typical transaction with Hibernate [44] and the timeout set to 1 second. That is, within 1 second, the transaction either returns with the outcome stored in the Boolean variable *success*, or throws an exception. In the case of an exception, the outcome of the transaction is unknown.[1] Exceptions can either mean the transaction is already committed/aborted, will later be rolled-back because of the timeout, or was simply lost and never even accepted at the server.

When a transaction throws an exception, developers have two options to recover from this unknown state: either periodically poll the database system to check if the transaction was executed, or “hack” the database system to get access to the persistent transaction log. The first option is difficult to implement without modifying the original transaction, as it is often infeasible to distinguish between an application's own changes and changes of other concurrent transactions. The second option requires detailed knowledge about the internals of the database system and is especially difficult in a distributed database system with no

[1] Hibernate supports *wasRolledBack()*, which returns *true* if the transaction rolled back. However, this only accounts for application-initiated rollbacks, not system rollbacks.

centralized log. In Section 4.2.2, I propose properties for make developing applications easier, especially with geo-replicated database systems.

4.2.2 The Dream: LAGA

As described previously, current transaction programming models like JDBC or Hibernate, leave the user in the dark when a timeout occurs. Furthermore, they provide no additional support for writing responsive applications. In this section, I describe the four properties a transaction programming model should have for uncertain environments, referred to as the ***LAGA*** properties for ***Liveness***, ***Assurance***, ***Guesses***, and ***Apologies***:

Liveness The most important property is that the application guarantees liveness and does not have to wait arbitrarily long for a transaction to finish. This property is already fulfilled by means of a timeout with current transaction programming models.

Assurance If an application decides to move ahead without waiting for the final outcome of the transaction (e.g., after the timeout), the application should have the assurance that the transaction will never be lost and that the application will at some point be informed about the final outcome of the transaction.

Guesses The application should be able to make informed decisions based on incomplete information, before the transaction even completes to reduce perceived latency, as, for instance, suggested by Helland and Campbell [43]. For example, if a transaction is highly likely to commit or abort, an application may choose to advance instead of waiting for transaction completion.

Apologies If a mistake was made on an earlier guess, the application should be notified of the error and the true transaction outcome, as suggested by Helland and Campbell [43], so the application can apologize to the user and/or correct the mistake.

The ***LAGA*** properties describe a transaction programming model, and are orthogonal to the transaction guarantees by the underlying database system. For example, a database system can fully support the ACID properties (Atomicity, Consistency, Isolation, Durability), while the transaction programming model supports the ***LAGA*** properties.

4.2.3 The Future: PLANET Example

In contrast to the state-of-the-art transaction programming models, PLANET fulfills not only the ***Liveness*** property but all four properties. Listing 4.2 shows an example transaction using PLANET in the Scala programming language. The transaction is for a simple order purchasing action in an e-commerce website, such as Amazon.com. The code fragment outlines how the application can guarantee one of three responses to the user within 300ms: (1) an error message, (2) a “Thank you for your order” page, or (3) a successful order page,

```
1  val t = new Tx(300ms) ({
2    INSERT INTO Orders VALUES (<customer id>);
3    INSERT INTO OrderLines
4      VALUES (<order id>, <item1 id>, <amt>);
5    UPDATE Items SET Stock = Stock - <amt>
6      WHERE ItemId = <item1 id>;
7  }).onFailure(txInfo => {
8    // Show error message
9  }).onAccept(txInfo => {
10   // Show page: Thank you for your order!
11 }).onComplete(90%)(txInfo => {
12   if (txInfo.state == COMMITTED ||
13       txInfo.state == SPEC_COMMITTED) {
14     // Show success page
15   } else {
16     // Show order not successful page
17   }
18 }).finallyCallback(txInfo => {
19   if (!txInfo.timedOut) {
20     // Update via AJAX
21   }
22 }).finallyCallbackRemote(txInfo => {
23   // Email user the completed status
24 })
```

Listing 4.2: Order purchasing transaction using PLANET

given the status of the transaction at the timeout; Furthermore, it guarantees an email and AJAX notification when the outcome of the transaction is known. Here, I briefly explain the different mechanisms used in the example, before Section 4.3 describes the semantics of PLANET .

With PLANET, transaction statements are embedded in a transaction object (line 1–6). PLANET requires a timeout (line 1) to fulfill the ***Liveness*** property. After the timeout, the application regains control. PLANET exposes three transaction stages to the application, *onFailure*, *onAccept* and *onComplete*. These three stages allow the developer to appropriately react to the outcome of the transaction given its state at the timeout. Whereas the code for *onFailure* (line 7–8) is only invoked in the case of an error, and *onComplete* (line 11–13) is invoked only if the transaction outcome is known before the timeout, *onAccept* exposes a stage between failure and completion, with the promise that the transaction will not be lost, and the application will eventually be informed of the final outcome. Therefore, the *onAccept* stage satisfies the ***Assurance*** property.

Only one of the code fragments for *onFailure*, *onAccept*, or *onComplete* is executed within

the time frame of the timeout. In addition, *onComplete* can take a probability parameter that enables speculative execution with a developer-defined commit likelihood threshold (90% in the example) and thus, fulfills the ***Guesses*** property. Finally, the callbacks *finallyCallback-Remote* and *finallyCallback* support the ***Apologies*** property, by providing a mechanism for notifying the application about the final outcome of the transaction regardless of when the timeout happened.

Given the four properties, PLANET enables developers to write highly responsive applications in only a few lines of code. The next section describes the semantics of PLANET in more detail.

4.3 PLANET Simplified Transaction Programming Model

PLANET is a transaction programming model abstraction and can be used with different data models, query languages and consistency guarantees, similar to JDBC being used with SQL or XQuery, depending on database support. The key idea of PLANET is to allow developers to specify different *stage blocks* (callbacks) for the different stages of a transaction. This section describes the simplified transaction programming model of PLANET, which is essentially "syntactic sugar" for common stages and usage patterns. Section 4.4.1 describes the more general model, which provides the developer with full control and customization possibilities.

4.3.1 Timeouts & Transaction Stage Blocks

At its core, PLANET combines the idea of timeouts with the new concept of *stage blocks*. In PLANET, the timeout is always required, but can be set to infinity. Finding the right timeout is up to the developer and can be determined through user studies [73, 8]. Listing 4.2 shows an example with the timeout set to $300ms$.

PLANET simplified transaction programming model also defines three *stage blocks*, corresponding to the internal stages of the transaction, that follow an ordered progression of *onFailure*, then *onAccept*, then *onComplete* (see Figure 4.2). When the timeout expires, the application regains the thread of control, and depending on the state of the transaction, *only* the code for the latest defined *stage block* is executed. That is, for any given transaction, only one of the three stage blocks is ever executed. In the following, I describe the *stage blocks* and their guarantees in more detail.

onFailure. PLANET tries to minimize the uncertainty when a timeout occurs, but cannot entirely prevent it. In fact, for distributed database systems, it can be shown by a reduction to the Two Generals' Problem, that it is theoretically impossible to completely avoid uncertainty of the outcome. Therefore, PLANET requires the *onFailure* stage to be defined, which is similar to an exception code block. When nothing is known about the commit progress when the timeout expires, the *onFailure* code is executed. Reaching the *onFailure*

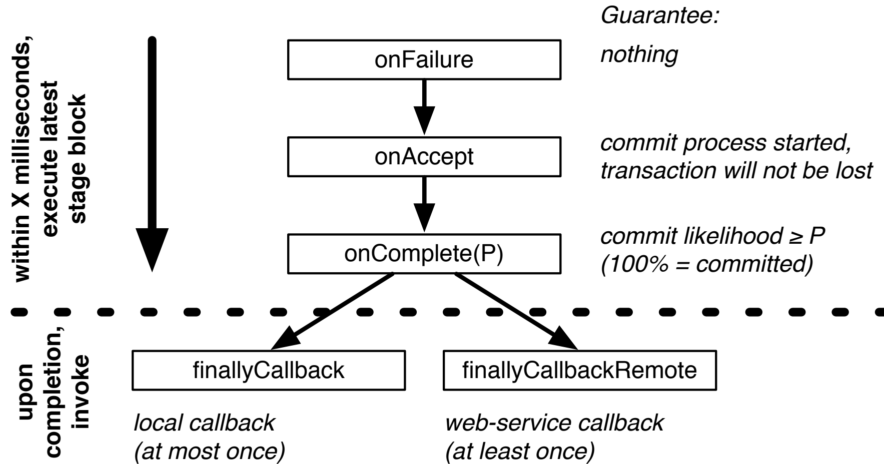


Figure 4.2: Client view of PLANET transactions

stage does not necessarily imply that the transaction will abort; instead the application may later be informed about a successfully committed transaction (see Section 4.3.3).

onAccept. When the transaction will not be lost anymore and will complete at some point, the transaction is considered *accepted*. Typically, this is after the system started the commit process. How strong the *not-be-lost* guarantee is depends on the implementation. For example, in a distributed database system, it could mean that at least one database server successfully received and acknowledged the commit proposal message (with the assumption that all servers eventually recover all acknowledged messages).

If the timeout expires, the *onAccept* stage is executed if the system is still attempting to commit the transaction, but the final outcome (i.e., abort or commit) is still unknown. If the later *stage block onComplete* is undefined, *onAccept* will be executed immediately after the transaction is *accepted* by the system, and not wait for the timeout. This feature is particularly useful for achieving very fast response times for transactions which will not abort from conflict (non-conflicting, append-only transactions). In contrast to the *onFailure* stage, if the *onAccept* stage is invoked, the system makes two important promises: (1) The transaction will eventually complete, and (2) the application will be later be informed about the final outcome of the transaction (see Section 4.3.3). Therefore, the *onAccept* stage satisfies the ***Assurance*** property.

onComplete. As soon as the final outcome of the transaction is known and the timeout has not expired, *onComplete* is executed. If the timeout did expire, an earlier stage, either *onFailure* or *onAccept*, was already executed, so *onComplete* will be disregarded.

An important point to the *onAccept* and *onComplete* stage blocks is that they both do not have to be defined. If only *onAccept* is defined, *onAccept* will be invoked as soon as the transaction is accepted, and the thread of control will return to the application. If only *onComplete* is defined, *onComplete* will be invoked when the transaction completes (or *onFailure* if it timed out). This allows developers to define flexible behavior for different

applications.

Finally, all *stage blocks* are always invoked with a transaction state summary (*txInfo* in Listing 4.2). The summary contains information about the transaction which includes the current state (*UNKNOWN*, *REJECTED*, *ACCEPTED*, *COMMITTED*, *SPEC_COMMITTED*, *ABORTED*), the time relative to the timeout (if the transaction *timed out*), and the updated commit likelihood (Section 4.3.2).

4.3.2 Speculative Commits using onComplete

PLANET provides the ***Guesses*** property by allowing developers to write applications that advance without waiting for the outcome of a transaction, if the expected likelihood of a successful commit is above some threshold P. This ability for applications to advance before the final transaction outcome and based on the commit likelihood is known as *speculative commits*. The developer enables speculative commits by using the optional parameter P of the *onComplete stage block*. For example, if the developer determines that a transaction should be considered as finished when the commit likelihood is at least 90%, then the developer would define the *stage block* as *onComplete(90%)* (shown in Listing 4.2). For a defined threshold P, PLANET will execute the *onComplete stage block* before the timeout, as soon as the commit likelihood of the transaction is greater than or equal to the threshold, which can greatly reduce transaction response times.

Obviously, the commit likelihood computation is dependent on the properties of the underlying system, and Sections 4.5.1 and 4.5.2 describe the model and statistics required for MDCC, a geo-replicated database system using the Paxos protocol. For most database systems, PLANET will calculate the commit likelihood using local statistics at the beginning of the transaction. However, for some database systems, it is also possible to re-evaluate the likelihood as more information becomes available during the execution of the transaction. Possible examples are: discovering a record of a multi-record transaction that has completed, and receiving responses from previous RPCs.

Of course, speculative commits are not suited for every application, and the commit likelihoods and thresholds vary significantly from application to application. However, I believe that speculative commits are a simpler programming construct for applications, which can already cope with eventual consistency. I envision many additional applications which can significantly profit from PLANET's speculative commits. For example, a ticket reservation system could use speculative commits to allow very fast response times, without risking significantly overselling a high-demand event like the Google I/O conference (see also [49] and [67]). In order to guide users in picking the right threshold, user experience studies or automatic cost-based techniques [49] can be leveraged.

4.3.3 Finally Callbacks and Apologies

It is possible that the application will not know the transaction's final outcome when the timeout expires, either because of a speculative commits, because *onComplete stage block* was

not defined, or because the transaction took longer than the timeout to finish. PLANET addresses this uncertainty with code blocks ***finallyCallback*** and ***finallyCallbackRemote***, which are special callbacks used to notify the application of the actual commit decision of the completed transaction. They are different from the other *stage blocks* because they are not restricted to execute within the timeout period, and they run whenever the transaction completes. The callbacks allow the developer to apologize and correct any incorrect behavior or state because of speculative commits, errors, or timeouts, and therefore satisfy the ***Apologies*** property.

In contrast to the *finallyCallback* code which can contain arbitrary code, the code for *finallyCallbackRemote* can only contain web-service invocations (e.g., REST calls), which can be executed anywhere in the system without requiring the outer application context. For *finallyCallback*, the system guarantees at-most-once execution. For example, a developer might use *finallyCallback* to update the web-page dynamically using AJAX about the success of a transaction after the timeout expires. However, *finallyCallback* might never be executed, in cases of application server failures. In contrast, *finallyCallbackRemote* ensures at-least-once execution as the web-service invocation can happen from any service in the system at the cost of reduced expressivity (i.e., only web-service invocations are allowed). The restrictions on *finallyCallbackRemote* allow for fault tolerant callbacks. Listing 4.2 shows an example of defining both callbacks; *finallyCallback* updates the application using AJAX and *finallyCallbackRemote* sends the order confirmation email. Like the *stage blocks*, *finallyCallback* and *finallyCallbackRemote* also have access to the current transaction summary, *txInfo*.

When speculative commits are used with a likelihood parameter $P < 100\%$, some transactions may experience *incorrect commits*. This occurs when the transaction commit likelihood is high enough (greater than P) and *onComplete* is invoked, but the transaction aborts at a later time. To apologize for *incorrect commits*, the final status is notified through one of the *finally* callbacks, thus satisfying the ***Apologies*** property.

4.3.4 PLANET vs. Eventual Consistency

Utilizing the *onAccept* block, coupled with the *finally* callbacks, is a valuable alternative to Eventual Consistency (EC) models [33]. Eventually consistent systems are typically used for their fast response times and high availability guarantees, but come with a high cost: potential data inconsistencies. In contrast, PLANET can also offer fast response times and high availability without sacrificing data consistency.

PLANET does not change the transaction semantics of the underlying database system; even with speculative commits the data would never be rendered inconsistent if the backend is strongly consistent. It only allows clients to deliberately proceed (using *onAccept* or speculative commits), even though the final decision (commit/abort) is not yet known. Furthermore, the *onAccept* stage guarantees (transactions will not be lost) can be implemented in a highly available fashion. Similar to eventual consistency, the application can still make inconsistent decisions, like informing a user about a successful order even though items are

sold out. But in contrast to eventual consistency, PLANET does not allow the database to expose an inconsistent state to other subsequent transactions or concurrent clients, and thus, prevents dependencies based on inconsistent data, which are particularly hard to detect and correct. This property makes it easier to write applications than with the eventually consistent transaction model: it isolates mistakes (e.g., a commit followed by an abort) to a single client/transaction allowing the developer to better foresee the implications.

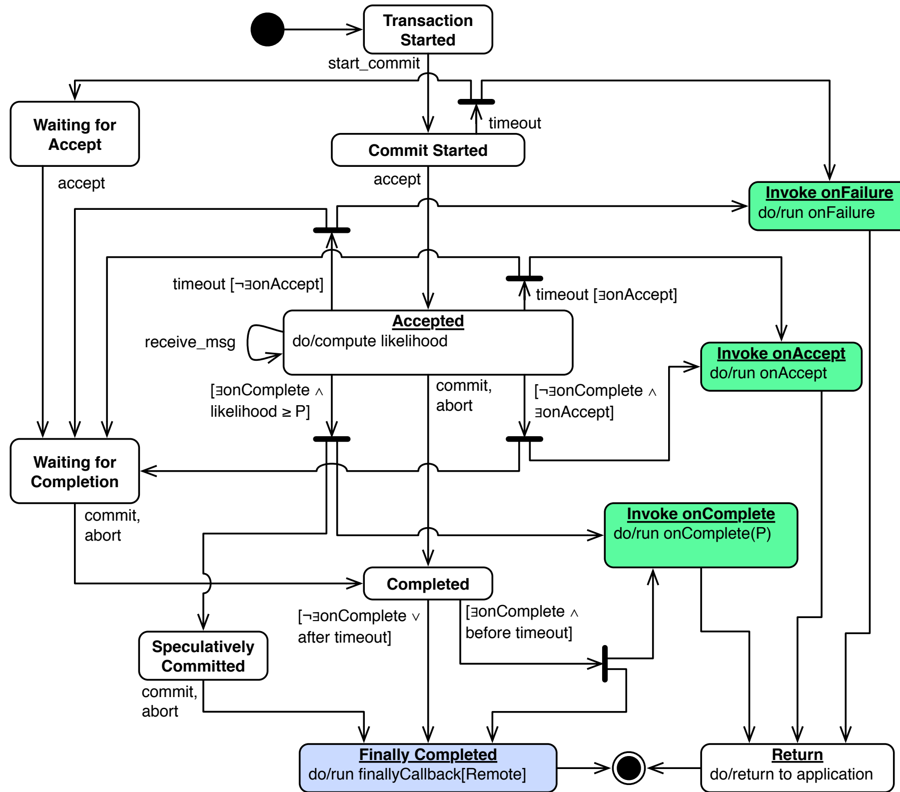


Figure 4.3: PLANET transaction state diagram

4.3.5 Life of a PLANET Transaction

Figure 4.3 formalizes the transaction programming model using a state diagram. State transitions are represented by edges and use the notation "*event[guard]/action*", which means *guard* must evaluate to *true* for the *event* to trigger the transition and the *action*. The dark perpendicular lines represent a fork in the state diagram, to capture parallel execution. The shaded states are the ones which execute user-defined stage blocks, and $\exists stageName$ and $\neg\exists stageName$ refer to whether or not the particular stage block was defined in the transaction code. This diagram shows that the commit likelihoods are computed every time a new message is received, and that timeouts fork execution so that the transaction continues

```
val t = new Tx(30000ms) ({
  UPDATE Accounts SET Balance = Balance - 100
    WHERE AccountId = <id>;
}).onFailure(txInfo => {
  // Show error message
}).onComplete(txInfo => {
  if (txInfo.success) {
    // Show money transfer
  } else {
    // Show failure page
  }
}).finallyCallbackRemote(txInfo => {
  if(txInfo.success && txInfo.timedOut) {
    // Inform service personnel
  }
})
```

Listing 4.3: ATM example using PLANET

to execute while the application regains control. For an example of a speculative commit, in the *Accepted* stage, when the *onComplete* stage block is defined and the likelihood is greater than P, the transition will fork so that one will run *onComplete* and return to the application, while the other will be in the *Speculatively Committed* state waiting for completion. Even though this state diagram may have many states, the client sees a far simpler view, without all the internal transitions, which can be simply explained using the state progression flow shown in Figure 4.2.

4.3.6 Usage Scenarios

PLANET is very flexible and can express many kinds of transactions. There are ad hoc solutions and systems for every situation, but PLANET is expressive enough to encapsulate the use cases into a single model for the developer, regardless of the underlying implementation. A document containing various use cases can be found in [67]. In addition to the e-commerce website motivating example in Listing 4.2, I describe two other use cases.

ATM Banking Code Listing 4.3 shows an example of an ATM transaction. The structure is very similar to a standard Hibernate transaction, where only failures and commits are reported on. This is because correctness is critical for money transfers, so waiting for the final outcome is the most appropriate behavior. Therefore, there is no *onAccept stage block*. When the timeout expires and the transaction is not completed, it is a failure. However, if the transaction times out and later commits, this means the user saw a failure message, but the transaction eventually committed successfully. The *finallyCallbackRemote* handles this problematic situation.

```
val t = new Tx(200ms) ({
  INSERT INTO Tweets VALUES (<user id>, <text>);
}).onFailure(txInfo => {
  // Show error message
}).onAccept(txInfo => {
  // Show tweet accept
})
```

Listing 4.4: Twitter example using PLANET

Twitter Code Listing 4.4 shows an example of a transaction for sending an update to a micro-blogging service, like Twitter. In contrast to the ATM example, these small updates are less critical and are not required to be immediately globally visible. Also, the developer knows that there will never be any transaction conflicts, since every transaction is essentially a record append, and not an update. Therefore, the transaction only defines the *onFailure* and *onAccept* code blocks, which means the developer is not concerned with the success or failure of the commit. This type of transaction easily provides the response times of eventually consistent systems, but at the same time never allowing the data to become inconsistent.

