Building scalable geo-replicated storage backends for web applications

by

Yair Sovran

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy Department of Computer Science New York University September 2012

Professor Jinyang Li

© Yair Sovran

All Rights Reserved, 2012

To Avital

&

my parents

Acknowledgements

First, and foremost, I would like to thank my advisor, Jinyang Li. I credit Jinyang for showing me the ropes and introducing me to the fascinating field of distributed systems. I couldn't have asked for a better introduction. Jinyang's sharp intellect and wealth of knowledge never cease to amaze me, and I will always be grateful for her guidance and encouragement.

I am grateful to all the current and former members of my PhD committee: Dennis Shasha, Lakshminarayanan Subramanian, Robert Grimm, Bryan Ford, and Marcos Aguilera.

I would like to express my special gratitude to Marcos Aguilera from Microsoft Research. Marcos has been a wonderful collaborator and an incredible person to work with and learn from. Marcos has been instrumental in developing and formalizing the ideas presented in this dissertation, and much of this work would not have been possible without his contributions.

I had the opportunity to collaborate with incredibly clever students at NYU, who contributed to the implementation of Walter and Lynx, developed applications to run on top of them, helped with experimentation, and through countless discussions, helped with the development of the ideas presented in this work. I would like to thank Russell Power, Nguyen Tran and Songbin Liu for their help and friendship.

I would also like to thank my mentors at Google, Sameer Bhola and Jeremy Chen, who over the course of two summers, gave me a glimpse of the real-world challenges of building and maintaining distributed systems.

I would like to thank my parents, brothers, and extended family for their support and love. I know that my time away from my loved ones, pursuing my degree, has been hard for them, as it was for me. Yet, they showed nothing but encouragement and support.

My greatest gratitude goes to my wife, Avital, who, on top of all the other major roles she plays in my life, was also my one-woman scientific advisory board and support group. Her constant source of love and understanding helped me push on and complete this work. Finally, I would like to thank my baby daughter, Romy, who has been my most joyous and gratifying distraction from doing work. I wouldn't have it any other way.

Abstract

Web applications increasingly require a storage system that is both scalable and can replicate data across many distant data centers or *sites*. Most existing storage solutions fall into one of two categories: Traditional databases offer strict consistency guarantees and programming ease, but are difficult to scale in a geo-replicated setting. *NoSQL* stores are scalable and efficient, but have weak consistency guarantees, placing the burden of ensuring consistency on programmers. In this dissertation, we describe two systems that help bridge the two extremes, providing scalable, geo-replicated storage for web applications, while also easy to program for.

Walter is a key-value store that supports transactions and replicating data across distant sites. A key feature underlying Walter is a new isolation property: *Parallel Snapshot Isolation* (PSI). PSI allows Walter to replicate data asynchronously, while providing strong guarantees within each site. PSI does not allow write-write conflicts, alleviating the burden of writing conflict resolution logic. To prevent write-write conflicts and implement PSI, Walter uses two new and simple techniques: preferred sites and counting sets.

Lynx is a distributed database backend for scaling latency-sensitive web applications. Lynx supports optimizing queries via data denormalization, distributed secondary indexes, and materialized join views. To preserve data constraints across denormalized tables and secondary indexes, Lynx relies on a novel primitive: *Distributed Transaction Chain* (DTC). A DTC groups a sequence of transactions to be executed on different nodes while providing two guarantees. First, all transactions in a DTC execute exactly once despite failures. Second, transactions from concurrent DTCs are interleaved consistently on common nodes.

We built several web applications on top of Walter and Lynx: an auction service, a microblogging service, and a social networking website. We have found that building web applications using Walter and Lynx is quick and easy. Our experiments show that the resulting applications are capable of providing scalable, low latency operation across multiple geo-replicated sites.

Contents

	Ded	cation	iv
	Ack	nowledgements	v
	Abst	ract	vii
	List	of Figures	xi
1	Intr	oduction	1
	1.1	Walter: A geo-replicated transactional key-value store	4
	1.2	Lynx: A scalable, eventually consistent database	6
	1.3	Contributions	9
	1.4	Dissertation organization	9
2	Ove	rview of Walter	11
3	Para	llel snapshot isolation	16
	3.1	Snapshot isolation	16
	3.2	Parallel Snapshot Isolation	19
4	Syst	em design and implementation of Walter	27
	4.1	Service	27
	4.2	Design and algorithms	31

5	Eval	uation of Walter	42
	5.1	Usability: Applications on top of Walter	42
	5.2	Performance evaluation	44
6	Over	rview of Lynx	55
	6.1	Application Challenges	55
	6.2	Overview	58
7	Lynx	's programming interface	61
	7.1	Creating and accessing tables	61
	7.2	Creating and accessing derived tables	63
	7.3	User-defined DTCs	65
8	Syste	em design and implementation of Lynx	67
	8.1	Design	67
	8.2	Implementation	73
	8.3	Correctness of Lynx's operations	73
9	Eval	uation of Lynx	88
	9.1	Usability: Applications on top of Lynx	88
	9.2	Performance evaluation	91
10	Rela	ted Work	100
	10.1	Walter	100
	10.2	Lynx	103
11	Cone	clusion	106

List of Figures

3.1	Specification of snapshot isolation.	17
3.2	Transaction outcome in snapshot isolation	17
3.3	Depiction of snapshot isolation	18
3.4	Specification of PSI.	19
3.5	Transaction outcome in PSI.	20
3.6	Commit ordering in PSI	20
3.7	Set operations in PSI specification.	22
3.8	Anomalies allowed by each isolation property	24
4.1	State kept on Walter servers	32
4.2	Executing Walter transactions	33
4.3	Walter Fast commit	34
4.4	Walter Slow commit	35
4.5	Walter Transaction replication	36
4.6	Basic C++ API for Walter and C++ and PHP examples	39
5.1	Transaction for <i>befriend</i> operation in WaltSocial.	42
5.2	Walter base read and write throughput	46
5.3	Aggregate transaction throughput on EC2	48
5.4	Fast commit latency on EC2 and our private cluster.	49

5.5	Replication latency for disaster-safe durability.	49
5.6	Latency of slow commit and replication.	51
5.7	Transaction size and throughput for Waltsocial operations.	53
5.8	Latency of WaltSocial operations.	53
5.9	Throughput of ReTwis using Redis and Walter.	54
6.1	A simple auction service	56
7.1	Schema of the partitioned <i>Users</i> and <i>Items</i> table	62
7.2	Accessing the Users table.	62
7.3	Specifying secondary indexes	63
7.4	Querying join table <i>Bids-Users</i> with secondary key	64
7.5	Specifying a user-defined DTC.	64
8.1	An overview of different components in Lynx and their interactions	68
8.2	Updating a secondary key on row insertion	71
8.3	Updating a join key value on row insertion	71
9.1	DTC execution throughput for several workloads.	92
9.2	Distribution of DTC completion latency.	92
9.3	L-Twitter throughput for different workloads.	95
9.4	L-Twitter operation latency.	95
9.5	L-RUBiS throughput for different workloads.	96
9.6	L-RUBiS operation latency.	97
9.7	Lynx aggregate throughput with geo-replication	98

Chapter 1

Introduction

Web applications are everywhere. The astounding growth of the World Wide Web, along with advances in Internet technology, have paved the way for for Internet applications that provide an increasing variety of services. These online activities include, among others, sending and reading email, blogging, editing documents collaboratively, and connecting with people on social networks. The trend towards web-based applications is only getting stronger.

There are two major concerns when building a popular web application. First, it must be able to scale up quickly to handle a rapidly growing user base. For example, the popular photo sharing application Instagram saw a nearly exponential growth to over 30 million users in less than two years [58]. Another popular application—the microblogging site Twitter—has grown, in under six years, to serving over 140 million users who post on average 340 million messages (or *Tweets*) a day [16].

Second, a well-designed web application should remain available in the face of different types of failures. As these applications run in the "cloud", the most damaging type of failure is the outage of an entire data center. To tolerate a data center failure, applications must *geo-replicate*, that is, they must replicate their data across data centers in different geographical regions. The need for geo-replication becomes evident after every major data center outage. For example, errors in a

single infrastructure component in Amazon's EC2 service triggered a snowball effect, on April 2011, that caused a disruption of service for Amazon's entire US East region [12]. Since the EC2 service does not, by default, replicate data across regions on behalf of its users, applications that did not implement their own geo-replication functionality experienced a disrupted service for up to 47 hours.

Achieving scalability and geo-replication is not an easy task. Web applications are commonly constructed using a multi-tiered design, where application servers use a storage tier to store and share data. Scaling the application tier is easily achieved by running application severs on many machines across multiple data centers, but it is much harder to scale the storage tier and have it support geo-replication. To be scalable, a storage system must divide data into a large number of partitions spread across many machines. As applications often need to access data belonging to multiple partitions, the storage system must coordinate access across different partitions, which comes at a performance cost. The cost of such coordination increases substantially when data is replicated across geographically distant data centers with tens or hundreds of milliseconds of communication delays.

There are no satisfactory storage solutions for building scalable and geo-replicated web applications. Small web-sites often use traditional databases as their storage tier. Traditional databases provide strong consistency guarantees, and are easy to program for, but they face issues when scalability and geo-replication are required. For example, faced with the inability of their PostgreSQL database backend to scale, the engineers at Instagram resorted to using manually-partitioned PostgreSQL tables, result caching via memcached, and duplication of photo timelines in a Redis keyvalue store [58]. The resulting system has no consistency guarantees: data kept in different systems (e.g., memcached, Redis, PostgreSQL) can become arbitrarily out of sync [58, 73]. This greatly complicates application development and can lead to a bad user experience. For replication, traditional databases commonly rely on master-slave schemes. These schemes are limited as a geo-replication solution in that backup sites are either read-only, or they provide no consistency guarantees across partitions.

The goal of this dissertation is to develop scalable, geo-replicated storage backends for web applications that are also easy to program for. Our backends support transactions, which we view as an essential programming primitive for simplifying application development. Both systems in this work explore the tradeoffs of relaxing typical ACID (Atomicity Consistency Isolation Durability) consistency for the benefit of scalable high performance in a geo-replicated setting. We show through examples and experiments that in the context of web applications, as long as the consistency level is strong enough to hide most anomalies from users, these tradeoffs make sense: They make it easy to quickly write applications that can achieve scalable high-performance, in a geo-replicated setting. This thesis develops two storage systems along this theme: One is called Walter, and the other is called Lynx.

The first system presented in this work is Walter, a geo-replicated key-value store. The focus of our work on Walter was to develop the strongest possible weak consistency model that can be efficiently implemented in a geo-replicated setting. For this goal we developed a novel isolation property called *Parallel Snapshot Isolation* (PSI). PSI allows Walter to replicate data asynchronously, while providing strong guarantees within each site. We demonstrate how Walter utilizes PSI to provide efficient transactional access to geo-replicated data.

The second system we propose is Lynx, a scalable geo-replicated database. The emphasis of our work on Lynx is on scalability and a user-friendly programming interface. Lynx retains many of the desirable programming features of a relational database, such as a flexible query interface. Lynx achieves scalable performance in a geo-replicated setting by providing a relaxed consistency model where the state of distributed secondary indexes, materialized join views, and denormalized data is permitted to temporarily lag behind the state of main tables, but eventually reflects the same state. Lynxes uses a novel primitive called *Distributed Transaction Chain* (DTC) to achieve its consistency guarantee with good performance.

The rest of this chapter is organized as follows: Section 1.1 and 1.2 discuss Walter and Lynx in more detail. We summarize our contributions in section 1.3. The chapter concludes with a description of the organization of the rest of the dissertation.

1.1 Walter: A geo-replicated transactional key-value store

In the first part of this thesis, we investigate a relaxed consistency model called *Parallel Snapshot Isolation* (PSI). This model is proposed for supporting efficient transactions over georeplicated data. We build a key-value store called Walter, as well as several web applications, to demonstrate the performance and usefulness of PSI.

Existing geo-distributed key-value stores provide no transactions or only restricted transactions (see Section 10). Without transactions, an application must carefully coordinate access to data to avoid race conditions, partial writes, overwrites, and other hard problems that cause erratic behavior. Developers must address these same problems for many applications. With transactions, developers are relieved from concerns of atomicity, consistency, isolation, durability, and coordination. For example, in a social networking application, one may want to remove user A from B's friends list and vice versa. Without transactions, developers must write code carefully to prevent one removal from happening without the other. With transactions, developers simply bundle those updates in a transaction.

To realize strong consistency, transactions must synchronously replicate data across geographically distant data centers, resulting in significantly increased operation latency. Our new isolation property PSI provides a better balance between consistency and latency [48, 94], as appropriate for web applications. In such applications, a user might log into the site closest to her, where she accesses application servers, ad servers, authentication servers, etc. These hosts should observe a consistent storage state. For example, in a social network, a user expects to see her own posts immediately and in order. For that reason, the storage system should provide a strong level of consistency among hosts in her site. Across sites, weaker consistency is *acceptable*, because users can tolerate a small delay for their actions to be seen by other users. A weaker consistency is also *desirable*, so that transactions can be replicated across sites asynchronously (lazy replication).

Eventual consistency [79, 86] is often the property provided by asynchronous replication. When different sites update the same data concurrently, there is a conflict that must be resolved by application logic. This logic can be complex, and we want to avoid forcing it upon developers.

With PSI, hosts within a site observe transactions according to a consistent snapshot and a common ordering of transactions. Across sites, PSI enforces only causal ordering, not a global ordering of transactions, allowing the system to replicate transactions asynchronously across sites. With causal ordering, if Alice posts a message that is seen by Bob, and Bob posts a response, no user can see Bob's response without also seeing Alice's original post. Besides providing causal ordering, PSI precludes write-write conflicts (two transactions concurrently writing to the same object) so that developers need not write conflict resolution logic.

To prevent write-write conflicts and implement PSI, Walter relies on two techniques: *preferred sites* and *counting sets*. In web applications, writes to an object are often made by the user who owns the object, at the site where this user logs into. Therefore, we assign each object to a *preferred site*, where objects can be written more efficiently. For example, the preferred site for the wall posts of a user is the site closest to the user. Preferred sites are less restrictive than primary sites, as we discuss in Section 2.

Preferred sites may not always suffice. For example, a friends list can be updated by users in many sites. The second technique in Walter to avoid conflicts is to use a new simple data type called a counting set (cset), inspired by commutative data types [60]. A cset is like a set, except that each element has an integer count. Unlike sets, csets operations are commutative, and so they never conflict [52]. Therefore, transactions with csets can commit without having to check for conflicts across sites. When developing applications for Walter, we used csets extensively to store friend lists, message walls, photo albums, and message timelines. We found that csets were

versatile and easy to use.

Walter uses multi-version concurrency control within each site, and it can quickly commit transactions that write objects at their preferred sites or that use csets. For other transactions, Walter resorts to two-phase commit to check for conflicts. We found that the latter type of transaction can be avoided in the applications we built.

Using Walter as the storage system, we build WaltSocial, a Facebook-like social networking application, and we port a third-party Twitter-clone called ReTwis [13]. We find that the transactions provided by Walter are effective and efficient. Experiments on four geographic locations on Amazon EC2 show that transactions have low latency and high throughput. For example, the operation to post a message on a wall in WaltSocial has a throughput of 16500 ops/s and the 99.9-percentile latency is less than 50 ms.

1.2 Lynx: A scalable, eventually consistent database

The second part of this thesis develops Lynx, a scalable, geo-replicated database designed for achieving low-latency queries.

One of the lessons we have learned from writing applications for Walter is that a key-value store, though simple to program with, lacks desirable features for issuing efficient queries, such as secondary index queries and joins. As a result, we chose to design Lynx on top of a relational database, and provide these added features to applications.

In Lynx, each table is split into partitions which are spread across many machines. For better fault-tolerance, Lynx can also optionally replicate data across several geographically distributed data centers. As web applications demand low latency operation, Lynx allows for three common query optimization patterns: denormalization [64], distributed secondary indexes, and materialized join views. These optimizations essentially pre-compute results so that a query can be satisfied by contacting just one machine, minimizing the latency and overhead for common read operations.

Lynx uses a new primitive, called the distributed transaction chain (DTC), to update secondary indexes and join tables. It also exposes the DTC primitive to the application programmer to update denormalized data in related tables.

Denormalization and index/view generation impose consistency constraints among data partitions managed by different machines. One way of enforcing these constraints is to use distributed ACID transactions. But this is a heavy hammer, and comes at a high price [17, 48]. Distributed ACID transactions require tight coordination among machines that manage different data partitions, lengthening the tail latency of operations. When data is replicated across data centers, such coordination requires communication between data centers, further increasing operation latency.

Instead of ACID, Lynx offers the weaker guarantee that denormalized data, indexes and join views are eventually consistent with the main data. We show that for a variety of applications this guarantee is sufficient for correct operation.

Achieving Lynx's consistency guarantee without distributed transactions is a challenge. When a logical operation is broken up into a series of independent steps, both failure and concurrency can result in permanent inconsistency between related tables and indexes. Lynx addresses these challenges through its new DTC primitive. A DTC bundles a group of local transactions and executes them in order, providing two guarantees. First, either all transactions in a chain finish successfully or, if any transaction aborts, each of the previously executed transactions is un-done with a compensating action. Second, if desired, transactions from different chains can be ordered: if two chains X and Y start in the same partition and X executes before Y in that partition, then X executes before Y in any partition where they both execute.

DTC is more powerful than a persistent message queue, a popular technique for modifying data on different machines—examples of this include Amazon's Simple Queue Service [3] and eBay's message queue [74] among others. Compared to DTC, the queue interface is low-level and transfers the burden of handling the difficult cases onto the programmers: programmers must not only design applications to explicitly enqueue and dequeue transactions, they must also ensure that

no transaction is executed more than once after failure recovery and that the arbitrary interleaving of transactions of different logical operations does not lead to inconsistencies.

Lynx is easy to use: programmers define table schemas using a SQL-like syntax and use a Python client library to access distributed tables. To help programmers optimize application performance, Lynx provides mechanisms to (a) control how various database tables should be partitioned and replicated across data centers, (b) specify which secondary indexes and join view tables to generate, and (c) let programmers write user-defined DTCs to update denormalized data in related tables.

Using Lynx, we have built three applications: an auction service, which we have ported from the RUBiS benchmark [1, 22]; a Twitter-like microblogging service; and a Facebook-like social networking site. All three applications are quick to build, demonstrating the ease-of-use of the Lynx API. They use DTCs to maintain the consistency of denormalized tables and rely heavily on secondary indexes and join views to optimize read operations. Experiments have shown that these applications have scalable performance and low latency operations. When scaling the number of Lynx servers used from 1 to 15, our Twitter-like application achieves 8x throughput increase in a mixed workload of read and write operations. Furthermore, all operations finish with a median latency of 10 ms and a maximum of less than 60 ms.

1.3 Contributions

The contributions of this dissertation are:

- We define Parallel Snapshot Isolation, an isolation property well-suited for geo-replicated web applications. PSI provides a strong guarantee within a site; across sites, PSI provides causal ordering and precludes write-write conflicts.
- We describe the design and implementation of Walter, a geo-replicated transactional keyvalue store that provides PSI. Walter can avoid common write-write conflicts without crosssite communication using two simple techniques: preferred sites and csets. We give distributed protocols to execute and commit transactions in Walter.
- We describe the design and implementation of Lynx, a scalable, geo-replicated database that provides an efficient query interface for large-scale, latency-sensitive web applications.
- We propose *Distributed Transaction Chains* and demonstrate how they can be used to implement efficient queries in a distributed database.
- We build five web applications: two on top of Walter, and three on top of Lynx. We show that these application are easy to build and optimize using the chosen storage backend, and that they work well under its consistency guarantees, while achieving high performance in a geo-replicated setting.

1.4 Dissertation organization

This dissertation is organized as follows: Chapter 2 gives an overview of Walter. Chapter 3 introduces Parallel Snapshot Isolation in the context of Snapshot Isolation. Chapter 4 describes Walter's operation in detail: Its programming interface, design, algorithms, and its implementation. Chapter 5 is an evaluation of Walter that consists of a description of our experience writing applications for it; and an experimental evaluation of the Walter prototype and the applications

running on top of it. Chapter 6 switches to discuss Lynx and contains an overview of the system. Chapters 7 describe Lynx's programming interface and its design, and presents proofs of the correctness of Lynx's derived tables' operations. Chapter 9 evaluates Lynx's usability and performance. Chapter 10 discusses related work, and Chapter 11 concludes.

Chapter 2

Overview of Walter

Setting. A geo-replicated storage system replicates objects across multiple sites. The system is managed by a single administrative entity. Machines can fail by crashing; addressing Byzantine failures is future work. Network partitions between sites are rare: sites are connected by highly-available links (e.g., private leased lines or MPLS VPNs) and there are redundant links to ensure connectivity during planned periods of link maintenance (e.g., using a ring topology across sites). We wish to provide a useful back-end storage system for web applications, such as social networks, web email, social games, and online stores. The storage system should provide reliability, a simple interface and semantics, and low latency.

