Coordination Avoidance in Distributed Databases

Peter Bailis

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-206
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-206.html

October 30, 2015

Copyright © 2015, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Coordination Avoidance in Distributed Databases

By

Peter David Bailis

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph M. Hellerstein, Co-Chair
Professor Ion Stoica, Co-Chair
Professor Ali Ghodsi
Professor Tapan Parikh

Fall 2015

Coordination Avoidance in Distributed Databases

Copyright 2015
by
Peter David Bailis

Abstract

Coordination Avoidance in Distributed Databases

by

Peter David Bailis

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Co-Chair
Professor Ion Stoica, Co-Chair

The rise of Internet-scale geo-replicated services has led to upheaval in the design of modern data management systems. Given the availability, latency, and throughput penalties associated with classic mechanisms such as serializable transactions, a broad class of systems (e.g., "NoSQL") has sought weaker alternatives that reduce the use of expensive coordination during system operation, often at the cost of application integrity. When can we safely forego the cost of this expensive coordination, and when must we pay the price?

In this thesis, we investigate the potential for coordination avoidance—the use of as little coordination as possible while ensuring application integrity—in several modern data-intensive domains. We demonstrate how to leverage the semantic requirements of applications in data serving, transaction processing, and web services to enable more efficient distributed algorithms and system designs. The resulting prototype systems demonstrate regular order-of-magnitude speedups compared to their traditional, coordinated counterparts on a variety of tasks, including referential integrity and index maintenance, transaction execution under common isolation models, and database constraint enforcement. A range of open source applications and systems exhibit similar results.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

I would like to acknowledge my advisors: Ali Ghodsi, Joe Hellerstein, and Ion Stoica. Ali has been a thoroughly conscientious and tenacious collaborator and mentor. Ali pushed our work to greater levels of technical depth, introducing me to the tradition of distributed computing and encouraging both precision and pursuit of principle. His unceasing support and thoughtful advice bolstered both the quality of this research as well as my spirit and my faith in the research process. Joe encouraged me to find my own balance between what Melville calls "audacity and reverence." I am especially grateful for Joe's support and patience in allowing me freedom during my graduate experience as well as Joe's careful commentary on this document. Ion has proven an example of industriousness and perseverance. His persistent encouragement to "find the nugget" deepened my appreciation for simplicity in design as well as clarity of thought and writing.

I am also grateful to several individuals who made major contributions to the research in this thesis. Alan Fekete, who visited us twice on sabbatical, was essential in our focus on database safety properties, from isolation guarantees to constraints. Alan is an exemplar of modesty despite technical brilliance. I especially admire his desire to keep learning well into his career as well as his uncanny ability to comb through formalism with diligence and grace. Mike Franklin was a dear collaborator at the beginning and end of this work, and his leadership has been an inspiring example for me. Tapan Parikh, my outside dissertation committee member, provided a valuable, human-centric perspective on this work.

I have been fortunate to overlap with a multitude of wonderful researchers during my time at Berkeley. In particular, Neil Conway and Peter Alvaro were both great friends and colleagues, and our shared enthusiasm for distributed databases, cooking, and nature was a great source of inspiration for me. I especially admire Peter's quiet sense of wonder and reverence for the literature as well as Neil's admiration for and cultivation of engineering and design as high craft. Shivaram Venkataraman was an unflappably positive collaborator in our earliest days of graduate studies and became a terrific labmate and travel companion, from Santa Cruz to Turkey and Chania. Patrick Wendell was an invaluable sounding board regarding the interfaces between academia and industry as well as an unforgettable roommate. I could always count on a fun conversation (or dinner outing) with Kay Ouster-

hout, Aurojit Panda, Colin Scott, and the NetSys lab. Justine Sherry was instrumental in cultivating a positive culture within our cohort and was a superbly capable co-founder of @TinyToCS. Evan Sparks, Dan Haas, Daniel Crankshaw, Sanjay Krishnan, Jiannan Wang, and the many other members of the Berkeley Database Seminar were always game for a good conversation. The many members of the Berkeley Database group, Cloud seminar, CCN group, AMPLab, and BOOM project made for wonderful, joyful company.