4.4 Advanced PLANET Features

The following section first describes the generalized version of PLANET, which gives developers even more freedom, and then, the PLANET admission control feature.

4.4.1 Generalized Transaction Programming Model

While the simplified model in Section 4.3 supports many of the common cases, there may be situations when the developer wants more fine-grained control. This section describes the fully generalized transaction programming model which supports the simplified model in Section 4.3, but also provides more control for the developer. The generalized model has only one *stage block*, *onProgress*, and has the two *finally* callbacks, *finallyCallback* and *finallyCallbackRemote*. The *stage blocks* of the simplified model in Section 4.3 are actually just "syntactic sugar" for common cases using the generalized model. Listing 4.5 is equivalent to the simplified Listing 4.2, but uses the generalized *onProgress* block instead.

4.4.1.1 onProgress

The *onProgress stage block* is useful for the application to get updates or notifications on the progress of the executing transaction. Whenever the transaction state changes, the *onProgress* block is called with the transaction summary, containing information about the transaction status and other additional information about the transaction (e.g., the commit

```
val t = new Tx(300ms) ({
  // Transaction operations
}).onProgress(txInfo => {
  if (txInfo.timedOut) {
    if (txInfo.state == ACCEPTED) {
      // onAccept code
    } else {
      // onFailure code
    }
    FINISH_TX  // finish the transaction.
  } else {  // not timedOut
    if (txInfo.commitLikelihood > 0.90) {
          // onComplete(90%) code
      FINISH_TX  // finish the transaction.
    }
  }
}).finallyCallback(txInfo => {
  // Callback: Update status via AJAX
}).finallyCallbackRemote(txInfo => {
  // Callback: Update status via email
})
```

Listing 4.5: PLANET general transaction programming model. Equivalent to Listing 4.2

likelihoods). This means the stage may be called multiple times during execution. By getting notifications on the transaction status, the application can make many informed decisions on how to proceed.

A special feature for the code defined in the *onProgress* block is that the developer can return a special code *FINISH_TX* to signal to the transaction handler that the application wants to stop waiting and wishes to move on. If the code returns *FINISH_TX*, the application will get back the thread of control and get notified of the outcome with a *finally* callback. If the application does not want the thread of control, the transaction handler will continue to wait for a later progress update.

4.4.1.2 User-Defined Commits

Using the generalized PLANET transaction programming model and the exposed transaction state, the developer has full control and flexibility on how transactions behave. For example, *onProgress* allows informing the user details about the progress of a buying transaction;: a website could first show, “trying to contact the back-end”, then move on to “booking received”, until it shows “order successfully completed”. This is not possible in the simplified model.

The generalized model is a very powerful construct; ultimately, it allows developers to redefine what a commit means. For example, for one transaction, a developer can choose to emulate asynchronously replicated systems by defining the commit to occur when the local data center storage nodes received the updates (assuming that the transaction summary contains the necessary information), while choosing to wait for the final outcome for another transaction.

4.4.2 Admission Control

With commit likelihoods, it becomes very natural to also use these likelihoods for admission control. PLANET implements admission control with the hope of improving performance by preventing wasted resources or thrashing. If the system computes a transaction commit likelihood which is too low, that means there is a high chance the transaction will abort. If that is the case, it may be a better idea not to actually execute the transaction and potentially waste resources in the system such as CPU cycles, disk I/O, or extraneous RPCs. In addition to improving resource allocation, not attempting transactions with low likelihoods reduces contention on the involved records, which can lead to improving the chances for other transactions to commit on those records for some consistency protocols (e.g., MDCC [50]). Currently, PLANET supports two policies for admission control: Fixed and Dynamic.

Fixed(threshold, attempt_rate) Whenever the transaction commit likelihood is less than the *threshold*, the transaction is attempted with probability *attempt_rate*. For example, *Fixed(40,20)* means when the commit likelihood is less than 40%, the transaction will be attempted 20% of the time. If *attempt_rate* is 100%, the policy is equivalent to using no admission control.

Dynamic(threshold) The *Dynamic* policy is similar to the *Fixed* policy, where the attempt rate is not fixed, but related to the commit likelihood. Whenever the likelihood, L, is lower than the *threshold*, the transaction is attempted with probability L. For example, a *Dynamic(50)* policy means all transactions with a likelihood L less than 50%, will be attempted with probability L. If the *threshold* is 0, the policy is equivalent to using no admission control. Section 4.6.7 further investigates these parameters.

When a transaction is rejected by the system, PLANET does not actively retry the transaction. However, with PLANET, the developer may choose to retry rejected transactions (the transaction summary contains the necessary information). This may lead to starvation, but the developer can define how retries are done, and implement retries with exponential backoff to mitigate starvation.

The PLANET admission control technique using commit likelihoods does not preclude using other methods such as intermittent probing [42]. In fact, admission control of PLANET can augment existing techniques by using record access rates to improve the granularity of information, and by using commit likelihoods to identify types of transactions and access patterns.

4.5 Geo-Replication

The PLANET transaction programming model can be implemented on any transactional database system as long as the required statistics and transaction states are exposed. It is even possible to use PLANET for non-transactional key/value stores, where the commit likelihood would represent the likelihood of an update succeeding without lost updates (see also Section 4.5.1.3). However, the benefits of PLANET are most pronounced with geo-replicated, strongly consistent, transactional database systems such as Megastore [13], Spanner [31] or MDCC [50, 46]. Figure 4.1 shows the long and unpredictable delays between data centers (on Amazon EC2) these systems have to deal with. In the following, I show how PLANET can be implemented on an existing geo-replicated database system, MDCC, for which two implementations are available [50, 46]. However, the results from this study are transferable to other systems, like Megastore or COPS[56], by adjusting the likelihood models.

4.5.1 Conflict Estimation for MDCC

MDCC is a distributed, geo-replicated transactional database system designed along the lines of Megastore [13]. A typical MDCC database deployment is distributed across several storage nodes, and the nodes are fully replicated across multiple data centers. Besides the small changes to the underlying MDCC (see Section 4.5.2), most of PLANET is implemented in the client-side library.

In MDCC, every record has a master that replicates updates to remote data centers using the Paxos consensus protocol [53] similar to Megastore's Paxos implementation. However, MDCC is able to avoid Megastore's limitations of being only able to execute one transaction at a time per partition and has significantly higher throughput by using a finer grained execution strategy [50, 46]. Furthermore, MDCC supports various read-modes (snapshot-isolation, read-committed) and proposes several optimizations including a *fast* protocol to reduce the latency of commits at the cost of additional messages in the case of conflicts. For the remainder of this section, many of the optimizations are ignored and I focus on the default setting (read-committed) of MDCC, without the *fast* protocol. Only the MDCC classic protocol is modeled, as this configuration is more similar to other well-known systems like Megastore.

4.5.1.1 The MDCC Classic Protocol

In its basic configuration, the MDCC protocol provides read-committed isolation, and ensures all write-write conflicts are detected similar to snapshot isolation, but only provides atomic durability (i.e., all or none of the updates are applied) and not atomic visibility (i.e., consistent reads). This read guarantee has proven to be very useful, because read-committed is still the default isolation level in most commercial and open-source database systems (e.g., MS SQL Server, PostgreSQL, Oracle).

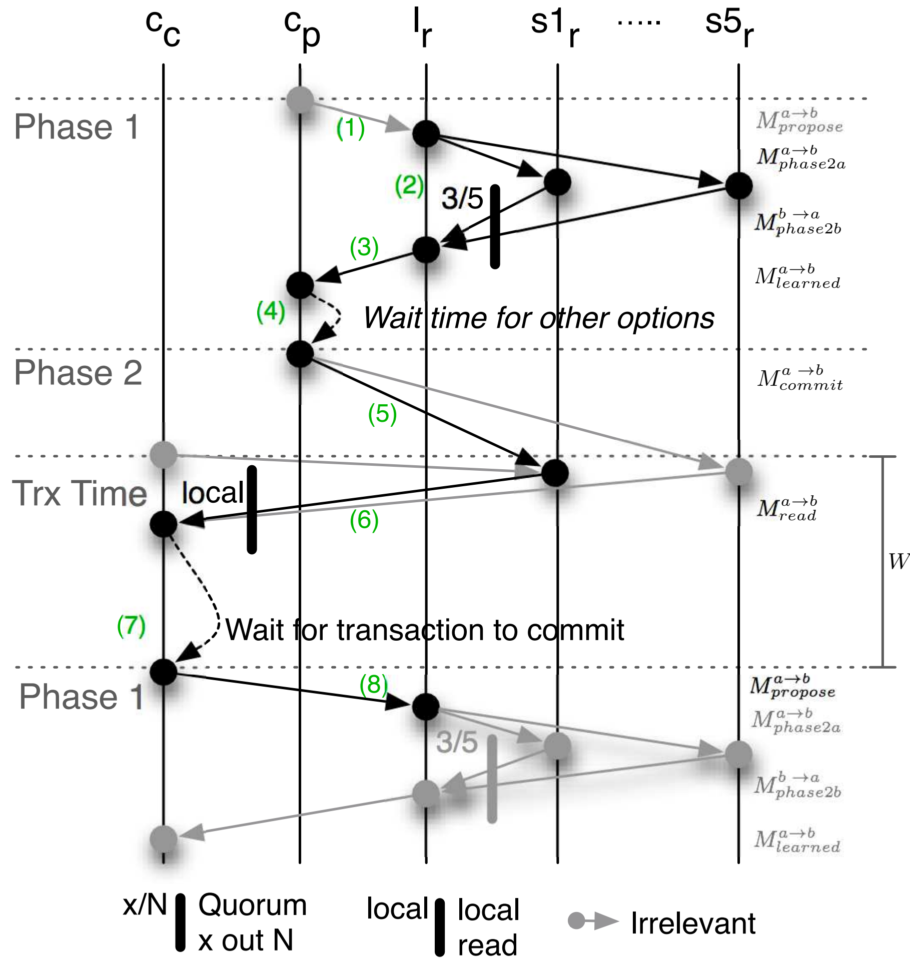


Figure 4.4: Sequence diagram for the MDCC classic protocol

In the MDCC classic protocol, the transaction manager (typically, the client) acquires an *option* per record update in a transaction using Multi-Paxos. The option is learned either as accepted or rejected using Multi-Paxos from a majority of storage nodes (note that even rejecting an option requires learning the option). A learned (but not yet visible) option prevents other updates on the same record from succeeding (i.e., does the write-write conflict detection) and can best be compared to the first phase of two-phase commit (2PC) as it prepares the nodes to commit. However, in contrast to 2PC, the transaction is immediately committed if all the options have successfully been *learned* using Paxos. If all options are learned as accepted, the transaction manager has no choice and must commit the transaction. Similarly, it has to abort the transactions if one of the options is learned as aborted. In this chapter, the MDCC optimization of broadcasting all messages to avoid a second phase is not considered.

The sequence diagram of the protocol is shown in Figure 4.4. As a first step, the client

(assumed to be C_p) proposes the option to any record-leader L_r (i.e, the master responsible for learning the option) involved in the transaction shown as step (1). Afterwards, the leader executes the Paxos round by sending a Paxos *phase2a* message to all storage nodes and waits for the majority of *phase2b* answers, visualized as step (2) for 5 storage $s1_r \dots s5_r$ nodes in Figure 4.4. If the option is learned by a majority, the leader sends a *learned* message with the success value to the transaction manager on the client C_p, shown as step (3). The transaction manager now has to wait for all *learned* messages, one per update in the transaction, indicated as step (4).[2] If all options are learned successfully, the transaction is committed and the client is allowed to move on. However, the updates are not yet visible to other clients. To make the updates visible, the transaction manager sends a *commit visibility* message to all involved storage nodes, shown as step (5). Note that the transaction manager/client C_p, the record leader L_r, and the storage nodes $S1_r \dots S5_r$ may all be in different data centers. If the client, leader and at least one storage node are co-located in the same data center, the commit will only take a single round-trip between remote data centers (local round-trips are less significant).

4.5.1.2 Commit Likelihood Model for MDCC

Next, I show how the commit likelihood for the described protocol is modeled. The key idea is estimating the time it takes to propagate the updates of a preceding transaction, so that the current transaction does not conflict with it. Given this duration, the likelihood of another transaction interfering can be calculated by considering the update rate per record. Section 4.5.2 describes how the system is able to collect the necessary statistics and precompute a lot of the calculations.

In the remainder of this section, the following symbols are used[3].

- $\{a, b, c, l\} \in \{1 \dots N\}$, where 1 to N are the data centers
- $C \in \{1 \dots N\}$, stochastic variable of the data center of the client, with c being an instance of the variable
- $L \in \{1 \dots N\}$, stochastic variable of the data center of the master (i.e., leader) with l being an instance of the variable
- $M^{a \to b} \in \mathbb{R}$, stochastic variable corresponding to the delay (message and processing time) of sending a message from data center a to data center b
- $R \in \mathbb{N}$, stochastic variable corresponding to the number of records inside a transaction
- $X(t) \subseteq \mathbb{N}$, stochastic variable corresponding to the number of expected updates for a given record and a time interval t.

[2]Actually, if messages come from the same storage nodes, they can be batched together. However, this optimization is not modeled.

[3]A stochastic variable is described in the short form $X \in \mathbb{R}$ instead of defining the function $X : \Omega \to \mathbb{R}$

- $W \in \mathbb{R}$, the processing time after reading the value before starting the commit
- $\Theta \in \{commit, abort\}$, stochastic variable corresponding to the commit or abort of a transaction

To simplify the model, it is assumed that transactions are issued independently, and the records as part of a transaction are also chosen independently. The protocol defines that a transaction is committed if all options are successfully learned. Therefore, the likelihood of a commit can be estimated by calculating the likelihood of successfully learning every option. In turn, learning an option can only be successful if no concurrent option is still pending (i.e., not committed). The first goal is therefore to derive a stochastic variable describing the required time for a previous transaction to commit and become visible.

A transaction for a record r is in conflict only from the moment it requested the option at the leader until it becomes visible, shown as steps (2) to (5). The leader executes the Paxos round by sending a *phase2a* message to all storage nodes and waiting for their *phase2b* responses. The message delay is modeled as a stochastic variable $M^{l,b}$, which simply adds the two stochastic variables for sending and receiving the *phase2a* and *phase2b* message from the leader's data center l to some other data center b:

$$M^{l,b} = M^{l \to b}_{phase2a} + M^{b \to l}_{phase2b} \tag{4.1}$$

The combined distribution for the round-trip requires convoluting the distributions for *phase2a* and *phase2b*. However, the leader only needs to wait for a majority q out of N responses. Assuming, that the leader sends the learning request to all N data centers, waiting for a quorum of answers corresponds to waiting for the maximum delay for all possible combination for picking n out of N responses and can be expressed as:

$$Q^l = \left\{ max\left(x_1 M^{l,1}, \ldots, x_N M^{l,N}\right) \mid x_i \in \{0,1\}; \sum_{i=1}^{N} x_i = q \right\} \tag{4.2}$$

Deriving the distribution for Q^l requires integrating over the maximum of all possible combinations of $M^{l,b}$. After the option has been successfully learned, the message delay to notify the transaction manager c_p can be modeled by adding the stochastic variable $M^{l \to c_p}_{learned}$ describing the delay to send a learned message:

$$Q^{l,c_p} = Q^l + M^{l \to c_p}_{learned} \tag{4.3}$$

Unfortunately, even though the duration of time to learn an option for a single record is reflected, the transaction is only committed if all the options of the transaction are successfully acquired. This entails waiting for the learned message from all involved leaders $(l_1, \ldots, l_r)$, with r being the number of updates. Furthermore, after the transaction manager received all learned messages, the update only becomes visible, if the *commit visibility* message arrives at least at the data center of the current transaction c_c before a local read is done for the record (assume only local reads). Given the locations of the leaders $(l_1, \ldots, l_r)$

and the previous client's data center c_p, the delay is modeled by taking the maximum of all Q^{l,c_p}, one for each leader, and adding the commit message time:

$$U^{c_c}\Big(c_p, (l_1, \ldots, l_r)\Big) = max\left(Q^{l_1,c_p}, \ldots, Q^{l_r,c_p}\right) + M^{c_p \rightarrow c_c}_{commit} \tag{4.4}$$

Equation 4.4 represents the stochastic variable describing the time to make an update visible for any given record. However, once the update is visible the current transaction still needs to read it and send its option to the leader. Therefore, the transaction processing time w is added, along with the stochastic variable for sending the *propose* message to the leader l.

$$\Phi^{c_c,l}(c_p, (l_1, \ldots, l_r)) = U^{c_c}(c_p, (l_1, \ldots, l_r)) + w + M^{c_c,l}_{propose} \tag{4.5}$$

Note that w is not a stochastic variable. Instead w is the measured time from requesting the read over receiving the response until committing the transaction. This allows factoring W out of equation 4.5 and only consider it in the next step.

Given the location c_p of the transaction manager of the previous transaction and all the involved leaders $(l_1, \ldots, l_r)$, $\Phi^{c_c,l}$ describes the time in which no other update should arrive to allow the current transaction to succeed. Unfortunately, these values are unknown. Therefore all possible instantiations of c_p and $(l_1, \ldots, l_r)$ need to be considered. By assuming independence between the inputs, the likelihood to finish within time t is described as:

$$\begin{aligned} P^{c_c,l}(t) = \sum_{\substack{\tau \in \mathbb{N} \\ l_1 \ldots l_\tau, c_p \in 1 \ldots N}} \Bigg\{ & P(R = \tau) P(L_1 = l_1) \cdots P(L_\tau = l_\tau) \\ & P(C_p = c_p) P\bigg(\Phi^{c_c,m}\Big(c_p, (m_1, \ldots, m_\tau)\Big) = t\bigg)\Bigg\} \end{aligned} \tag{4.6}$$

For a single record transaction, the likelihood of committing the transaction is equal to the likelihood of successfully learning the option. Given the likelihood $P(X(t) = 0)$ of having zero other updates in the time interval t, the likelihood of committing the current transaction can be expressed by multiplying the likelihood of finishing within time t and having no updates within t for all possible values of t:

$$P^{c_c,l}(\Theta = commit) = \int_0^\infty P(X(\gamma) = 0) P^{c_c,m}(\gamma)\, \mathrm{d}\gamma \tag{4.7}$$

Since w is a constant, all stochastic variables up to this point can be considered independent of the current transaction, by factoring out w from Φ and considering it as part of the time as:

$$\Phi^{c_c,l}_W(c_p, (l_1, \ldots, l_r)) = U^{c_c}(c_p, (l_1, \ldots, l_r)) + M^{c_c,l}_{propose} \tag{4.8a}$$

$$P^{c_c,l}(\Theta = commit) = \int_w^\infty P(X(\gamma) = 0) P^{c_c,m}(\gamma - w)\, \mathrm{d}\gamma \tag{4.8b}$$

Finally, in order to generalize the likelihood of committing a single record transaction to a transaction with multiple records, the likelihood that all updates are acquired successfully needs to be calculated. Assuming independence between records, this can simply be done by multiplying the likelihood of success for each record φ inside the transaction:

$$P^{c_c,(l_1 \ldots l_\varphi)}(\Gamma = commit) = \prod^{\varphi} P^{(c_c,l)}(\Theta = commit) \tag{4.9}$$

Note that $P^{c_c,(l_1 \ldots l_\varphi)}$ assumes that the current data center c_c as well as all involved leaders $(l_1 \ldots l_\varphi)$ are already known. However, in contrast to the previous transaction, this information is accessible for the current transaction because the involved records are known.

Even though it may look expensive to derive all the distributions for the various message delays and to do the actual computation, it turned out to be straightforward. This is mainly due to the fact that most of the measured statistics and convolution computations are independent of the current transaction and can be done off-line instead of online. Furthermore, some of the distributions can be simplified as the variance does not play an important role. I describe the implementation of the model and the simplifications in Section 4.5.2.

4.5.1.3 Other Protocol Models

Even though I only showed the model for the classic protocol of [50], it should be obvious that it is possible to model the conflict rate for other systems as well. For example, a model similar to PBS [11] can be used to estimate the likelihood of losing updates in a typical eventually consistent, quorum protocol as used by distributed key/value stores, Cassandra or Dynamo [33]. Furthermore, the model can be restricted to be more Megastore-like by assuming updates per partition instead of per record. Finally, the model could be adapted slightly to model more classical two-phase commit implementations by introducing extra wait delays.

4.5.2 System Statistics and Computations

I only had to make small changes to my existing implementation of MDCC in order to support the PLANET transaction programming model. Most of the changes to the system collect statistics on the characteristics of transactions. By gathering statistics on various attributes of the deployed system, the measurements can be used to calculate useful estimations such as estimated duration or commit likelihood, using the model in Section 4.5.1.