Why transactions? We illustrate the benefit of transactions in a social networking application, where users post photos and status updates, befriend other users, and write on friends' walls. Each site has one or more application servers that access shared user data. When Alice adds a new photo album, the application creates an object for the new album, posts a news update on Alice's wall, and updates her album set. With transactions, the application groups these writes into an atomic unit so that failures do not leave behind partial writes (atomicity) and concurrent access by other servers are not intermingled (isolation). Without transactions, the application risks exposing undesirable inconsistent state to end users. For example, Bob may see the wall post that Alice

has a new album but not find the album. Developers can sometimes alleviate these inconsistencies manually, by finding and ensuring proper ordering of writes. For example, the application can create the new album and wait for it to be replicated before posting on the wall. Then, concurrent access by Bob is not a problem, but a failure may leave behind an orphan album not linked to any user. The developer can deal with this problem by logging and replaying actions—which amounts to implementing rudimentary transactions—or garbage collecting dangling structures. This non-transactional approach places significant burden on developers.

We are not the first to point out the benefits of transactions to data center applications. Sinfonia uses transactions for infrastructure services [18, 19], while Percolator [71] uses them for search indexing. Both systems target applications on a single site, whereas we target geo-replicated applications that span many sites.

One way to provide transactions in a geo-replicated setting is to partition the data across several databases, where each database has its primary at a different site. The databases are replicated asynchronously across all sites, but each site is the primary for only one of the partitions. Unfortunately, with this solution, transactions cannot span multiple partitions, limiting their utility to applications.

Key features. Walter provides a unique combination of features to support geo-replicated web applications:

- *Asynchronous replication across sites*. Transactions are replicated lazily in the background, to reduce latency.
- *Efficient update-anywhere for certain objects*. Counting sets can be updated efficiently anywhere, while other objects can be updated efficiently at their preferred site.
- Freedom from conflict-resolution logic, which is complex and burdensome to developers.
- *Strong isolation within each site*. This is provided by the PSI property, which we cover below.

Existing systems do not provide some of the above features. For instance, eventually consistent systems such as [79, 86] require conflict-resolution logic; primary-copy database systems do not support any form of update-anywhere. We discuss related work in more detail in Chapter 10.

Overview of PSI. Snapshot isolation [25] is a popular isolation condition provided by commercial database systems such as Oracle and SQLServer. Snapshot isolation ensures that (a) transactions read from a snapshot that reflects a single commit ordering of transactions, and (b) if two concurrent transactions have a write-write conflict, one must be aborted. By imposing a single commit ordering, snapshot isolation forces implementations to coordinate transactions on commit, even when there are no conflicts (Section 3.1).

Parallel snapshot isolation extends snapshot isolation by allowing different sites to have different commit orderings. For example, suppose site A executes transactions T_1, T_2 and site B executes transactions T_3, T_4 . PSI allows site A to first incorporate just T_1, T_2 and later T_3, T_4 , while site B first incorporates T_3, T_4 and later T_1, T_2 . This flexibility is needed for asynchronous replication: site A (or site B) can commit transactions T_1, T_2 (or T_3, T_4) without coordinating with the other site and later propagate the updates.

Although PSI allows different commit orderings at different sites, it still preserves the property of snapshot isolation that committed transactions have no write-write conflicts, thereby avoiding the need for conflict resolution. Furthermore, PSI preserves causal ordering: if a transaction T_2 reads from T_1 then T_1 is ordered before T_2 at every site. We give a precise specification of PSI in Chapter 3.

We believe PSI provides strong guarantees that are well-suited for web applications. Intuitively, PSI provides snapshot isolation for all transactions executed within a single site. PSI's relaxation over snapshot isolation is acceptable for web applications where each user communicates with one site at a time and there is no need for a global ordering of all actions across all users. In a social networking application, Alice in site A may post a message at the same time as Bob in site B. Under PSI, Alice may see her message first before seeing Bob's message, and Bob sees the opposite ordering, which is reasonable since Alice and Bob post concurrently. As another example, in an auction application, PSI allows bids on different objects to be committed in different orders at different sites. (In contrast, snapshot isolation requires the same ordering at all sites.) Such relaxation is acceptable since the auction application requires bid ordering on each object separately, not across all objects.

Avoiding conflicts efficiently. To avoid write-write conflicts across sites, and implement PSI, Walter uses two techniques.

- Preferred sites. Each object is assigned a preferred site, which is the site where writes to the object can be committed without checking other sites for write conflicts. Walter executes and commits a transaction quickly if all the objects that it modifies have a preferred site where the transaction executes. Objects can be updated at any site, not just the preferred site. In contrast, some database systems have the notion of a primary site, which is the only site that can update the data. This notion is more limiting than the notion of a preferred site. For instance, suppose objects O_1 and O_2 are both replicated at sites 1 and 2, but the primary of O_1 is site 1 while the primary of O_2 is site 2. A transaction executing on site 1 can read both objects (since they are both replicated at site 1), but because the primary of O_2 is not site 1, the transaction can write only O_1 —which is limiting to applications. In practice, this limitation is even more severe because database systems assign primary sites at the granularity of the whole database, and therefore non-primary sites are entirely read-only.
- *Conflict-free counting set objects.* Sometimes an object is modified frequently from many sites and hence does not have a natural choice for a preferred site. We address this problem with counting set (cset) objects. Transactions in Walter support not just read and write operations, but also operations on csets. Csets have the desirable property that transactions concurrently accessing the cset object never generate write-write conflicts. A cset is similar

to a multiset in that it keeps a count for each element. But, unlike a multiset, the count could be negative [52]. A cset supports an operation add(x) to add element x, which increments the counter of x in the cset; and an operation rem(x) to remove x, which decrements the counter of x. Because increment and decrement commute, add and rem also commute, and so operations never conflict.

For example, a group of concurrent cset operations can be ordered as add(x), add(y), rem(x) at one site, and ordered as rem(x), add(x), add(y) at another site. Both reach the final state containing just y with count 1. Note that removing element x from an empty cset results in -1 copies of element x, which is an *anti-element*: later addition of x to the cset results in the empty cset.

Chapter 3

Parallel snapshot isolation

In this chapter, we precisely specify PSI—the guarantee provided by Walter—and we discuss its properties and implications. We start by reviewing snapshot isolation and explaining the frame-work that we use to specify properties (Section 3.1). Then, we give the exact specification of PSI and discuss its properties (Section 3.2.1). We next explain how to extend PSI to include set operations (Section 3.2.2). We then explain how developers can use PSI (Section 3.2.3) and csets (Section 3.2.4) to build their applications.

3.1 Snapshot isolation

We specify snapshot isolation by giving an abstract specification code that an implementation must emulate. The specification code is centralized to make it as simple as possible, whereas an implementation can be distributed, complex, and more efficient. An implementation code satisfies the specification code if both codes produce the same output given the same input (e.g., [63]). The input is given by calls to operations to start a transaction, read or write data, commit a transaction, etc. The output is the return value of these operations. Many clients may call the operations of the specification concurrently, resulting possibly in many outstanding calls; however, the body of each

operation startTx(x) x.startTs ← new monotonic timestamp return ok operation write(x, oid, data) append ⟨oid, DATA(data)⟩ to x.updates return ok operation read(x, oid) return state of oid from x.updates and Log up to timestamp x.startTs operation commitTx(x) x.commitTs ← new monotonic timestamp x.status ← chooseOutcome(x) if x.status = COMMITTED then append x.updates to Log with timestamp x.commitTs return x.status

Figure 3.1: Specification of snapshot isolation.

function chooseOutcome(x)
if some write-conflicting transaction has committed after x started
then return ABORTED
else if some write-conflicting transaction has aborted after x started
or is currently executing
then return (either ABORTED OR COMMITTED) // non-deterministic choice
else return COMMITTED

Figure 3.2: Transaction outcome in snapshot isolation.

operation is executed one at a time, using a single thread.

The specification is given in Figures 3.1 and 3.2 and depicted in Figure 3.3. It is assumed that clients start a transaction x with x initially \perp , then perform a sequence of reads and/or writes, and then try to commit the transaction. The behavior is unspecified if any client fails to follow this discipline, say by writing to a transaction that was never started. To start transaction x, the code obtains a new monotonically increasing timestamp, called the *start timestamp* of x. The timestamp is stored as an attribute of x; in the code, x is passed by reference. To write an object in transaction x, the code uses the update buffer—to check for any updates to the object written by the transaction itself—as well as a snapshot of the state when the transaction began. To determine the snapshot, the code maintains a *Log* variable with a sequence of object ids, data, and timestamps for the writes of previously-committed transactions. Only committed transaction's start timestamp. To commit transaction x, the code obtains a new monotonically increasing the updates in *Log* up to the transaction's start timestamp. To commit transaction x, the code obtains a new monotonically increasing timestamp.



Figure 3.3: Depiction of snapshot isolation. The writes of T_1 are seen by T_3 but not T_2 as T_2 reads from a snapshot prior to T_1 's commit.

called the *commit timestamp* of x. It then determines the outcome of a transaction according to the function in Figure 3.2. This function indicates the cases when the outcome is abort, commit, or either one chosen nondeterministically.¹ The code considers what happens after x started: if some write-conflicting transaction committed then the outcome is abort, where a *write-conflicting transaction* is one that writes an object that x also writes. Otherwise if some write-conflicting transaction has aborted or is currently executing—meaning it has started but its outcome has not been chosen—then the outcome is either abort or commit, chosen nondeterministically. Otherwise, the outcome is commit. If the outcome is commit, the writes of x are appended to Log with x's commit timestamp.

Note that the specification keeps internal variables—such as the log, timestamps, and other attributes of a transaction—but an implementation need not have these variables. It needs to emulate only the return values of each operation.

The above specification of snapshot isolation implies that any implementation must satisfy two key properties [91, Page 362]:

SI property 1. (Snapshot Read) All operations read the most recent committed version as of the time when the transaction began.

SI property 2. (*No Write-Write Conflicts*) *The write sets of each pair of committed concurrent transactions must be disjoint.*

Here, we say that two committed transactions are *concurrent* if one of them has a commit timestamp between the start and commit timestamp of the other.

¹Nondeterminism in specifications allows implementations to have either behavior.

operation startTx(x) $x.startTs \leftarrow$ new monotonic timestamp return ok **operation** write(x, oid, data)append (oid, DATA(data)) to x.updates return OK **operation** read(x, oid)**return** state of oid from x.updates and Log[site(x)] up to timestamp x.startTs **operation** commitTx(x) $x.commitTs[site(x)] \leftarrow$ new monotonic timestamp $x.status \leftarrow chooseOutcome(x)$ if x.outcome = COMMITTED append x.updates to Log[site(x)] with timestamp x.commitTs[site(x)] return x.status **upon** $[\exists x, s: x.status = COMMITTED and x.commitTs[s] = <math>\bot$ and $\forall y \text{ such that } y. \textit{commitTs}[\textit{site}(x)] < x.\textit{startTs} : y.\textit{commitTs}[s] \neq \bot$] $x.commitTs[s] \leftarrow$ new monotonic timestamp append x.updates to Log[s] with timestamp x.commitTs[s]

Figure 3.4: Specification of PSI.

Snapshot isolation is inadequate for a system replicated at many sites, due to two issues. First, to define snapshots, snapshot isolation imposes a total ordering of the commit time of all transactions, even those that do not conflict². Establishing such an ordering when transactions execute at different sites is inefficient. Second, the writes of a committed transaction must be immediately visible to later transactions. Therefore a transaction can commit only after its writes have been propagated to all remote replicas, thereby precluding asynchronous propagation of its updates.³ We define PSI to address these problems.

3.2 Parallel Snapshot Isolation

3.2.1 Specification of PSI

We define PSI as a relaxation of snapshot isolation so that transactions can propagate asynchronously and be ordered differently across sites. Note that the PSI specification does not refer to

²For example, suppose A=B=0 initially and transaction T_1 writes $A\leftarrow 1$, transaction T_2 writes $B\leftarrow 1$, and both commit concurrently. Then T_1 and T_2 do not conflict and can be ordered arbitrarily, so either (A=1, B=0) or (A=0, B=1) are valid snapshots for transactions to read. However, it is illegal for both snapshots to occur, because snapshot isolation either orders T_1 before T_2 or vice versa.

³A variant called weak snapshot isolation [36] allows a transaction to remain invisible to others even after it commits, but that does not address the first issue above.

function chooseOutcome(x)
if some write-conflicting transaction has committed at site(x) after x started
or is currently propagating to site(x) // text has definition of "propagating"
then return ABORTED
else if some write-conflicting transaction has aborted after x started
or is currently executing
then return (either ABORTED or COMMITTED)
else return COMMITTED



Figure 3.5: Transaction outcome in PSI.

Figure 3.6: PSI allows a transaction to have different commit times at different sites. At site A, committed transactions are ordered as T1, T2. Site B orders them differently as T2, T1.

preferred sites, since they are relevant only to the implementation of PSI. The specification code is given in Figures 3.4 and 3.5 and depicted in Figure 3.6. As before, the specification is abstract and centralized—there is a single thread that executes the code without interleaving—but we expect that implementations will be distributed. Each transaction x has a site attribute denoted site(x). There is a log per site, kept in a vector *Log* indexed by sites. A transaction has one commit timestamp per site. A transaction first commits locally, by writing its updates to the log at its site; subsequently, the transaction propagates to and commits at the remote sites. This propagation is performed by the upon statement which, at some non-deterministic time, picks a committed transaction x and a site s to which x has not been propagated yet, and then writes the updates of x to the log at s. (For the moment, we ignore the second line of the upon statement in the code.) As Figure 3.5 shows, a transaction is aborted if there is some write-conflicting transaction that has committed at site(x) after x started or that is currently propagating to site(x); a transaction y is propagating to a site s if its status is committed but it has not yet committed at site s—that is, y.status=COMMITTED and y.commitTs[s]= \perp . Otherwise, if there is some concurrent writeconflicting transaction that has not committed, the outcome can be abort or commit. Otherwise, the outcome is commit. The outcome of a transaction is decided only once: if it commits at its site, the transaction is not aborted at the other sites. In Section 4.2.7, we discuss what to do when a site fails.

The above specification contains code that may be expensive to implement directly, such as monotonic timestamps and checks for write conflicts of transactions in different sites. We later give a distributed implementation that can avoid these inefficiencies.

From the specification, it can be seen that PSI replaces property 1 of snapshot isolation with the following:

PSI property 1. (*Site Snapshot Read*) All operations read the most recent committed version at the transaction's site as of the time when the transaction began.

Intuitively, a transaction reads from a snapshot established at its site. In addition, PSI essentially preserves property 2 of snapshot isolation. To state the exact property, we say two transactions T_1 and T_2 are *concurrent at site* s if one of them has a commit timestamp at s between the start and commit timestamp of the other at s. We say the transactions are *somewhere-concurrent* if they are concurrent at *site*(T_1) or at *site*(T_2).

PSI property 2. (*No Write-Write Conflicts*) *The write sets of each pair of committed somewhereconcurrent transactions must be disjoint.*

This property prevents the lost update anomaly (Section 3.2.3). The specification of PSI also ensures causal ordering:

PSI property 3. (*Commit Causality Across Sites*) If a transaction T_1 commits at a site A before a transaction T_2 starts at site A, then T_1 cannot commit after T_2 at any site.

This property is ensured by the second line of the upon statement in Figure 3.4: x can propagate to a site s only if all transactions that committed at x's site before x started have already propagated to s. The property prevents a transaction x from committing before y at a remote site when x has observed the updates of y. The property also implies that write-conflicting transactions are committed in the same order at all sites, to prevent the state at different sites from diverging permanently.

3.2.2 PSI with cset objects

In the specification of PSI in Section 3.2.1, transactions operate on objects via read and write operations, but it is possible to extend the specification to support objects with other operations. We give the extension for cset objects, but this extension should apply to any object with commutative operations. To add an element to a cset, the code appends an entry $\langle setid, ADD, id \rangle$ to the transaction's update buffer (*x.updates*) and, on commit, appends this entry to the log. Similarly, to remove an element from a cset, the code appends entry $\langle setid, DEL, id \rangle$. To read a cset, the code computes the state of the cset: for each element, it sums the number of ADD minus the number of DEL in the log and the update buffer, thus obtaining a count for each element. Only elements with a non-zero count are returned by the read operation. Because the operations to add and remove elements in a cset commute, these operations do not cause a write conflict. Note that a cset object does not support a write operation since it does not commute with ADD. Figure 3.7 shows the code of the specification.

A cset may have many elements, and reading the entire cset could return large amounts of data. It is easy to extend the specification with an operation *setReadId* to return the count of a chosen element on a cset, by simply computing the state of the cset (using the log) to extract the count of

that element. operation setAdd(x, setid, id) append (setid, ADD(id)) to x.updates return ок operation setDel(x, setid, id) append (setid, DEL(id)) to x.updates return ок operation setRead(x, setid) return state of setid from x.updates and Log[site(x)] up to timestamp x.startTs

Figure 3.7: Set operations in PSI specification.

3.2.3 Using PSI

One way to understand an isolation property is to understand what type of anomalous behavior it allows, so that developers know what to expect. In this section, we consider PSI from that standpoint, and we compare it against snapshot isolation and serializability. It is well-known that the weaker a property is, the more anomalous behaviors it has, but at the same time, the more efficiently it can be implemented. The anomalies allowed by PSI can be seen as the price to pay for allowing asynchronous replication.

Figure 3.8 shows various anomalies and whether each isolation property has those anomalies. Eventual consistency is very weak and allows all anomalies. The first three anomalies are well-known (e.g., [50]). Snapshot isolation and PSI prevent dirty and non-repeatable reads, because a transaction reads from a snapshot, and they prevent lost updates because there are no write-write conflicts. Snapshot isolation allows the state to fork, because two or more transactions may read from the same snapshot and make concurrent updates to different objects. We call this a *short fork*, also known as *write skew*, because the state merges after transactions commit. With PSI, the state may remain forked after transactions commit (when they execute in different sites), but the state is later merged when the transactions propagate across sites. Due to its longer duration, we call this a *long fork*. A *conflicting fork* occurs when the states diverges due to conflicting updates, which is not allowed by PSI.

Long forks are acceptable in web applications when users in a site do not expect their updates to be instantly visible across all sites. If the user wants to know that her updates are visible everywhere, she can wait for her transaction to commit at all sites. In some cases, the fork may be noticeable to users: say, Alice posts a message on her social network wall saying that she is the first to flag a new promotion; she then confirms her statement by reading her friend's walls and seeing nothing there. With a long fork, Bob could be simultaneously doing the same thing from a different site, so that both Alice and Bob believe they posted their message first. One way

Anomaly	Serializability	Snapshot Isolation	PSI	Eventual Consis- tency
Dirty read	No	No	No	Yes
Non-repeatable read	No	No	No	Yes
Lost update	No	No	No	Yes
Short fork	No	Yes	Yes	Yes
Long fork	No	No	Yes	Yes
Conflicting fork	No	No	No	Yes

Dirty read. A transaction reads the update made by another transaction that has not yet committed; the other transaction may later abort or rewrite the object, making the data read by the first transaction invalid. *Example.* Initially A=0. T_1 writes $A \leftarrow 1$ and $A \leftarrow 2$ and commits; concurrently, T_2 reads A=1.

Non-repeatable read. A transaction reads the same object twice—once before and once after another transaction commits an update to it—obtaining different results. *Example.* Initially A=0. T_1 writes $A \leftarrow 1$ and commits; concurrently T_2 reads A=0 and then reads A=1.

Lost update. Transactions make concurrent updates to some common object, causing one transaction to lose its updates. *Example.* Initially A=0. T_1 reads A=0, writes $A\leftarrow 1$, and commits. Concurrently, T_2 reads A=0, writes $A\leftarrow 2$, and commits.

Short fork. Transactions make concurrent disjoint updates causing the state to fork. After committing, the state is merged back. *Example.* Initially A=B=0. T_1 reads A=B=0, writes $A\leftarrow 1$, and commits. Concurrently, T_2 reads A=B=0, writes $B\leftarrow 1$, and commits. Subsequently, T_3 reads A=B=1.

Long fork. Transactions make concurrent disjoint updates causing the state to fork. After they commit, the state may remain forked but it is later merged back. *Example*. Initially A=B=0. T_1 reads A=B=0, writes $A \leftarrow 1$, and commits; then T_2 reads A=1, B=0. T_3 and T_4 execute concurrently with T_1 and T_2 , as follows. T_3 reads A=B=0, writes $B \leftarrow 1$, and commits; then T_4 reads A=0, B=1. Finally, after T_1, \ldots, T_4 finish, T_5 reads A=B=1.



Conflicting fork. Transactions make concurrent conflicting updates causing the state to fork in a way that requires application-specific or ad-hoc rules to merge back. *Example.* Initially A=0. T_1 writes $A\leftarrow1$ and commits. Concurrently, T_2 writes $A\leftarrow2$ and commits. Some external logic determines that the value of A should be 3, and subsequently T_3 reads A=3.

Figure 3.8: Anomalies allowed by each isolation property.

to avoid possible confusion among users is for the application to show an "in-flight" mark on a freshly posted message; this mark is removed only when the message has been committed at all sites. Then, when Alice sees the mark, she can understand that her in-flight message may not yet be visible to all her friends.