Many others provided helpful feedback during the course of this work, including: Shel Finkelstein and Pat Helland, my guides to real-world inconsistency (and "outconsistency") in enterprise databases; Daniel Abadi, who never fails to elicit a fascinating conversation; Phil Bernstein, whose feedback is consistently top-notch and provides astute historical context; Doug Terry, whose pioneering work on weak replication and whose kind encouragement were both significant inspirations to me; Mike Stonebraker, whose unflagging energy is contagious, and who provides frequent and welcome reminders to consider the market for and impact of my research; Eric Brewer, Sam Madden, and Scott Shenker, who have provided gracious support and thoughtful guidance; and Divy Agrawal, Amr El Abaddi, and Sharad Mehotra, for rousing conversations about semantics-based concurrency control.

I am also grateful to the many people building, operating, and managing distributed systems and databases in the field who provided feedback on and inspiration for this work, including Michael R. Bernstein, Rick Branson, Mark Callaghan, Adrian Colyer, Sean Cribbs, Jonathan Ellis, Alex Feinberg, Andy Gross, Coda Hale, Colin Jones, Evan Jones, Kyle Kingsbury, Adam Marcus, Caitie McCaffrey, Christopher Meiklejohn, Mike Miller, Jeremiah Peschka, Mark Phillips, Henry Robinson, Mehul Shah, Xavier Shay, Justin Sheehy, Ines Sombra, Kelly Sommers, and Sriram Srinivasan. It is an exciting and perhaps unparalleled time in the history of data management, and this dialogue with practice has greatly enriched this work as well as my experience and enthusiasm during my studies.

While my graduate studies have been brief, I feel that my training in research began long before I arrived at Berkeley. I am especially indebted to Vijay Janapa Reddi, Margo Seltzer, David Brooks, Radhika Nagpal, Matt Welsh, Justin Werfel, and Christopher Goodrich for taking the chance to work with me, teaching me the process and joy of research at an early stage of my career, and encouraging the development of intellectual audacity and perseverance. Over time, I have come to realize how rare such experiences are and how privileged I am to have had such a group of individuals in my life to enable them.

This work was supported in part by a National Science Foundation Graduate Fellowship under Grant DGE 1106400 and by a Berkeley Fellowship for Graduate Study from the UC Berkeley Graduate Division.

Finally, I am deeply grateful for support and encouragement from my family and friends.

Chapter 1

Introduction

This thesis examines the design of robust and efficient distributed database systems.

Over the past decade, the challenges of distributed systems design have become increasingly mainstream. The rise of new application domains such as Internet services and the continued decrease of storage and computing costs have led to massive increases in request and data volumes. To address these trends, application developers have frequently turned to distributed systems designs. Scale-out application programming frameworks, data serving systems, and data processing platforms enjoy unprecedented popularity today [31, 38, 179]. Coupled with the introduction of inexpensive and elastic cloud computing [24], these factors have led distribution and replication to become common features in modern application and system designs. As a result, a growing class of software must address the difficulties of robust operation over computer networks, which include communication delays, partial failures, and inherent uncertainty about global system state [97].

In many applications, the difficulties of distributed systems design are relegated to a database tier. Application development best practices delegate the management of application state to a database back-end: application programmers implement application logic, while a back-end data infrastructure system handles data storage, query processing, and concurrency control [215]. This separation of concerns means database systems frequently act as the keystone of reliable distributed applications (Figure 1.1). Thus, database systems must directly address the challenges of distribution, replication, and fault tolerance—while providing a user-friendly interface for application developers and end users.

In this work, we investigate a conceptually simple class of designs for robust distributed databases: we study the design of databases that allow applications to make non-trivial progress independent of network behavior. That is, we study the design of database systems that provide *coordination-free* execution: whenever database clients can access at least