The model in Section 4.5.1 defines various required statistics. However, most of the statistics can be collected on a system-wide level and be approximated. Specifically, the distribution of the stochastic variable of equation 4.8a can be entirely precomputed for all possible master/client configurations. Afterwards, given the current number of records inside the current transaction, their leader location, and the update arrival rate per record, the computation of final probability reduces to a look-up to find the distribution of the stochastic variable from data center c_c to master l and integrating over the time as shown in

equations 4.8b and 4.9. Furthermore, in practice, the integration itself is simplified by using histograms for the statistics. In this section, I describe the required statistics.

4.5.2.1 Message Latencies

Instead of individual per-message-type statistics, I simplified the model and assumed that the message delays are similar for all message types. Therefore, the round-trip latencies between data centers are measured by sending a simple RPC message to storage nodes in all the data centers. The clients keep track of histograms of latencies for every data center. In order to disseminate this information to other clients in the system, the clients send their histograms in the RPC message to the storage nodes. The storage nodes aggregate the data from the different clients and data centers and send the information back with the response to the clients. Furthermore, I implemented a window based histogram approach [49], and aged out old round trip values, in order to better approximate the current network conditions.

4.5.2.2 Transaction Sizes

The distribution of transaction sizes is collected in a similar way as the data center round trip latencies described in the previous section. Whenever a transaction starts, the application server stores the size in a local histogram, and occasionally distributes the histograms by sending the data to some storage nodes throughout the distributed database system. The distribution of transaction sizes is useful for estimating transaction durations and is used in the conflict estimation models.

4.5.2.3 Record Access Rates

The likelihood equation 4.8b needs the likelihood of zero conflicting updates. As a simplification, I assume the update-arrival rate follows a Poisson process, so it is sufficient to compute the average arrival rate as the λ parameter.

The update-arrival rate for individual records is measured on the storage nodes. On the servers, the number of accesses to a particular record is counted with *bucket* granularity, to reduce the size of the collected data. Also, only the most recent buckets are stored for each record to reduce the storage overhead. In my implementation, the configured size of a *bucket* is 10 seconds, and only the 6 latest buckets are maintained, which is aggregated using the arithmetic mean. Using these buckets of accesses, the arrival rate described in Section 4.5.1 can be calculated without significant space overhead ([49] describes in more detail the required overhead).

4.5.2.4 Computations

Given the message latencies and the transaction size statistics, it is possible to convolute all these statistics according to equation 4.8a. The result is an $N \times N$ matrix with one entry

per data-center pair. This matrix is very compact and can be stored on every storage node and client, allowing a very efficient likelihood computation.

Whenever a transaction is started, it uses the current statistics and precomputed values, and computes the commit likelihood of the transaction, using equation 4.9. The computation is negligible, and adds virtually no overhead compared to the overall latencies between data centers. Also, whenever a new message or response is received, the client re-computes the likelihood with the new information. This allows the commit likelihood to become more accurate as more information is revealed during the commit process.

4.6 Evaluation

I evaluated PLANET on top of MDCC, which was deployed across five different data centers with Amazon EC2. My evaluation shows that (1) PLANET significantly reduces the uncertainty of transactions with timeouts, (2) speculative commits and admission control notably improve the overall throughput and latency, (3) the prediction model for MDCC is accurate enough for various conflict rates, and (4) the dynamic admission control policy generally provides the best throughput for a variety of configurations.

4.6.1 Experimental Setup

I implemented PLANET on top of my implementation of MDCC [50], and modified it according to Section 4.5.2. I deployed the system in five different data centers of Amazon EC2: US West (Northern California), US East (Virginia), EU (Ireland), Asia Pacific (Tokyo), Asia Pacific (Singapore). Each data center has a full replica of the database (five times replicated), partitioned across two m1.large servers per data center. Clients issuing transactions are evenly distributed across all five data centers, and are on m1.large servers, separate from the data storage nodes. Clients behave in the open system model, so they issue transactions at a fixed rate, in order to achieve a global target throughput. For all the experimental runs, clients ran for 3 minutes after a 2 minute warm-up period, and recorded throughput, response times, and statistics. I further configured PLANET to consider a transaction as accepted as soon as the first storage node confirmed the transaction proposal message.

4.6.2 TPC-W-like Buy Transactions

I used a TPC-W-like benchmark for all of the experiments. TPC-W is a transactional benchmark which simulates clients interacting with an e-commerce website. TPC-W defines several read and write transactions, but for my purposes, I only test the TPC-W order buying transaction. Many TPC-W transactions focus on reads, which are orthogonal to the transaction programming model. The buy transaction randomly chose 1–4 items, and purchases them by decrementing the stock levels (similar to the code shown in Listing 4.2).

To focus on database write transactions, credit card checks are ignored, and only a single Items table with the same attributes as defined in the TPC-W benchmark is used.

4.6.3 Reducing Uncertainty With PLANET

Using PLANET's *onAccept stage block* can reduce the amount of uncertainty that applications may experience. To evaluate the effectiveness, I used the buy transaction with varying timeouts from 0ms to 1500ms. The clients used a uniformly random access pattern, and the fixed client rate was set to 200 TPS, for moderate contention. The Items table had 20,000 items, and both speculative commits and admission control were disabled.

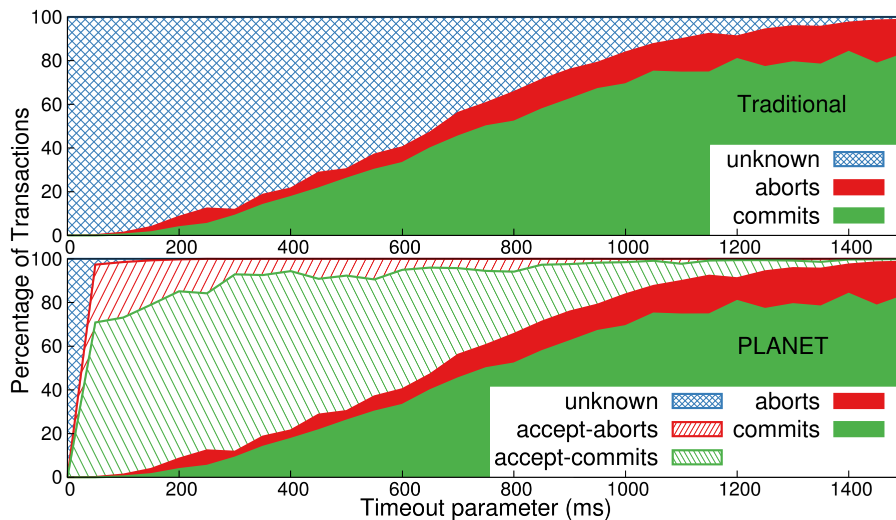


Figure 4.5: Transaction outcomes, varying the timeout (20,000 items, 200 TPS)

Figure 4.5 shows the breakdown of transaction outcomes for the different timeout values, for PLANET and a traditional JDBC model with normal timeouts.[4] The solid areas of the graph show the percentage of transactions of which the outcomes are known when the timeout expires. All other portions of the graph (striped, crosshatched) represent transactions which have not finished before the timeout. For the traditional model, there can be a large percentage of transactions with an unknown state when the timeout expires (blue crosshatched area in top graph). So, for a given timeout value, a larger crosshatched area means the application is more frequently in the dark and more users will be presented with an error.

[4]The transaction latencies are not a property of PLANET itself but of the underlying database system, MDCC. Also, all presented latencies in this chapter are not directly comparable to the latencies shown in Chapter 3 as it focuses on MDCC optimizations such as fast Paxos and commutativity, instead of the classic protocol, and uses different workloads (no contention, less load, etc).

With PLANET, transactions quickly reach the accepted stage, with the promise that they will eventually complete and the user will be informed of the final outcome. The *accept-commits* and *accept-aborts* areas of the PLANET graph are those transactions which were *accepted* before the timeout, but completed after the timeout. By being notified through *finallyCallback*, applications can discover the true outcome of transactions even if they do not complete before the timeout, and this can drastically reduce the level of uncertainty. For the red and green striped areas, the application can present the user with a meaningful message that the request was received and that the user will be notified about the final outcome, instead of showing an error message. Furthermore, in contrast to the traditional model, for the transactions which only reach the *onFailure* stage within the timeout (blue crosshatched area in the bottom graph), the finally callbacks may be invoked as long as the transaction is not actually lost due to a failure.

As Figure 4.5 shows, PLANET is less sensitive to the timeout parameter using *onAccept* and *finallyCallback*, because it allows for providing more meaningful responses to the user and for learning the transaction outcome even after the timeout.

4.6.4 Overall Performance

In order to show the overall benefits of PLANET, I used the TPC-W-like buy transaction, with the clients executing at a fixed rate, to achieve a fixed target aggregate throughput. The transactions used a 5 second timeout, and no *onAccept* stage. To simulate non-uniform access of very popular items, a hotspot access pattern was utilized, where 90% of transactions accessed an item in the hotspot. The Items table had 200,000 items and the hotspot size was varied to vary the transaction conflict rates. The PLANET system enabled admission control with the *Dynamic(50)* policy, so when the likelihood of commit, L, is less than 50%, then the transaction is attempted with probability L. The PLANET transaction also enabled speculative commits with a value of 0.95, so when the likelihood of commit is at least 95%, the transaction is considered committed.

Figure 4.6 shows the commit and abort rates for different hotspot sizes, with a client target throughput of 200 TPS. The figure shows that as the hotspot size grows (decreasing conflict rates), PLANET achieves similar throughput with the standard system, with abort rates around 1%–2% with the uniform access. As the hotspot sizes shrink (i.e., more conflicts are created), the abort rates increase because of the increasing conflict rates, and PLANET begins to experience higher commit throughput. When the hotspot is 200 items, PLANET has a commit rate of 58.2%, but the standard system only has a commit rate of 17.1%. The better commit rate is explained by the *Dynamic(50)* policy, which has the biggest impact at roughly 800 or fewer hotspot items. The effects of different admission control parameters are further studied in Section 4.6.7.

Figure 4.7 shows the average commit response times for the different target throughputs. The PLANET response times are all faster than those of the standard system, because PLANET can take advantage of faster, speculative commits. As the hotspot size increases from 200 to 6400 items, the average PLANET response times (green solid line) increase

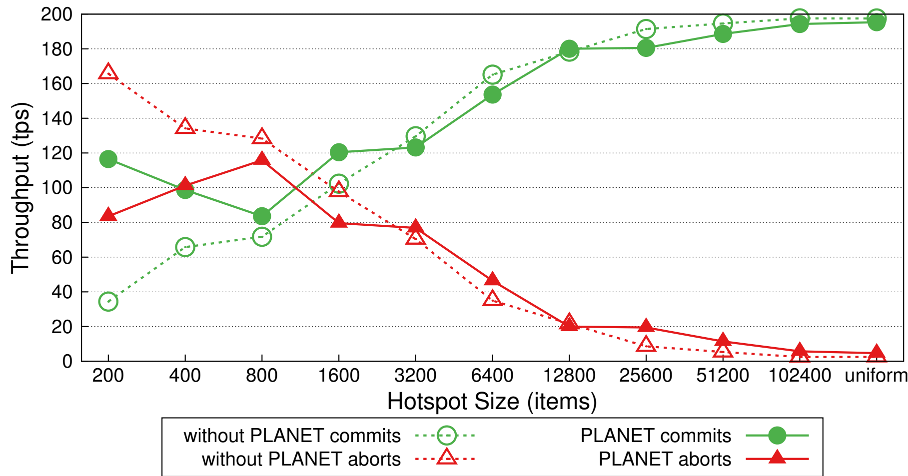


Figure 4.6: Commit & abort throughput, with variable hotspot (200,000 items, 200 TPS)

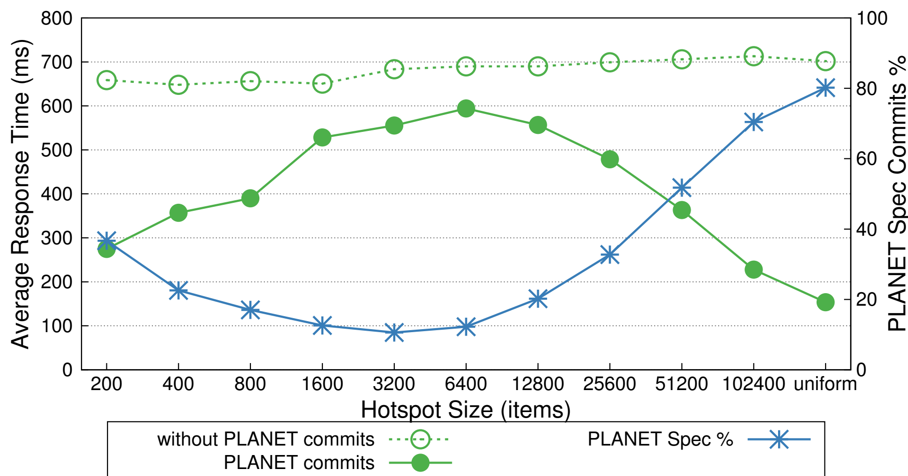


Figure 4.7: Average response time, with variable hotspot (200,000 items, 200 TPS)

because reducing the conflict rate in the hotspot increases the commit likelihoods and fewer transactions will be rejected. This means fewer transactions will be able to run in the less contended portion of the data, and be able to experience speculative commits. However, as the hotspot size increases further from 6400 items, the hotspot is then large enough where even the transactions accessing the hotspot begins to experience faster, speculative commits and the response time decreases again. Figure 4.7 also shows the percentage of commits which are speculative, and demonstrates that PLANET response times are low when a larger percentage of speculative commits are possible.

Overall, the throughput and response time are significantly improved by PLANET. The next sub-sections study in more detail the impact of contention, speculative commits and

admission control on the overall performance.

4.6.5 Performance Under High Contention

To further investigate the performance under high contention, the client request rates (Client Transaction Rate) were varied with a fix set of Items (50,000) and hotspot (100 items). Figure 4.8 shows the successful commit transaction throughput (i.e., goodput) of the PLANET system , for various client requests rates. The PLANET system outperforms the standard system for all the client throughputs and achieves up to 4-times more throughput at higher requests rates. For the standard system, the throughput peaks at around 40 TPS, where as with PLANET, the peak throughput is around 163 TPS. The abort rates for PLANET ranged from 44% to 75.8% at 600 TPS (hence, the difference between request rate and actual commit throughput). Without PLANET, the abort rates ranged from 56.7% to 94.1% at 600 TPS. Again, PLANET admission control is the main reason for the improved goodput. Admission control prevents thrashing the system, uses resources to attempt more likely transactions, improves commit throughput, and improves the goodput within the hotspot by reducing the contention.

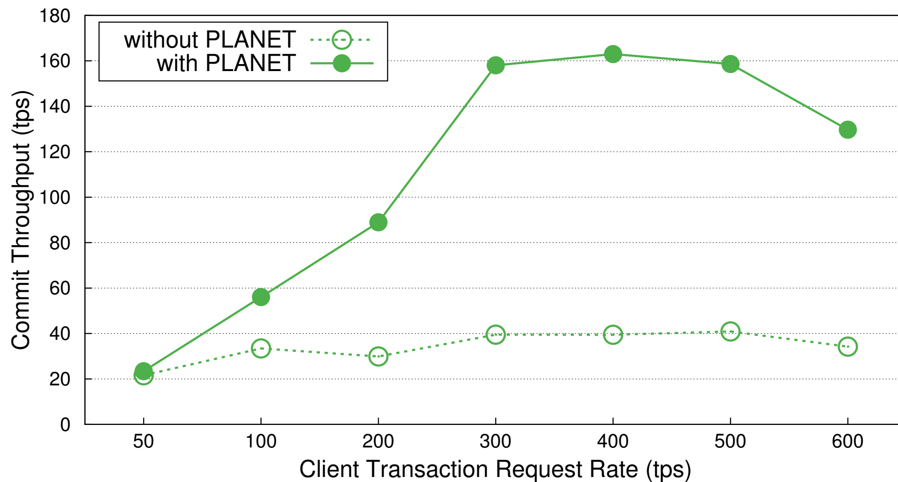


Figure 4.8: Commit throughput, with variable client rate (50,000 items, 100 hotspot)

Figure 4.9 shows the cumulative distribution functions (CDF) for committed transaction response times for experimental runs of various client request rates. It shows that the latencies for PLANET transactions are lower than the latencies for transactions not using PLANET. The main reason for the reduced response times with PLANET is that speculative commits are utilized. At 300 TPS, about 46.2% of all commits could commit speculatively, therefore greatly reducing response times. Many of the transactions not in the hotspot (cold-spot) are able to commit speculatively, with a commit likelihood greater than 0.95 because of the low contention. Therefore, the commit likelihoods of cold-spot transactions are high.

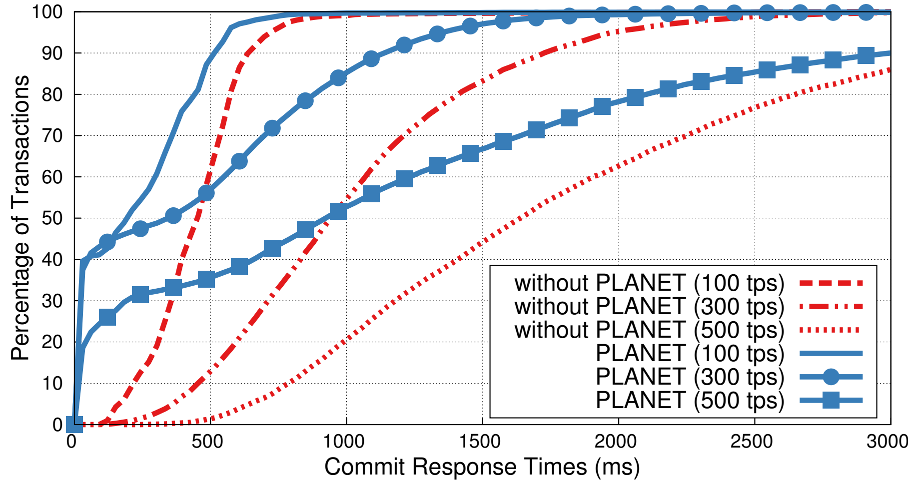


Figure 4.9: Commit response time CDF (50,000 items, 100 hotspot)

At low load of 100 TPS, 95.5% of transactions in the cold-spot could commit speculatively, and at high load of 500 TPS, about 20.2% of transactions were speculative commits. These results show that the speculative commits of PLANET can improve the response times.

4.6.6 Speculative Commits

In order to study the prediction model in more detail, I ran experiments with the benchmark, but with the transaction size at 1 item, a 5 second timeout, and no *onAccept* stage. To better evaluate only the speculative model, admission control is disabled and a more balanced contention scheme is used by selecting uniformly items from the Items table, which was varied in size from 1,000 to 10,000 items. The transactions were defined to speculatively commit when the likelihood was at least 0.95 and the client transaction request rate was set to 200 TPS.

Figure 4.10 shows the breakdown of the different commits types, for the different data sizes. In the figure, standard commits are labeled as "Normal", speculative commits are labeled as "Spec", and speculative commits which are incorrect are labeled as "Incorrect Spec". When the data size is large and there is low contention on the records (10,000 items), most of the transactions can commit speculatively. At 10,000 items, about 77.3% of transactions could commit speculatively because of the high likelihood of success. When the data size is small and there is high contention (1,000 items), most of the transactions cannot take advantage of speculative commits. At 1,000 items, only 0.1% of transactions could commit speculatively. This occurs because as contention increases, the records have higher access rates, so it becomes less likely that a transaction would have a commit probability of at least 0.95.

Since speculative commits finish the transaction early, before the final outcome, sometimes the commit can be wrong. It is clear in Figure 4.10, that the fraction of incorrect

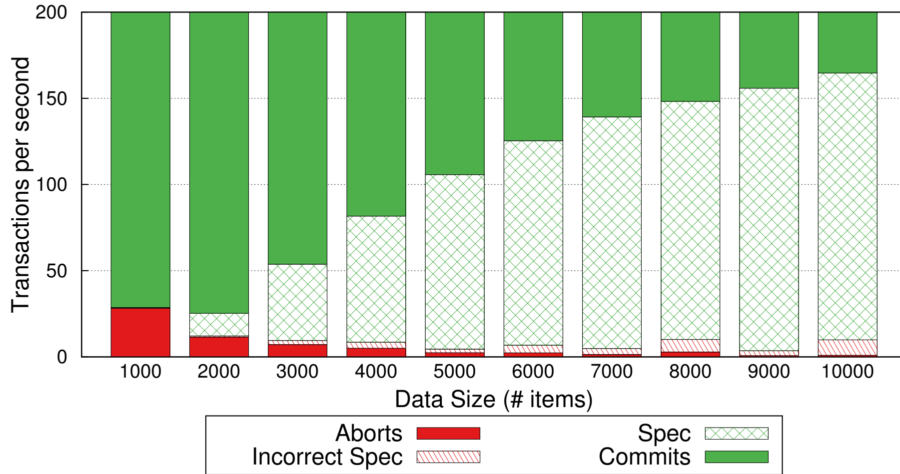


Figure 4.10: Transaction types, with variable data size (200 TPS, uniform access)

commits is not very large. The transaction defines speculative commits with a likelihood of at least 0.95, so ideally only about 5% of speculative commits would be incorrect. For all the data sizes greater than 1,000 items, the rates of incorrect speculative commits were between 1.8% and 5.8%. For 1,000 items, 25.6% of speculative commits were incorrect. The higher error rate for 1,000 items can be explained by the fact that not many transactions commit speculatively (only 39 in 3 minutes), and high contention makes it difficult to predict the commit likelihood accurately. However, most of the error rates are similar to, or better than the expected rate of 5%.