Having discussed the anomalies of PSI, we now discuss ways that an application can use and benefit from PSI.

Multi-object atomic updates. With PSI, updates of a transaction occur together, so an application can use a transaction to modify many objects without exposing partial updates on each object.

Snapshots. With PSI, a transaction reads from a fixed consistent snapshot, so an application can use a transaction to ensure that it is reading consistent versions of different objects.

Read-modify-write operations. Because PSI disallows write-write conflicts, a transaction can implement any atomic read-modify-write operation, which reads an object and writes a new value based on the value read. Such operations include atomic increment and decrement of counters, atomic appends, and atomic edits.

Conditional writes. A particularly useful type of read-modify-write operation is a conditional write, which writes an object only if its content or version matches a value provided by the application. With PSI, this is performed by reading the object, evaluating the condition and, if it is satisfied, writing the object. This scheme can be extended to check and write many objects at once.

3.2.4 Using cset operations

A cset is a mapping from ids to counts, possibly negative. The mapping indicates how many times the element with a given id appears in the cset. There are two ways to use csets. First, when the count is useful to the application, a cset can be used as is. For example, a cset can keep the number of items in a shopping cart or inventory, the number of accesses to a data item, or the number of references to an object.
The second way to use a cset is as a conventional set, by hiding the counts from the user. For example, a cset can keep a list of friends, messages, active users, or photo albums. In these cases, the count has no meaning to the user. The application should be designed to keep the counts of elements at zero or one: the application should not add an element to a cset when the element is already present, or remove an element from a cset when the element is not there. In some cases, however, concurrent updates may cause the count to raise above one or drop below zero. For example, a user may add the same friend to her friends list, and do so concurrently at two different sites: the application sees a count of zero in both sites, and so it adds the friend once at each site. This situation is rare, because there must be updates to the *same* element in the *same* cset, and those updates must be concurrent, but it may happen. This is addressed by treating a count of one or more as present in the set, and count of zero or less as absent from the set. For example, when showing the list to the user, friends with negative counts are excluded. When the user adds a friend, if the count is negative, the application adds the friend enough times for the count to be one. When removing a friend, the application removes her enough times for the count to be zero. This is done by the application, transparently to the user.

Chapter 4

System design and implementation of Walter

This chapter provides an in-depth description of the design and implementation of Walter. Section 4.1 describes how clients view and use Walter. Section 4.2 explains the operation of Walter by reviewing its design and protocols. Section 4.3 discusses Walter's implementation.

4.1 Service

Each site contains a Walter server and one or more application clients. Walter stores keyvalue object pairs grouped in containers (Section 4.1.1), where each container is replicated across multiple sites. The Walter client interface is exposed as a user-level library with functions to start transactions, read and write data, and commit transactions (Section 4.1.2). Walter provides fault tolerance by replicating data across sites (Section 4.1.3), and it allows users to trade-off durability for availability (Section 4.1.4).

4.1.1 Objects and containers

Walter stores objects, where an object has a key and a value. There are two types of objects: regular and cset. In a regular object, the value is an uninterpreted byte sequence, while in cset object, the value is a cset.

Each object is stored in a *container*, a logical organization unit that groups objects with some common purpose. For example, in a Web application, each user could have a container that holds all of her objects. To reduce space overhead, all objects in a container have the same preferred site, and Walter stores this information only once, as an attribute of the container. Administrators choose the preferred site to be the site most likely to modify the objects. For example, each user may have a designated site where she logs into the system (if she tries to log into a different site, she is redirected), and this would be the preferred site of her objects.

Object ids consist of a container id and a local id. The container id indicates to which container the object belongs, and the local id differentiates objects within a container. Since the container id is part of the object id, the container of an object cannot be changed.

4.1.2 Interface

Walter provides a client library for starting a transaction, manipulating objects, and committing a transaction, with the PSI semantics and operations explained in Sections 3.2.1 and 3.2.2. For regular objects, the available operations are read and write; for cset objects, the available operations are read, add element, and delete element.

Walter replicates transactions asynchronously, and the interface allows a client to receive a callback when (a) the transaction is disaster-safe durable (Section 4.1.4), and (b) the transaction is globally visible, meaning it has been committed at all sites.

4.1.3 Replication

Walter provides both durability and availability by replicating data within a single site and across multiple sites. Replication is transparent to clients: all the replicas of an object have the same object id, and the system accesses the replica closest to the client. An object need not be replicated at all sites and clients can read objects even if they are not replicated at the local site, in which case Walter fetches the data from a remote site.¹ A transaction commits at every site, even where it is not replicated, following the semantics of PSI in Section 3.2.1: once a transaction is committed at a site, reads from that site see the effects of the transaction. Administrators choose how many replicas and where they are. These settings are stored as attributes of a container, so all objects of a container are replicated similarly.

4.1.4 Durability and availability

Walter provides two levels of durability:

(*Normal Durability*) When a transaction commits at its site, writes have been logged to a replicated cluster storage system [47, 59, 75, 87], so writes are not lost due to power failures. Data may be lost if an entire data center is wiped out by a disaster.

(*Disaster-safe Durability*) A transaction is considered *disaster-safe durable* if its writes have been logged at f+1 sites, where parameter f determines the desired fault tolerance level: up to f sites may fail without causing data loss. The default value of f is 1.

If an entire site *s* fails temporarily or is unreachable due to cross-site network issues, it may have transactions that were locally committed but not yet propagated to other sites. In that case, the application has two choices:

(Conservative) Wait for the site s to come back online, so that it can propagate the missing trans-

¹In the PSI specification, data is replicated at every site, but an implementation need not do that, as long as it behaves identically in terms of responses to operations.

actions. But then clients cannot write to objects whose preferred site is *s* until *s* comes back online—a loss of availability for some writes.

(*Aggressive*) Sacrifice the durability of a few committed transactions at site *s* for better availability, by replacing site *s* and abandoning its non-propagated transactions. Technically, this choice violates PSI, but one could extend the PSI definition to allow for lost committed transactions when a site fails or disconnects. Applications can wait for important transactions to be marked disastersafe durable before confirming them to users.

Availability within a site comes from the availability of the cluster storage system: if the Walter server at a site fails, the system starts a new server, which can access the same cluster storage system. Availability under network partitions or disasters comes from cross-site replication. If a site fails, an application can warn users before they are redirected to another site, because users may see a different system state at the new site due to the semantics of PSI. In practice, the state at different sites diverges by only a few seconds.

4.2 Design and algorithms

This section describes Walter's design, emphasizing the protocols for executing and committing transactions. We first give an overview of the basic architecture (Section 4.2.1) and object versioning (Section 4.2.2). We then explain how to execute transactions (Section 4.2.3) and how to commit certain common transactions quickly (Section 4.2.4). Next, we explain how to commit other transactions (Section 4.2.5) and how transactions are replicated asynchronously (Section 4.2.6). Lastly, we consider failure recovery (Section 4.2.7) and scalability (Section 4.2.8).

4.2.1 Basic architecture

There are multiple sites numbered 1, 2, ... Each site contains a local Walter server and a set of clients. A client communicates with the server via remote procedure calls implemented by the API library. The server executes the actual operations to start and commit transactions, and to access objects.

Walter employs a separate *configuration service* to keep track of the currently active sites, and the preferred site and replica set for each object container. The configuration service tolerates failures by running as a Paxos-based state machine replicated across multiple sites. A Walter server confirms its role in the system by obtaining a lease from the configuration service, similar to what is done in [32, 85]. The lease assigns a set of containers to a preferred site, and it is held by the Walter server at that site. A Walter server caches the mapping from a container to its replica sites to avoid contacting the configuration service at each access. Incorrect cache entries do not affect correctness because a server rejects requests for which it does not hold the corresponding preferred site lease.

At Server_i: // *i* denotes the site number CurrSeqNo_i: integer with last assigned local sequence number CommittedVTS_i: vector indicating for each site how many transactions of that site have been committed at site *i* History_i[oid]: a sequence of updates of the form $\langle data, version \rangle$ to oid, where version = $\langle j:n \rangle$ for some *j*, *n* GotVTS_i: vector indicating for each site how many transactions of that site have been received by site *i*

Figure 4.1: Variables at server on each site.

4.2.2 Versions and vector timestamps

The PSI specification is centralized and uses a monotonic timestamp when a transaction starts and commits. But monotonic timestamps are expensive to produce across multiple sites. Thus, to implement PSI, Walter replaces them with version numbers and vector timestamps. A version number (or simply *version*) is a pair (*site*, *seqno*) assigned to a transaction when it commits; it has the site where the transaction executed, and a sequence number local to that site. The sequence number orders all transactions within a site. A vector timestamp represents a snapshot; it contains a sequence number for each site, indicating how many transactions of that site are reflected in the snapshot. A transaction is assigned a vector timestamp *startVTS* when it starts. For example, if *startVTS* = $\langle 2, 4, 5 \rangle$ then the transaction reads from the snapshot containing 2 transactions from site 1, 4 from site 2, and 5 from site 3.

Given a version $v = \langle site, seqno \rangle$ and a vector timestamp *startVTS*, we say that v is *visible* to *startVTS* if *seqno* \leq *startVTS*[*site*]. Intuitively, the snapshot of *startVTS* has enough transactions from *site* to incorporate version v.

Figure 4.1 shows the variables at the server at site *i*. Variable $CurrSeqNo_i$ has the last sequence number assigned by the server, and $CommittedVTS_i[j]$ has the sequence number of the last transaction from each site *j* that was committed at site *i*. We discuss $History_i$ and $GotVTS_i$ in Sections 4.2.3 and 4.2.6.

```
At Server;:
                // i denotes the site number
operation startTx(x)
  x.tid \leftarrow unique transaction id
  x.startVTS \leftarrow CommittedVTS_i
  return ok
operation write(x, oid, data): add (oid, DATA(data)) to x.updates; return OK
operation setAdd(x, setid, id): add (setid, ADD(id)) to x.updates; return OK
operation setDel(x, setid, id): add (setid, DEL(id)) to x.updates; return OK
operation read(x, oid)
  if oid is locally replicated
  then return state of oid reflecting x.updates and
       all versions in History, [oid] visible to x.startVTS
  else return state of oid reflecting x.updates,
       the versions in History_{site(oid)} [oid] visible to x.startVTS, and
      the versions in History<sub>i</sub>[oid] visible to x.startVTS
```

operation *setRead*(*x*, *setid*): same as *read*(*x*, *oid*)

Figure 4.2: Executing transactions.

4.2.3 Executing transactions

To execute transactions, the server at each site *i* maintains a history denoted $History_i[oid]$ with a sequence of writes/updates for each object *oid*, where each update is tagged with the version of the responsible transaction. This history variable is similar to variable *Log* in the PSI specification, except that it keeps a list per object, and it has versions not timestamps. When a transaction *x* starts, Walter obtains a new start vector timestamp *startVTS* containing the sequence number of the latest transactions from each site that were committed at the local site. To write an object, add to a cset, or remove from a cset, Walter stores this update in a temporary buffer *x.updates*. To read an object, Walter retrieves its state from the snapshot determined by *startVTS* and any updates in *x.updates*. Specifically, for a regular object, Walter returns the last update in *x.updates* or, if none, the last update in the history visible to *startVTS*. For a cset object, Walter computes its state by applying the updates in the history visible to *startVTS* and the updates in *x.updates*.

The above explanation assumes an object is replicated locally. If not, its local history $History_i[oid]$ will not have all of the object's updates (but it may have some recent updates). Therefore, to read such an object, Walter retrieves the data from the object's preferred site and merges it with any updates in the local history and in *x.updates*. To write, Walter buffers the write in *x.updates* and, upon commit, stores the update in the local history while it is being replicated to other sites; after

```
At Server;:
                  // i denotes the site number
function unmodified(oid, VTS): true if oid unmodified since VTS
function update(updates, version)
  for each \langle oid, X \rangle \in updates do add \langle X, version \rangle to History_i[oid]
operation commitTx(x)
   x. writeset \leftarrow {oid : \langle oid, DATA(*) \rangle \in x. updates }
                                                                // * is a wildcard
  if \forall oid \in x. writeset : site(oid) = i then return fastCommit(x)
  else return slowCommit(x)
function fastCommit(x)
  if \forall oid \in x.writeset : unmodified(oid, startVTS) and oid not locked then
    x.seqno \leftarrow ++CurrSeqNo_i
                                         // vertical bar indicates atomic region
     update(x.updates, \langle i, x.seqno \rangle)
    wait until CommittedVTS<sub>i</sub>[i] = x.seqno-1
     \textit{CommittedVTS}_{i}[i] \leftarrow x.\textit{seqno}
     x outcome \leftarrow COMMITTED
    fork propagate(x)
  else x.outcome ← ABORTED
  return x.outcome
```

```
Figure 4.3: Fast commit.
```

that, the local history can be garbage collected. Figure 4.2 shows the detailed pseudocode executed by a server. Recall that clients invoke the operations at the local server using a remote procedure call (not shown). The code is multi-threaded and we assume that each line is executed atomically.

4.2.4 Fast commit

For transactions whose write-set has only objects with a local preferred site, Walter uses a fast commit protocol. The write-set of a transaction consists of all oids to which the transaction writes; it excludes updates to set objects. To fast commit a transaction x, Walter first determines if x can really commit. This involves two checks for conflicts. The first check is whether all objects in the write-set are unmodified since the transaction started. To perform this check, Walter uses the start vector timestamp: specifically, we say that an object *oid* is *unmodified since* x.startVTS if all versions of *oid* in the history of the local site are visible to x.startVTS. The second check is whether all objects in the write-set of x are unlocked; intuitively, a locked object is one being committed by the slow commit protocol (Section 4.2.5). If either check fails, then x is aborted. Otherwise, Walter proceeds to commit x, as follows. It assigns a new local sequence number to x, and then applies x's updates to the histories of the modified objects. Walter then waits until the local transaction with preceding sequence number has been committed. This typically happens

```
At Server;:
                 // i denotes the site number
function slowCommit(x)
  // run 2pc among preferred sites of updated objects
  sites \leftarrow {site(oid) : oid \in x.writeset}
  pfor each s \in sites do
                                   // pfor is a parallel for
     vote[s] \leftarrow remote call prepare(x.tid,
                 \{oid \in x.writeset : site(oid) = s\}, x.startVTS
  if \forall s \in sites : vote[s] = YES then
    x.seqno \leftarrow ++CurrSeqNo_i
                                         // vertical bar indicates atomic region
     update(x.updates, \langle i, x.seqno \rangle)
     wait until CommittedVTS<sub>i</sub>[i] = x.seqno – 1
     CommittedVTS<sub>i</sub>[i] \leftarrow x.seqno
     release locks (at this server) with owner x.tid
     x.outcome \leftarrow COMMITTED
    fork propagate(x)
  else
     pfor each s \in \textit{sites} such that \textit{vote}[s] = \text{YES} do remote call \textit{abort}(x.\textit{tid})
     x.outcome \leftarrow ABORTED
  return x.outcome
function prepare(tid, localWriteset, startVTS)
  if \forall oid \in localWriteset : oid not locked and unmodified(oid, startVTS) then
    for each oid ∈ localWriteset do lock oid with owner tid
     return YES
  else return NO
function abort(tid)
  release locks (at this server) with owner tid
```

Figure 4.4: Slow commit.

quickly, since sequence numbers are assigned in commit order. Finally, transaction x is marked as committed and Walter propagates x to remote sites asynchronously as described in Section 4.2.6. Figure 4.3 shows the detailed pseudocode. The notation site(oid) denotes the preferred site of oid. As before, we assume that each line is executed atomically. A vertical bar indicates a block of code with multiple lines that is executed atomically.

4.2.5 Slow commit

Transactions that write a regular object whose preferred site is not local must be committed using the slow commit protocol, which employs a type of two-phase commit among the preferred sites of the written objects (not across all replicas of the objects). The purpose of two-phase commit is to avoid conflicts with instances of fast commit and other instances of slow commit. To commit a transaction x, the server at the site of the transaction acts as the coordinator in the twophase protocol. In the first phase, the coordinator asks the (servers at the) preferred sites of each written object to vote based on whether those objects are unmodified and unlocked. If an object

```
At Server;:
                  // i denotes the site number
function propagate(x)
 send (PROPAGATE, x) to all servers
 wait until \forall oid \in x. writeset: received \langle PROPAGATE-ACK, x. tid \rangle
      from f+1 sites replicating oid including site(oid)
 mark x as disaster-safe durable
 send (DS-DURABLE, x\rangle to all servers
 wait until received \langle VISIBLE, x.tid \rangle from all sites
 mark x as globally visible
when received \langle \mathsf{PROPAGATE}, x \rangle from \mathit{Server}_j and
   GotVTS_i \ge x.startVTS and GotVTS_i[j] = x.seqno-1 do
 if i \neq j then update(items in x.updates replicated in this site, \langle j : x.seqno \rangle)
// when i = j, update has been applied already when transaction committed
 GotVTS_i[j] = x.seqno
 send (PROPAGATE-ACK, x.tid) to Server,
when received (DS-DURABLE, x) and (PROPAGATE, x) from Server<sub>j</sub> and
   Committed VTS<sub>i</sub> \geq x. start VTS and Committed VTS<sub>i</sub>[j] = x. seqno-1 do
 CommittedVTS<sub>i</sub>[j] \leftarrow x.seqno
 release all locks with owner x.tid
 send (VISIBLE, x.tid) to Server<sub>i</sub>
```

Figure 4.5: Transaction replication.

is modified at the preferred site, then an instance of fast commit conflicts with x; if the object is locked at the preferred site, then another instance of slow commit conflicts with x. If either case occurs, the site votes "no", otherwise the site locks the objects and votes "yes". If any vote is "no", the coordinator tells the sites to release the previously acquired locks. Otherwise, the coordinator proceeds to commit x as in the fast commit protocol: it assigns a sequence number to x, applies x's updates to the object histories, marks x as committed, and propagates x asynchronously. When x commits, a site releases the acquired locks when x is propagated to it. Figure 4.4 shows the detailed pseudocode.

4.2.6 Asynchronous propagation

After a transaction commits, it is propagated asynchronously to other sites. The propagation protocol is simple: the site of a transaction x first copies the objects modified by x to the sites where they are replicated. The site then waits until *sufficiently many sites* indicate that they received (a) transaction x, (b) all transactions that causally precede x according to x.startVTS, and (c) all transactions of x's site with a smaller sequence number. "Sufficiently many sites" means at least f+1 sites replicating each object including the object's preferred site, where f is the disaster-safe

tolerance parameter (Section 4.1.4). At this point, x is marked as disaster-safe durable and all sites are notified. Transaction x commits at a remote site j when (a) site j learns that x is disaster-safe durable, (b) all transactions that causally precede x are committed at site j, and (c) all transactions of x's site with a smaller sequence number are committed at site j. When x has committed at all sites, it is marked as globally visible. The pseudocode is shown in Figure 4.5. Vector $GotVTS_i$ keeps track of how many transactions site i has received from each other site. Note that when a site i receives a remote transaction and updates the history of its objects, the transaction is not yet committed at i: it commits only when $CommittedVTS_i[j]$ is incremented. The code omits simple but important optimizations: when server i propagates transaction x to a remote server, it should not send all the updates of x, just those updates replicated at the remote server. Similarly, when it sends a DS-DURABLE message, a server need not include the updates of x again.

4.2.7 Handling failures

Recovering from client or server failure. If a client crashes, its outstanding transactions are aborted and any state kept for those transactions at the server is garbage collected. Each server at a site stores its transaction log in a replicated cluster storage system. When a Walter server fails, the replacement server resumes propagation for those committed transactions that have not yet been fully propagated.

Handling a site failure. An entire site s may fail due to a disaster or a power outage. Such failure is problematic because there may be committed transactions at s that were not yet replicated at other sites. As explained in Section 4.1.4, Walter offers two site recovery options: conservative and aggressive. Recall that the conservative option is to wait for s to come back online, while the aggressive option is to remove s and reassign the preferred site of its containers to another site. To remove a failed site, Walter uses the configuration service (Section 4.2.1). Each configuration indicates what sites are active. Before switching to a new configuration that excludes site s, the

configuration service must find out the transactions committed by s that will survive across the configuration change. Transaction x of site s survives if x and all transactions that causally precede x and all transactions of s with a smaller sequence number have been copied to a site in the new configuration. The configuration service queries the sites in the new configuration to discover what transactions survive. Then, it asks each site to discard the replicated data of non-surviving transactions and, in the background, it completes the propagation of surviving transactions that are not yet fully replicated. Finally, the configuration service reassigns the preferred site of containers of s to another site, by having another site take over the appropriate leases. While reconfiguration is in progress, sites that are still active continue to commit transactions, except transactions that write to objects whose preferred site was s, which are postponed until those objects get a new preferred site.

Re-integrating a previously failed site. When a previously removed site s recovers, it must be reintegrated into the system. The configuration service starts a new reconfiguration that includes s. To switch to the new configuration, s must first synchronize with its replacement site s' to integrate modifications committed by s'. Once synchronization is finished, s takes over the lease for being the preferred site for the relevant containers, and the new configuration takes effect.