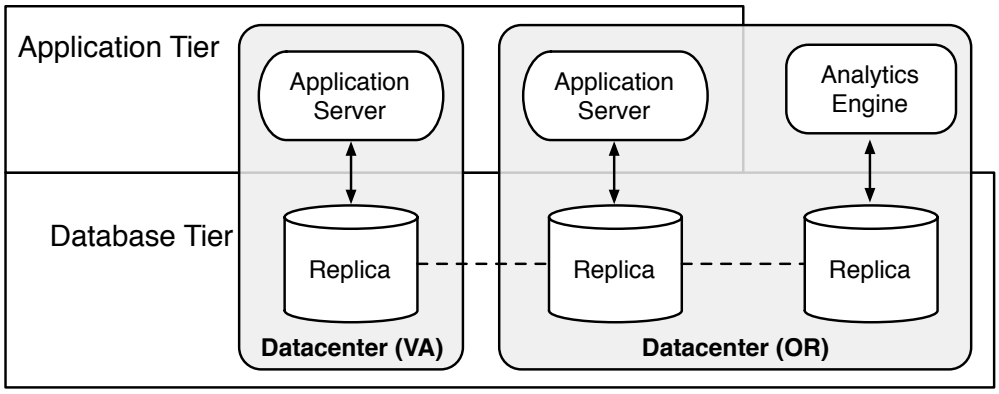


Figure 1.1: An illustration of a distributed, replicated database and its relation to application servers and end users. In modern distributed databases, data is stored on several servers that may be located in geographically distant regions (e.g., Virginia and Oregon, or even different continents) and may be accessed by multiple database clients (e.g., application servers, analytics frameworks, database administrators) simultaneously. The key challenge that we investigate in this thesis is how to minimize the amount of synchronous communication across databases while providing "always on," scalable, and high performance access to each replica.

one copy of database state, they are guaranteed to make non-trivial progress. This means that, in the presence of communication failures between servers, each client's operations may still proceed, providing "always on" functionality, or guaranteed *availability*. Even when database servers are able to communicate with one another, communication is not required. This allows *low latency* operation: each client's requests can be processed using locally accessible resources. Finally, coordination-free execution ensures *scalability*: as more servers are added, the database can make use of them without disrupting existing servers. This results in increased capacity. Thus, coordination-free execution offers attractive performance and availability benefits while ensuring that, subject to the availability of additional resources, additional requests can be serviced on demand.

This coordination-free database system design represents a departure from classic database system designs. Traditionally, database systems exposed a *serializable transaction* interface: if user operations are bundled into transactions, or groups of multiple operations over multiple data items, a database providing *serializability* guarantees that the result of executing the transactions is equivalent to some serial execution of the transactions [53]. Serializability is remarkably convenient for programmers, who do not need to reason about concurrency or distribution. However, serializability is inconvenient for databases: serializability (provably) requires coordination [92], negating the benefits of coordination-free execution. Intuitively, enforcing a serial ordering between transactions precludes the ability to guarantee the transactions' independent progress when executed on multiple servers.

As a result, increased application demands for availability, low latency, and scalability have led to a recent schism within mainstream database system designs [174, 179]. A proliferation of database systems, often called "NoSQL" systems, forego serializability (and other "strong" semantics) in order to provide coordination-free execution. However, in turn, these NoSQL systems provide few, if any semantic guarantees about their query results and about database state (also called *safety properties*; e.g., "no two users share a username") [38, 54, 156]. In contrast with serializability, which guarantees that safety properties preserved by individual transactions will be preserved under their serial composition, these NoSQL stores leave the enforcement of application safety to the application—a challenging and error-prone proposition [20, 132]. Thus, database users and application programmers today are left with one of two options: ensure safety via serializability and coordination, or forego safety within the database but enjoy the benefits of coordination-free execution.

In this thesis, we examine this apparent tension between ensuring application safety properties within the database and enjoying the scalability, availability, and performance benefits of coordination-free execution. Is enforcing safety properties always at odds with coordination-free execution? If we rely on serializability, the answer is yes. Instead, we consider an alternative: the enforcement of non-serializable semantic guarantees in a coordination-free manner. By examining the safety properties of today's database-backed applications, we determine whether coordination is strictly necessary to enforce them and explore implementations that limit the use of coordination. Thus, this thesis explores the relationship between correctness—according to useful safety guarantees—and coordination. More precisely, what is the coordination cost of a given safety guarantee? What is the minimum communication we must perform to enforce various correctness criteria?

1.1 Coordination Avoidance

Our primary goal in this thesis is to build database systems that coordinate only when strictly required in order to guarantee application safety. To realize this goal, we examine which semantic guarantees databases *can* provide under coordination-free execution—and explore implementations that fulfill this potential. We study a range of guarantees from both classic database systems as well as guarantees required by modern database-backed applications. Today, most of these guarantees are either implemented using coordination or are not provided by coordination-free systems. However, we demonstrate that many of these requirements can be correctly implemented without coordination and provide algorithms and system implementations for doing so. This enables what we call *coordination avoidance*: the use of as little coordination as possible while maintaining safety guarantees on behalf of database users.