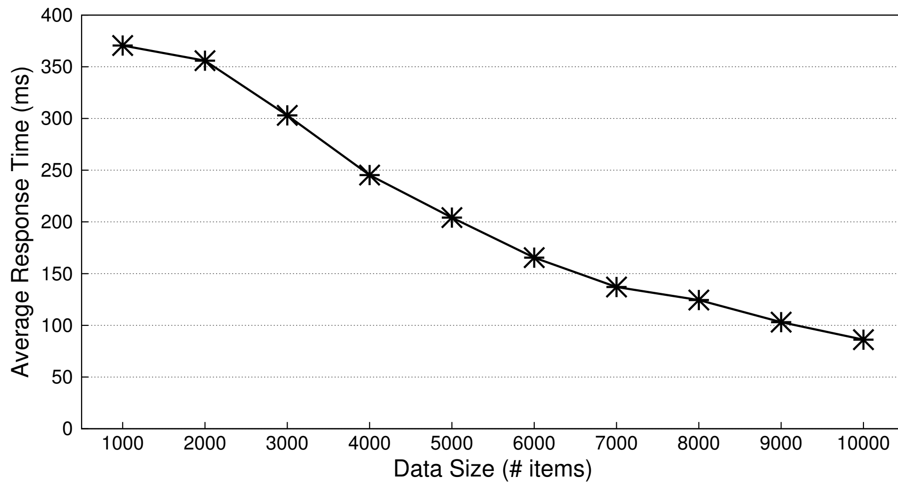


Figure 4.11: Average commit latency, with variable data size (200 TPS, uniform access)

Figure 4.11 shows the average transaction response times (including aborts) for the same

setup as Figure 4.10. The graph shows the expected: larger data sizes lower the response times because more transactions can commit speculatively. I conclude that even this simple prediction model provides enough accuracy and is able to significantly lower the total response times by using speculative commits of PLANET.

4.6.7 Admission Control

Finally, I studied the effects of PLANET's admission control more closely, by running experiments with the benchmark with smaller data sizes, and without speculative commits. Transaction size was set to 1 item, the data size was set to 25,000 items, and the hotspot size was set to 50 items. I ran the experiments with different client request rates, and varied the parameters for the *Fixed* and *Dynamic* policies, to observe how the parameters are affected by different access rates. For *Fixed(threshold,attempt˙rate)*, the *attempt˙rate* was varied for a few constant values of the *threshold*. For *Dynamic(threshold)*, the *threshold* was varied.

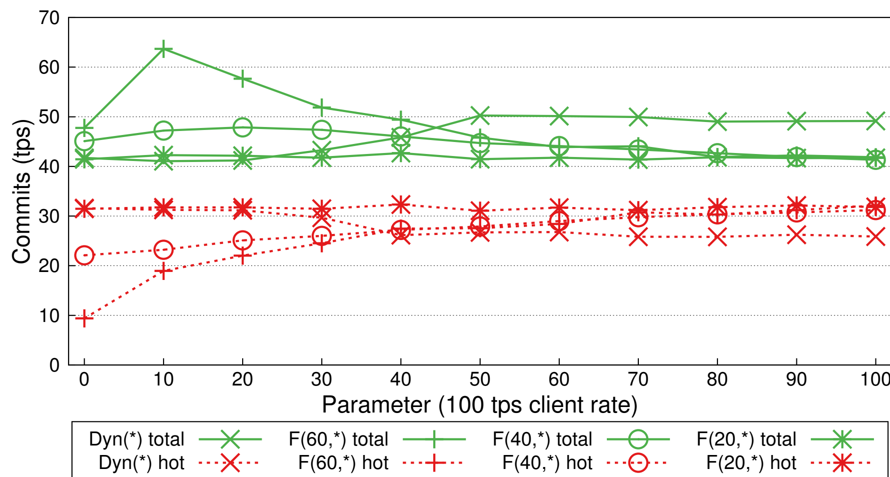


Figure 4.12: Admission control, varying policies (100 TPS, 25,000 items, 50 hotspot)

Figures 4.12 and 4.13 show the commit rates for the policies with client throughputs of 100 TPS and 400 TPS, respectively. The *Dynamic(*)*, *Fixed(20,*)*, *Fixed(40,*)*, and *Fixed(60,*)* policies were tested, represented by *Dyn(*)*, *F(20,*)*, *F(40,*)*, and *F(60,*)* in the graphs, where "∗" refers to the parameter varied (X-axis). The figures show the total commit rates (solid green lines), and the hotspot commit rates (dashed red lines), while varying parameters.

In general, for a 100 TPS request rate (Figure 4.12) all policies behave similarly. At 100 TPS, the contention level is not strong enough for the admission control policies to really show an impact. However, they are three configurations, which stand out: *Fixed(60,*)*, *Fixed(40,*)* and *Dyn(*)*. *Fixed(60,*)* and *Fixed(40,*)* overcompensates for *attempt˙rate* near 0% as they reject too many hotspot transactions causing them to drop significantly below the maximum

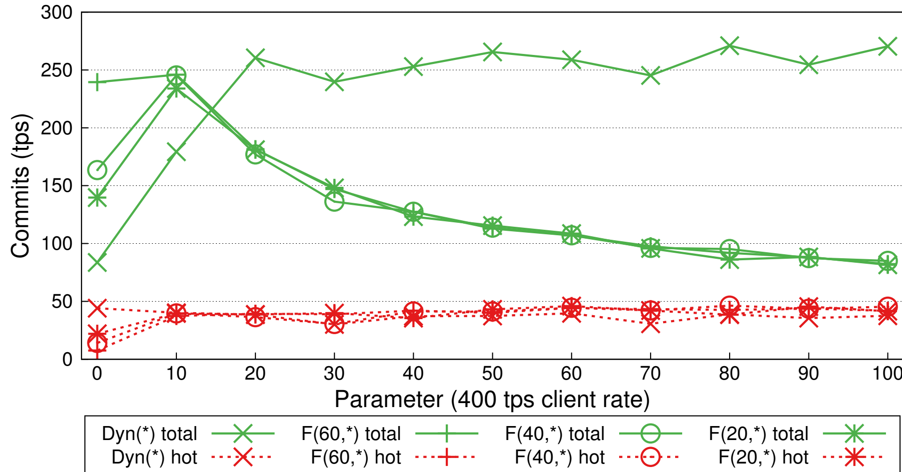


Figure 4.13: Admission control, varying policies (400 TPS, 25,000 items, 50 hotspot)

hotspot throughput of 30 TPS. However, with an attempt rate of 10%, *Fixed(60,10)* achieves the highest total throughput in this setup. The reason is, that *Fixed(60,10)* is too aggressive in rejecting low commit likelihood transactions and influences the workload to a more uniform workload, whereas the other policies still attempt and commit more transactions in the hotspot region (around 30 TPS). This is undesirable, since the admission control technique should fully utilize the hotspot instead of underutilizing it. In contrast, the *Dyn(*)* always utilizes the hotspot at around 30 TPS while providing a good overall throughput.

For a 400 TPS request rate (Figure 4.13) the situation is different. The dynamic policy performs poorly with a threshold near 0%, whereas the *Fixed* strategies do well, the high threshold (60%) in particular. The reason is simple. Recall a *Fixed(T,A)* policy means that when the commit likelihood is less than T%, the transaction will be attempted A% of the time. With a setting of *Fixed(60,0)* the admission control is most aggressive, whereas the *Fixed(60,10)* will admit some transactions accessing the hotspot and increase the overall performance. In contrast, a *Dyn(0)* policy actually refers to a setup without any admission control. It is more appropriate to compare the *Dyn(60)* point with all the data points of *Fixed(60,*)*, as a *Dyn(60)* policy means that all transactions with a likelihood L, less than 60% will be attempted L% of the time.

In general, the *Dynamic(100)* policy, which means that all transactions are tried in proportion to their commit likelihood, performed very well in both experiments. This also shows that the prediction model is accurate enough to allow the *Dynamic* policy to make good decisions. Lower thresholds for the *Dynamic* strategy essentially accept more risky transactions into the system. In the experiments, the *Dynamic* policy performed similarly for all thresholds greater than 50%.

For all configurations, using admission control always resulted in a higher total commit rate than when not using admission control. With admission control, PLANET can back off

from the hotspot, not spend resources and thrash the commit protocol, and try to execute transactions which have a much better chance to succeed (transactions not accessing the hotspot). These experiments show the sensitivity analysis for the parameters and policies, and show that the default policy for PLANET, *Dynamic* with a high threshold, works well for a range of environments.

In summary, PLANET enables application developers to write responsive applications by speculatively committing transactions and using the *onAccept* stage. Also, when the system performs admission control with commit likelihoods, resources can be spent for transactions more likely to commit, resulting in higher throughput.

4.7 Related Work

The PLANET transaction programming model enables application developers to write latency sensitive applications in high variance environments. In [43], the authors describe three types of design patterns in eventually consistent and asynchronous environments: memories, guesses, and apologies. PLANET supports these design patterns with speculative commits, and finally callbacks.

This chapter described a conflict estimation model to predict the likelihood of commit for a write-set, for an optimistic concurrency control system. Similarly, the work in [83] developed models for two-phase locking, and the models can be implemented in PLANET to compute the commit likelihoods, for PLANET to work in systems using two-phase locking.

[62, 77, 49] all use probabilistic models to limit divergence of replicas or inconsistencies with respect to caching. PLANET uses probabilistic models to predict the likelihood of commit, instead of possible data inconsistencies or divergence.

There have been many studies on transaction admission control, or load control in database systems. In [42], the “optimal” load factor is approximated by adaptively probing the performance of the system with more or less load, and admission control prevented thrashing of the system. [22, 59, 82] have all studied the effects of thrashing and admission control with two-phase locking concurrency control. These solutions require keeping track of the global number of transactions or locks held by transactions to make admission control decisions, which is difficult in geo-replicated distributed database systems. The authors of [37] implemented a proxy for admission control by rejecting new transactions which may surpass the system capacity computed offline. PLANET differs from these solutions by using commit likelihoods to make decisions on admission control.

There have been previous proposals for system implementations that perform optimistic commit, sometimes needing compensation if the optimistic decision was wrong [55, 47]. PLANET is orthogonal to this, with speculative commits at the language level that allow the application programmer awareness of the commit likelihood, so a principled decision can trade-off the benefits of fast responses against the occasional compensation costs. Any optimistic commit protocol could be used to implement the PLANET model, so long as PLANET can predict the probability of eventual success.

Several systems support time-outs for transactions, such as JDBC drivers or Hibernate, but they only support simple timeouts with no further guarantees. Also, with most models, after the timeout expires, the transaction outcome is unknown without an easy way to discover it. Some models allow setting various timeouts for different stages of the transaction (e.g., Galera, Oracle RAC), but how the timeouts effect the user application is not obvious. In contrast, PLANET provides a solid foundation for developers to implement highly responsive transactions using the guess and apology pattern, as well as minimizes the uncertain state for which the application does not know anything about the transaction.

4.8 Conclusion

High variance and high latency environments can be common with recent trends towards consolidation, scalable cloud computing, and geo-replication. Transactions in such environments can experience unpredictable response times or unexpected failures, and the increased uncertainty makes developing interactive applications difficult. I proposed a new transaction programming model, Predictive Latency-Aware NEtworked Transactions (PLANET), that offers features to help developers build applications. PLANET exposes the progress of the transactions to the application, so that applications can flexibly react to unexpected situations while still providing a predictable and responsive user experience. PLANET's novel commit likelihood model and user-defined commits enable developers to explicitly trade-off between latency and consistent application behavior (e.g., apologizing for moving ahead too early), making PLANET the first implementation of the previously proposed *guesses* and *apologies* transaction design pattern [43]. Furthermore, likelihoods can be used for admission control to reject transactions which have a low likelihood to succeed, in order to better utilize resources and avoid thrashing. I evaluated PLANET in a strongly consistent, synchronous geo-replicated system and showed that using the speculative commits and admission control features of PLANET can improve the throughput of the system, and decrease the response times of transactions.

When implemented in a strongly consistent, synchronous geo-replicated system, PLANET can offer the lower latency benefits of eventual consistency. While eventually consistent systems choose to give up data consistency or multi-record transactions for improved response times, PLANET can improve transaction response times while still keeping data consistent.

Chapter 5

A New Scalable View Maintenance Algorithm

5.1 Introduction

In Chapter 3 and Chapter 4, I described new techniques of supporting and interacting with scalable transactions for distributed database systems. However, applications also need to read and query the transactional data in an effective way. Modern distributed partitioned stores commonly split up their data in order to handle the growing size of data and query load in workloads. Partitioning data and load across many machines is an effective technique to scale out, and because of this nearly limitless scalability, partitioned stores have become very popular. There are two commonly used types of partitioned stores: sharded transactional systems, and distributed key-value stores.

Sharded transactional systems [20, 31, 13] partition the data across multiple instances of transactional database systems, thus allowing efficient queries and transactions within a single partition. For example, Facebook uses a large array of MySQL instances to store and serve its social graph data. Distributed key-value stores [24, 33, 30, 6, 7, 58] also partition their data, but limit operations to single rows, or key-value pairs.

Partitioning data is a common technique to scale data stores, however, there are some limitations for applications using these partitioned stores. For sharded database systems, the transactions are usually limited to a single partition, and cross-partition transactions require heavy coordination that limits scalability, and are strongly discouraged. Also, joins are only supported within a single partition, and cross-partition joins are done by external applications. Key-value stores have similar limitations, where transactions or updates are only supported per single key-value pair, and joins have to be performed by applications.

When an application needs joins using partitioned stores, it either has to perform the join itself, or issue the join for a single partition, and neither scenario is ideal. Using materialized views is widely used in traditional database system designs to make joins easy and effective to evaluate. Materialized views are precomputed results of queries and they can be

an effective way to support global joins without requiring applications to re-implement a join algorithm. Materialized join views allow applications to query the join results directly for reduced response times, and can also help with data warehousing and analytics tasks [40]. However, most partitioned stores do not natively support materialized views, mainly because maintaining views may require expensive, cross-partition transactions. In order for applications to use materialized views, the applications themselves must deal with maintaining them, and if they are not careful, anomalies and inconsistencies will arise in the views.

Surprisingly there has been little work on extending incremental view maintenance techniques to distributed and scalable environments. There are many incremental view maintenance techniques [23, 17, 68, 41] that assume single-server database systems or require global transactions. These techniques are not directly applicable to distributed settings, because distributed transactions are expensive or not always available, and increasing the size of transactions will likely cause more conflicts and limit the scalability of the system. Related data warehouse techniques [87, 2, 84, 26] depend on a single server to process all the updates and maintain all the views, so this can become a bottleneck and limit the scalability. In summary, existing techniques for maintaining views are not scalable to be appropriate for modern distributed database systems.

In this chapter, I propose SCALAVIEW, a new technique for scalable maintenance for Select-Project-Join (SPJ) views. I investigate view maintenance for partitioned stores in a distributed and scalable environment, how to achieve convergence consistency, and how to reduce view staleness. Perfect, complete consistency is not a goal of this work. Instead, a view is allowed to be a little stale, but reducing this staleness is a goal. However, views should exhibit convergence consistency, which means that the correct state will eventually appear in the view. Because partitioned stores do not natively support views and applications must maintain views, I study the types of anomalies that can prevent convergence consistency, and new techniques that prevent and correct the anomalies.

I describe the goals of a scalable view maintenance algorithm in Section 5.3. Section 5.4 describes some existing solutions for view maintenance. Potential anomalies of scalable maintenance are discussed in Section 5.5. Design decisions and components of SCALAVIEW are described in Section 5.6 and the algorithm is presented in Section 5.7. SCALAVIEW is evaluated in Section 5.9, and I discuss other related work in Section 5.10.

5.2 Motivation

Partitioned stores have become very popular for their scalability properties. By adding more servers to the configuration of the system, the system can quickly support more data storage, and more query workload volume. However, if materialized views are not available, applications may no longer be scalable even when using scalable partitioned stores [10]. Therefore, materialized views can be very beneficial for many workloads.

In this section, I describe a simple example of when a materialized view would be beneficial. A Twitter-like micro-blogging service has *users* who can *post* content, as well as *follow*

other users. The database schema would look like the following:

```
Users(uid INT, name STRING)
Posts(uid INT, ts TIMESTAMP, post STRING)
Follows(uid1 INT, uid2 INT)
```

Each time a user, *user* wants to see the stream of posts from their following users, an example of the join query that would have to be issued is:

```
SELECT * FROM Posts P, Follows F
WHERE F.uid1 = user AND F.uid2 = P.uid
```

Since many modern scalable partitioned stores force the client application to execute joins, it would have to query the *Follows* table for all the following users, and then query *Posts* for the posts for each of the following users. This type of query access could become expensive, especially when there are many users trying to read their own streams. However, by using a materialized view, this access could become a much easier query, because the view would essentially be a cache of the join. The following is the schema for a materialized view table.

```
Streams(uid1 INT, uid2 INT, ts TIMESTAMP,
        post STRING)
```

This view precomputes all the streams for all of the users, so when a particular user wants to see their stream, the query is a simple lookup in the view table, with the query:

```
SELECT * FROM Streams S WHERE S.uid1 = user
```

By using a materialized view, each user simply has to issue this query to the view, instead of having to compute a join each time. The application can avoid imposing scalability limitations to the underlying system, by avoiding computing the join for each user. Unfortunately, current distributed database systems do not have scalable solutions for maintaining materialized views. Therefore, developing a scalable mechanism for maintaining materialized join views would benefit many distributed database systems.

5.3 Goals for Scalable View Maintenance

Modern partitioned stores are becoming increasingly popular, mainly for their scalability and fault tolerance properties. When designing an algorithm for view maintenance in a scalable system, it is important to not negate the benefits of the underlying system. In this section 5.3, I discuss specific goals for a scalable view maintenance algorithm, so that it does not limit the scalability or fault tolerance properties of distributed database systems.

5.3.1 Deferred Maintenance

One of the main goals is to maintain materialized views asynchronously from the base update. That is, no global transaction includes both the base changes and the view changes. As opposed to early materialized view work [23, 17] that included the view maintenance operations in the base transaction, deferred maintenance [28, 75, 88] decouples the view operations from the base transaction.

Deferred maintenance is useful because some applications may be able to tolerate the staleness of views due to the asynchronous nature of the maintenance. This is particularly true for data warehouse applications, since updates to the data warehouse are usually deferred, and sometimes batched up for an entire day.

Also, for many large-scale distributed database systems, arbitrary transactions are not supported. Some transactions are limited to either a single record, or a single partition. Therefore, including view maintenance operations in the base transaction is often not even possible. However, even if distributed transactions are supported, coupling view maintenance operations with the base transaction may not always be desirable. By adding more operations to the base transaction, it will increase the read and write sets of the transaction, and result in more contention, greater conflict rates, and slower response times. Therefore, deferred view maintenance is a goal for a scalable algorithm.

5.3.2 Scalable & Distributed

Since the materialized views are in a scalable, distributed data store, the view maintenance algorithm should also be scalable and distributed. Distributed systems allow the base data to be easily scalable to handle larger amounts of data and load. However, if the view maintenance algorithm is not scalable, the maintenance could become a bottleneck and may not be able to keep up with the aggregate throughput. Therefore, the maintenance algorithm should be distributed and scalable.

To be able to support scalable view maintenance, the algorithm should not depend on global state or global ordering. Requiring global ordering could limit the scalability, since this is usually achieved by a single point of serialization, coordination, or locking. A single point of serialization or ordering can also adversely affect the fault tolerance of the system. Therefore, the view maintainers should be distributed across multiple machines, and views should not have to be maintained at a single server like a traditional data warehouse.

5.3.3 Incremental Maintenance

Another goal for scalable view maintenance is for the maintenance to be incremental, and to avoid re-computation. Since materialized views are like precomputed results of queries, one way to maintain the views is to re-compute the entire result. However, re-computation should be avoided since it could be very expensive, especially for large tables and large join views.

By just examining and updating the data that changed, incremental view maintenance can be very quick and cost effective.

5.3.4 Reduce Staleness

Another important goal is to reduce staleness of the materialized views. Since views will be updated asynchronously, there will always be some delay between the base data update and the view update. However, the goal is to try to reduce the propagation delay from the base update to the view. The experiments in Section 5.9 focus on measuring this staleness.