4.2.8 Scalability

Walter relies on a single server per site to execute and commit transactions, which can become a scalability bottleneck. A simple way to scale the system is to divide a data center into several "local sites", each with its own server, and then partition the objects across the local sites in the data center. This is possible because Walter supports partial replication *and* allows transactions to operate on an object not replicated at the site—in which case, the transaction accesses the object at another site within the same data center. We should note that PSI allows sites to diverge; to avoid exposing this divergence to users, applications can be designed so that a user always log into

Method	Description		
void start()		start transaction	
int commit()		try to commit	
int abort()		abort	
int read(Oid o, char **buf)		read object	
int write(Oid o, char *buf, int len)	write object		
Oid newid(ContainerId cid, OType	get new oid		
int setAdd(Oid cset, Id id)		add <i>id</i> to <i>cset</i>	
int setDel(Oid cset, ld id)		delete <i>id</i> from <i>cset</i>	
int setRead(Oid cset, IdSetIterator **iter)		read cset	
int setReadId(Oid cset, Id id, int *answer)		read <i>id</i> in <i>cset</i>	
C++ Example: Tx x; x.start(); len = x.read(o1, &buf); err = x.write(o2, buf, len); res = x.commit();	PHP Exar \$x = wa\$ \$buf = w \$err = w \$res = w	mple: StartTx(); /aRead(\$x, \$o1); aWrite(\$x, \$o2, \$buf); /aCommit(\$x);	

Figure 4.6: Basic C++ API for Walter and C++ and PHP examples.

the same local site in the data center. Another approach to scalability, which we do not explore in this dissertation, is to employ several servers per site and replace the fast commit protocol of Section 4.2.4 with distributed commit.

4.3 Implementation

The Walter implementation has a client-side library and a server, written in C++, with a total of 30K lines of code. There is also a PHP interface for web development with 600 lines of code. The implementation differs from the design as follows. First, each Walter server uses direct-attached storage devices, instead of a cluster storage system. Second, we have not implemented the scheme to reintegrate a failed site (Section 4.2.7): currently, the administrator must invoke a script manually to do that. Third, the client interface, shown in Figure 4.6, differs cosmetically from the specification in Section 3.2.1, due to the specifics of C++ and PHP. In C++, there is a Transaction class and operations are methods of this class. Functions read, setRead, and setReadId return the data via a parameter (the C++ return value is a success indication). setRead provides an iterator for the ids in a cset. setReadId indicates the count of an identifier in a cset. commit can optionally inform the client via

supplied callbacks—not shown—when the transaction is disaster-safe durable and globally visible (i.e., committed at all sites). There is a function newid to return a fresh oid, explained below.

There are no specialized functions to create or destroy objects. Conceptually, all objects always exist and are initialized to *nil*, without any space allocated to them. If a client reads a never-written object, it obtains *nil*. Function *newid* returns a unique oid of a never-written object of a chosen type (regular or cset) in a chosen container. Destroying a regular object corresponds to writing *nil* to it, while destroying a cset object corresponds to updating its elements so that they have zero count. There are some additional functions (not shown), including (a) management functions for initialization, shutdown, creating containers, and destroying containers; and (b) functions that combine multiple operations in a single RPC to the server, to gain performance; these include functions for reading or writing many objects, and for reading all objects whose ids are in a cset. The functions to create and destroy containers run outside a transaction; we expect them to be used relatively rarely. Identifiers for containers and objects are currently restricted to a fixed length, but it would be easy to make them variable-length.

The server stores object histories in a persistent log and maintains an in-memory cache of recently-used objects. The persistent log is periodically garbage collected to remove old entries. The entries in the in-memory cache are evicted on an LRU basis. Since it is expensive to reconstruct csets from the log, the eviction policy prefers to evict regular objects rather than csets. There is an in-memory index that keeps, for each object, a list of updates to the object, ordered from most to least recent, where each update includes a pointer to the data in the persistent log and a flag of whether the data is in the cache. To speed up system startup and recovery, Walter periodically checkpoints the index to persistent storage; the checkpoint also describes transactions that are being replicated. Checkpointing is done in the background, so it does not block transaction processing. When the server starts, it reconstructs the index from the checkpointed state and the data in the log after the checkpoint.

To improve disk efficiency, Walter employs group commit to flush many commit records to

disk at the same time. To reduce the number of threads, the implementation makes extensive use of asynchronous calls and callbacks when it invokes blocking and slow operations. To enhance network efficiency, Walter propagates transactions in periodic batches, where each batch remotely copies all transactions that committed since the last batch.

The protocol for slow commit may starve because of repeated conflicting instances of fast commit. A simple solution to this problem is to mark objects that caused the abort of slow commit and briefly delay access to them in subsequent fast commits: this delay would allow the next attempt of slow commit to succeed. We have not implemented this mechanism since none of our applications use slow commit.

Chapter 5

Evaluation of Walter

In this chapter we present an evaluation of Walter. We first demonstrate the usability of Walter by describing our experience with writing applications on top of it. We then proceed to experimentally evaluate the performance of Walter and the applications it runs.

5.1 Usability: Applications on top of Walter

Using Walter, we built a social networking web site (WaltSocial) and ported a third-party Twitter-like application called *ReTwis* [13]. Our experience suggests that it is easy to develop applications using Walter and run them across multiple data centers.

WaltSocial. WaltSocial is a complete implementation of a simple social networking service, supporting the common operations found in a system such as Facebook. These include *befriend*, *status-update*, *post-message*, *read-info* as well as others. In WaltSocial, each user has a profile object for storing personal information (e.g., name, email, hobbies) and several cset objects: a

Tx x; x.start(); x.read(oidA, &profileA); x.read(oidB, &profileB); (* continues in next column *)

x.setAdd(profileA.friendlist, oidB); x.setAdd(profileB.friendlist, oidA); success = x.commit();

Figure 5.1: Transaction for befriend operation in WaltSocial.

friend-list has oids of the profile objects of friends, a *message-list* has oids of received messages, an *event-list* has oids of events in the user's activity history, and an *album-list* has oids of photo albums, where each photo album is itself a cset with the oids of photo objects.

WaltSocial uses transactions to access objects and maintain data integrity. For example, when users A and B *befriend* each other, a transaction adds A's profile oid to B's friend-list and vice versa (Figure 5.1). To *post-message* from A to B, a transaction writes an object m with the message contents and adds its oid to B's message-list and to A's event-list.

Each user has a container that stores her objects. The container is replicated at all sites to optimize for reads. The system directs a user to log into the preferred site of her container. User actions are confirmed when transactions commit locally.

ReTwis. ReTwis is a Twitter-clone written in PHP using the Redis key-value store [10]. Apart from simple get/put operations, this application makes extensive use of Redis's native support for certain atomic operations, such as adding to or removing from a list, and adding or subtracting from an integer. In Redis, cross-site replication is based on a master-slave scheme. For our port of ReTwis, we replace Redis with Walter, so that ReTwis can update data on multiple sites. We use Walter transactions and csets to provide the equivalent atomic integer and list operation in Redis.

For each user, ReTwis has a timeline that tracks messages posted by the users that the user is following. In the original implementation, a user's timeline is stored in a Redis list. When a user posts a message, ReTwis performs an atomic increment on a sequence number to generate a postID, stores the message under the postID, and appends the postID to each of her followers' timelines. When a user checks postings, ReTwis displays the 10 most recent messages from her timeline. To port ReTwis to use Walter, we make several changes: we use a cset object to represent each user's timeline so that different sites can add posts to a user's timeline without conflicts. To post a message, we use a transaction that writes a message under a unique postID, and adds the postID to the timeline of every follower of the user. We found the process of porting ReTwis to Walter to be quite simple and straightforward: a good programmer without previous Walter experience wrote the port in less than a day. Transactions allow the data structure manipulations built into Redis to be implemented by the application, while providing competitive performance (Section 5.2.7).

5.2 Performance evaluation

We evaluate the performance of Walter and its applications (WaltSocial, ReTwis) using Amazon's EC2. The highlights of our results are the following:

- Transactions that modify objects at their preferred sites commit quickly, with a 99.9-percentile latency of 27ms on EC2. Committed transactions are asynchronously replicated to remote sites within twice the network round-trip latency.
- Transactions that modify csets outside of their preferred sites also commit quickly without cross-site coordination. WaltSocial uses csets extensively and processes user requests with a 99.9-percentile latency under 50ms.
- The overhead for supporting transactions in Walter is reasonable. ReTwis running on Walter has a throughput 25% smaller than running on Redis in a single site, but Walter allows ReTwis to scale to multiple sites.

5.2.1 Experimental setup

Unless stated otherwise, experiments run on Amazon's EC2 cloud platform. We use machines in four EC2 sites: Virginia (VA), California (CA), Ireland (IE), and Singapore (SG), with the following average round-trip latencies within and across sites (in ms):

	VA	CA	IE	SG
VA	0.5	82	87	261
CA		0.3	153	190
IE			0.5	277
SG				0.3

Within a site, the bandwidth between two hosts is over 600 Mbps; across sites, we found a bandwidth limit of 22 Mbps.

We use extra-large EC2 virtual machine instances, with 7 GB of RAM and 8 virtual cores, each equivalent to a 2.5 GHz Intel Xeon processor. Walter uses write-ahead logging, where commit logs are flushed to disk at commit time. Since one cannot disable write-caching at the disk on EC2, where indicated we run experiments on a private cluster outside of EC2, with machines with two quad core Intel Xeon E5520 2.27 GHz processors and 8 GB of RAM.

Each EC2 site has a Walter server, and we run experiments with different numbers of sites and replication levels, as shown below:

Experiment name	Sites	Replication level
1-site	VA	none
2-sites	VA, CA	2
3-sites	VA, CA, IE	3
4-sites	VA, CA, IE, SG	4

Our microbenchmark workload (Sections 5.2.2–5.2.5) consists of transactions that read or write a few randomly chosen 100-byte objects. (Changing the object size from 100 bytes to 1 KB yields similar results.) We choose to evaluate small transactions because our applications, WaltSocial and ReTwis, only access a few small objects in each transaction. We consider a transaction to be disaster-safe durable when it is committed at all sites in the experiment.

5.2.2 Base performance

We first evaluate the base performance of Walter, and compare it against Berkeley DB 11gR2 (BDB), a commercial open-source developer database library. The goal is to understand if Walter provides a usable base performance.

Benchmark setup. We configure BDB to use B-trees with default pagesize and snapshot isolation; parameters are chosen for the best performance. We configure BDB to have two replicas with asynchronous replication. Since BDB allows updates at only one replica (the primary), we set up the Walter experiment to also update at one site. To achieve good throughput in BDB, we must use many threads at the primary to achieve high concurrency. However, with many threads, EC2 machines perform noticeably worse than private machines. Therefore, we run the primary BDB replica in our private cluster (with write-caching at the disk enabled), and the other replica at the CA site of EC2. We do the same for Walter. Clients and the server run on separate hosts. For BDB, we use an RPC server to receive and execute client requests.

The workload consists of either read or write transactions each accessing one 100-byte object. We populate BDB and Walter with 50,000 keys, which fits in the 1 GB cache of both systems. Walter includes an optimization to reduce the number of RPCs, where the start and commit of each transaction are piggybacked onto the first and last access, respectively. Thus, transactions with one access require just one RPC in Walter and in BDB.

Results. Figure 5.2 shows that throughput of read and write transactions of Walter is comparable to that of BDB. Read throughput is CPU-bound and mainly limited by the performance of our RPC library in both systems. Walter's read throughput is slightly lower because it does more work than

Name	Read Tx throughput	Write Tx throughput
Walter	72 Ktps	33.5 Ktps
Berkeley DB	80 Ktps	32 Ktps

Figure 5.2: Base read and write transaction throughput.

BDB by acquiring a local lock and assigning a start timestamp vector when a transaction starts. The commit and replication latency of BDB and Walter are also similar and not shown here (see Section 5.2.3 for Walter's latency).

5.2.3 Fast commit on regular objects

This microbenchmark evaluates the performance of transactions on regular objects, using fast commit.

Benchmark setup. The experiments involve one to four sites. Objects are replicated at all sites, and their preferred sites are assigned evenly across sites. At each site, we run multiple clients on different hosts to issue transactions as fast as possible to its local Walter server. There are several workloads: *read-only*, *write-only*, and *mixed*. Read-only or write-only transactions access one or five 100-byte objects. The mixed workload consists of 90% read-only transactions and 10% write-only transactions.

Result: throughput. Figure 5.3 shows Walter's aggregate throughput across sites as the number of sites varies. Read throughput is bounded by the RPC performance and scales linearly with the number of sites, reaching 157 Ktps (thousands of transactions per second) with 4 sites. Write throughput is lower than read throughput due to lock contention within a Walter server. Specifically, when a transaction commits, a thread needs to acquire a highly contended lock to check for transaction conflicts. Moreover, write throughput does not scale as well as read throughput as the number of sites increases. This is because data is replicated at all sites, so the amount of work per write transaction grows with the number of sites. Yet, the cost of replication is lower than that of committing because replication is done in batches. Thus, the write throughput still grows with the number of sites, but not linearly. Note that the read and write throughput for transactions of size 1 in Figure 5.3 is only 50–60% of that in Figure 5.2 as a result of running this experiment on EC2 instead of the private cluster. In the mixed workload, performance is mostly determined





by how many operations a transaction issues on average. For example, when there are 90% readonly transactions each reading one object and 10% write-only transactions each writing 5 objects, a transaction issues on average only 1.4 requests to the server. As a result, a relatively high aggregate throughput of 80 Ktps is reached across 4 sites.

Result: latency. We measure the fast commit latency for write-only transactions accessing 5 objects. We record the time elapsed between issuing a commit and having the server acknowledge the commit completion. Figure 5.4 shows the latency distribution measured on EC2, and in our private cluster with and without write caching at the disk. The measurements were taken for a moderate workload in which clients issued enough requests to achieve 70% of maximal throughput. The points at the lower-end of the distributions in Figure 5.4 show latencies that we observe in a lightly loaded system.

Because there is no cross-site coordination, fast commit is quick: On EC2 the 99-percentile latency is 20 ms and the 99.9-percentile is 27 ms. Since the network latency within a site is low at 0.5 ms, the commit latency is dominated by the effects of queuing inside the Walter server and of flushing the commit log to disk when committing transactions at a high throughput. Figure 5.4 also shows the effect of disabling write-caching at the disk, measured on our private cluster. Even in that case, the 99.9-percentile latency of a fast commit is under 90 ms.

The latency for a committed transaction to become disaster-safe durable is dominated by the network latency across sites. As shown in Figure 5.5, the latency is distributed approximately uniformly between $[RTT_{max}, 2 * RTT_{max}]$ where RTT_{max} is the maximum round-trip latency between VA and the other three sites. This is because Walter propagates transactions in batches to maximize throughput, so a transaction must wait for the previous batch to finish.

The latency for a committed transaction to become globally visible is an additional RTT_{max} after it has become disaster-safe durable (not shown).

5.2.4 Fast commit on cset objects

We now evaluate transactions that modify csets.

Benchmark setup. We run the 4-site experiment in which each transaction modifies two 100-byte objects at the preferred site and adds an id to a cset with a remote preferred site.

Results. The latency distribution curve for committing transactions (not shown) is similar to the curve corresponding to EC2 in Figure 5.4. This is because transactions modifying csets commit via the same fast commit protocol as transactions modifying regular objects at their preferred site. Across 4 sites, the aggregate throughput is 26 Ktps, which is lower than the single-write transaction throughput of 52 Ktps shown in Figure 5.3. This is because the cset transactions issue 4 RPCs (instead of 1 RPC for the transactions in Figure 5.3), to write two objects, modify a cset, and commit.

5.2.5 Slow commit

We now evaluate the slow commit protocol for transactions modifying objects with different preferred sites. Unlike fast commit, slow commit requires cross-site coordination.

Benchmark setup. We run the 4-site experiments and have clients issue write-only transactions at the VA site. We vary the size of a transaction from 2 to 4 objects. Each object written has a different preferred site: the first, second, third, and fourth object's preferred sites are VA, CA, IE, and SG respectively.

Results. Figure 5.6 shows the commit latency (left-most three lines) and the latency for achieving disaster-safe durability (right-most three lines). The commit latency is determined by the round-trip time between VA and the farthest preferred site of objects in the writeset. This is because slow commit runs a two-phase protocol among the preferred sites of the objects in the writeset. For example, for transactions of size 3, the commit latency is 87 ms, which is the round-trip time from VA to IE. The latency for disaster-safe durability is the commit latency plus the replication latency. The replication latency is the same as for fast commit: it is uniformly distributed between [RTT_{max} , 2 * RTT_{max}], where RTT_{max} is the round-trip time between VA and SG.

To optimize performance, applications should minimize the use of slow commits. Both Walt-Social and ReTwis avoid slow commits by using csets.



5.2.6 WaltSocial performance

Transactions make it easy to develop WaltSocial. Our experiments also show that WaltSocial achieves good performance.

Workload setup. The WaltSocial experiments involve 4 sites in EC2. We populate Walter with 400,000 users, each with 10 status updates and 10 wall postings from other users. We run many application clients at each site, where each client issues WaltSocial operations. An operation corresponds to a user action, and it is implemented by executing and committing a transaction that reads and/or writes several data objects (Section 5.1). We measure the latency and aggregate throughput for each operation. We also evaluate two mixed workloads: mix1 consists of 90% *read-info* operations and 10% update operations including *status-update, post-message* and *befriend*; mix2 contains 80% *read-info* operations and 20% update operations.

Operation throughput. Figure 5.7 shows the throughput in thousands operations per second (Kops/s) for each WaltSocial operation and for the mixed workloads. The *read-info* operation issues read-only transactions; it has the highest aggregate throughput at 40 Kops/s. The other operations issue transactions that update objects; their throughput varies from 16.5 Kops/s to 20 Kops/s, depending on the number of objects read and written in the transactions. The mixed workloads are dominated by *read-info* operations, hence their throughput values are closer to that of *read-info*. The achieved throughput is likely sufficient for small or medium social networks. To handle larger deployments, one might deploy several sites per data center to scale the system (Section 4.2.8).

Operation latency. Figure 5.8 shows the latency of WaltSocial operations when the system has a moderate load. Operations finish quickly because the underlying transactions involve no cross-site communication: transactions always read a local replica for any object and transactions that update data use cset objects. The 99.9-percentile latency of all operations in Figure 5.8 is below 50 ms. As each WaltSocial operation issues read/write requests to Walter in series, the latency is affected by the number of objects accessed by different WaltSocial operations. The *read-info* operation

Operation	# objs+csets read	# objs written	# of csets written	Throughput (1000 ops/s)
read-info	3	0	0	40
befriend	2	0	2	20
status-update	1	2	2	18
post-message	2	2	2	16.5
mix1	2.9	0.5	0.3	34
mix2	2.8	0.7	0.5	32

Figure 5.7: Transaction size and throughput for Waltsocial operations.

involves fewest objects and hence is faster than other operations.

5.2.7 **ReTwis performance**

We compare the performance of ReTwis using Walter and Redis as the storage system, to assess the cost of Walter.

Workload setup. The Walter experiments involve one or two sites. Redis does not allow updates from multiple sites, so the Redis experiments involve one site. Since Redis is a semi-persistent key-value store optimized for in-memory operations, we configure both Walter and Redis to commit writes to memory. We run multiple front-end web servers (Apache 2.2.14 with PHP 5.3.2) and client emulators at each site. We emulate 500,000 users who issue requests to post a message (*post*), follow another user (*follow*), or read postings in their own timeline (*status*). The mixed workload consists of 85% *status*, 7.5% *post* and 7.5% *follow* operations.

Throughput comparison. Figure 5.9 shows the aggregate throughput (Kops/s) for different work-





Figure 5.9: Throughput of ReTwis using Redis and Walter.

loads when running ReTwis with Walter and Redis. As can be seen, with one site, ReTwis with Walter has similar performance as ReTwis with Redis: the slowdown is no more than 25%. For example, the throughput of the *post* operation for Walter (1 site) is 4713 ops/s, compared to 5740 ops/s for Redis. But ReTwis with Walter can use multiple sites to scale the throughput. For example, the throughput of *post* using ReTwis with Walter on two sites is 9527 ops/s—twice the throughput of one site.

Chapter 6

Overview of Lynx

In this chapter we proceed to describe Lynx. Although the main challenges that Lynx sets to solve are similar to those motivating Walter's design, Lynx focuses more on intra-site scalability, and on efficiently implementing a rich query interface. We begin by discussing the specific application challenges that Lynx addresses.

6.1 Application Challenges

The design of Lynx is motivated by the performance and consistency challenges facing web applications running atop a distributed database backend. To make the discussion concrete, we illustrate these challenges with an example application, a simplified online auction service modeled on the RuBIS benchmark [1].