Thesis Statement: *Many semantic requirements of database-backed applications can be efficiently enforced without coordination, thus improving scalability, latency, and availability.*

To achieve this goal, we first develop a new, general rule for determining whether a coordination-free implementation of a given safety property exists, called *invariant confluence*. Informally, invariant confluence determines whether the result of executing operations on independent copies of data can be combined (or "merged") into a single, coherent (i.e., *convergent*) copy of database state. Given a set of operations, a safety property that we wish to maintain over all copies of database state, and a merge function, invariant confluence tells us whether coordination-free execution is possible. Invariant confluence is both necessary and sufficient: if invariant confluence holds, a coordination-free, convergent implementation exists. If invariant confluence does not hold, no system can implement the semantics while also providing coordination-free, convergent execution.

Given this invariant confluence criterion, we examine a set of common semantic guarantees found in database systems and database-backed applications today to determine whether a coordination-free execution strategy for enforcing them exists. We perform several case studies, which we describe in detail below. In many cases, although existing implementations of these guarantees may rely on coordination, we show that coordination-free implementations of the semantics exist. This provides the scalability, performance, and availability benefits of coordination-free execution—but without compromising desirable semantic guarantees. None of these guarantees is serializable, but all of them correspond to existing or emerging demands from applications today.

Our use of invariant confluence recognizes latent potential for coordination-free execution in existing and emerging applications. In demonstrating this potential, we highlight a need for conscientious consideration of coordination in the design of database systems: in many invariant confluent scenarios, traditional and/or legacy implementations designed for non-distributed environments over-coordinate and fail to capture the potential for coordination-free execution. Conversely, when invariant confluence does not hold, understanding when a coordination-free implementation does not exist spares system designers the effort of searching for a more efficient implementation when in fact such an implementation does not exist.

Of course, simply recognizing that a coordination-free implementation exists does not by itself lead to coordination-free systems. Rather, we must also find a coordination-free implementation of invariant confluent semantics. Therefore, we present the design, implementation, and evaluation of several sets of invariant confluent guarantees. We demonstrate order-of-magnitude improvements in latency and throughput over traditional algorithms, validating the power of coordination-free systems design. In addition to these case studies, we also present more general design principles for realizing coordination avoidance in prac-

tice. We note the importance of separating visibility from progress, ensuring composability of operations, and controlling visibility via multiversioning.

By example. As a simple example, consider the common Read Committed (RC) isolation guarantee [9], which is the default semantics in fifteen of eighteen popular relational databases, including Oracle, SAP Hana, and Microsoft SQL Server (Chapter 4). Informally, Read Committed ensures that users never read non-final writes to data; if, within a transaction, a user sets her username to "Sally" and, subsequently, sets her username to "Sal," no other user should ever read that the user's username is "Sally." The classic strategy for implementing Read Committed isolation dates to the 1970s and relies on locking: when our user wants to update her username record, she acquires a mutually exclusive lock on the record [125]. This is a reasonable strategy for a single-node database, but the coordination required to implement this mutual exclusion can be disastrous in a modern distributed environment: while our first user holds the lock on her username record, all other users who wish to also access the same username record must wait.

While coordination is sufficient to enforce RC isolation (via locking), is it strictly necessary? We can apply the principle of invariant confluence here: informally, if each individual user never observes non-final writes (i.e., each individual read-write history is valid under RC isolation), then their collective behavior (i.e., the "merged" histories) will not exhibit any reads of non-final writes. Thus, insofar as each individual user respects RC isolation, all users will, and so it is invariant confluent. As a result, there must be a coordination-free algorithm for enforcing RC isolation; now we must find one. One strategy is to store multiple versions of each record and mark each version with a special bit recording whether the corresponding write is a final write or not. If the database only shows users records that have been marked as final writes upon transaction commit, users will never observe non-final writes. Moreover, users can create versions, mark them as final, and read versions marked as final entirely concurrently, on separate copies of state, achieving our goal of a coordination-free implementation of RC isolation.