5.3.5 Convergence Consistency

Considering the goals for scalable, distributed, and asynchronous view maintenance of joins, maintaining consistency of the views is more difficult. In this environment, global transactions, global ordering or global state in the distributed system are not available, so much of the previous work on techniques for consistent materialized views are not directly applicable. However, previous work on consistency of materialized views [88] have presented the following varying levels of consistency.

Convergence After all activity has ceased, the state of the view is consistent with the state of the base data.

Weak Consistency Every state of the view is consistent with some valid state of the base data. However, it may be in a different order.

Consistency Every state of the view is consistent with a valid state of the base data, and they occur in the same order.

Completeness There is a complete, order-preserving mapping between the base data states and the view states.

In this work, the goal is for the materialized views to have **convergence** consistency. There are use cases of views that can tolerate convergence consistency. For example, if there is reporting or aggregation done on a large result, it may not be important to have the exact answer, but a close estimate may be enough. However, accumulating errors would still not be acceptable, so convergence is a useful property. In Section 5.5, I show threats to convergence in this scalable, distributed, and asynchronous setting, and how I propose to prevent them.

5.3.6 Reduce Data Amplification

The amount of space overhead required to maintain views is another aspect to consider. The amount of data amplification, or extraneous data that is not the base data, can be quite large when dealing with massive data sets. Therefore, a solution should not require too much extraneous data in order to correctly maintain materialized views.

5.3.7 Provide General Solution

The solution should provide a general framework for materialized views. This means, the requirements from the underlying system should not be so great as to exclude too many existing distributed database systems. Also, strict assumptions in the workload should be avoided. For example, there are prior solutions for only equi-join views, or local indexes, or domain-specific web indexes. However, I seek a solution that is more general and applicable to a wider range of underlying systems and use cases.

5.4 Existing Maintenance Methods

There are many methods for maintaining materialized views in databases, and in this section, I discuss two classes of techniques applicable for scalable partitioned stores.

5.4.1 Centralized Algorithm

Most view maintenance algorithms are centralized algorithms that either execute on the same server as a single-server database system, or execute on a single data warehouse installation. In both cases, the maintenance algorithm has access to certain global state of the system. For distributed database systems, the data warehouse techniques are most applicable. In these algorithms, all the updates in the system are sent to the data warehouse, where a sequencer assigns a global timestamp to every message. Then the algorithms use this stream of updates to maintain views either single-thread as STROBE [87] or SWEEP [2], or multi-threaded as POSSE [61] or PVM [84]. Even the multi-threaded algorithms need to synchronize access to global state to provide correct views. Therefore, this synchronization could become a bottleneck, thus limiting scalability.

5.4.2 Co-Located Indexes

An existing method for scalable, distributed view maintenance for equi-joins is a co-locating scheme, similar to the methods found in PNUTS [3] from Yahoo, or Lynx [85] from Microsoft Research. With this method, intermediate indexes are created and maintained for each table involved in the join. The indexes are partitioned on one of the join keys, so that matching records from different tables are co-located, so executing a local join is possible. After the local join, additional indexes may have to be maintained and re-partitioned for remaining join keys specified in the view.

This is a simple technique that maintains join views in a distributed and scalable way. However, there are some limitations that suggest opportunities for a different solution. First of all, this technique only works for equi-joins and is not general enough to support other join conditions. Also, since re-partitioning must co-locate matching records for a join key, skew in the data could be problematic. If a co-located partition cannot fit on a single machine, multi-partition updates or joins would be required, and that aspect has not been investigated

by previous work. Another potential drawback is that intermediate indexes and results must be maintained in addition to the base tables and the materialized join view. Therefore, data duplication required for the scheme to work could potentially be high, and this could be a limiting factor for large distributed deployments.

5.5 Possible Anomalies

The main idea behind incremental view maintenance techniques is to avoid re-computation of the entire view, which could be quite costly. In typical incremental algorithms such as ECA [88], STROBE [87], or SWEEP [2], queries are issued in response to an update to a base table. These queries access other base tables required for the join view. However, due to the asynchronous nature of the view maintenance, anomalies are possible that threaten convergence unless extra techniques are used. In this section, I describe the types of anomalies that must be prevented during distributed view maintenance.

5.5.1 Incremental Maintenance Operations

A base table can be modified with an insert or a delete operation. A table can also be modified with an update, but for the purposes of this thesis, an update is modeled as a delete followed by an insert. Incremental view maintenance operations for base inserts and deletes are below.

5.5.1.1 Insert

When a base record is inserted, the view maintainer must perform operations to determine which records to insert into the view. This set of view maintenance operations is called **Insert-Join**. For the **Insert-Join** maintenance task, the maintainer first issues queries to the other tables involved in the join, using data from the base insert. After all the query responses, the view maintainer joins the records from the responses to construct the view records to insert. Finally, the set of view records is inserted into the view. In pseudocode listing 6, lines 2–10 show the pseudocode for **Insert-Join**.

5.5.1.2 Delete

When a base record is deleted, the view maintainer must perform operations to determine which records to delete from the view. There are two different methods of determining which records to delete, **Delete-Join** and **Delete-Direct**.

The **Delete-Join** task is similar to the **Insert-Join** task. The maintainer first issues queries to the other tables involved in the join, using data from the base delete. The maintainer joins the records from the responses to construct the view records to delete. Listing 6 lines 11–19 show the pseudocode for **Delete-Join**.

Algorithm 6 View Maintenance Tasks

```
1: V : T_1 ⋈ ... ⋈ T_n

2: procedure ProcessInsertJoin(ΔT_x)
3:     ΔV ← ΔT_x
4:     for i ≠ x do
5:         Query T_i for records matching ΔV
6:         Receive ΔT_i
7:         ΔV ← ΔT_i ⋈ ΔV
8:     end for
9:     Insert ΔV into View V
10: end procedure

11: procedure ProcessDeleteJoin(ΔT_x)
12:     ΔV ← ΔT_x
13:     for i ≠ x do
14:         Query T_i for records matching ΔV
15:         Receive ΔT_i
16:         ΔV ← ΔT_i ⋈ ΔV
17:     end for
18:     Delete ΔV from View V
19: end procedure

20: procedure ProcessDeleteDirect(ΔT_x)
21:     Read View V for records matching ΔT_x
22:     Receive ΔV
23:     Delete ΔV from View V
24: end procedure
```

An alternative method of deleting view records is to use the information from the deleted base record, to directly delete from the view table. This **Delete-Direct** task reads the view table to find view records that joins with the delete record, and then deletes them. Listing 6 lines 20–24 show the pseudocode for **Delete-Direct**.

5.5.2 When Anomalies Are Possible

Section 5.5.1 describes the basic view maintenance tasks for incrementally updating views. If these tasks were executed in a single transaction with the base insert or delete, and the transactions were executed with a serializable schedule, then there would be no anomalies or inconsistencies with the database. However, maintaining views in base transactions is not desirable because it adversely affects the base update rate, and transactions are not available

in many modern distributed database systems. Therefore, the view maintenance tasks are logically part of the base transactions, but executed without any transaction mechanisms such as locking or coordination, and this can lead to anomalies.

View anomalies between concurrent view maintenance tasks are a threat when there are conflicting operations and there is a cycle in the conflict dependency graph [16]. When the read-write, write-read, or write-write conflicts between maintenance tasks produce a cycle in the dependency graph, there is no equivalent serializable schedule, so anomalies would be present. Simply executing the view maintenance tasks without special handling will not be able to prevent or correct the anomalies, since no locking or coordination mechanisms are utilized.

5.5.3 Example of an Anomaly

In this section, I describe how a possible anomaly may happen. Say there are two relations R and S and the view V is defined as $R \bowtie S$. R starts as empty and S contains a single tuple. The schema and contents of tables R and S are shown below:

R

a	b

S

b	c
2	3

V

a	b	c

The following sequence is possible, which leads to an anomaly in the view. For the notation, up_i denotes a base update i, and v_j denotes a view maintenance task j triggered by a base update up_j.

1. up_1 inserts a new record $(1, 2)$ into R, then triggers the corresponding view task v_1.
2. v_1 reads S for matching records for the inserted record $(1, 2)$, and receives S tuple $(2, 3)$.
3. up_2 deletes record $(2, 3)$ from S, then triggers the corresponding view task v_2.
4. v_2 reads R for matching records for the deleted record $(2, 3)$, and receives R tuple $(1, 2)$.
5. v_2 attempts to delete joined record $(1, 2, 3)$ from V, but nothing happens since it does not exist.
6. v_1 inserts joined record $(1, 2, 3)$ into V.
7. The final result of the view V contains $(1, 2, 3)$, but it is not consistent with the base tables R and S.

This sequence of events, or history, leads to an anomaly in the view, because the resulting view V not equivalent to $R \bowtie S$. If you consider up_1 and v_1 as a "logical" transaction T_1, and up_2 and v_2 as a separate "logical" transaction T_2, the conflicts can be visualized. The

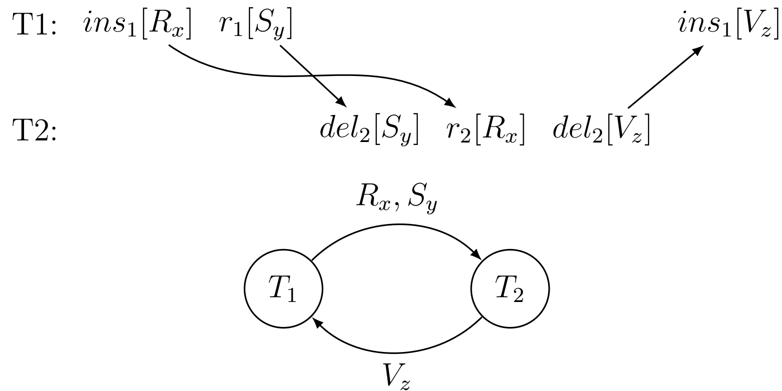


Figure 5.1: History and corresponding conflict dependency graph for example anomaly

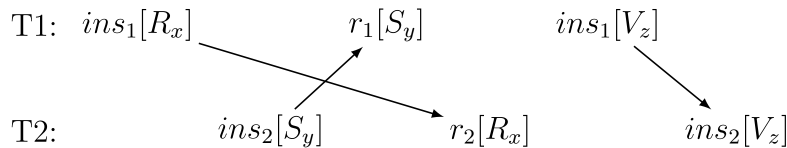
history of operations is shown in figure 5.1. In denoting the sequence of operations, the notation $ins_1[R_x]$ (or $del_1[R_x]$) means transaction T_1 inserts (or deletes) record x in table R, and $r_2[S_y]$ means transaction T_2 reads record y from table S. T_1 writes R_x before T_2 reads R_x, and T_1 reads S_y before T_2 write S_y. However, T_2 writes V_z before T_1, so the conflict dependency graph looks like figure 5.1. There is a cycle in the conflict dependency graph, so this sequence of operations results in an anomaly.

5.5.4 Classifying Anomalies

To investigate what types of anomalies are possible and when they may occur, all the combinations of two concurrent view maintenance tasks are considered, and examine the schedules with cycles in the conflict graphs. In this section, I describe the different types of anomalies possible and how they may occur.

5.5.4.1 Insert-Join & Insert-Join

When two concurrent **Insert-Join** tasks, T_1 and T_2, both read the other transaction's base insert, the conflict schedule looks like this:



Both T_1 and T_2 will insert the same record V_z into the view. This anomaly is called **Duplicate-Insert-Anomaly**.

Another type of cycle and anomaly is possible when the **Insert-Join** task uses an asynchronous global index to read other values from other tables. When reading from an asyn-

chronous global index, it is possible that maintenance tasks read stale versions of the index, and that can cause this schedule:

T1: $ins_1[R_x]$ $r_1[S.idx_y]$ $ins_1[R.idx_x]$

T2: $ins_2[S_y]$ $r_2[R.idx_x]$ $ins_2[S.idx_y]$

When both **Insert-Join** tasks read the stale values from the global indexes, they both do not read any rows to use to eventually insert into the view. Therefore, nothing will be inserted into the view, and that leads to the **Stale-Index-Anomaly**.

5.5.4.2 Insert-Join & Delete-Join

For concurrent **Insert-Join** T_1, and **Delete-Join** T_2, multiple conflict graphs with cycles are possible. The first cycle is caused by this schedule:

T1: $ins_1[R_x]$ $r_1[S_y]$

T2: $del_2[S_y]$ $r_2[R_x]$ $del_2[V_z]$

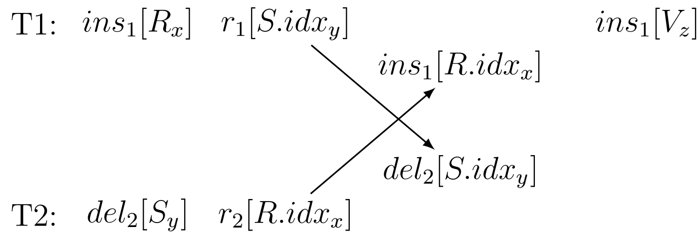
In this schedule, the **Insert-Join** T_1 does not read record S_y because it is deleted, so T_1 does not update the view. T_2 reads R_x and attempts to delete the view record V_z, but it does not exist, since it was never inserted by T_1. This is called a **Delete-DNE-Anomaly**. A separate conflict schedule causes a different cycle:

T1: $ins_1[R_x]$ $r_1[S_y]$ $ins_1[V_z]$

T2: $del_2[S_y]$ $r_2[R_x]$ $del_2[V_z]$

With this schedule, **Delete-Join** T_2 tries to delete the view record V_z before T_1 inserts the record. Therefore, when T_1 inserts the view record V_z, it will remain in the view even though it should not exist. This is called **Late-Insert-Anomaly**. Section 5.5.3 shows an example of this anomaly.

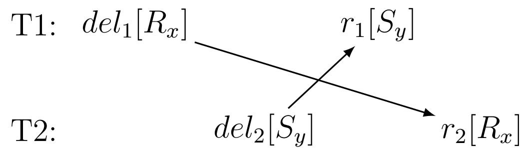
If asynchronous global secondary indexes are read for querying other tables, another anomaly is possible.



With this conflict cycle, both the view maintenance tasks do not observe the results of the other task, so the **Insert-Join** task inserts the view record, but the **Delete-Join** does not delete the record. This causes a **Stale-Index-Anomaly**.

5.5.4.3 Delete-Join & Delete-Join

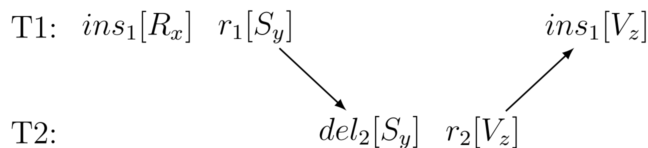
Below is the schedule with a cycle for two concurrent **Delete-Join** tasks.



This occurs when both tasks try to read records after they have already been deleted, so both tasks determine that nothing should be deleted from the view. This called a **Missing-Delete-Anomaly**.

5.5.4.4 Insert-Join & Delete-Direct

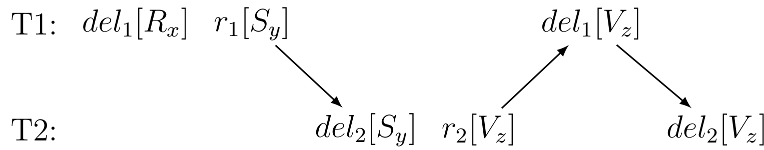
The history below shows the schedule that produces a cycle in the dependency graph.



This anomaly occurs when **Insert-Join** reads the record before it is deleted, but inserts the record into the view after the **Delete-Direct** task tries to delete it. Therefore, the view record remains inserted even though the corresponding base record has been deleted. This anomaly is called **Early-Read-Anomaly**.

5.5.4.5 Delete-Join & Delete-Direct

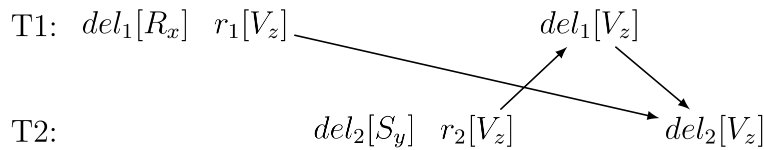
This history below shows the schedule with a cycle for concurrent **Delete-Join** and **Delete-Direct** tasks.



In this schedule, both tasks attempt to delete the same view record, so this is called a **Duplicate-Delete-Anomaly**.

5.5.4.6 Delete-Direct & Delete-Direct

This history below shows the schedule with a cycle for two concurrent **Delete-Direct** tasks.



In this schedule, both tasks attempt to delete the same view record, so this is called a **Duplicate-Delete-Anomaly**.

5.6 Scalable View Maintenance

In order to support scalable view maintenance of SPJ views for modern, scalable partitioned stores, solutions should not have to depend on a centralized server, algorithm, or a serialization point. While distributed systems can easily scale by adding capacity, centralized servers or serialization services may lead to bottlenecks in the system. This is undesirable because bottlenecks could prevent the view maintenance from keeping up with the load, with no way of increasing capacity. Therefore, scalable solutions should be distributed and not depend on centralized algorithms or global state. By allowing distributed execution, but still preventing and correcting the anomalies described in Section 5.5.4, SCALAVIEW enables scalable maintenance for join views. In this section I discuss components for the SCALAVIEW solution for maintaining materialized views.

5.6.1 System Architecture Requirements

In order to support scalable maintenance of join views, I made several design decisions for the system architecture. In this section, I describe the different parts of the system that enable the SCALAVIEW algorithm.

5.6.1.1 Partitioned with Record Monotonicity

The SCALAVIEW technique for view maintenance is designed for a horizontally partitioned distributed database system that supports monotonicity per record. These properties are found in most modern partitioned stores.

5.6.1.2 Transactions not Supported or Used

Distributed transaction support is assumed to unavailable, or transactions are not used if the underlying system does supports them. Many scalable systems do not support full, arbitrary transactions, so SCALAVIEW does not depend on them. Also, asynchronous maintenance is one of the major goals for a scalable algorithm. Even if transactions are supported by the system, they are not utilized, because including view operations in the base transaction will increase the conflict footprint of the transaction. Increasing contention can negatively affect the concurrency of base updates to the database.

5.6.1.3 No Centralized Serialization or Global State

In order to achieve scalable view maintenance, SCALAVIEW does not depend on a centralized server to process update, or any globally consistent state of the entire system. Therefore, there is no total ordering of messages or updates. By not requiring global state, all the processing does not have to occur on a single machine, which could lead to bottlenecks in the system.

5.6.1.4 Primary Keys and Versions

Every table must define a primary key and every record has a record version number. Materialized join views must also have primary keys and record versions. However, for each join view record, the primary key and the record version are inferred from the base records that make up the joined view record. Therefore, for all records in a join view, the composite primary key is constructed from the keys of the base records, and the composite record version is constructed from the base record version numbers.

5.6.1.5 Independent Maintenance per Record

When any base record in the database is updated, a trigger is asynchronously initiated to execute a view maintenance task. The view maintenance task processes the updates in order for a particular record, and issues additional queries to incrementally update views. The updates have to be processed in order, but only within each record. Each view maintenance task is independent from the tasks of any other record, so at any given time, there could be many tasks executing on many different servers, for different records in the system. Therefore, SCALAVIEW is scalable since there is no centralized view maintainer, nor a global ordering of updates.

5.6.2 Components Techniques for Scalable View Maintenance

The distributed view maintenance triggers described in Section 5.6.1 allows for scalable maintenance of join views, but the anomalies described in Section 5.5 are still possible.

Therefore, techniques are needed in order to prevent and correct the anomalies. The foundation of SCALAVIEW is a simple compensation approach, similar to common data warehouse techniques [88, 87, 2]. In the rest of this section, I discuss additional details of the components of SCALAVIEW to address the anomalies that may arise in a distributed setting without depending on global state or coordination (proofs can be found in Section 5.8).

5.6.2.1 View Record Composite Primary Key

Since every record has a primary key, each view record is uniquely identified. Therefore, all materialized join views do not have duplicates. Since every record is uniquely identified, there is no ambiguity in inserting or deleting a view record. Using primary keys from base records to avoid duplicates in join views has been proposed in previous work [17].

Since every insert or delete is uniquely identified, that means duplicate inserts and deletes can be detected. Therefore, if a particular record is inserted multiple times, the SCALAVIEW algorithm will only insert the record once, since a record cannot be inserted multiple times. Multiple deletes of a record are resolve in the same say, so using the primary key prevents both **Duplicate-Insert-Anomaly** and **Duplicate-Delete-Anomaly**.

5.6.2.2 View Record Tombstones

Using delete tombstones is a mechanism for dealing with out-of-order concurrent updates to the same join view record, and prevents the **Delete-DNE-Anomaly** and **Late-Insert-Anomaly**. These anomalies occur because a concurrent insert and delete are attempting to update the same view record, but the operations occur in a schedule that is not conflict serializable. These schedules are described in Section 5.5.4. These types of anomalies occur because the deleting maintenance task attempts to delete a view record that does not exist. So, instead of causing a potential anomaly, if the view record to delete does not exist, a delete tombstone is created. Delete tombstones are specific to a particular version of the record. Therefore, for a particular record, there may exist delete tombstones for some of the versions, while non-existent for others.