Our example auction service stores its persistent state in three tables, shown in Figure 6.1. The table schemas are the same as those in RuBIS. The *Users* table maintains information about each user including a user ID, username etc. The *Items* table stores information about each item on sale, such as the item ID, a description of the item, the current highest bid on the item, and the corresponding high bidder. The *Bids* table stores information about each bid including the

User	S		1	ltems				
uid	name	location		item_id	descrip- tion	seller	high bidder	high price
123	alice	CA		345	Nikon N50	666	123	\$200
549	bob	NY		575	Cute puppy 123			
666	eve	NJ		Bids				
				bid_id	bidder	item	bid_	price
				1	549	345	\$100	D
			ĺ	2	123	345	\$200)

Figure 6.1: A simple auction service example consisting of three tables, *Users, Items*, and *Bids* monetary amount, the user who placed the bid, and the item for which the bid was placed. All three tables are horizontally partitioned, i.e., split into multiple pieces based on their respective primary keys and spread across multiple machines.

When building a latency-sensitive web service, the application developer must optimize for frequently occurring read workloads. Three techniques are commonly used for this:

- *Denormalized schemas* In a normalized schema all data is stored once. Denormalized schemas store redundant information to speed read access for certain queries. For example, in the schema of Figure 6.1, the two columns *high_bidder* and *high_price* have been denormalized: the highest bidder/price of an item could be calculated by selecting all bids with a given item-id in the *bids* and computing the maximum. With the denormalized schema, obtaining the current high-bidder of an item is much faster as it requires only a single row lookup.
- Secondary indexes To speed up queries that enumerate the bids placed on an item or submitted by a user, it is desirable to have secondary indexes on the *item* and *bidder* columns for the *Bids* table. These secondary indexes need to scale as well as the main *Bids* table itself, hence, they should also be partitioned (based on *item* or *bidder*).
- *Materialized joins* Join operations across tables are particularly expensive in a distributed setting. Suppose a common query in the auction service is to list the names of all users who have placed a bid on a given item. This query requires doing a distributed join of the *Bids* table with *Users*. But because these tables are partitioned differently, this requires the database to

query all partitions of the *Bids* table to return a result. Materialized join views reduce the cost of these queries by pre-computing the join result, and maintaining duplicated information from the *Users* and *Bids* tables.

All above techniques impose consistency constraints (i.e. correctness invariants) between partitions managed by different machines. For example, the denormalized *high_bidder* and *high_price* columns should match the actual highest bid recorded in the *Bids*. Similarly, secondary indexes and join tables must correctly reflect changes to the main tables.

We can maintain these consistency constraints with ACID transactions, however, doing so comes at a performance cost. Distributed protocols for achieving serializability require tight coordination among machines [26], increasing the tail latency of both reads and writes. The latency increase is especially pronounced when data is replicated across multiple data centers, mandating cross data-center coordination.

Our example application can actually be satisfied with a weaker guarantee: that the consistency constraints among different table partitions are eventually satisfied.¹ For our auction service it is acceptable to read a high bid price from *Items* that is lower than the true high price according to the *Bids* table when there are concurrent bidding operations. Similarly, it is okay for updates to secondary and join tables to "lag" behind that of the main tables. By contrast, our auction service will be incorrect if the high price in *Items* does not eventually reflect that of *Bids*, or if secondary and join tables permanently diverge from the corresponding main tables. Many other web applications share this property of requiring only eventual consistency across partitions.

Two challenges make it difficult to enforce consistency constraints across partitions when using only local transactions: failure handling and concurrency.

Machine failures may result in a permanently inconsistent state if not handled properly. As a write operation needs to update denormalized data or secondary/join tables in different partitions,

¹This is a different notion of eventual consistency than the simple property that all replicas of a data item eventually converge to the same value. Here, eventual consistency also refers to the property that the consistency constraints among different data items are eventually satisfied.

the failure of a server might cause some but not all of the updates to complete, preventing the eventual consistency across partitions.

In the absence of serialized operations, concurrent operations may be ordered differently on different partitions. For updates that commute with each other, such arbitrary interleaving does not matter. For example, for concurrent chains placing bids on the same item, the execution order of different chains inserting to *Bids* does not need to match that of them modifying *Items* because computing the maximum bid price is commutative. However, not all updates commute. Suppose operation O_1 changes the username of uid 123 to "Alice_123" and O_2 changes the name of uid 123 to "Alice_abc" concurrently. If O_1 's modification is ordered before that of O_2 for *Users*, but vice-versa for the join table uniting *Users* and *Bids*, the join table and *Users* will permanently diverge.

6.2 Overview

We designed Lynx to address the challenges outlined above, and to make it easy to write scalable, high-performance web applications.

Scaling. Lynx achieves scalability by partitioning each database table into a large number of logical data partitions according to its primary key. Each data partition may be stored by a different machine which handles all read/write requests to that partition. The number of partitions is static, but this is not a problem because users can choose a large number of partitions and assign many partitions to the same server. As the system grows, the user can reassign the partitions to be in separate servers.

Geo-replication. Lynx works both with and without geo-replication. When geo-replication is enabled, Lynx performs full replication of data across all data centers, a mode of operation common to many geo-replicated systems [62, 80]. All read operations are handled by the local data center. Inspired by the design of Walter, Lynx associates each data partition with a "home"

data center (site). The server responsible for the partition at its home site orders all the writes to that partition and synchronously replicates them to other data centers.

Support for derived tables. Lynx supports optimizing queries via secondary indexes and materialized joins. A programmer may specify indexes and materialized joins as part of their table schema. These are provided by Lynx by what we call *derived tables*. A derived table is, as its name implies, a table whose contents are derived from one or more tables. There are two types of derived tables: index and join. An index derived table stores a secondary index for the table; the key for the derived table is the index column. A join derived table stores the pre-computed join of two tables; the key for the derived table's key. When the underlying table(s) changes, Lynx automatically updates derived tables, but this is not done in an atomic way, since we want to avoid distributed transactions. For that reason, programmers must structure an application so that the freshness difference between derived and main tables does not result in correctness problems. To force programmers to be aware of this difference, Lynx exposes derived tables as a set of read-only tables that are queried independently.

When a main table is modified, Lynx automatically updates related derived tables. Lynx helps the programmer reason about the relative freshness of related tables by enforcing a specific ordering of updates to different derived tables. Specifically, a main table T is more up to date than its secondary index table, and a secondary table of T is more up to date than any join table of T.

Consistency guarantee and mechanism. Lynx provides strong consistency and ACID transactions within each data partition. For consistency constraints across partitions, Lynx provides the weaker guarantee that all derived tables and denormalized data are eventually consistent with the main tables.

Lynx addresses the challenges of enforcing consistency constraints across partitions via its new primitive, Distributed Transaction Chain (DTC). A DTC groups together a series of transactions, each modifying a single data partition, and guarantees that all of them will be eventually executed despite failures. If any transaction in the chain incurs a user-initiated abort, DTC ensures that all the previously executed transactions are undone by corresponding compensating transactions. To address concurrent ordering issues, a DTC can provide an ordering constraint, which ensures that hops from different chains maintain their relative order of execution across partitions. Specifically, if two chains X and Y start in the same partition and X executes before Y in that partition, then X executes before Y in any partition where they both execute. This ordering constraint is essential to preserve consistency across partitions when chains do not commute with each other. As an example, suppose we have two DTCs, where one adds a sales order, and the other deletes the order. These chains modify a sales table and then insert an event log into a log table. Without the ordering constraint the log table might indicate the sales order exists (delete occurs before add), but no such sale exists in the sales table (delete occurs after add).

Lynx uses DTCs extensively internally to update secondary indexes and join tables as their corresponding main tables are modified. By leveraging the fault-tolerance and ordering behavior of DTCs, Lynx can maintain the consistency between derived tables and main tables in the face of failures and concurrency. Furthermore, Lynx exposes DTCs to application programmers to allow them to maintain consistency properties across their own denormalized data.

Chapter 7

Lynx's programming interface

Lynx's programming interface consists of a client-side library in Python and a set of table schema specifications. Our Python client library is inspired by model-view-controller style web frameworks. Like Django, Lynx mediates application access to Lynx via a set of Python classes for each table.

We design Lynx's API to give programmers sufficient control so that they can best optimize application performance while also benefiting from the consistency guarantees provided by Lynx. Specifically, Lynx gives programmers control in three areas: (i) how to partition each table and what home site to associate with each partition (§7.1), (ii) which secondary indexes and join tables are created and which derived table are used for specific queries (§7.2), and (iii) creating their own DTCs to modify denormalized data consistently (§7.3).

7.1 Creating and accessing tables

Creating partitioned tables. Programmers define partitioned tables using a SQL-like syntax. Figure 7.1 shows the schema of the *Users* and *Items* table for our auction service example. The programmer first defines the set of logical partitions that tables are split into. In this example, there
```
USE_PARTITIONS SIZE=1000, SHARD_BY=mod, NODES=db_config
CREATE TABLE Users
{PRIMARY=uid} (
    uid integer,
    username varchar(16),
    location varchar(16)
);
CREATE TABLE Items
{PRIMARY=item_id COLOCATE=seller} (
    item_id integer,
    description varchar(100),
    seller integer,
    high_bidder integer,
    high_price float
);
```

Figure 7.1: Schema of the partitioned *Users* and *Items* table.

```
import linked_dbc
# auto-generated class definitions for table schemas
import auction_schema

def get_user_name(uid):
    ctx = get_db_context()
    user = Users().lookup(context=ctx, id=uid)
    return user.name

def add_user(name, location):
    ctx = get_db_context()
    Users().insert(
        context=ctx, uid=new_local_id(),
        name=name, location=location)
```

Figure 7.2: Accessing the Users table.

are 1000 partitions and they are spread across a group of servers determined by the configuration named db_config. Each main table is always partitioned by its primary key.

Accessing main tables. Figure 7.2 gives an example of querying and modifying the *Users* table. The programmer accesses Lynx within a given context, obtained using get_db_context. The context maintains the client's network connections to servers and keeps cached information mapping logical partitions to servers. Table operations lookup, insert, update are all performed using the primary key for the table. Lynx only supports table modification based on the primary key. By default, operations return as soon as the main table is modified but before the corresponding derived tables are updated. The programmer can instead request to block until all derived tables have been updated.

Figure 7.3: Specifying secondary indexes and join tables. *Bids* has two secondary keys bidder and item. *Bids-Users* is a join table uniting *Bids* and *Users* with join key Bids.bidder.

Controlling home sites. When Lynx is used with geo-replication, it uses the notion of home sites (a site is a data center) to avoid conflicting updates at different sites efficiently [81]. Each logical partition has a home site, which is where the rows associated with the partition can be modified. Other sites must forward updates to the partition's home site.

Programmers may wish to control the home site of newly created data items, so it matches the site where the application is running. To do so, the programmer uses the new_local_id function, which returns a new primary key in a local partition served by the current site. Lynx also allows users to specify that two tables share the same partitioning. In Figure 7.1, we specify that each row of the *Items* table should co-locate with its seller column. Any new entry to the *Items* table will be assigned a primary key that maps to the same partition as the seller column. As a result, an auction item may be modified by its seller at his home site.

By default, insert and update return as soon as the replica of the main table is modified at its home site. The programmer can change this behavior to block until all replicas have been written.

7.2 Creating and accessing derived tables

Creating derived tables Figure 7.3 gives the schema for the *Bids* table with secondary keys on bidder and item and the join table *Bids-Users*. We define *Bids-Users* by specifying the join

```
def get_bidders_names(itemid):
    ctx = get_db_context()
    results = Bids-Users().query(
        context=ctx, Bids_item=itemid)
    return [ r.Users_name for r in results ]
```

Figure 7.4: Querying join table Bids-Users with secondary key.

```
def submit_bid(bidder, itemid, price):
  a1 = dtc.action(
   procedure=_insert_bid,
   arguments=[bidder, itemid, price]
   return_point=True)
  a2 = dtc.action(
   procedure=_update_item,
   arguments=[bidder, itemid, price])
  chain = dtc.new_chain(a1,a2)
  ctx = get_db_context()
  chain.execute(ctx, order=False)
@dtc.partition(Bids, bidder)
def _insert_bid(ctx, bidder, itemid, price):
 Bids().insert(
    context=ctx, bidder=bidder, item=itemid, price=price)
@dtc.partition(Items, itemid)
def _update_item(ctx, bidder, itemid, price):
 item = Items().lookup(context=ctx, item=itemid)
 if item.high_price < price:
    Items().update(
      context=ctx, item=itemid,
     high_bidder=bidder, high_price=price)
```

Figure 7.5: Specifying a user-defined DTC.

key (Bids.bidder=Users.uid). Secondary tables always duplicate all columns of the main table, but Lynx lets programmers choose the set of columns to be duplicated in the join table. One can also specify secondary keys for the join table, e.g., in Figure 7.3, *Bids-Users* has Bids.item as its secondary key. We can use this secondary table to quickly display the names of bidders on an item.

Accessing derived tables. Derived tables are exposed to the programmer in the same manner as main tables. All derived tables are read-only and do not accept the insert, update or delete methods. Figure 7.4 gives an example of querying the join table *Bids-Users* based on its secondary key.

7.3 User-defined DTCs

Lynx lets programmers specify user-defined DTCs to maintain the consistency of denormalized data in different tables. A user-defined DTC consists of a sequence of *hops*, where each hop is expressed as a Python procedure. A hop can read or write data in a single partition and its database accesses are performed in a local transaction. The programmer may write hops that abort the chain in the middle; in the case, the programmer must also write a corresponding compensating action for each previous hop in the chain.

Figure 7.5 gives an example of a user-defined DTC. Our auction example invokes the submit_bid function when a user places a bid. This function constructs a DTC out of two procedures, _insert_bid and _update_item. _insert_bid inserts the bid into *Bids* and _update_item update the maximum bid on the item if the new bid contains a higher bid price. Lynx requires programmers to specify the partition accessed by each hop. This allows Lynx to transfer a DTC to the server responsible for each hop. The partition hint is specified as a Python function annotation, as seen in Figure 7.5 and the hint typically consists of the name of the table to be accessed and an index key value. For example, _update_item will execute on the partition containing the row identified by item in *Items* table.

Because updates to the max bid in *Items* always commute, the relative ordering of concurrent submit_bid operations in different partitions does not matter. We therefore set order=False when starting the chain to allow the system to execute concurrent chain hops in any order.

The programmer can also decide whether to block waiting for the completion of the chain. In Figure 7.5, we set return_point=True in _insert_bid, allowing the client library to return as soon as the first hop completes. Returning to the user as soon as his bid is recorded reduces the latency of submit_bid, but users might notice a small delay between when a bid is submitted and when it shows up as an item's top price. For an auction service, this is likely to be an acceptable trade-off. If this is not the case, the programmer can also choose to block until _update_item

completes at the cost of increased latency.

Chapter 8

System design and implementation of Lynx

This chapter begins with a tour of Lynx's design (Section 8.1). We then discuss Lynx's implementation (Section 8.2), and conclude the chapter by providing formal proofs of the correctness of Lynx's derived tables' operations (Section 8.3).

8.1 Design

8.1.1 Architecture overview

A Lynx deployment consists of many *server* machines running in one or more data centers, a large collection of *clients* linked into application servers, as well as a separate configuration service.

The *configuration service* keeps track of the mapping from every logical data partition to its responsible server in each data center and the partition's home site. The configuration service runs as a Paxos-based state machine replicated across multiple sites to tolerate faults. Each Lynx server obtains a lease from each of its responsible data partitions from the configuration service, similar to the design in [32, 85].

Figure 8.1 shows the interaction between Lynx clients and servers. Each Lynx client fetches



Figure 8.1: An overview of different components in Lynx and their interactions. the mapping of partitions to servers from the configuration service and caches the information locally so that it can directly contact the responsible server for accessing the database. Stale cache information does not affect correctness since each server will only handle requests for partitions it is currently responsible for.

Each Lynx server runs a database system locally to store partition data. A client contacts the Lynx server process for all its write requests but may connect directly to the underlying database for better read performance. Upon receiving a write request, the Lynx server generates the required DTC for updating secondary or join tables if necessary. When running in a geo-replicated setting, a server in the home site also replicates any local writes to the remote servers.

8.1.2 Executing DTCs

When transferring and executing DTCs from server to server, we must ensure that each hop in a chain is executed exactly once and that the required ordering constraint is preserved for each chain.

We employ two mechanisms to guarantee exactly-once execution. First, a Lynx server implements a queue module to reliably store DTCs received from other servers or clients. Second, a Lynx server inserts an execution record (consisting of a chain id and its hop number) into a history table maintained by the local database as part of the local transaction for executing a hop in the chain.

The naive design to avoid duplicate execution makes the server check the history table before running each hop to ensure that it has not been previously executed. This is inefficient because it requires an additional lookup in the database's history table for every operation. We use a more efficient design that leverages the local queue module to avoid duplicates. Specifically, the queue module offers three APIs, *push* to enqueue a DTC, *borrow* to temporarily dequeue a DTC for execution (or transfer to another server), *release* to remove a DTC from the queue. The queue module logs the *push* and *release* events synchronously to an on-disk log file in batches. During normal operation, each worker thread in the server borrows a DTC to execute against the local database and releases the DTC upon completion. Whenever a server restarts after a failure, the queue log files are read to determine the set of operations that have been pushed, but not yet released. The server replays these actions by consulting the history table kept in the local database: if a queue item X is not present in the history table then it must be applied to the database. If the item is found in the history table then it can be safely discarded. The queue module also avoids duplicates that might arise during the transfer of DTCs from one server to another in the face of network and server failures; we do not discuss the details here.

We enforce the ordering constraint of chains using pair-wise sequencers among n logical data partitions. In particular, for each partition i that a server is responsible for, the server maintains n counters, $ctr_{i\rightarrow 1}$, $ctr_{i\rightarrow 2}$, ..., $ctr_{i\rightarrow n}$, which it increments and assigns as sequence numbers to a DTC. The server also keep tracks of the latest sequence number that it has processed on partition ifrom other partitions, $done_{1\rightarrow i}$, $done_{2\rightarrow i}$,..., $done_{n\rightarrow 1}$.

All chains obeying the ordering constraint are required to have a pre-determined trajectory, i.e., the set of partitions to be accessed are known at the start of the chain's execution. Suppose a chain of length m is to execute at partitions 1,2,...,m. The Lynx server executing the chain at partition 1 generates m-1 sequence numbers, one for each of the remaining hops, by incrementing one on the corresponding *ctr* counters. These sequence numbers are attached to the chain as

 $seq_{1\rightarrow2}$, $seq_{1\rightarrow3}$, ..., $seq_{1\rightarrow m}$ and are used to order execution on the chain's subsequent hops. The server responsible for partition *i* compares the chain's sequence number, $seq_{1\rightarrow i}$ with its local value $done_{1\rightarrow i}$. If the chain's sequence number is the next in line, i.e., $seq_{1\rightarrow i} = done_{1\rightarrow i} + 1$, then the server executes the chain immediately and increments $done_{1\rightarrow i}$ accordingly. Otherwise, it blocks the chain until those "holes" in the sequence $1 \rightarrow i$ have been filled up by the arrival and execution of other chains. Should a chain incurs a user-initiated abort at a server, the server sends empty actions to the chain's subsequent hops to "release" its sequence numbers. Lynx's sequencing mechanism is very efficient as it does not incur any additional communication between servers during chains' execution.

When working in a geo-replicated setting, each hop of the chain executes in the home site of the partition. Specifically, the responsible server in the partition's home site first executes the hop locally and then synchronously replicates it to remote data centers¹. Once replication completes, the server proceeds to transfer the DTC to its next hop server.

8.1.3 Maintain derived tables

Secondary indexes. Whenever a server receives a client request to modify a main table, it creates a DTC to update both the main table and its derived tables². We explain this process using an example main table T with primary key (K_0) and two secondary indexes (K_1, K_2). As illustrated in Figure 8.2, the DTC to insert a new row in T consists of three hops that insert the row into the appropriate partitions for tables T, T₋ K_1 , T₋ K_2 , respectively. The DTCs for deleting an existing row and for updating a column that is not a secondary key are similar to that of the insert.

Lynx handles updating a secondary key of an existing row in T differently. Suppose one updates

¹Databases like PostgreSQL and MySQL have built-in master-slave replication. Unfortunately, they do not allow separate replica groups for different partitions managed by the same database instance. Thus, we choose to implement synchronous replication ourselves in Lynx.

²Note that these are DTCs generated automatically by the system, not the user-generated DTCs. They both use the same machinery to execute.



Figure 8.2: The DTCs for inserting a new row and updating an existing row's secondary key value in table T which has two secondary tables T_K1, T_K2.



Figure 8.3: The DTCs for inserting a new row and updating an existing row's join key value. To create join table LT-RT, both main tables LT and RT have a secondary table, LT_K_{join} , RT_K_{join} , corresponding to the join key K_{join} .

a row with the primary key value k_0 to have a new secondary key value k'_1 . The resulting DTC consists of four hops, as shown in Figure 8.2. First, the chain reads the old secondary key value k_1 from the main table and updates the value to k'_1 . Second, the chain deletes the old row from secondary table T_K_1 with key k_1 . Third, the chain inserts the new row into T_K_1 with key k'_1 . Lastly, the chain updates the other secondary table T_K_2 .

The use of DTC guarantees that all hops for updating secondary tables will be executed in the event of failure. Furthermore, since insert/delete/update operations to the same main table row do not commute with each other, the ordering constraint of the DTCs is crucial for maintaining the correctness of secondary tables in the face of concurrent operations.