While this example is relatively simple, it demonstrates the power of rethinking legacy implementations of important semantics. In Chapter 4, we demonstrate how a slightly modified protocol achieves orders of magnitude improvements in performance on modern cloud computing infrastructure.

In the remainder of this chapter, we outline the key contributions of this work and describe the structure of the remainder of this thesis.

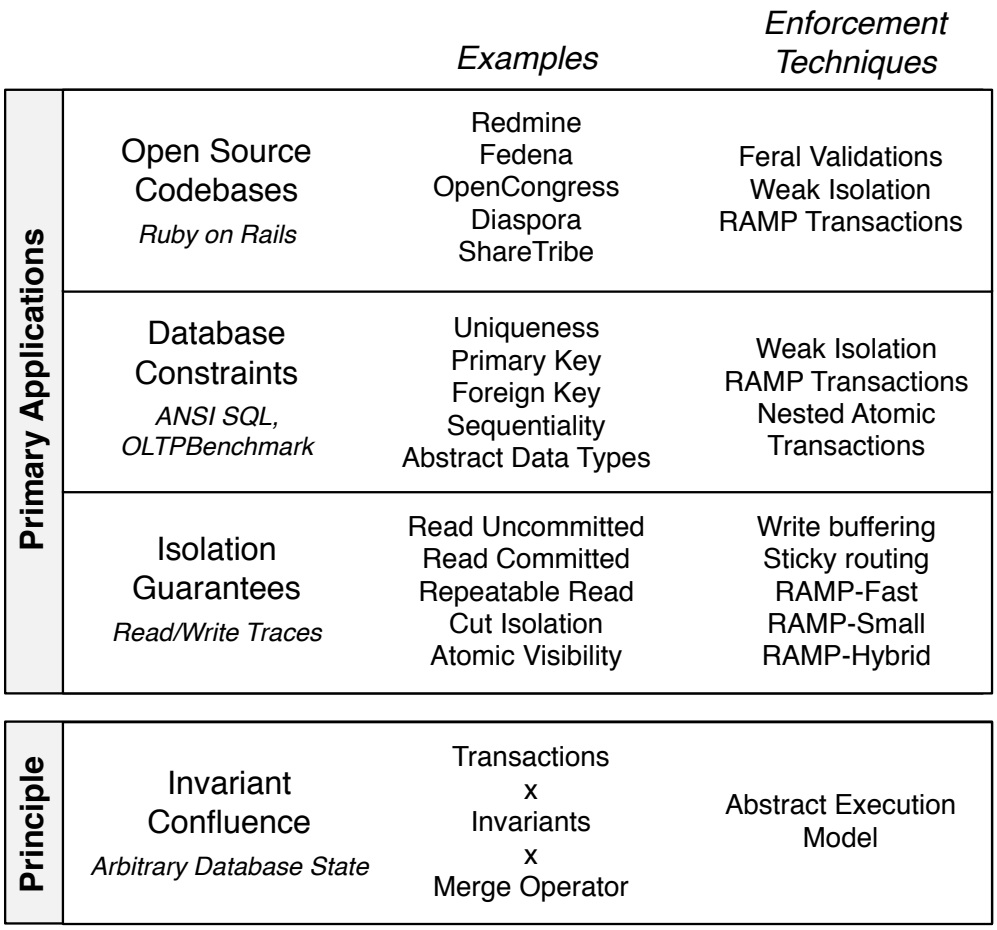


Figure 1.2: In this thesis, we develop the principle of Invariant Confluence, a necessary and sufficient condition for safe, convergent, coordination-free execution, and apply it to a range of application domains at increasing levels of abstraction: database isolation, database constraints, and safety properties from modern database-backed applications. Each guarantee that is invariant confluent is guaranteed to have at least one coordination-free implementation; we investigate the design of several implementations in this work, which operate at the database infrastructure tier (Figure 1.1).

1.2 Primary Contributions

In this section, we summarize the primary contributions of this work.

Coordination-free execution and Invariant Confluence. We identify coordination-free execution as fundamental to available, low latency, and scalable system execution. To do so, we present a system model and show a direct correspondence between these desirable system properties and the ability to guarantee non-trivial progress in an asynchronous network.