When reading a particular version of a record and a delete tombstone exists, the record is simply not read. When a concurrent insert is processed for a version of a record and the delete tombstone already exists, the insert is ignored, because the tombstone is signifying that the record should not exist. If the insert for a particular record version arrives after the delete tombstone, this means that there was an out-of-order execution of the operations on the same version of the record. Therefore, the insert should be ignored, and the record should not exist. As a result, **Delete-DNE-Anomaly** and **Late-Insert-Anomaly** are both handled by using delete tombstones.

5.6.2.3 Re-Read the Read-Set

Another technique for handling some of the possible anomalies is to re-read the read set for the **Insert-Join** task and *compensate* for the differences in the two read sets. With certain

schedules of operations, a **Insert-Join** will insert view records, while a concurrent **Delete-Direct** may not be able to delete the records, and results in the **Early-Read-Anomaly**.

In the standard **Insert-Join** task, it reads records from other tables involved in the join (read set RS_1), joins the matching records to compute the view records, and finally inserts the records into the view. When using the re-read technique, after the task writes records into the view, it then reads it read set again, into RS_2. Then, the first read set, RS_1, and second read set, RS_2, are compared, and any records in the first read set which does not appear in the second read set are gathered as $RS_{deleted} = RS_1 - RS_2$. These records which appear in RS_1 and not RS_2 are records which were deleted concurrently, but could potentially remain in the view. Then, the task uses $RS_{deleted}$ to compute the join view records, and then *deletes* those records from the view. The view records computed from $RS_{deleted}$ are view records which were already inserted by this task, but should not remain the view because of other concurrent deletes in the system. The view records deletes could result in duplicate deletes or deletes that do not exist, but those issues are handled by using the primary keys. This is similar to existing compensation techniques [88, 87, 2, 84, 26], but without having to depend on centralized state or update timestamps.

5.6.2.4 Query Old Versions

Without coordination, concurrent **Delete-Join** tasks can cause a **Missing-Delete-Anomaly**, leaving view records which should not remain. In order to prevent anomalies to remain in the materialized join view, **Delete-Join** tasks use a new mechanism to query old versions. Whenever **Delete-Join** tasks query other tables involved in the join, instead of reading the latest versions of the records, the tasks request older versions as well, and include them in their read set. The task uses this new read set in the same way as before, for querying other tables for matching records, and to compute the join view records to delete. The resulting delete set may include versions of join view records which do not exist, since now the read sets include old versions of base records, but extraneous or duplicate view records will be ignored or be considered as delete tombstones. By reading old versions, the **Delete-Join** will be able to delete records from the view that could be missed if only reading the latest versions, and would prevent the **Missing-Delete-Anomaly**.

5.6.2.5 Trigger View Operations After Index Updates

Asynchronous index updates can cause the **Stale-Index-Anomaly**, because concurrent reads of the index may be stale. In order to prevent these types of anomalies from happening, the view maintenance tasks can force a partial order of the operations. After a base table is updated and a view maintenance task is triggered, instead of updating all indexes and views asynchronously, the tasks update all the indexes of the table first, and then update the views asynchronously. The view tasks use a synchronization barrier to maintain all indexes first, and then proceed to maintain the views afterwards. The operations in the view task are still not atomic, but by introducing the barrier, it forces a partial order for

all the indexes to be updated before maintaining any views for a particular update to a record. This synchronization barrier is not a barrier between multiple maintenance tasks for different records, but only for parallelism within a single view maintenance task. By forcing this partial order, this is enough to prevent the **Stale-Index-Anomaly**, without requiring locking or coordination.

5.7 SCALAVIEW Algorithm

In this section, I present the SCALAVIEW algorithm and discuss its various properties.

5.7.1 Pseudocode of Algorithm

Here, I describe the pseudocode of the SCALAVIEW algorithm in listing 7, 8, and 9. The pseudocode is broken up into parts, where listings 7 and 8 describe the algorithm for incrementally maintaining views for each row update, and listing 9 describes the algorithm for the data servers for responding to queries and row updates.

The process *ProcessRowUpdate* in listing 7 is asynchronously triggered for every row insert, delete or update to maintain the views. Lines 2–4 update the indexes before any of the view maintenance tasks for preventing the **Stale-Index-Anomaly**.

In algorithm 8, the lines 11–15 read the read set again, and lines 16–18 compute the difference and apply the compensating deletes. These techniques are necessary to correct the **Early-Read-Anomaly**.

Line 23 shows where the algorithm queries old versions for the **Delete-Join** task. Querying old versions is necessary to prevent the **Missing-Delete-Anomaly**.

In the data server algorithm 9, the line 11 stores delete tombstones if the delete for a view record is processed before the corresponding insert, and the line 16 returns the older versions of a record if requested.

5.7.2 Discussion of SCALAVIEW

With the component techniques described in Section 5.6.2, materialized views for joins can be maintained in a scalable way, for partitioned stores with SCALAVIEW. Since there is no centralized server that has to process a queue of all the updates in the system, and the view maintenance is distributed among the servers of the system, a maintenance bottleneck is not imposed. If the client update rate is too high for the algorithm to keep up, additional servers can added in order to scale out maintenance throughput, but for a centralized algorithm, scaling out is not possible. Also, locking and explicit coordination are not employed, to avoid scalability limitations on the system. Because these constructs are not used, SCALAVIEW has a lower set of system requirements of the underlying partitioned store, so it can be implemented on a wide range of systems.

Algorithm 7 SCALAVIEW Algorithm. Part 1

```
1: procedure ProcessRowUpdate(ΔT_x)
2:     for idx ∈ T_x.indexes do
3:         Update Index idx with ΔT_x
4:     end for
5:     switch ΔT_x do
6:         case Insert[new] :
7:             run ProcessInsertJoin(new)
8:         case Delete[old] :
9:             if useDeleteJoin then
10:                run ProcessDeleteJoin(old)
11:            else
12:                run ProcessDeleteDirect(old)
13:            end if
14:        case Update[old, new] :
15:            run ProcessDeleteJoin(old)
16:            run ProcessInsertJoin(new)
17: end procedure
```

SCALAVIEW is designed to be applicable to modern scalable database systems, such as distributed key-value stores. Also, it is important to note that one of the intentional goals of SCALAVIEW is to be deferred, or asynchronous, maintenance, as described in Section 5.3. Therefore, the techniques and semantics of SCALAVIEW cannot be directly compared with traditional, transactional, synchronous techniques.

Compared with the co-located indexes method, SCALAVIEW does add more computing and querying overhead. However, the co-located indexes method only supports equi-joins, where as the SCALAVIEW algorithm described in Section 5.7.1 can support other joins with non-equality conditions or even similarity joins. Some other aspects of the two algorithms are evaluated in Section 5.9.

There are some drawbacks to the SCALAVIEW algorithm for view maintenance. Because there is no serialization or total ordering or coordination, anomalies are possible as described in Section 5.5.4. However, with the techniques described in Section 5.6.2 and the algorithm in Section 5.7.1, the anomalies will either be prevented or corrected.

5.8 Proofs

In this section, I show proofs of how the techniques in SCALAVIEW prevent the anomalies described in Section 5.5.

Algorithm 8 SCALAVIEW Algorithm. Part 2

```
1: procedure ProcessInsertJoin(ΔT_x)
2:     RS_1 = ∅
3:     RS_2 = ∅
4:     for i ≠ x do
5:         Query T_i for records matching RS_1
6:         Receive ΔT_i
7:         RS_1 = RS_1 ∪ ΔT_i
8:     end for
9:     ΔV ← compute join from RS_1
10:    Insert ΔV into View V
11:    for i ≠ x do
12:        Query T_i for records matching RS_2
13:        Receive ΔT_i
14:        RS_2 = RS_2 ∪ ΔT_i
15:    end for
16:    RS_deleted = RS_1 − RS_2
17:    ΔV_deleted ← compute join from RS_deleted
18:    Delete ΔV_deleted from View V
19: end procedure

20: procedure ProcessDeleteJoin(ΔT_x)
21:    ΔV ← ΔT_x
22:    for i ≠ x do
23:        Query Versions T_i for records matching ΔV
24:        Receive ΔT_i
25:        ΔV ← ΔT_i ⋈ ΔV
26:    end for
27:    Delete ΔV from View V
28: end procedure

29: procedure ProcessDeleteDirect(ΔT_x)
30:    Read View V for records matching ΔT_x
31:    Receive ΔV
32:    Delete ΔV from View V
33: end procedure
```

Algorithm 9 Algorithm for Data Server

```
1:  storage : internal storage for row data

2:  procedure WriteRow(op)
3:      if op.key ∉ storage then
4:          storage.add(op.key, op.row)
5:          return
6:      end if
7:      switch op do
8:          case Insert[new] :
9:              storage.add(op.key, op.row)
10:         case Delete[old] :
11:             storage.deleteOrTombstone(op.key, op.row)
12: end procedure

13: procedure ReadRow(key, oldVersions)
14:     rowData ← storage.get(key)
15:     if oldVersions then
16:         return rowData.oldVersions()
17:     else
18:         return rowData.latestRow()
19:     end if
20: end procedure
```

Claim 1. *Composite primary keys prevents* ***Duplicate-Insert-Anomaly*** *and* ***Duplicate-Delete-Anomaly****.*

Proof. By definition, primary keys uniquely determine every record in a table. Therefore, any duplicate inserts or duplicate deletes to the identical record can be detected and prevented by using primary keys. □

Claim 2. *Using delete tombstones prevents* ***Delete-DNE-Anomaly****.*

Proof. Section 5.5.4.2 shows the schedule in which the **Delete-DNE-Anomaly** occurs. The anomaly occurs because the **Delete-Join** task attempts to delete a record which does not exist and will not exist. If using tombstones, a delete tombstone is stored whenever a delete of a non-existent record is attempted. When reading a delete tombstone, the record is not read, so the tombstone behaves as if the record does not exist. The results in the view is equivalent to a serial schedule where there is no concurrency, and either the **Insert-Join** task completes first (view record is deleted successfully), or the **Delete-Join** task completes first (task leaves a delete tombstone). Therefore, using delete tombstones prevents the **Delete-DNE-Anomaly**. □

Claim 3. *Using delete tombstones prevents* ***Late-Insert-Anomaly****.*

Proof. Section 5.5.4.2 shows the schedule in which the **Late-Insert-Anomaly** occurs. This anomaly occurs because the view delete of **Delete-Join** is out-of-order with the view insert of **Insert-Join**. The insert is too late and happens after the delete, so the record remains in the view, which is the incorrect result. Delete tombstones solves this out-of-order problem by essentially making the insert and delete commutative. There are only two possible orderings of the insert and delete: either the insert occurs first or the delete occurs first. If the insert occurs first, the following delete operation will simply delete the record, and no anomaly will occur. If the delete occurs first, a delete tombstone will be saved, the following insert will observe the tombstone, and the insert will effectively be ignored, and no anomaly will occur. Delete tombstones allow the insert and delete of the same view record to be commutative, so the corresponding conflict edge is no longer required in the conflict graph, and there would be no cycle. Therefore, delete tombstones prevent the **Late-Insert-Anomaly**. □

Claim 4. *Re-reading the read-set prevents* ***Early-Read-Anomaly****.*

Proof. Section 5.5.4.4 shows the schedule in which the **Early-Read-Anomaly** occurs. The conflict cycle exists when the **Insert-Join** task reads before the **Delete-Direct** task deletes the record ($r_1[S_y] \rightarrow del_2[S_y]$), and the **Delete-Direct** task reads the view before the **Insert-Join** task inserts the view record ($r_2[V_z] \rightarrow ins_1[V_z]$).

When re-reading the read-set, the **Insert-Join** task reads its read-set again after updating the view ($reread_1[S_y]$), and deletes from the view any records determined from the differences in the read-sets. So, all conflicting deletes that occur between $r_1[S_y]$ and $ins_1[V_z]$ will be reflected in the second read-set with $reread_1[S_y]$. Therefore, the resulting conflict edge is $del_2[S_y] \rightarrow reread_1[S_y]$. The original conflict edge, $r_1[S_y] \rightarrow del_2[S_y]$, is no longer relevant because concurrent deletes are captured in the re-read. Therefore, the relevant conflict edges are:

$$r_2[V_z] \rightarrow ins_1[V_z]$$
$$del_2[S_y] \rightarrow reread_1[S_y]$$

There is no cycle with the schedule, and by re-reading the read-set, the results are equivalent to a serial schedule where **Delete-Direct** completes before **Insert-Join**. Therefore, this technique prevents **Early-Read-Anomaly**. □

Claim 5. *Querying old versions prevents* ***Missing-Delete-Anomaly****.*

Proof. Section 5.5.4.3 shows the conflict cycle for with two concurrent **Delete-Join** tasks. This anomaly occurs because both tasks attempt to read the other record after the record has been deleted by the other task. The conflict edges are:

$$del_1[R_x] \rightarrow r_2[R_x]$$
$$del_2[S_y] \rightarrow r_1[S_y]$$

This leads to both tasks not being able to determine the records to delete from the view. When the **Delete-Join** task requests old versions when issuing maintenance queries, the task is able to retrieve deleted versions of records. By reading older versions, the order in which $del_1[R_x]$ and $r_2[R_x]$ occur is not important, and likewise for $del_2[S_y]$ and $r_1[S_y]$. Those conflict edges are no longer relevant, so there that conflict cycle no longer exists.

However, the view delete sets for the **Delete-Join** tasks will be inflated by using older versions, and there could be conflicting updates between view deletes. Those conflicting view deletes will look like duplicate deletes or non-existent deletes. However, other techniques already handle these types of anomalies. Claims 1 and 2 show how primary keys and tombstones prevent those anomalies. Therefore, querying old versions in **Delete-Join** tasks prevents the **Missing-Delete-Anomaly**. □

Claim 6. *Forcing maintenance of indexes before views prevents* ***Stale-Index-Anomaly****.*

Proof. The Sections 5.5.4.1 and 5.5.4.2 show conflict cycles that result in the **Stale-Index-Anomaly**. For both of these scenarios, the cause for the anomaly is that asynchronous indexes are queried during the maintenance queries. When both concurrent tasks read the stale indexes, both tasks do not observe the other task's update, so neither is aware o the other task, and neither is able to correctly update the view. There is no equivalent serial schedule where both tasks do not observe the results of the other.

Without loss of generality, we examine the case of concurrent **Insert-Join** tasks. If we force maintaining indexes before any view maintenance operations, this imposes a partial ordering in the operations of the **Insert-Join** tasks. So, the following ordering relationships with each task are enforced:

$$ins_1[R.idx_x] \rightarrow r_1[S.idx_y]$$
$$ins_2[S.idx_y] \rightarrow r_2[R.idx_x]$$

Looking at the two concurrent reads $r_1[S.idx_y]$ and $r_2[R.idx_x]$ of the tasks, there is no imposed ordering between them, but one of the reads will happen before the other. Without loss of generality, assume $r_1[S.idx_y]$ happens before $r_2[R.idx_x]$, $(r_1[S.idx_y] \rightarrow r_2[R.idx_x])$, so by transitivity:

$$ins_1[R.idx_x] \rightarrow r_1[S.idx_y] \rightarrow r_2[R.idx_x]$$
$$ins_1[R.idx_x] \rightarrow r_2[R.idx_x]$$

Therefore, the read $r_2[R.idx_x]$ will observe the other tasks insert, $ins_1[R.idx_x]$. When forcing all index maintenance before any view maintenance operations, this will force one of the tasks to be able to observe the other task, thus preventing the **Stale-Index-Anomaly**.

Imposing the partial order may cause other anomalies, but they are equivalent to the cases when the indexes are *not* queried during maintenance. This is because all view maintenance is triggered after the index update, just like how all view maintenance is triggered after the base table update when not using indexes. The overall structure of operations in the two scenarios are identical with respect to the conflicting operations. So any potential anomalies have already been considered and are prevented from other techniques discussed in Section 5.6.2. Therefore, maintaining indexes before views prevents anomalies. □

5.9 Evaluation

In this section, I evaluate various aspects of the SCALAVIEW approach to view maintenance for join views. I compare SCALAVIEW with a centralized algorithm, as well as the co-located index approach. This section shows the results of the experiments and how well SCALAVIEW achieves the goals for scalable view maintenance.

While the asynchronous, incremental, and convergence goals from Section 5.3 are properties of the system and algorithm, I evaluate SCALAVIEW on how well it achieves the other goals. Specifically, I performed experiments to evaluate how SCALAVIEW performs for reducing view staleness, reducing data amplification, more general joins, and scaling out with more servers. The rest of this section describes some of the implementation details, and describes the experimental results.

5.9.1 Implementation

I implemented SCALAVIEW on a distributed key-value store developed in the AMPLab at UC Berkeley. The underlying key-value store is an in-memory, horizontally partitioned store, which has support for inner-join queries, and asynchronous triggers. While SCALAVIEW was implemented on the distributed system from the AMPLab, it is possible to implement it with other underlying systems which satisfy the requirements discussed in Section 5.6.1.

In order to evaluate SCALAVIEW, I compared it with a centralized algorithm, and the co-located indexes method. Both of these other methods were implemented in the same system and the same asynchronous triggers mechanism, so the underlying data store is the same for all three methods.

5.9.1.1 Centralized Algorithm

In order to run a centralized algorithm for comparison, I implemented the ideas of PVM [84]. This algorithm was chosen because it uses similar compensation techniques, and also has a notion of parallelism by using multiple threads. However, PVM requires stronger system

```
CREATE TABLE R(rid INT, rdata STRING, PRIMARY KEY(rid))
CREATE TABLE S(rid INT, tid INT, sdata STRING,
               PRIMARY KEY(rid, tid))
CREATE TABLE T(tid INT, tdata STRING, PRIMARY KEY(tid))
CREATE VIEW V AS SELECT * FROM R, S, T
                 WHERE R.rid = S.rid AND S.tid = T.tid
```

Figure 5.2: Simple three table join for the micro-benchmark

assumptions, since it requires FIFO and sequential assumptions in how queries are handled. Also, I experimented with several levels of parallelism for my implementation of PVM and determined that using 2048 threads had good results on the hardware used. Therefore, for all results for the centralized algorithm is my implementation of PVM using 2048 threads.

5.9.1.2 Co-Located Indexes

I also implemented a co-located indexes technique within the distributed key-value store developed in the AMPLab at UC Berkeley. This algorithm is similar to the techniques of PNUTS [3] from Yahoo, or Lynx [85] from Microsoft Research. For each join key, the relevant tables are partitioned on that join key to create co-located indexes, so that local joins are performed. Then, after the local join is materialized, additional partitioning may be required, if there are other join keys.

5.9.1.3 Experiments

For the experiments, I use a simple synthetic micro-benchmark consisting of 3 base tables. Most of the experiments use a simple equi-join view, which is defined on the base tables, and the schemas are defined in figure 5.2.

I use this simple 3-way join to demonstrate the characteristics of SCALAVIEW. In the experiments, many clients issue insert, update, and delete operations to random rows in the database. Clients do not issue any read queries, because evaluating maintenance of join views is the primary objective, and reads do not affect maintenance of views. Throughout the experimental runs, various metrics are measured such as throughput, and staleness of view updates. All experiments run on Amazon EC2 in the US West (Oregon) region, and all the servers used cr1.8xlarge instances. The cr1.8xlarge instances are memory-optimized instances with 32 virtual CPUs and 244 GB of memory. All servers are running in the same availability zone, unless otherwise noted.

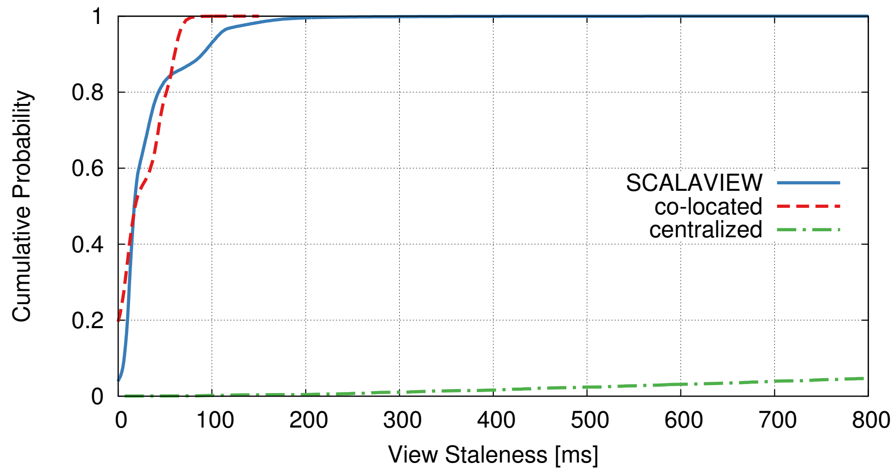


Figure 5.3: View staleness CDF

5.9.2 View Staleness

I first evaluate how well the algorithms satisfy the goal for reducing view staleness. SCALAVIEW and the other techniques in comparison are all asynchronous techniques. This means the view update does not execute transactionally with the base update, but is updated at a later time. Therefore, there will always be a delay between the base update and when that update is reflected in the join view. I measure view staleness for the different view maintenance algorithms. In the experiment, the partitioned data store was configured to use 4 data servers (partitions), and the clients issued write operations at an aggregate rate of $5,000$ operations/second.