Join tables. In the most general case, Lynx needs to maintain the join table LT-RT where the join key K_{join} is a secondary key for both the left main table (LT) and the right main table (RT).

We explain the join process for modifying LT. The top chain in Figure 8.3 handles the insertion

of a new row into LT with the join key value k. The first hop of the chain inserts the row in LT. The chain's second hop is destined for the partition determined by the join key value k. Because Lynx requires that both LT and RT be spread across the same set of logical partitions, all three table splits (LT_K_{join} , RT_K_{join} , LT-RT) for k reside in the same logical partition. Therefore, at the second hop, the chain is able to insert the new row into the secondary table LT_K_{join} , read the set of tuples Y matching the join key value k_j from RT_K_{join} , and insert the set of tuples joining the new row and set Y into LT-RT, all in one local transaction. The DTC to delete an existing row first deletes the row from LT and then deletes the row in LT_K_{join} and the set of rows in $LT-RT_K_{join}$ on partition k in one local transaction. Updating an existing row's join key value is like performing an insert following a delete, as illustrated in Figure 8.3.

LT may have secondary indexes other than the join key. If so, the DTC to modify LT contains additional hops to update those secondary tables before accessing the join key partition containing LT_K_{join} , RT_K_{join} , and LT-RT. Additionally, the join table LT-RT itself may have secondary indexes. Lynx updates these secondary tables by spawning new DTCs at the last hop of the join chain. In principle, for every row modified in LT-RT, a DTC is created to update the corresponding secondary table partitions. These DTCs execute in parallel. As an optimization, Lynx batches those updates that traverse the same sequence of partitions into a single DTC.

The correctness of the join process is assured by two features in the system. First, the ordering constraint of the underlying DTCs enables concurrent modifications on the same row of LT to interleave correctly. Second, the local transaction for accessing the affected rows in LT_K_{join} , RT_K_{join} , and LT-RT enable concurrent modifications from LT and RT that affect the same set of rows in LT-RT to interleave correctly. Section 8.3 formally proves these claims.

8.2 Implementation

Both the client and server implementation of Lynx are written in Python. Each server uses PostgreSQL as its local database backend. Our prototype provides a parser that translates a Lynx schema into a collection of native SQL schema files used by PostgreSQL. The parser also generates Python classes that serve as the data access interface in the client library. These auto-generated classes hide from programmers the complexity of Lynx's internal data layout and the partitioned nature of the underlying storage. The construction of DTCs for updating secondary and join tables is part of the server implementation. For user-defined chains, the client library marshals the DTC (including its procedure definition, partition hint and arguments) and transfers it to the server for storage and execution.

Two details in the implementation are worth mentioning. First, we impose a "soft" limit on the capacity of each server's DTC queue. If the queue exceeds its limit, the *push* operation is delayed by a time period proportional to the current queue length, slowing down clients' requests. Doing so ensures that those DTCs already admitted into the system finish in a bounded amount of time. Second, we built our own custom full-featured RPC library for transferring DTCs. Our RPC library achieves much better performance than several existing RPC implementations for Python (including the standard xmlrpc library).

8.3 Correctness of Lynx's operations

8.3.1 Model

Database model. For simplicity, we restrict our discussion to two table R and S. All proofs extend naturally to a larger number of tables. Let a table row be represented by a tuple r. Tuple columns in Lynx do not all have the same semantics. In a row $r = (k_0, \ldots, k_n, c_1, \ldots, c_m)$ the columns k_0, \ldots, k_n are index columns referred to as *partition keys* (or just *keys*). k_0 serves as a

row's primary key and has the additional constraint that it is unique across a table R. We denote the attribute values of tables $(\rho_0, \rho_1, \ldots, \rho_w)$ where w = m + n + 1. For brevity, we use several shorthand notations: We substitute the values c_1, \ldots, c_m with a single notation data, as a stand-in for all columns which are not partition keys. Also, we use r to denote both a single row r and the set containing r as its only member $(\{r\})$. A table R is a set containing all rows r: $R = \bigcup_i r_i$. A table R may have several copies denoted R^0, \ldots, R^n corresponding to partition keys. The table R^0 is a main table, whereas $R^j, j = 1, \ldots, n$ are secondary tables. Each table R^j may be partitioned according to the key k_j into P_j disjoint partitions: $R^j = \bigcup_{p=1}^{P_j} R_p^j$ such that $\forall l, p : l \neq p, R_p^j \cap R_l^j = \emptyset$. **Basic Data Operations.** We consider three basic database write operations on tables: insert, delete and update. It is illegal to update k_0 for any r in a table. An insert to a table is an addition of a row r to R. A deletion of a row is the removal of r from R. An update simply updates a row rin place. In addition, we support two other operators. A selection operator and a join operator. A selection operator $\sigma_{\varphi}(R)$ selects all rows from R that satisfy the selection predicate φ . We only allow predicates of the form $\{k_j = x\}$, where k_j is a partition key and x is an attribute value. The join operator is discussed below.

Partitioning. Lynx requires that each key $k_j, j = 0, ..., n$ has a corresponding partitioning *pure* function $part_j(x), K^j \xrightarrow{part_j} 1, ..., P^j$, where K^j is the key-space of k_j and P^j is the number of partitions of R^j .

Joins. We use the symbol \bowtie_{φ} to denote an operator that joins two tables R and S ($R \bowtie_{\varphi} S$) according to a join predicate φ . Lynx joins have the following constraints: They are inner equi-joins whose join predicates may only use partition keys. In addition, Lynx requires that join predicate columns k_r and k_s from two tables R and S have the same partitioning function ($part_r = part_s$). For simplifying the notation, we assume that the joined tables R^j and S^l have the same set of partitioning keys k_0, \ldots, k_n , hence we can denote the join of R^j and S^j as $R \bowtie_{k_j} S$. (Note however, that Lynx does not perform *natural* joins and that it support joins of different partitioning keys, as

long as they have the same partitioning function.) The extension of all proofs to the general case are trivial using the required notation.

Execution. The state of the database, as modelled above, can be changed by issuing the supported update operations. We assume all requests are valid, as defined by each algorithm. In the case of the centralized algorithms, a single thread of execution processes requests serially, as they arrive. In this case, we do not model a user issuing the requests. Instead, we view users' requests as a predefined sequence of operations H. In the distributed version of the algorithms, each node is assigned a single partition domain. A partition domain node, denoted N_p^j has a single thread of execution and can only access data that resides on partition T_p^j , for any table T. Both in the centralized version of the algorithms and in the distributed one, we add a *transaction* primitive. Transactions are started with a BEGIN TRANSACTION call and end with a COMMIT call (we do not need to consider abort cases). The consistency level provided by these transactions in *serializability*, with the existing restriction that only access to local data is allowed. (In the centralized case, this is the entire database.) A thread of execution can delay any part of an operation until some condition c holds. In that case the thread switches to service other requests (without violating transactional guarantees). The thread returns to the blocked execution automatically and immediately when the condition becomes true. We use the notation *WaitUntil(c)* for this waiting behavior. Nodes communicate using a reliable asynchronous message passing framework. Messages are always eventually delivered (the use of DTCs in Lynx, guarantees this property). We do not assume a FIFO communication channel. Instead, when FIFO is needed it is handled by the logic of the protocol. Finally, we use in the definitions the notion of termination of execution. This time is well-defined for a centralized algorithm. For a distributed algorithm, we say that the algorithm has terminated when all nodes have finished to process any outstanding requests and there are no messages in transit. We say that the last node to terminate is the node that, in terms of real time, finishes its execution last when there are no messages in transit.

8.3.2 Correctness of secondary tables operations

We prove the correctness of Lynx's secondary tables operations by first formally defining our correctness criteria (definition 8.1). We then proceed to prove that Lynx's operations satisfy this definition by first specifying a centralized version of the secondary tables operations (algorithm 1). We prove the correctness of the centralized algorithm. We then give a partitioned version of the algorithm (algorithm 2), and show that it is equivalent to the centralized one. This conceptual step is needed for clearly defining what it means for a centralized version and a distributed one to operate on the same input of users' requests. We then give Lynx's full distributed algorithm (algorithm 1). We prove that it is equivalent to the centralized version, thus showing that it is correct.

The following definition states what it means for secondary tables' state to converge to the state of main tables. Additionally, it requires that each row ends up in the right partition, thus guaranteeing that subsequent queries by a secondary key will be both efficient, and, after convergence, complete.

Definition 8.1 (Secondary keys integrity). We say that an algorithm preserves *secondary keys integrity* if at the end of any execution of valid update operations *H*, the following holds:

1. (Replication Integrity) For j = 1, ..., n:

(a)
$$\bigcup_{p=1}^{P_j} R_p^j = \bigcup_{p=1}^{P_0} R_p^0$$

(b) $\forall l, p, l \neq p : R_p^j \cap R_l^j = \emptyset$

2. (Partitioning Integrity) $\forall (\rho_0, \rho_1, \dots, \rho_n, data) \in \bigcup_{p=1}^{P_0} R_p^0$, and for $j = 1, \dots, n$ $(\rho_0, \rho_1, \dots, \rho_n, data) \in R_{part_j(\rho_j)}^j$

The first property of definition 8.1 requires that all rows of the main tables exist in secondary tables and that there are no other rows in secondary tables. It also states that a row r cannot be

present in two different partitions. The second property requires that each row r ends up in the right partition of each secondary table in accordance with its partitioning key and the partitioning function.

Algorithm 1 gives a centralized specification of Lynx's update logic.

Proposition 4. Algorithm 1 preserves secondary keys integrity.

Proof. The complete proof follows an induction on the number of operations in H. We only discuss the non-trivial cases: From the logic of *ExecuteRowOperation* it can be seen that the cases of *INSERT* and *DELETE* trivially satisfy definition 8.1. In the case of an *UPDATE*, immediately after line 1.15 is executed, property 1 of definition 8.1 does not hold anymore, and possibly, property 2 too. If condition 1.17 if true, line 1.18 satisfies property 1 again, which is the only one that was temporarily false. Otherwise, lines 1.20 and 1.21 make conditions 1 and 2 hold again. In all cases when procedure *ExecuteRowOperation* terminates, the prefix of operations from H that have been executed to that point leave the database in a state that satisfies definition 8.1.

The input to algorithm 1 is a sequence H of user operations with their arguments, which is a total ordering of these operations. The real input to the distributed version of the algorithm, which follows the real execution of Lynx, is a set of operations that users issue to different nodes. The ordering of this input is only a partial order, which is defined according to the order in which these operations are received on each node. To translate an input to the distributed version to an input to the centralized version, we allow a partial order of user requests H to be extended to a total order H' in any arbitrary way (e.g. by the real time in which operations were received on nodes). To justify this translation, we give algorithm 2 and prove proposition 5.

Definition 8.2. We say that two algorithms are *end-state equivalent* if once they terminate they leave the database in the exact same state (as expressed by the the equivalence of all the sets $R_n^j (j \ge 0)$ at the end of all executions of both algorithms).

Algorithm 1 Derived operations (centralized)

 P_0 **Require:** if $Op = UPDATE \lor Op = DELETE$ then $(\rho_0, \rho_1, \dots, \rho_n, data) \in \bigcup R_p^0$ **Require:** if Op = INSERT then $(\rho_0, \rho_1, \dots, \rho_n, data) \notin \bigcup R_p^0$ 1: **procedure** EXECUTEROWOPERATION($Op, \rho_0, \rho_1, \dots, \rho_n, data$) 2: **BEGIN TRANSACTION** 3: if Op = INSERT then Add $(\rho_0, \rho_1, \ldots, \rho_n, data)$ to $R^0_{part_0(\rho_0)}$ 4: 5: for $j \leftarrow 1, n$ do Add $(\rho_0, \rho_1, \ldots, \rho_n, data)$ to $R^j_{part_i(\rho_i)}$ 6: 7: end for else if Op = DELETE then 8: 9: Remove $(\rho_0, \rho_1, \ldots, \rho_n, data)$ from $R^0_{part_0(\rho_0)}$ for $j \leftarrow 1, n$ do 10: Remove $(\rho_0, \rho_1, \ldots, \rho_n, data)$ from $R^j_{part_i(\rho_i)}$ 11: end for 12: else if Op = UPDATE then 13: $(\rho_0, \hat{\rho_1}, \dots, \hat{\rho_n}, data) = \sigma_{\{k_0 = \rho_0\}}(R^0_{part_0(\rho_0)})$ 14: Update: $\sigma_{\{k_0=\rho_0\}}(R^0_{part_0(\rho_0)}) \leftarrow (\rho_0, \rho_1, \dots, \rho_n, data)$ 15: for $j \leftarrow 1, n$ do 16: if $k_j = \hat{k_j}$ then 17: Update: $\sigma_{\{k_0=\rho_0\}}(R^j_{part_i(\rho_i)}) \leftarrow (\rho_0, \rho_1, \dots, \rho_n, data)$ 18: 19: else Remove $(\rho_0, \hat{\rho_1}, \dots, \hat{\rho_n}, data)$ from $R^j_{part_i(\hat{\rho_i})}$ 20: Add $(\rho_0, \rho_1, \ldots, \rho_n, data)$ to $R^{j}_{part_i(\rho_i)}$ 21: 22: end if end for 23: end if 24: COMMIT 25: 26: end procedure **procedure** EXECUTION(*H*) 27: for $(Op, \rho_0, \rho_1, \ldots, \rho_n, data)$ in H do 28: EXECUTE **R**OW OPERATION ($Op, \rho_0, \rho_1, \ldots, \rho_n, data$) 29: end for 30: 31: end procedure

Algorithm 2 Derived operations (partitioned)

Require: ψ is a permutation of H that preserves all sub-sequences $H|_{k_0=x}$

- 1: **procedure** PARTITIONEDEXECUTION(H, ψ)
- 2: $H^{\iota} \leftarrow \psi(H)$
- 3: EXECUTION(H')
- 4: end procedure

Proposition 5. Algorithms Execution and PartitionedExecution are end-state equivalent (for any input H and any valid ψ).

Proof. When procedure *ExecuteRowOperation* gets a row $(\rho_0, \rho_1, \ldots, \rho_n, data)$ as input, regardless on the execution path that is taken, all update operations (INSERT, DELETE, UPDATE) operate on rows with the same primary key ρ_0 . Any two rows that don't share the same primary key are different set elements. In general, set operations on the same set with different set-members as arguments commute. Consider each set R_j^p and the sequence of operations that are applied to it in algorithms 1 and 2. Each of these sets goes through the exact set of update operations in algorithm 2 as it does in algorithm 1. The operations applied to sets in algorithm 2 either commute with one another or, the ones that don't commute, are applied in the exact same order in which they are applied in algorithm 1. This implies that all sets will have the exact same final state once all operations complete.

We proceed to give the final algorithm which formally captures Lynx's distributed execution. Algorithm 3 uses two new state variables kept on each node. Both are arrays indexed by all other nodes. *sequences* is used by nodes storing main partitions to issue pair-wise sequence numbers that accompany update messages, and are used to order secondary operations on their target node (which isn't always the direct recipient of the message). The array *lastSeen* is kept on nodes responsible for secondary partitions It is used to verify that operations are executed in the exact same order in which they were on main partitions.

Proposition 6. Algorithms PartitionedExecution and DistributedExecution are end-state equivalent, for any sequence of operations that is input to both algorithms in the same order, in terms of

Algorithm 3 Derived operations (distributed)

1: **procedure** DISRIBUTEDEXECUTION(H)

- 2: **for** $(Op, \rho_0, \rho_1, \dots, \rho_n, data)$ in H **do**
- 3: **SENDTO** $(N_{part_0(\rho_o)}^0, DistributedExecuteRowOperation, Op, \rho_0, \rho_1, \dots, \rho_n, data)$
- 4: **end for**
- 5: end procedure

real time.

Proof. Any two update operations on a set that have different items as arguments - commute. We show that for each secondary partition N_p^j , all non-commutative operations are applied on that partition, in any an execution of algorithm 3, in the same order they are applied on the same partition in algorithm 1. For every row, every operation on a primary partition $N_{part_0(\rho_0)}^0$ produces the exact same operations on each secondary partition $N_{part_i(\rho_i)}^j$ in algorithm 1 and in algorithm 3. Hence, the exact same operations are applied on each partition N_p^j in both algorithms. Assume for the sake of contradiction that there are two non-commutative operations Op_i and Op_j that execute in the order Op_i before Op_j on a secondary partition N_p^j in algorithm 1, but their equivalent operations \hat{Op}_i and \hat{Op}_i execute in algorithm 3 in the order: \hat{Op}_i before \hat{Op}_i , on the same partition. Now, in algorithm 1 no single execution of procedure ExecuteRowOperation modifies the same secondary partition N_p^j more than once, so Op_i had to execute in an earlier invocation of *ExecuteRowOp*eration. Since Op_i and Op_j do not commute in the particular case, they must both have, as an argument, a row with the same primary key ρ_0 . This implies that there are two corresponding triggering user operations in algorithm 3, both with row arguments containing ρ_0 , such that the one resulting in Op_i was received and processed by procedure DistributedExecuteRowOperation before the one resulting in $\hat{Op_i}$. Since both $\hat{Op_i}$ and $\hat{Op_i}$ are eventually applied on the same partition, the messages containing them, must have been issued a sequence number from the same index in sequences in $N_{part_0(\rho_0)}^0$ - either as nextSeq or as srcSeq (call these sequences seq_i and seq_j). It must be then that $seq_i < seq_j$. However, by assumption, \hat{Op}_j was applied before \hat{Op}_i on N_p^j . But this is impossible since both operations share the same source $N_{part_0(\rho_0)}^0$, so the assumed **Require:** On node N_p^0 , upon receiving *DistributedExecuteRowOperation* from user 6: **procedure** DISTRIBUTEDEXECUTEROWOPERATION($Op, \rho_0, \rho_1, \ldots, \rho_n, data$) 7: **BEGIN TRANSACTION** if Op = INSERT then 8: Add $(\rho_0, \rho_1, \ldots, \rho_n, data)$ to $R^0_{part_0(\rho_0)}$ 9: 10: for $j \leftarrow 1, n$ do $nextSeq = sequences[N_{part_j(\rho_j)}^j]$ 11: $sequences[N_{part_j(\rho_j)}^j] \leftarrow sequences[N_{part_j(\rho_j)}^j] + 1$ 12: SENDTO($N_{part_i(\rho_i)}^j$, SecondaryInsert, nextSeq, ($\rho_0, \rho_1, \ldots, \rho_n, data$)) 13: 14: end for else if Op = DELETE then 15: Remove $(\rho_0, \rho_1, \ldots, \rho_n, data)$ from $R^0_{part_0(\rho_0)}$ 16: for $j \leftarrow 1, n$ do 17: $nextSeq = sequences[N_{part_i(\rho_i)}^{j}]$ 18: $sequences[N_{part_j(\rho_j)}^j] \leftarrow sequences[N_{part_j(\rho_j)}^j] + 1$ 19: SENDTO($N_{part_j(\rho_j)}^j$, SecondaryDelete, nextSeq, ($\rho_0, \rho_1, \ldots, \rho_n, data$)) 20: end for 21: 22: else if Op = UPDATE then $(\rho_0, \hat{\rho_1}, \dots, \hat{\rho_n}, data) = \sigma_{\{k_0 = \rho_0\}}(R^0_{part_0(\rho_0)})$ 23: Update: $\sigma_{\{k_0=\rho_0\}}(R^0_{part_0(\rho_0)}) \leftarrow (\rho_0, \rho_1, \dots, \rho_n, data)$ 24: 25: for $j \leftarrow 1, n$ do $srcSeq = sequences[N_{part_j(\rho_j)}^j]$ 26: $sequences[N_{part_j(\rho_j)}^j] \leftarrow sequences[N_{part_j(\rho_j)}^j] + 1$ 27: if $k_j = \hat{k_j}$ then 28: SENDTO($N_{part_j(\rho_j)}^j$, SecondaryUpdate, srcSeq, ($\rho_0, \rho_1, \ldots, \rho_n, data$)) 29: else 30: $nextSeq = sequences[N^{j}_{part_{j}(\hat{\rho_{j}})}]$ 31: $sequences[N_{part_{j}(\hat{\rho}_{j})}^{j}] \leftarrow sequences[N_{part_{j}(\hat{\rho}_{j})}^{j}] + 1$ $src \leftarrow N_{part_{0}(\rho_{0})}^{0}$ 32: 33: $r \leftarrow (\rho_0, \rho_1, \dots, \rho_n, data)$ 34: SENDTO($N_{part_i(\hat{\rho}_i)}^j$, UpdateDelete, nextSeq, $\hat{\rho}_j$, r, src, srcSeq) 35: 36: end if end for 37: end if 38: 39: COMMIT 40: end procedure

```
Require: On Node N_p^j, upon receiving SecondaryInsert from node N_q^l
```

```
41: procedure SECONDARYINSERT(nextSeq, (\rho_0, \rho_1, \dots, \rho_n, data))
```

```
42: WAITUNTIL(lastSeen[N_q^l] = nextSeq - 1)
```

- 43: BEGIN TRANSACTION
- 44: Add $(\rho_0, \rho_1, \dots, \rho_n, data)$ to $R^j_{part_j(\rho_j)}$
- 45: $lastSeen[N_q^l] \leftarrow nextSeq$
- 46: COMMIT
- 47: end procedure

Require: On Node N_p^j , upon receiving *SecondaryDelete* from node N_q^l 48: **procedure** SECONDARYDELETE(*nextSeq*, ($\rho_0, \rho_1, \ldots, \rho_n, data$))

- 49: WAITUNTIL $(lastSeen[N_q^l] = nextSeq 1)$
- 50: BEGIN TRANSACTION
- 51: Delete $(\rho_0, \rho_1, \dots, \rho_n, data)$ from $R^j_{part_j(\rho_j)}$
- 52: $lastSeen[N_q^l] \leftarrow nextSeq$
- 53: COMMIT
- 54: end procedure

Require: On Node N_p^j , upon receiving *SecondaryUpdate* from node N_q^l 55: **procedure** SECONDARYUPDATE(nextSeq, ($\rho_0, \rho_1, \ldots, \rho_n, data$)) 56: WAITUNTIL($lastSeen[N_q^l] = nextSeq - 1$) 57: BEGIN TRANSACTION 58: Update: $\sigma_{\{k_0 = \rho_0\}}(R_{part_j(\rho_j)}^j) \leftarrow (\rho_0, \rho_1, \ldots, \rho_n, data)$ 59: $lastSeen[N_q^l] \leftarrow nextSeq$ 60: COMMIT 61: **end procedure** **Require:** On Node N_n^j , upon receiving *UpdateDelete* from node N_a^l 62: **procedure** UPDATEDELETE($nextSeq, \hat{\rho_j}(\rho_0, \rho_1, \dots, \rho_n, data), source, srcSeq$) WAITUNTIL($lastSeen[N_q^l] = nextSeq - 1$) 63: **BEGIN TRANSACTION** 64: delete from $N_{part_j(\hat{\rho_j})}^j$ using $\sigma_{(k_0=\rho_0)}$ 65: $nextSeq2 = sequences[N_{part_j(\rho_i)}^j]$ 66: $sequences[N_{part_j(\rho_j)}^j] \leftarrow sequences[N_{part_j(\rho_j)}^j] + 1$ 67: SENDTO($N_{part_j(\rho_j)}^j$, UpdateInsert, nextSeq2, ($\rho_0, \rho_1, \dots, \rho_n, data$), source, srcSeq) 68: $lastSeen[N_q^l] \leftarrow nextSeq$ 69: COMMIT 70: 71: end procedure

```
Require: On Node N_p^j, upon receiving UpdateInsert from node N_q^l
72: procedure SECONDARYMOVE(nextSeq, \hat{\rho_j}(\rho_0, \rho_1, \dots, \rho_n, data), source, srcSeq)
73: WAITUNTIL(lastSeen[N_q^l] = nextSeq - 1)
```

- 74: WAITUNTIL(lastSeen[source] = srcSeq 1)
- 75: BEGIN TRANSACTION
- 76: Add $(\rho_0, \rho_1, \dots, \rho_n, data)$ to $R^{j}_{part_j(\rho_j)}$
- 77: $lastSeen[N_q^l] \leftarrow nextSeq$
- 78: $lastSeen[source] \leftarrow srcSeq$
- 79: COMMIT

```
80: end procedure
```

execution order implies a violation of the blocking condition of one of the lines: 42, 49, 56, 63, 73, or 74, in algorithm 3. The above contradiction implies that all non-commutative operations are applied to all partitions in the same order in both algorithms. Combined with fact that the same set of operations is applied to all partitions in both algorithms, they must end up in the same state. \Box

Corollary 7. Algorithm DistributedExecution preserves secondary keys integrity.

8.3.3 Correctness of join operations

As described in section 8.1.3, Lynx maintains materialized joins tables that are updated to reflect the effect of database write operations on the desired join. For brevity, we provide a proof to the correctness of the join process triggered after an *INSERT* row operation on a main table. The proof for the *DELETE* row operation follows the exact same lines. An *UPDATE* operation has the exact same eventual effect as a *DELETE* followed by an *INSERT* of the updated row, hence, can be replaced by these two operations.

Whenever a row r with partition key k_j is inserted to a partition $R_{part_j(\rho_j)}^j$, a check is performed to see if there is a join specification $R \bowtie_{k_j} S$. If so, all rows from $S_{part_j(\rho_j)}^j$ with $k_j = \rho_j$ are selected. The Cartesian product of r and these rows is inserted to $(R \bowtie_{k_j} S)_{part(\rho_j)}$. The exact same procedure follows for an insertion of a row s (replace R with S above). Formally, the insertion of row r_i adds to the materialized join table the following rows: $r_i \times \sigma_{\{k_j = \rho_j\}}(S_{part_j(\rho_j)}^j)$. Since by construction of S^j , all rows with $k_j = x$ (for any attribute value x) reside on the same partition, and due to the semantics of the selection operation, the above set is equal to $r_i \times \sigma_{\{k_j = \rho_j\}}(S^j)$. The following proposition states that this process is correct and yields the desired join.

Proposition 8 (Correctness of the join process). For two tables $R^j = \bigcup_{r_i \in R^j} r_i$ and $S^j = \bigcup_{s_l \in S^j} s_l$, If a materialized join table T is constructed by adding the rows $r_i \times \sigma_{\{k_j = \rho_j\}}(S^j)$ ($r_i \times \sigma_{\{k_j = \rho_j\}}(S^j)$) whenever a row r_i (s_j) is inserted to R^j (S^j), then after all operations complete $T = R^j \bowtie_{k_j} S^j$, which is also equal to $R \bowtie_{k_j} S$. **Lemma 3** (Join decomposition). For two tables $R^j = \bigcup_{r_i \in R^j} r_i$ and S^j , the following holds:

$$R^{j} \bowtie_{k_{j}} S^{j} = \bigcup_{r_{i} \in R^{j}} \left(r_{i} \times \sigma_{\{k_{j} = \rho_{i_{j}}\}}(S^{j}_{part_{j}}(\rho_{j})) \right)$$

Proof of lemma 3.

$$r \bowtie_{k_j} S^j = \sigma_{\{k_j = \rho_j\}}(r \times S^j) = r \times \sigma_{\{k_j = \rho_j\}}(S^j)$$

In addition:

$$R^{j} \bowtie_{k_{j}} S^{j} = \left(\bigcup_{r_{i} \in R^{j}} r_{i}\right) \bowtie_{k_{j}} S^{j} = \bigcup_{r_{i} \in R^{j}} \left(r_{i} \bowtie_{k_{j}} S^{j}\right)$$

Combining the above two lines and using $\sigma_{\{k_j=\rho_j\}}(S^j_{part_j(\rho_j)}) = \sigma_{\{k_j=\rho_j\}}(S^j)$ we get:

$$R^{j} \bowtie_{k_{j}} S^{j} = \bigcup_{r_{i} \in R^{j}} \left(r_{i} \times \sigma_{\{k_{j} = \rho_{i_{j}}\}}(S^{j}_{part_{j}}(\rho_{j})) \right)$$

		н
		н
		н
-		

Proof of proposition 8. We prove by an induction on the total number of rows added to both tables R^j and S^j . The induction holds trivially for empty tables. Adding a single row r or s is an application of of lemma 3 with $R^j = r$ or $S^j = s$. We prove the induction step: Assume that a total of n - 1 rows were added to both R^j and S^j and that the materialized join table T is the result of running the Lynx join protocol after these n - 1 insertions such that $T = R^j \bowtie_{k_j} S^j$. Consider the *n*th row inserted. With out loss of generality, assume that the insertion is to table R^j . Denote $R^{j'}$ as the table R^j excluding the last row to be added. Let m be the number of rows that R^j contains after the addition of the new row, which we denote as r_m ($m \leq n$). This implies that $R^{j'}$ has m - 1 rows. We denotes these rows as r_i , where i is the order in which a row was added,

as determined by real time. By lemma 3:

$$R^{j'} \bowtie_{k_j} S^j = \bigcup_{i=1}^{m-1} \left(r_i \times \sigma_{\{k_j = \rho_{i_j}\}}(S^j) \right)$$

The addition of r_m will add to $R^{j'} \bowtie_{k_j} S^j$ the rows $r_m \times \sigma_{\{k_j = \rho_{m_j}\}}(S^j)$. The combined set is:

$$\{\bigcup_{i=1}^{m-1} \left(r_i \times \sigma_{\{k_j = \rho_{i_j}\}}(S^j) \right) \} \cup \{r_m \times \sigma_{\{k_j = \rho_{m_j}\}}(S^j) \} = \bigcup_{i=1}^m \left(r_i \times \sigma_{\{k_j = \rho_{i_j}\}}(S^j) \right) = R^j \bowtie_{k_j} S^j$$

where the last equation is another application of lemma 3.

The combination of propositions 8 and 6 yields the correctness of Lynx's join operations, in terms of end-state convergence. Proposition 8 guarantees that Lynx's materialized join table construction correctly reflects $R^j \bowtie_{k_j} S^j$, for every j. Proposition 6 guarantees that eventually $T^j = T^0$ for every table T and all j. Hence, the materialized join table reflects $R \bowtie_{k_j} S$, which is the property we require,

We have proved that once all update operations terminate, derived tables reflect the same data stored in main tables. This does not specify what is the intermediate state that is visible to queries issued to secondary tables while there are still update operations being processed. In fact, we can specify a somewhat stronger property regarding the state of derived tables (secondary and joins) than just eventual convergence. Recall that in Lynx, main tables represent the authoritative state of the database and derived tables expose a read-only interface to users. Replies to the user queries on secondary tables capture a state that we may call *single-query historical snapshot*. That is, every single secondary index query or join query always returns a snapshot of the state of the main tables. This may be a recent snapshot or one that captures an historical snapshot of main tables. Note that this does not imply that the union of secondary tables' partitions always reflects an historical state of the union of main tables partitions. Formally, $\bigcup_{p=1}^{P_j} R_p^j$ is not always an historical state of the

set $\bigcup_{p=1}^{P_0} R_p^0$. The reason for this anomaly is that updates of a secondary key on a main table are translated to a removal of a row and an insertion of a new row in two different partitions. However, since Lynx queries are always answered by a single partition, this anomaly cannot be observed via a single query.

Chapter 9

Evaluation of Lynx

In this chapter we present an evaluation of Lynx. As before, we first describe our experience with developing applications for Lynx, and then proceed to give an experimental evaluation of Lynx's base performance and the performance of its applications.

9.1 Usability: Applications on top of Lynx

We have implemented three applications using Lynx: an auction service (L-RUBiS), a microblogging service (L-Twitter), and a social network website (L-Social). All three applications work correctly under the consistency guarantees of Lynx and are easy to build and optimize.

Auction service. L-RUBIS is a port of the auction website in the RUBIS benchmark [1, 22]. The original RUBIS implementation is based on PHP using a local MySQL database. We ported RUBIS' schema for Lynx and re-wrote its PHP functions in Python to use Lynx's API for accessing the database.

L-RUBIS consists of ten partitioned tables (the example of Chapter 7 uses a subset of them.) We specify a total of 13 secondary indexes across tables and the maximum secondary indexes used per table is 3. We use one join table uniting the *User* and the *Comments* table which records users' comments. This join table allows L-RUBiS to quickly list those usernames who have commented on a given user. With the help of secondary and join tables, the majority of queries in L-RUBiS can be satisfied by querying one server machine.

Two user-defined DTCs in L-RUBiS are worth mentioning, one to process bidding requests (as seen in Chapter 7), and the other to handle user registration which must guarantee that users choose unique usernames. The original RUBiS imposes a uniqueness constraint on the username column in *Users*. Because Lynx does not guarantee the uniqueness of non-primary keys, L-RUBiS employs an additional table, called *Usernames*, containing all usernames that have ever been used, with username as its primary key. To register a user "alice", L-RUBiS uses a DTC to first check if "alice" already exists in *Usernames* and then add the user to the *Users* table. Because this DTC can abort only in the first hop, no compensating action is necessary. L-RUBiS waits for the entire chain to complete before returning from a user's registration request.

Microblogging. L-Twitter is a simplified clone of Twitter. The schema of L-Twitter is modeled after those used by Twitter [56] with three tables, *Users*, *Tweets*, and *Follow*. *Tweets* contains information about each tweet including its identifier, the creator, the 140-character content etc. *Follow* contains the uid of follower and followee for each link in the follower graph.

The real Twitter relies on manually partitioned MySQL with no support of secondary indexes and join tables and hence lacks a principled way to optimize queries. For example, to enumerate all tweets created by a given uid, the underlying query needs to contact every partition of the tweets table which it optimizes by stopping the query early after contacting a few recent partitions [56]. By contrast, L-Twitter performs the equivalent query efficiently by generating a secondary index on the *Tweets* table based on uid. Likewise, L-Twitter creates indexes based on both the follower and followee columns in *Follow* in order to query the follower graph efficiently.

A common operation in Twitter is to display a user's "timeline", the collection of tweets posted by those that the user follows. Twitter's original implementation on a single-node MySQL backend performs a join query between the *Follow* and *Tweets* [56]. In Twitter's current distributed implementation, joins are no longer supported and it resorts to maintaining the "timeline" of each user in memcached. L-Twitter sticks to the original implementation by creating a distributed join table *Tweets-Follow* based on the join key *Tweets.creator* = *Follow.followee*. By querying the secondary index *Follow.follower* on the *Tweets-Follow* table, L-Twitter can display a user's timeline by querying only one server.

Social networking. L-Social is a social networking application which implements many operations commonly found in a website like Facebook, such as be-friending users, updating one's status, posting to walls etc. L-Social consists of 6 tables, *Users, Friends, Friendrequests, Status, Wall* and *Events*, as well as two join tables, *Friends-Status* and *Friends-Events*.

Friendship is similar to the *Follow* table in L-Twitter except all friendship links are symmetric. To be-friend X, user Y must first inserts his request into *Friendrequests*. When X approves the request from Y, L-Social uses a DTC to delete Y's request from *Friendrequest*, add two links $(X \rightarrow Y, Y \rightarrow X)$ to *Friends* table, and add two events (one for X, one for Y) in the *Events* table announcing the newly formed friendship. By using a DTC instead of a distributed transaction, users browsing the friendship graph may observe asymmetric links temporarily. We think this is acceptable behavior for L-Social.

When user X posts a status message, L-Social uses a DTC to insert the new status into *Status* and to add an event "X has changed her status" in *Events*. Similarly, when user X posts on Y's wall, L-Social uses a DTC to insert the post into *Wall* and to add an event "X has posted on Y's wall" in *Events*.

9.2 Performance evaluation

We tested the performance of Lynx and the applications described above. The highlights are:

- Lynx scales well. The throughput increase for executing DTCs and updating derived tables is ~6× when scaling from 2 to 15 servers.
- Applications achieve scalable and low-latency operation. The maximum latency for L-Twitter and L-RUBiS is <60 ms and the median is <10 ms.
- Lynx enables L-Twitter to replicate its data in geographically distant sites while maintaining its low latency operation.

9.2.1 Experimental setup

We ran experiments on a local cluster of 18 machines with heterogeneous hardware configuration: 6 machines have two 16-core AMD Opteron 6272 processors with 32GB memory, 6 machines have two quad-core Intel Xeon E5520 processors with 8GB memory, and the remaining 6 machines have a single quad-core Intel Xeon X3360 processor with 4GB memory. All machines are equipped with either a 60GB OCZ Vertex-3 SSD or a 120GB Sandisk Extreme SSD. They are connected with each other via a commodity gigabit Ethernet switch.

On each server machine, we ran a single PostgreSQL instance (version 9.1) and a Lynx server. We configured PostgreSQL to use the local SSD for storage. The Lynx server also stores its log file on the SSD. In all experiments, we ran a large number of client processes to issue requests over the network to Lynx.

9.2.2 Microbenchmark

We evaluated Lynx using a set of microbenchmarks to help understand its baseline performance and scaling behavior.



Figure 9.1: DTC execution throughput for several workloads.



Figure 9.2: Distribution of DTC completion latency.

DTC throughput. Figure 9.1 shows Lynx's attainable throughput measured as DTCs per second, as the number of Lynx servers increases. We evaluated three types of DTCs. In the *simple* experiments, clients issued user-defined DTCs whose actions insert rows into main tables that have no derived tables. We controlled the length of a user-defined DTC by varying the number of single-insert actions that comprise the simple chain. In the secondary key and join experiments, clients issue requests to insert a row into a table with a secondary index and an additional join table, respectively.

As seen in Figure 9.1, Lynx achieves good but not perfect scaling. This is in part due to the heterogeneous nature of our cluster configuration. Because Python's multithreading support does not take full advantage of multiple cores, the throughput achievable by a Lynx server is limited by its single-core performance. Therefore, we see much better scaling from 2 to 10 servers (than scaling from 10 to 15 servers) because the experiments with fewer servers use only Intel machines which have better per-core performance than our AMD machines.

The aggregate chain throughput is largely dependent on the number of hops in the chain. A chain of length m results in m times the number of actions processed by Lynx, thus should achieve 1/m the throughput of DTCs with length 1. This can be seen in the 10-server experiments, where for example, the throughput for simple DTCs of length 2 is 10,000 chains/sec, which is about half of that for DTCs of length 1 with similar actions. The DTCs for the join experiments have 2 hops (see top chain in Figure 8.3). The performance for this workload is less than that of the simple chain of length 2 as the second action in the join chain is more complicated and requires more processing by the server.

DTC latency. Figure 9.2 shows the CDF of the latency for completing a DTC in experiments involving 10 servers. As the figure shows, the completion time of a chain is largely a function of its length. The median latency for finishing the first hop is 8 ms. The time to complete the second and fourth hop is roughly double or quadruple that of the first hop respectively. The vast

majority of write operations in our applications can return to the user immediately after finishing the first action of the chain, thus the observed latency for those write operations can be low as the completion latency seen for DTCs of length 1 in Figure 9.2.

Read throughput. In our prototype implementation, clients directly contact a server's PostgreSQL instance for processing read requests. As a result, the aggregate read throughput of the system is determined by PostgreSQL, which achieves 20,000 queries per second for a single server and 58,000 QPS for three servers (we did not have enough physical machines to run enough client processes to saturate the performance of more than three servers).

In contrast to the read-only workload, our Python-based server implementation cannot saturate write performance of PostgreSQL. PostgreSQL can service $\sim 12,000$ local insertions per second, while a single Lynx server processes 4000 chain actions per second due to both Python's high CPU overhead and its poor handling of multiple threads. Since most websites handle far more reads than writes (a typical ratio is 8:1 [2]), we think incurring Python's overhead on the write path is acceptable.

9.2.3 Application performance

Lynx makes it easy to develop scalable web applications. Our example implementations of L-Twitter and L-RUBiS both achieved good performance using Lynx.

L-Twitter. We populated the database with 100000 users, each with 10 existing tweets and 10 random followers. We evaluated the performance of three common L-Twitter operations: view-timeline for displaying the timeline of tweets that a user follows, follow for adding a follow relationship, and tweet for posting a tweet. We also evaluated a mixed workload consisting of 90% view-timeline operations, 5% follow and 5% tweet operations.

Figure 9.3 shows the throughput of the above workloads for a varying number of servers. The follow operation inserts a row into the *Follow* table which causes updates to two secondary tables,



Figure 9.3: L-Twitter throughput for different workloads.



Figure 9.4: L-Twitter operation latency.



Figure 9.5: L-RUBiS throughput for different workloads.

the join table *Tweet-Follow*, as well as the secondary table of *Tweet-Follow*. The resulting DTC has 4 hops. The *Tweets* table has one secondary index. The underlying DTC for tweet consists of a modification to the main table, the secondary and join table (in one step), and an additional step (executed in parallel) to update the secondary table of *Tweet-Follow*. The throughput for tweet and follow scales with the number of servers, achieving 7000 and 5000 ops/sec respectively in experiments involving 15 servers. The cost of updating join and secondary tables for tweet and follow pays off by allowing for fast view-timeline operations that require only a simple lookup in the secondary table of *Tweet-Follow*. As a result, the mixed workload can achieve much higher throughput (90% view-timeline), up to 35,000 ops/sec for 15 servers.

Lynx helps L-Twitter achieve very low operation latency. The read operation view-timeline contacts only one server. The write operations tweet and follow return as soon as the underlying DTC finishes executing its first hop. Figure 9.4 shows the operation latency distribution in exper-



Figure 9.6: L-RUBiS operation latency.

iments with 15 servers. All operations finish in less than 60 ms with the median latency less than 10 ms.

L-RUBIS. We evaluated the five most common RUBIS operations, three of which are writes (additem, add-comment, and submit-bid) and the remaining two are reads (view-item and viewcomments).

Figure 9.5 shows the throughput of different operations as well as a mixed workload with 90% reads and 10% writes. add-item inserts a row into the *Items* table, resulting in a DTC of length 2. add-comment inserts a row into the *Comments* table, resulting in a DTC of length 5. The submitbid operation involves a user-defined DTC of two actions each modifying a table with secondary tables, resulting in a total chain length of 5. All three operations scale as the number of servers increases, reaching 7500, 6000, and 4800 ops/sec for add-item, add-comment, and submit-bid in 15-server experiments. Compared to writes, the read operations are inexpensive; view-item performs one lookup in the *Items* table and view-comments looks up using the secondary key of the join table *Comments-Users*. Thus, L-RUBiS achieves high throughput in the mixed workload with 37,000 ops/sec for 15 servers.