Figure 5.3 shows the cumulative distribution functions of the view staleness of the different algorithms. The co-located indexes method has the least view staleness, followed by the SCALAVIEW algorithm, and then the centralized algorithm. The average delay from the base update to the view for the co-located indexes is 26.67ms, and the average delay for SCALAVIEW is 32.38ms. SCALAVIEW has a little bit more delay over the co-located method because it does have to issue more queries, since it is a more general technique. However, the view staleness for SCALAVIEW is still reasonable, with most of the delays being less than 100ms.

Figure 5.3 also shows how a centralized algorithm will not scale. The centralized algorithm exhibits very poor view staleness, because the algorithm is not able to keep up with the incoming update load. My specific implementation of a centralized algorithm was able to handle a few hundred operations/second, which is not fast enough to handle the $5,000$ operations/second from the clients. Because the centralized algorithm needs to access a global queue of updates and other global data structures, this creates thread contention and causes a bottleneck. Although additional optimizations could be implemented to remove some of the contention, but this potential bottleneck is present for all centralized algorithms. At

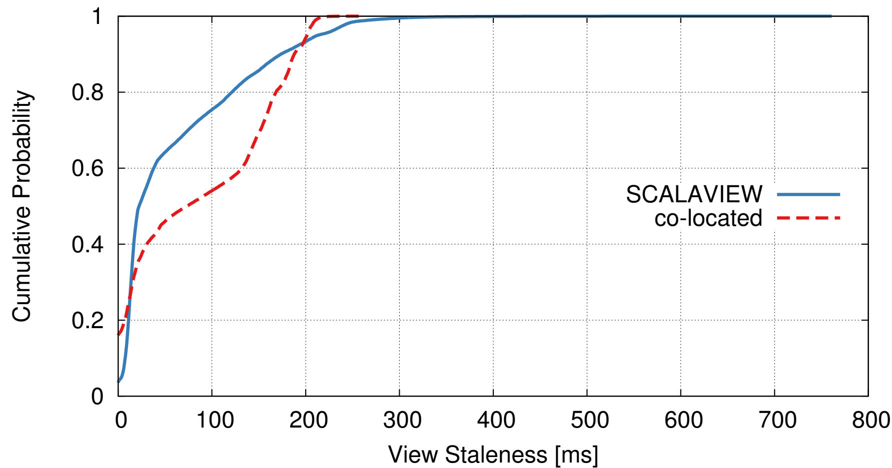


Figure 5.4: View staleness CDF (across three availability zones)

some point, a centralized algorithm is going to hit a throughput bottleneck, and will be unable to scale out to handle higher update rates, and this experiment demonstrates this limitation. The results of this experiment show that SCALAVIEW and co-located index techniques perform well to achieve the goal of reducing view staleness.

5.9.2.1 Separate Availability Zones

While the previous experiment places every server local to a single availability zone in Amazon EC2, figure 5.4 shows the same experiment with the servers spread out over three availability zones. Since the servers are no longer in the same availability zone, there is higher network latency and variance. In this scenario, the co-located technique actually provides staleness in the views (87.54ms) than SCALAVIEW (61.47ms). Using co-located indexes must first co-locate table R and S on join key rid, in order to locally materialize $R \bowtie S$. Then, another index for $R \bowtie S$ must be re-partitioned on join key tid to locally materialize $R \bowtie S \bowtie T$. This experiment shows that multiple levels of co-located indexes and re-partitioning is more sensitive to the network latencies, and that SCALAVIEW is comparable in view staleness can even exhibit lower delays.

5.9.3 Data Size Amplification

Next, I evaluate how well SCALAVIEW performs to satisfy the goal of reducing data size amplification. Different maintenance techniques require different amounts of extra data that needs to be duplicated or be kept in the system. The data for the base tables must be retained in the database, but any extra data required for the algorithms is the data amplification. For SCALAVIEW, keeping around the old versions is extraneous data. For the co-located

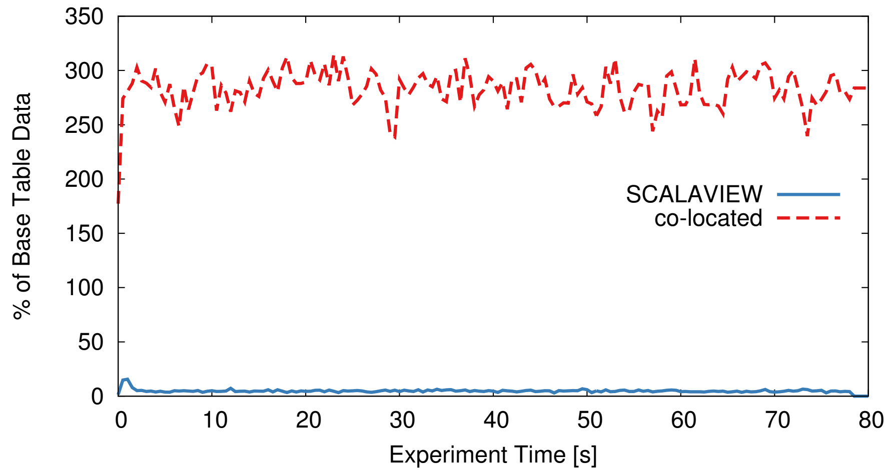


Figure 5.5: Data size amplification as a percentage of the base table data

```
CREATE VIEW V AS SELECT * FROM R, S, T
                 WHERE R.rid = S.rid AND
                       S.tid >= T.tid - c AND
                       S.tid <= T.tid + c
```

Figure 5.6: Band join query for micro-benchmark

technique, the co-located indexes are extraneous data. In this experiment, I measure the size of base table data, and the extraneous data in the system.

Figure 5.5 shows the data size amplification for the co-located and the SCALAVIEW techniques. The figure shows the amount of extra data as a percentage of the base table data, and SCALAVIEW requires much less extra data in order to operate. Throughout the experimental run, the co-located technique has on average 283% data size data amplification, while SCALAVIEW only uses about 4.7% on average. This experiment shows that SCALAVIEW requires far less extraneous data in order to maintain materialized join views.

5.9.4 Band Joins

Another goal for the SCALAVIEW algorithm is to be general enough to support more than equi-joins. While the co-located index only support equi-join views, SCALAVIEW can support non-equality join conditions. One example of a non-equi-joins is a band join [34], where a join condition looks like $S.y - c_1 \leq R.x \leq S.y + c_2$. In this experiment, the view staleness is measured for band joins for different values of c, with the modified view definition in figure 5.6.

Figure 5.7 shows the cumulative distribution functions of the view staleness for different

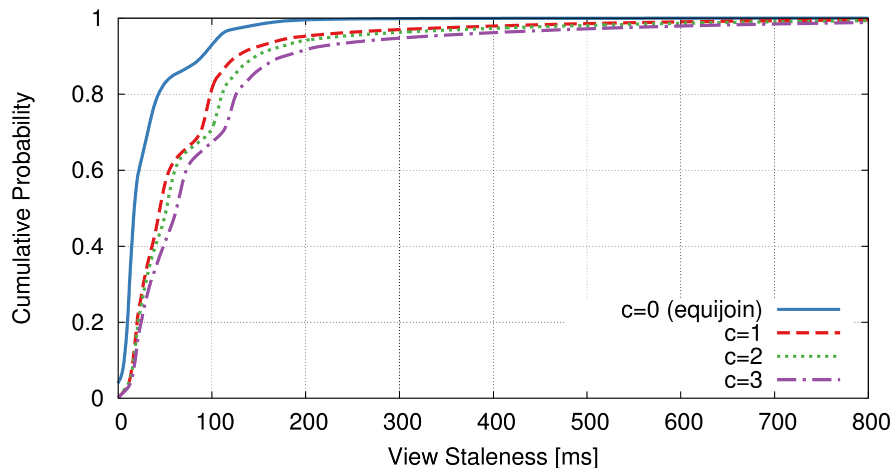


Figure 5.7: View staleness CDF for band joins

values of the band join parameter, c. When $c = 0$, the band join is equivalent to an equi-join. As the parameter c increases, the staleness in the view also increases as shown in the figure. The average staleness for the equi-join is 32.38ms, while the average staleness for the band join with $c = 3$ is 101.84ms. Larger band joins exhibiting more staleness is expected, because the queries required for the incremental maintenance are larger because of the range join condition. More rows are read and transferred in the larger band joins, so the additional processing will cause more staleness. However, the staleness is reasonable and expected, because for $c = 3$, the maintenance queries may read up to 7 times more data, and the staleness is about 3 times worse. This experiment demonstrates that SCALAVIEW is general enough for different join conditions, and can still perform well when non-equi-joins views are used.

5.9.5 Scalability

One of the main objectives for SCALAVIEW is to maintain materialized views in a scalable way, to avoid any bottlenecks in a scalable system. I performed an experiment to determine how the SCALAVIEW algorithm would scale out with more servers and clients. I executed the equi-join view experiment several times with varying number of servers, and client aggregate rate. For 2 servers, the client aggregate rate was set at $5,000$ write operations/second, and as the number of servers increased, the client rate and the amount of data partitioned to each server also increased proportionally. Therefore, at 16 servers, the client rate is $40,000$ operations/second. All the scaling experiments were performed in a multiple availability zone environment, because all the instances could not be reserved in the same availability zone.

Figure 5.8 shows the cumulative distribution functions of the view staleness for the different scaling factors. The graph shows that the view staleness measurements for the different

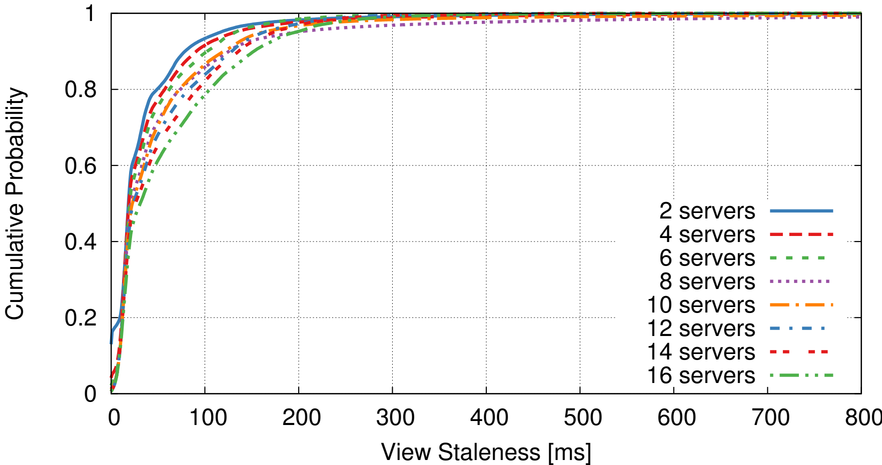


Figure 5.8: View staleness CDF scalability

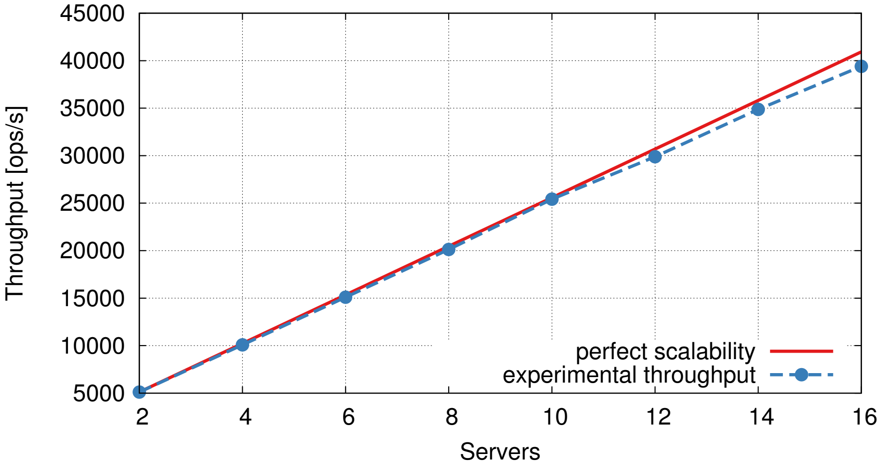


Figure 5.9: Throughput scalability

```
CREATE TABLE R(rid INT, sid INT, rdata STRING,
               PRIMARY KEY(rid, sid))
CREATE TABLE S(sid INT, sdata STRING,
               PRIMARY KEY(sid))
CREATE VIEW V AS SELECT * FROM R, S
                 WHERE R.sid = S.sid
```

Figure 5.10: Simple two table linear join query (Linear-2)

number of servers are generally similar to each other. However, the staleness does slowly increase as the scaling factor increases. When there are only 2 servers, the average staleness was 36.13ms, but when run with 16 servers, the average staleness in the view was 60.66ms. This is explained by the extra network messages that may have to access more servers for the maintenance queries. Since some of the queries need to access more servers, the queries will experience greater response times, especially in the case of distributing the servers across multiple availability zones. This causes a longer tail in response times, thus resulting in a longer tail in the view staleness as well. However, this experiment still shows that the algorithm is still quite scalable, and SCALAVIEW could scale even better if network latency aware optimizations and join techniques were implemented to mitigate the longer tails.

Figure 5.9 shows the throughput of the view maintenance operations for the different scaling factors. Here, the measured throughput is plotted in the graph, along with the theoretical throughput of perfect scalability. The experimental throughput plot nearly matches the perfect linear scalability throughput as the servers vary from 2 to 16. When the experiment was run with 16 servers, SCALAVIEW could scale to about 96.2% of the throughput of perfect linear scalability. Therefore, this experiment demonstrates that SCALAVIEW is able to scale out well with more machines and more client load, and it satisfies the scalability objective from Section 5.3.

5.9.6 Join Types

I ran another experiment with different types of joins to observe how SCALAVIEW handles different types of joins. Here, additional simple types of joins were used in the micro-benchmark to how they affect the staleness of the views. Figure 5.10 shows a simple linear join of two tables (Linear-2), and Figure 5.11 shows a simple linear join of three tables (Linear-3). The previously described join in Figure 5.2 is another type of three table join. It is a join of a star schema with a single fact table that references two other dimension tables (Star-3). Figure 5.12 is star schema join of four tables (Star-4). The micro-benchmark was run for each of the four join types, and the staleness was measured.

Figure 5.13 shows the CDF plots of the view staleness of the different join types. In general, the results are encouraging, since the staleness is pretty small for all the join types. Linear-2 exhibits the least staleness, which explained by the fact that there are only two

```
CREATE TABLE R(rid INT, sid INT, rdata STRING,
               PRIMARY KEY(rid, sid))
CREATE TABLE S(sid INT, tid INT, sdata STRING,
               PRIMARY KEY(sid, tid))
CREATE TABLE T(tid INT, tdata STRING, PRIMARY KEY(tid))
CREATE VIEW V AS SELECT * FROM R, S, T
                 WHERE R.sid = S.sid AND S.tid = T.tid
```

Figure 5.11: Simple three table linear join query (Linear-3)

```
CREATE TABLE R(rid INT, rdata STRING, PRIMARY KEY(rid))
CREATE TABLE S(sid INT, sdata STRING, PRIMARY KEY(sid))
CREATE TABLE T(tid INT, tdata STRING, PRIMARY KEY(tid))
CREATE TABLE U(rid INT, sid INT, tid INT, udata STRING,
               PRIMARY KEY(rid, sid, tid))
CREATE VIEW V AS SELECT * FROM R, S, T, U
                 WHERE U.rid = R.rid AND
                       U.sid = S.sid AND
                       U.tid = T.tid
```

Figure 5.12: Simple four table star join query (Star-4)

tables in the join. Star-3 and Star-4 both perform quite similar to each other. This is expected, since the query plans for those joins are both similar. If a fact table is updated, all the dimension tables can be queried in parallel. If a dimension table is updated, then the fact table must be queried before accessing the other dimension tables. Linear-3 showed the most staleness in the views. This was also expected because a lot of the incremental update plans had to employ a sequential chain of table queries. Also, secondary indexes were not created for the tables, so some queries to tables R and S took longer to complete.

5.10 Related Work

There have been significant previous work in the area of efficiently updating materialized views in databases. Early work in incremental view maintenance [23, 17, 68, 41] investigated new algorithms and queries to update materialized views with smaller updates, instead of re-computing the entire view. For these techniques, single-server database systems are assumed, and additional queries are triggered to run within the base table transactions, in order to update materialized views correctly. RAMP transactions [12] allow cross-partition atomicity for write sets, but for maintaining join views, the read and write sets could grow large. In

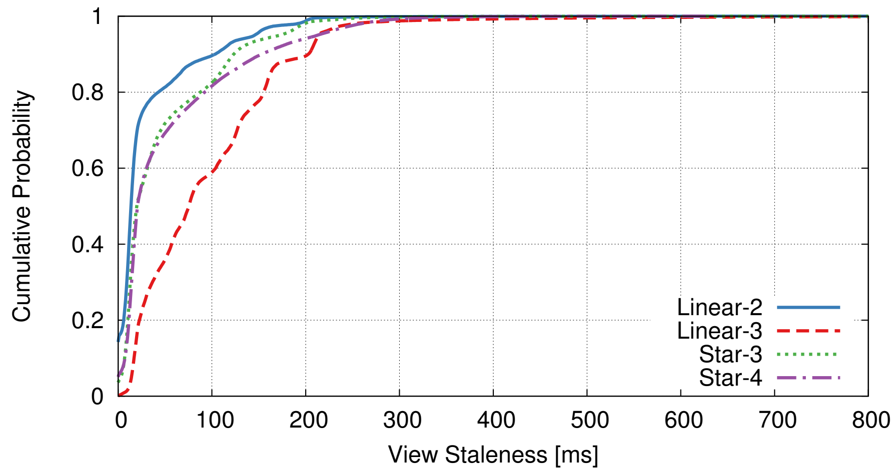


Figure 5.13: View staleness CDF for different join types

contrast, my work maintains views separately from the base transactions, and focus on large distributed systems.

While many of the incremental view maintenance techniques update views synchronously with the base transactions, other work has been done for asynchronous view maintenance, or deferred view maintenance. Segev [76, 75] studied deferred maintenance of views, but only investigated select-project views, and not joins. 2VNL [69] is a data warehouse technique for updating the views without locking. ECA [88] is a data warehouse technique which updates the views from a remote database source. Because of the deferred nature of the updates, it is possible to update views incorrectly and leave anomalies, but ECA solves the problem by using compensation queries to adjust the results of the incremental queries. Single-server, asynchronous view maintenance techniques [28, 86, 71] use intermediate auxiliary tables in order to incrementally update views. These techniques opt for some staleness in the views for faster base transactions and potential better system utilization. These are also single source or single-server algorithms that depend on global state and ordering in the database system. SCALAVIEW does not depend on having access to such global state or ordering.

In addition to deferred maintenance, other related work has investigated updating materialized views from many different distributed sources. Most of this work has focused on view maintenance in the context of data warehouses. In this setting, distributed data sources are typically separate from the data warehouse, and the views in the data warehouse are updated incrementally as updates from the sources arrive. STROBE [87] and SWEEP [2] both probe the sources for maintenance queries and also issue compensation queries to prevent anomalies. However, both STROBE and SWEEP are single-threaded sequential algorithms. POSSE [61] and PVM [84] uses multiple threads at the data warehouse to process updates in parallel, and DyDa [26] also handles schema changes. TxnWrap [25] maintains versions of source data and uses a multi-version concurrency control scheme at the data warehouse

for providing consistency. Typically in these data warehouse techniques, update ordering is achieved at data warehouse, before potential parallelism is exploited. In contrast, SCALAVIEW does not depend on a global sequence of updates.

Asynchronous and distributed view maintenance [3, 85] has been investigated for equi-joins in modern scalable partitioned stores. While previous techniques have focused on incremental view maintenance with a single or few view updaters, distributed view maintenance investigate the challenges when scaling out to many view updaters distributed on many machines. The main techniques for distributed view maintenance are related to asynchronous secondary index maintenance, and co-location of join key indexes. SCALAVIEW is able to support more general types of joins, and also does not require intermediate indexes or results.

5.11 Conclusion

Scalable partitioned stores have become popular for their scalability properties. These distributed systems can easily scale out by adding more servers to handle larger data sizes, and greater query load. However, join queries can limit the scalability and performance of these scalable partitioned stores. Materialized views are commonly used in traditional database systems to avoid running the same types of queries and joins repeatedly. However, in distributed partitioned stores, materialized views are not natively supported. Manual materialized views and view maintenance are always possible, but the results may not always be correct.

I proposed SCALAVIEW, a new scalable view maintenance algorithm for join views, which is asynchronous, incremental, and distributed. This algorithm is used to maintain materialized views natively in a scalable partitioned store. SCALAVIEW does not require a serializer, or a centralized server to process all the updates in the system, and therefore does not impose scalability bottlenecks. SCALAVIEW works asynchronously, so global distributed transactions or locking are not required and not used. I showed how, when views are maintained manually by applications, anomalies may arise in the views; I described how SCALAVIEW prevents and corrects these anomalies to achieve convergence consistency. I evaluated the performance, scalability and overhead of SCALAVIEW, and compared it with other distributed and centralized techniques. The experiments show SCALAVIEW can scale out well with more machines and load, and provides view staleness comparable to a simpler, but limited and more data amplifying, distributed technique of co-located indexes. The properties of SCALAVIEW make it very applicable for maintaining views in modern scalable partitioned stores.

Chapter 6

Conclusion

This chapter reviews the main contributions of this thesis. It also discusses possible areas of future work to further scalable transactions in distributed database systems.

6.1 Contributions

This thesis first described a new transaction commit protocol, MDCC, for distributed database systems. MDCC, or Multi-Data Center Consistency, is a new commit protocol that provides scalable ACID transactions in distributed database systems. While MDCC is applicable in a large range of deployment environments, it is particularly beneficial in a wide-area network setting, because it is optimized to reduce commit response times by reducing the number of message rounds. MDCC provides transaction durability by synchronously replicating to multiple data centers, so the will still be available even when an entire data center fails. MDCC is able to achieve its goals for faster commits by taking advantage of two observations of workloads: transaction conflicts are rare, or transaction conflicts commute with each other. By exploiting these two properties, and adapting the generalized Paxos protocol for transactions, MDCC supports distributed transactions for scalable database systems.

Next, I introduced a new transaction programming model, PLANET, for distributed transactions. When developers have to deal with distributed transactions, the underlying environment is different from local transaction environment. Instead of using a traditional, single server database system, modern applications tend to use distributed database systems. With distributed systems, the environment can be less predictable and less reliable than a single-site system. Also, communication and coordination costs in distributed systems lead to slower response times. PLANET, or Predictive Latency-Aware NEtworked Transactions, is a new transaction programming model that aims to alleviate some of the difficulty in interacting with distributed transactions. PLANET provides staged feedback about transactions, and exposes greater visibility of transaction state so applications and developers can better adapt to unpredictable environments. This can improve the experience with user-facing applications during unexpected periods of high latency in the deployment. PLANET

is also, to the best of my knowledge, the first transaction programming model to implement the *guesses and apologies* paradigm as suggested by Helland and Campbell [43]. PLANET enables *guesses* by means of *commit likelihoods* and *speculative commits*, and this thesis presents a commit likelihood model for Paxos-based geo-replicated protocol such as MDCC. I also showed how admission control can use *commit likelihoods* in order to use system resources more effectively. With PLANET, developers can build flexible applications to cope with unexpected sources of latency without sacrificing the end-user experience.

Finally, in this thesis, I proposed SCALAVIEW, a new algorithm for scalable distributed view maintenance of join views. SCALAVIEW is a distributed view maintenance algorithm that is scalable because it does not depend on synchronous coordination or a globally centralized service. Materialized views can speed up queries, especially for scalable systems such as key-value stores that do not have support for queries such as joins. Storing precomputed results can greatly benefit queries by avoiding repeated computation. I described how traditional approaches for view maintenance can restrict scalability because of the centralized nature of the algorithms. I also discussed how and why naïve, distributed algorithms can result in anomalies in the data. If anomalies remain in the data, then the view will never converge to the *correct* answer. SCALAVIEW is a distributed algorithm that achieves convergent consistency for materialized join views in scalable data stores. This thesis showed how the various techniques of SCALAVIEW can prevent and correct potential anomalies that may arise in the distributed maintenance of views.

As a whole, this thesis described several new algorithms and techniques for scalable transactions in distributed database systems. However, there are certain limitations and opportunities for advancement to the approaches introduced in this thesis, and the next section discusses those points.

6.2 Future Work

This section discusses various areas for extension to the techniques for scalable transactions described in this thesis.

6.2.1 Stronger Levels of Consistency for MDCC

Exploring stronger levels of consistency is a natural extension to MDCC. MDCC has a default isolation level of read-committed without lost updates, and that is stronger than the default level of several popular single-server database systems. However, it may be possible to extend MDCC to support stronger levels such as Snapshot Isolation, and even Serializability. Providing stronger consistency will require additional coordination, and studying the various trade-offs between stronger semantics and scalability is important.

6.2.2 Dynamic Quorums

MDCC is a transaction commit protocol that is based on Generalized Paxos, a quorum protocol. In this thesis, MDCC assumed a relatively uniform and static deployment environment and workload. However, further investigation is possible related to the quorum membership and quorum sizes in the protocol. For example, by allowing quorum properties to change along with the variable nature in the distributed environment or the user workload, the database system could adjust to the different conditions to provide an optimal configuration for faster distributed transactions. Determining how quorums could adapt while still providing correctness is a great opportunity for future work.

6.2.3 Extensions to PLANET

PLANET provides a flexible transaction programming model for developers to cope with unexpected latency issues with distributed transactions. However, there are opportunities to discover other paradigms for helping developers deal with distributed transactions. While PLANET is flexible for many different use cases, it is possible that there are other transaction stages useful for application development. Also, exposing additional system statistics may be beneficial to the developer, by enabling more adaptive control over the transaction behavior.

6.2.4 Stronger Levels of Consistency for SCALAVIEW

There are opportunities for further study for stronger levels of consistency in SCALAVIEW as well. SCALAVIEW is able to achieve convergence consistency, but stronger levels should be investigated for scalable view maintenance. In particular, global snapshots of consistent views could be very beneficial to applications and developers, and is an opportunity for further studies.

6.2.5 Integration with Data Analytics Frameworks

Scalable transactions are important for modifying data in distributed database systems, but data analytics is necessary for extracting value out of the data. Therefore, integration with data analytics engines is a crucial next step to investigate. Supporting materialized views is one good way of enabling analytics on transactional data, but further studies are necessary. By bringing the analytics closer to the source of transactional data, the latency can be reduced, thus allowing faster insights and decisions. There are many challenges to investigate to bridge the gap between OLTP and OLAP systems.

6.3 Conclusion

With the advent of large-scale applications and emergence of cloud computing, demands on data management systems are greater than ever. Existing techniques for executing and

interacting with transactions work well for the traditional model of single server database systems, but need to be revisited for distributed and scalable database systems. New techniques are required to achieve performant and scalable transactions for the modern cloud environment. My thesis proposed new scalable algorithms and methods for distributed transactions in scalable database systems. Therefore, distributed transactions can be supported in distributed database systems without having to abandon scalability.

Bibliography

[1] Michael Abd-El-Malek et al. "Fault-Scalable Byzantine Fault-Tolerant Services". In: *Proc. of SOSP*. Brighton, United Kingdom, 2005.

[2] D. Agrawal et al. "Efficient View Maintenance at Data Warehouses". In: *Proc. of SIGMOD*. SIGMOD '97. Tucson, Arizona, USA: ACM, 1997, pp. 417–427. ISBN: 0-89791-911-4.

[3] Parag Agrawal et al. "Asynchronous View Maintenance for VLSD Databases". In: *Proc. of SIGMOD*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 179–192. ISBN: 978-1-60558-551-2.

[4] *Amazon EC2 Outage, April 2011*. `http://aws.amazon.com/message/65648/`.

[5] *Amazon RDS Multi-AZ Deployments*. `http://aws.amazon.com/rds/mysql/\#Multi-AZ`.

[6] *Apache Cassandra*. `http://cassandra.apache.org`.

[7] *Apache HBase*. `http://hbase.apache.org`.

[8] Michael Armbrust et al. "PIQL: Success-Tolerant Query Processing in the Cloud". In: *PVLDB* 5.3 (2011), pp. 181–192.

[9] Michael Armbrust et al. "A View of Cloud Computing". In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782.

[10] Michael Armbrust et al. "Generalized Scale Independence Through Incremental Precomputation". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 625–636. ISBN: 978-1-4503-2037-5.

[11] Peter Bailis et al. "Probabilistically Bounded Staleness for Practical Partial Quorums". In: *Proc. VLDB Endow.* 5.8 (2012), pp. 776–787. ISSN: 2150-8097.

[12] Peter Bailis et al. "Scalable Atomic Visibility with RAMP Transactions". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 27–38. ISBN: 978-1-4503-2376-5.

[13] Jason Baker et al. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". In: *CIDR*. 2011.

[14] Daniel Barbará and Hector Garcia-Molina. "The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems". In: *VLDB J.* 3.3 (1994), pp. 325–353.

[15] Hal Berenson et al. "A Critique of ANSI SQL Isolation Levels". In: *Proc. of SIGMOD*. San Jose, California, United States, 1995. ISBN: 0-89791-731-6.

[16] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 0-201-10715-5.

[17] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. "Efficiently Updating Materialized Views". In: *Proc. of SIGMOD*. SIGMOD '86. Washington, D.C., USA: ACM, 1986, pp. 61–71. ISBN: 0-89791-191-1.

[18] Peter Bodik et al. "Characterizing, Modeling, and Generating Workload Spikes for Stateful Services". In: *Proc. of SoCC*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 241–252. ISBN: 978-1-4503-0036-0.

[19] Matthias Brantner et al. "Building a database on S3". In: *Proc. of SIGMOD*. Vancouver, Canada, 2008. ISBN: 978-1-60558-102-6.

[20] Nathan Bronson et al. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 49–60.

[21] Christian Cachin et al. "Secure and Efficient Asynchronous Broadcast Protocols". In: *Advances in Cryptology-Crypto 2001*. Springer. 2001, pp. 524–541.

[22] Michael J. Carey, Sanjay Krishnamurthi, and Miron Livny. "Load Control for Locking: The 'Half-and-Half' Approach". In: *PODS*. 1990, pp. 72–84.

[23] S. Ceri and J. Widom. "Deriving Production Rules for Incremental View Maintenance". In: *Proc. of VLDB*. 1991.

[24] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (June 2008), 4:1–4:26. ISSN: 0734-2071.

[25] Songting Chen, Bin Liu, and Elke A. Rundensteiner. "Multiversion-based View Maintenance over Distributed Data Sources". In: *ACM Trans. Database Syst.* 29.4 (Dec. 2004), pp. 675–709. ISSN: 0362-5915.

[26] Songting Chen, Xin Zhang, and Elke A Rundensteiner. "A compensation-based approach for view maintenance in distributed environments". In: *Knowledge and Data Engineering, IEEE Transactions on* 18.8 (2006), pp. 1068–1081.

[27] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782.

[28] Latha S. Colby et al. "Algorithms for Deferred View Maintenance". In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. Montreal, Quebec, Canada: ACM, 1996, pp. 469–480. ISBN: 0-89791-794-4.

[29] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proc. of SoCC*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0.

[30] Brian F. Cooper et al. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proc. VLDB Endow.* 1 (2 2008), pp. 1277–1288. ISSN: 2150-8097.

[31] James C. Corbett et al. "Spanner: Google's Globally-Distributed Database". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 261–264. ISBN: 978-1-931971-96-6.

[32] Carlo Curino et al. "Workload-Aware Database Monitoring and Consolidation". In: *Proc. of SIGMOD*. SIGMOD '11. Athens, Greece: ACM, 2011, pp. 313–324. ISBN: 978-1-4503-0661-4.

[33] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *Proc. of SOSP*. Stevenson, Washington, USA, 2007. ISBN: 978-1-59593-591-5.

[34] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. "An Evaluation of Non-Equijoin Algorithms". In: *Proceedings of the 17th International Conference on Very Large Data Bases*. VLDB '91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 443–452. ISBN: 1-55860-150-3.

[35] Dan Dobre et al. "HP: Hybrid paxos for WANs". In: *Dependable Computing Conference*. IEEE. 2010, pp. 117–126.

[36] Jennie Duggan et al. "Performance Prediction for Concurrent Database Workloads". In: *Proc. of SIGMOD*. SIGMOD '11. Athens, Greece: ACM, 2011, pp. 337–348. ISBN: 978-1-4503-0661-4.

[37] Sameh Elnikety et al. "A method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites". In: *WWW*. 2004, pp. 276–286.

[38] *Google AppEngine High Replication Datastore*. `http://googleappengine.blogspot.com/2011/01/announcing-high-replication-datastore.html`.

[39] Jim Gray and Leslie Lamport. "Consensus on Transaction Commit". In: *TODS* 31 (1 2006), pp. 133–160. ISSN: 0362-5915.

[40] Ashish Gupta and Inderpal Singh Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". In: *IEEE Data Eng. Bull.* 18.2 (1995), pp. 3–18.

[41] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. "Maintaining Views Incrementally". In: *SIGMOD Rec.* 22.2 (June 1993), pp. 157–166. ISSN: 0163-5808.

[42] Hans-Ulrich Heiss and Roger Wagner. "Adaptive Load Control in Transaction Processing Systems". In: *VLDB*. 1991, pp. 47–54.

[43] Pat Helland and David Campbell. "Building on Quicksand". In: *CIDR*. 2009.

[44] *Hibernate*. `http://www.hibernate.org/`.

[45] *How Big Is Facebook's Data?* http://techcrunch.com/2012/08/22/how-big-is-facebooks-data-2-5-billion-pieces-of-content-and-500-terabytes-ingested-every-day/.

[46] Hiranya Jayathilaka. *MDCC - Strong Consistency with Performance.* http://techfeast-hiranya.blogspot.com/2013/04/mdcc-strong-consistency-with-performance.html. 2013.

[47] Bettina Kemme et al. "Processing Transactions over Optimistic Atomic Broadcast Protocols". In: *Proc. of ICDCS*. IEEE. 1999, pp. 424–431.

[48] Donald Kossmann, Tim Kraska, and Simon Loesing. "An Evaluation of Alternative Architectures for Transaction Processing in the Cloud". In: *Proc. of SIGMOD*. Indianapolis, Indiana, USA, 2010. ISBN: 978-1-4503-0032-2.

[49] Tim Kraska et al. "Consistency Rationing in the Cloud: Pay only when it matters". In: *PVLDB* 2.1 (2009), pp. 253–264.

[50] Tim Kraska et al. "MDCC: Multi-Data Center Consistency". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 113–126. ISBN: 978-1-4503-1994-2.

[51] Leslie Lamport. "Fast Paxos". In: *Distributed Computing* 19 (2 2006), pp. 79–103. ISSN: 0178-2770.

[52] Leslie Lamport. *Generalized Consensus and Paxos*. Tech. rep. MSR-TR-2005-33. Microsoft Research, 2005. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=64631.

[53] Leslie Lamport. "Paxos Made Simple". In: *SIGACT News* 32.4 (2001), pp. 51–58. ISSN: 0163-5700.

[54] Leslie Lamport. "The Part-Time Parliament". In: *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998), pp. 133–169.

[55] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. "An Optimistic Commit Protocol for Distributed Transaction Management". In: *Proc. of SIGMOD*. SIGMOD '91. Denver, Colorado, USA: ACM, 1991, pp. 88–97. ISBN: 0-89791-425-2.

[56] Wyatt Lloyd et al. "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS". In: *Proc. of SOSP*. Cascais, Portugal, 2011. ISBN: 978-1-4503-0977-6.

[57] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. "Mencius: Building Efficient Replicated State Machines for WANs". In: *Proc. of OSDI*. San Diego, California: USENIX Association, 2008, pp. 369–384.

[58] *MongoDB*. http://www.mongodb.org.

[59] Axel Mönkeberg and Gerhard Weikum. "Conflict-driven Load Control for the Avoidance of Data-Contention Thrashing". In: *ICDE*. 1991, pp. 632–639.

[60] *New Tweets per second record, and how!* `https://blog.twitter.com/2013/new-tweets-per-second-record-and-how`.

[61] Kevin O'Gorman, Divyakant Agrawal, and Amr El Abbadi. "Posse: A Framework for Optimizing Incremental View Maintenance at Data Warehouse". In: *Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery*. DaWaK '99. London, UK, UK: Springer-Verlag, 1999, pp. 106–115. ISBN: 3-540-66458-0.

[62] Chris Olston, Boon Thau Loo, and Jennifer Widom. "Adaptive Precision Setting for Cached Approximate Values". In: *SIGMOD Conference*. 2001, pp. 355–366.

[63] Patrick E. O'Neil. "The Escrow Transactional Method". In: *TODS* 11 (4 1986), pp. 405–430. ISSN: 0362-5915.

[64] M Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer Science & Business Media, 2011.

[65] Stacy Patterson et al. "Serializability, not Serial: Concurrency Control and Availability in Multi-Datacenter Datastores". In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1459–1470. ISSN: 2150-8097.

[66] Fernando Pedone. "Boosting System Performance with Optimistic Distributed Protocols". In: *IEEE Computer* 34.12 (Feb. 19, 2002), pp. 80–86.

[67] *PLANET*. `http://planet.cs.berkeley.edu`.

[68] X. Qian and Gio Wiederhold. "Incremental Recomputation of Active Relational Expressions". In: *IEEE Trans. on Knowl. and Data Eng.* 3.3 (Sept. 1991), pp. 337–341. ISSN: 1041-4347.

[69] Dallan Quass and Jennifer Widom. "On-line Warehouse View Maintenance". In: *Proc. of SIGMOD*. SIGMOD '97. Tucson, Arizona, USA: ACM, 1997, pp. 393–404. ISBN: 0-89791-911-4.

[70] Masoud Saeida Ardekani et al. *Non-Monotonic Snapshot Isolation*. English. Research Report RR-7805. INRIA, 2011, p. 34. URL: `http://hal.inria.fr/hal-00643430/en/`.

[71] Kenneth Salem et al. "How to Roll a Join: Asynchronous Incremental View Maintenance". In: *Proc. of SIGMOD*. SIGMOD '00. Dallas, Texas, USA: ACM, 2000, pp. 129–140. ISBN: 1-58113-217-4.

[72] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance". In: *Proc. VLDB Endow.* 3.1-2 (Sept. 2010), pp. 460–471. ISSN: 2150-8097.

[73] Eric Schurman and Jake Brutlag. *Performance Related Changes and their User Impact*. Presented at Velocity Web Performance and Operations Conference. 2009.

[74] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. "Scalaris: Reliable Transactional P2P Key/Value Store". In: *Erlang Workshop*. 2008.

[75] A. Segev and J. Park. "Updating Distributed Materialized Views". In: *IEEE Trans. on Knowl. and Data Eng.* 1.2 (June 1989), pp. 173–184. ISSN: 1041-4347.

[76] Arie Segev and Weiping Fang. "Currency-Based Updates to Distributed Materialized Views". In: *Proceedings of the Sixth International Conference on Data Engineering.* Washington, DC, USA: IEEE Computer Society, 1990, pp. 512–520. ISBN: 0-8186-2025-0.

[77] Shetal Shah, Krithi Ramamritham, and Prashant J. Shenoy. "Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers". In: *IEEE Trans. Knowl. Data Eng.* 16.7 (2004), pp. 799–812.

[78] Dale Skeen. "Nonblocking Commit Protocols". In: *Proc. of SIGMOD*. SIGMOD '81. Ann Arbor, Michigan: ACM, 1981, pp. 133–142. ISBN: 0-89791-040-0.

[79] Yair Sovran et al. "Transactional storage for geo-replicated systems". In: *Proc. of SOSP*. 2011.

[80] Michael Stonebraker. "The Case for Shared Nothing". In: *HPTS*. 1985,

[81] D. B. Terry et al. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 172–182. ISBN: 0-89791-715-4.

[82] Alexander Thomasian. "Thrashing in Two-Phase Locking Revisited". In: *ICDE*. 1992, pp. 518–526.

[83] Alexander Thomasian. "Two-Phase Locking Performance and Its Thrashing Behavior". In: *TODS* 18.4 (1993), pp. 579–625.

[84] Xin Zhang, Lingli Ding, and Elke A Rundensteiner. "Parallel multisource view maintenance". In: *VLDB J.* 13.1 (2004), pp. 22–48.

[85] Yang Zhang et al. "Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems". In: *Proc. of SOSP*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 276–291. ISBN: 978-1-4503-2388-8.

[86] Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. "Lazy Maintenance of Materialized Views". In: *Proc. of VLDB*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 231–242. ISBN: 978-1-59593-649-3.

[87] Yue Zhuge, Hector Garcia-Molina, and Janet L Wiener. "The Strobe Algorithms for Multi-Source Warehouse Consistency". In: *Parallel and Distributed Information Systems, 1996., Fourth International Conference on*. IEEE. 1996, pp. 146–157.

[88] Yue Zhuge et al. "View Maintenance in a Warehousing Environment". In: *Proc. of SIGMOD*. SIGMOD '95. San Jose, California, USA: ACM, 1995, pp. 316–327. ISBN: 0-89791-731-6.