All five common operations of L-RUBiS can return to the user after executing at the first server.


Figure 9.7: The aggregate throughput of L-Twitter running on one and two geographically separate sites. Each site consists of 4 Lynx servers.

Therefore, as shown in Figure 9.6, the operation latency is very low with the maximum latency being less than 60 ms and the median less than 10 ms, similar to that of L-Twitter.

9.2.4 Geo-replication performance

We also evaluated L-Twitter in a geo-replicated setting. To prepare L-Twitter for geo-replication, we assigned each user with a home site where all logins of the user are re-directed to. In addition, we specify that entries in the *Tweets* and *Follow* are co-located with the creating user. As a result of this organization, the tweet operation inserts to a *Tweet* table partition located at the user's home site and can return to the user without blocking for remote replication. Similarly, follow can also return quickly. Once replication to remote sites completes, the underlying DTC for tweet or follow proceeds to update the secondary table of *Tweet* or *Follow*, the join table *Tweet-Follow* as well as the join table's secondary index. Once the secondary table of *Tweet-Follow* has been updated, the follower's view-timeline operation will see the new tweet.

We evaluated the latency of L-Twitter by simulating two geographically distant sites on our local cluster. Specifically, we artificially imposed a delay of 100 ms between two groups of servers,

each with 4 machines. As expected, all operations return with low latency (with a median of 10 ms) because they do not need to wait for the remote site. It takes \sim 400 ms for a tweet to show up in followees' timelines, as the underlying DTC requires three roundtrip times between the two sites to replicate data for each of its three updates. Figure 9.7 shows the aggregate throughput of different operations. Since all tables are replicated twice, scaling from one to two sites does not yield better throughput for write operations. By contrast, since each view-time reads from a local replica of the join table, its throughput roughly doubles with two sites.

Chapter 10

Related Work

10.1 Walter

Transactions in data centers. Early transactional storage for data centers include Bigtable [32], Sinfonia [19], Percolator [71], and distributed B-trees [18]. Unlike Walter, these systems were designed for a single data center only.

Storage systems that span many data centers often do not provide transactions (e.g., Dynamo [39]), or support only restricted transactional semantics. For example, PNUTS [35] supports only one-record transactions. COPS [62] provides only read-only transactions. Megastore [24] partitions data and provides the ACID properties within a partition but, unlike Walter, it fails to provide full transactional semantics for reads across partitions.

Transactions in disconnected or wide-area systems. Perdis [45] is an object store with a checkout/check-in model for wide-area operations: it creates a local copy of remote data (check-out) and later reconciles local changes (check-in), relying on manual repair when necessary. For systems with mobile nodes, tentative update transactions [49] can commit at a disconnected node. Tentative commits may be aborted later due to conflicts when the hosts re-connect to servers, which requires reconciliation by an external user. In contrast to the above systems, Walter does not require burdensome operations for manual repair or reconciliation. Mariposa [83] is a wide-area system whose main focus is on incentivizing a site to run third-party *read-only queries*.

Database replication. There is much work on database replication, both commercially and academically. Commercial database systems support master-slave replication across sites: one site is the primary, the others are mirrors that are often read-only and updated asynchronously. When asynchronous mirrors are writable, applications must provide logic to resolve conflicts. On the academic side, the database replication literature is extensive; here we summarize relevant recent work. Replication schemes are classified on two axes [49]: (1) who initiates updates (primarycopy vs update-anywhere), and (2) when updates propagate (eager vs lazy). With *primary-copy*, objects have a master host and only the master initiates updates; with update-anywhere, any host may initiate updates. With *eager replication*, updates propagate to the replicas before commit; with lazy replication, replicas receive updates asynchronously after commit. All four combinations of these two dimensions are possible. Eager replication is implemented using distributed two-phase commit [27]. Later work considers primary-copy lazy replication and provides serializability by restricting the placement of each object's primary [33], or controlling when secondary nodes are updated [29, 69]. Update-anywhere lazy replication is problematic because conflicting transactions can commit concurrently at different replicas. Thus, recent work considers hybrids between eager and lazy replication: updates propagate after commit (lazy), but replicas also coordinate during transaction execution or commit to deal with conflicts (eager). This coordination may involve a global graph to control conflicts [23, 31], or atomic broadcast to order transactions [57, 70]. Later work considers snapshot isolation as a more efficient alternative to serializability [36, 42, 43, 61, 72, 92]. Walter differs from the above works because they ensure a stronger isolation property-serializability or snapshot isolation-which inherently requires coordination across sites to commit, whereas Walter commits common transactions without such coordination.

Federated transaction management considers techniques to execute transactions that span multiple database systems [76]. This work differs from Walter because it does not consider issues involving multiple sites and its main concern is to minimize changes to database systems, rather than avoiding coordination across sites.

Relaxed consistency. Some systems provide weaker consistency, where concurrent updates cause diverging versions that must be reconciled later by application-specific mechanisms [39, 67, 86]. *Eventual consistency* permits replicas to diverge but, if updates stop, replicas eventually converge again. Weak consistency may be tolerable [88], but it can lead to complex application logic. Inconsistency can also be quantified and bounded [20, 53, 94], to improve the user experience. Fork consistency [65] allows the observed operation history to fork and not converge again; it is intended for honest clients to detect the misbehavior of malicious servers rather than to provide efficient replication across sites.

Commutative data types. Prior work has shown how to exploit the semantics of data types to improve concurrency. In [89], abstract data types (such as sets, FIFO queues, and a bank account) are characterized using a table of commutativity relations where two operations conflict when they do not commute. In [46, 77], a lock compatibility table is used to serialize access to abstract data types, such as directory, set or FIFO queue, by exploiting the commutativity of their operations. Because these works aim to achieve serializability, not all operations on a set object are conflict-free (e.g., testing the membership of element a conflicts with the insertion of a in the set). As a result, operating on sets require coordination to check for potential conflicts. In contrast, since we aim to achieve the weaker PSI property, operations on Walter's cset objects are always free of conflicts, allowing each data center to read and modify these csets without any remote coordination.

Letia et al. [60] have proposed the use of commutative replicated data types to avoid concurrency control and conflict resolution in replicated systems. Their work has inspired our use of csets. Subsequent recent work [78] provides a theoretical treatment for such data types and others—which are together called conflict-free replicated data types or CRDTs—proposing sufficient conditions for replica convergence under a newly-defined strong eventual consistency model. While that work concerns replication of single operations/objects at a time, not transactions, one could imagine using general CRDTs with PSI and our protocols to replicate transactions efficiently. U-sets [78, 93] are a type of set in which commutativity is achieved by preventing a removed element from being added again. In contrast, csets achieve commutativity by augmenting elements with counts. Csets are similar to Z-relations [52], which are mappings from tuples to integers, used to allow for decidability of equivalence of queries in the context of query optimization.

Escrow transactions [68] update numeric data, such as account balances, by holding some amount in escrow to allow concurrent commutative updates. By exploiting commutativity, such transactions resemble transactions with csets, but they differ in two ways. First, escrow transactions operate on numeric data. Second, escrow transactions must coordinate among themselves to check the amounts in escrow, which does not serve our goal of avoiding coordination across distant sites.

10.2 Lynx

Distributed database. Pioneering work in distributed database, such as Gamma [41], Bubba [28], R* [66], Teradata and Tandem [40], aim to the provide the same transactional update and query interface that were well established in centralized database systems. Specifically, they support distributed transactions through two phase commit [28, 41, 66] and employ a distributed dataflow graph to execute a complex relational query across many machines [28, 40, 41].

Most modern day distributed databases position themselves as either data warehouses (an alternative name is big data analytics engine) or OLTP (online transaction processing) systems [82]. Greenplum [6], Vertica [14], Aster Data [5] and ParAccel [9] fall in the first category and they optimize for executing batched complex analytics queries, using techniques similar to those in MapReduce [38], Dryad [54] or DryadLINQ [95]. LazyBase [34] optimizes data analytics by batching writes and updating materialized secondary indexes in epochs. H-store [55, 84] and VoltDB [15] (a commercial variant of H-Store) are designed for OLTP workload. H-store is a main-memory distributed database supporting distributed transactions via two-phase commits [55]. Lynx targets at web applications which share the same low latency requirement as OLTP workload. Unlike traditional OLTP systems, Lynx focuses on helping programmers optimize online read queries through the use of denormalization, secondary keys, and materialized joins. Most web applications do not need ACID guarantees to work correctly, thereby providing opportunity for Lynx to efficiently update derived tables via DTCs.

Distributed NoSQL stores. Many recent systems achieve scalable performance by providing a much simpler data model and query interface than relational databases, e.g. BigTable [32], H-Base [7], MongoDB [8], Megastore [24], Dynamo [39], DynamoDB [21], PNUTS [35], Redis [10], Cassandra [4], and COPS [62]. These systems have limited support for online queries. None of them supports materialized joins or queries using pre-computed secondary indexes. For some [8, 11], querying via a secondary key often involves contacting every server in the system. The recent HyperDex store [44] offers efficient range queries over multiple attributes. By contrast, Lynx does not support range queries across multiple partitions. Only a few NoSQL systems offer more than local transactions: Sinfonia [19] provides mini-transactions and Percolator [71] provides distributed transactions with snapshot isolation. Neither systems are targeted at low latency web applications.

Workflow Management [90]. The DTC primitive bears resemblance to workflow systems for managing application workflows like travel planning or insurance claim processing. An application workflow naturally consists of multiple activities, each of which executes as a transaction. Like DTC, workflow systems guarantee that the series of activities are executed completely and exactly once. Although DTC shares some similarity with workflow systems, its usage is quite different. While workflow systems are meant to manage sophisticated application workflows, Lynx uses

DTC to decompose a single logical operation for writing to a table and updating its derived tables and denormalized data. Consequently, DTCs must also provide the ordering constraint in addition to eventual execution in order to guarantee consistency.

Alternatives to ACID. The database community has done much work to explore weaker notions of correctness than global serializability in distributed multidatabase systems [30]. The most related piece of work is [46] by Garcia-Molina. He proposes dividing operations into a series of smaller steps and uses compensating transactions to undo the effects of committed earlier steps. The goal of his work is to allow users to exploit their semantic knowledge in an organized fashion to enable more concurrency in the system. DTC is inspired by Molina's work and pushes his idea further: apart from compensating actions, DTC also provides the ordering constraint so that non-commutative actions from different chains interleave correctly in the face of concurrency, an important property for updating derived tables correctly. The idea of compensating transactions also appears in [37, 51] as Spheres of Control (SoC) where problematic computations can be automatically invalidated.

Chapter 11

Conclusion

This dissertation presented two storage systems that were specifically designed and built to address the requirements of large-scale web applications. Both systems, Walter and Lynx are georeplicated and employ a relaxed, yet relatively strong consistency model. The chosen consistency guarantees of both systems allow the implementations to achieve high performance at large scale, and make the development of web application on top the systems simple and rapid.

Walter is a transactional geo-replicated key-value store. A key feature behind Walter is Parallel Snapshot Isolation (PSI), a precisely-stated isolation property that permits asynchronous replication across sites without the need for conflict resolution. Walter relies on techniques to avoid conflicts across sites, thereby allowing transactions to commit locally in a site. PSI thus permits an efficient implementation, while also providing strong guarantees to applications. We have demonstrated the usefulness of Walter by building a Facebook-like social networking application and porting a third-party Twitter clone. Both applications were simple to implement and achieved reasonable performance.

Lynx targets intra-site scalability, in addition to geo-replication. It is a distributed database for building scalable web applications. Lynx supports distributed secondary indexes and materialized joins which help programmers optimize queries for low latency operation. Lynx maintains its derived tables using DTC, which executes a series of transactions at different nodes while guaranteeing fault-tolerance and correct interleaving. Lynx also exposes the DTC primitive to programmers for maintaining other types of denormalized data in the application. Lynx provides the consistency guarantee that denormalized data and derived tables are eventually consistent with each other. We have demonstrated the usefulness of Lynx by building an auction service, a microblogging and a social networking website.

Bibliography

- [1] http://rubis.ow2.org/index.html as of Oct 2010.
- [2] Private data on database access patterns from a popular dating website.
- [3] Amazon simple queue service. http://aws.amazon.com/sqs.
- [4] Apache cassandra database. http://cassandra.apache.org/.
- [5] Aster data. http://www.asterdata.com.
- [6] Greenplum. http://www.greenplum.com.
- [7] Hbase: Hadoop database. http://hbase.apache.org.
- [8] MongoDB. http://www.mongodb.com.
- [9] Paraccel analytic database. http://www.paraccel.com.
- [10] Redis: an open-source advanced key-value store. http://redis.io.
- [11] Riak (open source distributed database). http://wiki.basho.com/Riak.html.
- [12] Summary of the amazon ec2 and amazon rds service disruption in the us east region. http://aws.amazon.com/message/65648.
- [13] A Twitter clone for the Redis key-value database. http://retwis.antirez.com.
- [14] Vertica systems. http://www.vertica.com.
- [15] VoltDB, 2011. http://voltdb.com.
- [16] Twitter turns six, 2012. http://blog.twitter.com/2012/03/twitter-turns-six.html.
- [17] Daniel Abadi. Replication and the latency-consistency tradeoff. DBMusings. http://dbmsmusings.blogspot.com/.

- [18] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed B-tree. In *International Conference on Very Large Data Bases*, pages 598–609, August 2008.
- [19] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. ACM Transactions on Computer Systems, 27(3):5:1–5:48, November 2009.
- [20] Rafael Alonso, Daniel Barbará, and Hector Garcia-Molina. Data caching issues in an information retrieval system. ACM Transactions on Database Systems, 15(3):359–384, September 1990.
- [21] Amazon. DynamoDB: a fully managed NoSQL database service. http://aws.amazon.com/dynamodb/.
- [22] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *Workshop on Workload Characterization*, 2002.
- [23] Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool. Replication, consistency, and practicality: are these mutually exclusive? In *International Conference on Management* of Data, pages 484–495, June 1998.
- [24] Jason Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In 5th Conference on Innovative Data Systems Research, pages 223–234, January 2011.
- [25] Hal Berenson et al. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data*, pages 1–10, May 1995.
- [26] Philip Bernstein and Nathan Goodman. Concurrency control in distributed database systems. ACM Computing Survey, 13(2):185–221, 1981.
- [27] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

- [28] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly available parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2, 1990.
- [29] Yuri Breitbart et al. Update propagation protocols for replicated databases. In *International Conference on Management of Data*, pages 97–108, June 1999.
- [30] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of multidatabase transaction management. *The International Journal on Very Large Data Bases*, 1992.
- [31] Yuri Breitbart and Henry F. Korth. Replication and consistency in a distributed environment. *Journal of Computer and System Sciences*, 59(1):29–69, August 1999.
- [32] Fay Chang et al. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation*, pages 205–218, November 2006.
- [33] Parvathi Chundi, Daniel J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *International Conference on Data Engineering*, pages 469–476, February 1996.
- [34] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A. N. Soules, and Alistair Veitch. LazyBase: Trading freshness for performance in a scalable database. In *EuroSys*, 2012.
- [35] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *International Conference on Very Large Data Bases*, pages 1277–1288, August 2008.
- [36] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation.In *International Conference on Very Large Data Bases*, pages 715–726, September 2006.
- [37] S.H. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2), 1978.
- [38] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, 2004.

- [39] Giuseppe DeCandia et al. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, October 2007.
- [40] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 36(6), 1992.
- [41] David Dewitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui i Hsiao, and Rick Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2:44–62, 1990.
- [42] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *European Conference on Computer Systems*, pages 117–130, April 2006.
- [43] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *European Conference on Computer Systems*, pages 399–412, March 2007.
- [44] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store for cloud computing. Technical report, CS Dept, Cornell University, 2011.
- [45] Paulo Ferreira et al. Perdis: design, implementation, and use of a persistent distributed store. In *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*, chapter 18.
 Springer-Verlag, February 2000.
- [46] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.
- [47] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In ACM Symposium on Operating Systems Principles, pages 29–43, October 2003.
- [48] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services. ACM SIGACT News, 33(2):51–59, June 2002.
- [49] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *International Conference on Management of Data*, pages 173–182, June 1996.

- [50] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.
- [51] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 1993.
- [52] Todd J. Green, Zachary G. Ives, and Val Tannen. Reconcilable differences. In *International Conference on Digital Telecommunications*, pages 212–224, March 2009.
- [53] Hongfei Guo et al. Relaxed currency and consistency: How to say "good enough" in SQL.In *International Conference on Management of Data*, pages 815–826, June 2004.
- [54] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference* on Computer Systems (EuroSys), 2007.
- [55] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.
- [56] Nick Kallen. Big data in real time at twitter. QCon San Francisco 2010:The annual international software development conference. http://www.slideshare.net/nkallen/q-con-3770885.
- [57] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, September 2000.
- [58] Mike Krieger. Scaling instagram, 2012. http://www.scribd.com/doc/89025069.
- [59] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In International Conference on Architectural Support for Programming Languages and Operating Systems, pages 84–92, October 1996.
- [60] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. In *International Workshop on Large Scale Distributed Systems and Middleware*, October 2009.

- [61] Yi Lin, Bettina Kemme, Marta Pati no Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *International Conference on Management of Data*, pages 419–430, June 2005.
- [62] Wyatt Lloyd, Michael Freedman, Michael Kaminsky, and David Andersen. Don't settle for eventual: Stronger consistency for wide-area storage with cops. In ACM Symposium on Operating Systems Principles, October 2011.
- [63] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
- [64] Ryan Mack. Building timeline: Scaling up to hold your life story. Facebook Engineering Notes. http://www.facebook.com/note.php?note_id=10150468255628920.
- [65] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage.In ACM Symposium on Principles of Distributed Computing, pages 108–117, July 2002.
- [66] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4), 1986.
- [67] Lily B. Mummert, Maria R. Eblig, and Mahadev Satyanarayanan. Exploiting weak connectivity for mobile file access. In ACM Symposium on Operating Systems Principles, pages 143–155, December 1995.
- [68] Patrick E. O'Neil. The escrow transactional method. ACM Transactions on Database Systems, 11(4):405–430, December 1986.
- [69] Esther Pacitti, Pascale Minet, and Eric Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *International Conference on Very Large Data Bases*, pages 126–137, September 1999.
- [70] Marta Patino-Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso.
 MIDDLE-R: Consistent database replication at the middleware level. ACM Transactions on Computer Systems, 23(4):375–423, November 2005.
- [71] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Symposium on Operating Systems Design and*

Implementation, pages 251–264, October 2010.

- [72] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *International Middleware Conference*, pages 155–174, October 2004.
- [73] Dan Ports, Austin Clements, Irene Zhang, Samuel Madden, and Barbara Liskov.
 Transactional consistency and automatic management in an application data cache. In USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2010.
- [74] Dan Pritchett. Base: An acid alternative. In ACM Queue, 2008.
- [75] Yasushi Saito et al. FAB: building distributed enterprise disk arrays from commodity components. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58, October 2004.
- [76] Ralf Schenkel et al. Federated transaction management with snapshot isolation. In Workshop on Foundations of Models and Languages for Data and Objects, Transactions and Database Dynamics, pages 1–25, September 1999.
- [77] Peter Schwarz and Alfred Spector. Synchronizing shared abstract types. ACM Transactions on Computer Systems, 2(3):223–250, August 1984.
- [78] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. October 2011.
- [79] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. Zeno: Eventually consistent byzantine fault tolerance. In *Symposium on Networked Systems Design and Implementation*, pages 169–184, April 2009.
- [80] Jason Sobel. Scaling out. http://www.facebook.com/note.php?note_id=23844338919.
- [81] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.
- [82] M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. In *Proceedings of the 2005 International Conference on Data Engineering*, 2005.
- [83] Michael Stonebraker et al. Mariposa: a wide-area distributed database system. In

International Conference on Very Large Data Bases, pages 48–63, January 1996.

- [84] Michael Stonebraker, Samuel R. Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.
- [85] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, Frans Kaashoek, and Robert Morris. Simplifying wide-area application development with WheelFS. In Symposium on Networked Systems Design and Implementation, pages 43–58, April 2009.
- [86] Douglas B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In ACM Symposium on Operating Systems Principles, pages 172–183, December 1995.
- [87] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In ACM Symposium on Operating Systems Principles, pages 224–237, October 1997.
- [88] W. Vogels. Data access patterns in the amazon.com technology platform. In *International Conference on Very Large Data Bases*, page 1, September 2007.
- [89] William Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.
- [90] Gerhard Weikum and Gottfried Vossen. Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
- [91] Gerhard Weikum and Gottfried Vossen. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann, 2009.
- [92] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *International Conference on Data Engineering*, pages 422–433, April 2005.

- [93] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionart problems. In ACM Symposium on Principles of Distributed Computing, pages 233–242, August 1984.
- [94] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, August 2002.
- [95] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI)*, 2008.