We subsequently develop the invariant confluence property, a necessary and sufficient condition for ensuring that a safety guarantee can be guaranteed under coordination-free, convergent execution of a given set of transactions. This is the first necessary and sufficient condition for these properties that we have encountered. In effect, invariant confluence lifts traditional partitioning arguments from distributed systems from the domain of event traces to the domain of arbitrary application logic and constraints over data. We use this property to examine and optimize a range of semantics from database systems (Figure 1.2), which we describe in turn below.

Coordination-free Isolation Guarantees. While transactional guarantees are often associated with serializability and its necessarily coordinated implementations, most databases in practice provide weaker forms of transactional isolation, or variants of admissible read-write interleavings (Figure 1.2). In a survey of 18 "ACID" and "NewSQL" databases, we find that only three offered serializability by default and only nine offered it as as an option at all. We investigate the weaker models offered by these databases and show that many are invariant confluent. For example, common isolation models such as Read Committed (default in eight databases) and ANSI Repeatable Read isolation are invariant confluent. Many guarantees, like Read Committed, arbitrated *visibility* but not concurrency (much like causal consistency). The resulting taxonomy is one of the first unified treatments of weak isolation, distributed register consistency, session guarantees, and coordination requirements in the literature. Using these results, we implement a coordination-free prototype implementing weak isolation guarantees that achieves up to two order-of-magnitude latency reductions when deployed across datacenters.

In addition to investigating existing isolation guarantees, we examine new guarantees. A number of applications leverage database-provided functionality for enforcing referential integrity, secondary indexing, and multi-get and multi-put operations, yet there is no coordination-free mechanism for enforcing them. Accordingly, a number of practitioner reports on systems (e.g., from Facebook, Google, LinkedIn, and Yahoo!) specifically highlight these use cases as scenarios where, lacking a coordination-free algorithm, architects explicitly sacrificed correctness for latency and availability. In response, we develop a new, invariant confluent isolation model called Read Atomic (RA) isolation and set of scalable, coordination-free algorithms called Read Atomic Multi-Partition (RAMP) Transactions for addressing the isolation requirements of these use cases. Informally, RA guarantees *atomic visibility* of updates: once one write from a transaction is visible, all writes will be visible. Existing protocols for achieving RA isolation (or stronger), such as distributed locking, couple atomic visibility with mutual exclusion; RAMP achieves the former without the cost of the latter. RAMP uses limited multi-versioning to allow clients to operate concurrently over the same data items while ensuring that readers can correctly detect and repair incom-

plete writes. Across a range of workloads (including high contention scenarios), RAMP transactions incur limited overhead and outperform existing mechanisms for achieving RA isolation while scaling linearly.

Coordination-free Database Constraints and Application Criteria. Moving from read-write traces to higher-level semantic properties (Figure 1.2), we examine the integrity constraints and invariants offered by databases today—including the foreign key constraints addressed by RAMP but also row-level check constraints, uniqueness constraints, and constraints on abstract data types—and classify each as invariant confluent or not. Many are invariant confluent, so we subsequently apply this classification to a number of database workloads from the OLTPBenchmark suite [98]. Many invariants in these workloads pass the invariant confluence test as well. For example, in the TPC-C benchmark, the gold standard for transaction processing performance, ten of twelve invariants are invariant confluent under the workload. Given this classification, we develop a database prototype and coordination-avoiding execution strategy for TPC-C that, on a cluster of 200 servers, achieves a 25-fold improvement in throughput over the prior best result (over 12.7M New-Order transactions per second).

While we are able to find invariant confluent database integrity constraints and benchmarks, are *real* applications invariant confluent? Moreover, are invariants a practical choice of correctness criteria? To answer these questions, we examine open source web applications to inspect their safety properties (Figure 1.2). We find that popular web programming frameworks—including Ruby on Rails, Django, and Spring/Hibernate—have introduced support for *validations*, or declarative, application-level invariants. We subsequently analyze the use of validations in 67 of the most popular Ruby on Rails applications on GitHub and find that, in fact, validation usage is fourteen times more common than the use of database transactions. Moreover, more than 86.9% (of over 9900 invariants) are invariant confluent. However, the remainder are *not* invariant confluent and therefore require coordination. For these invariants, we profile the incidence of constraint violations both with and without validations. In addition to demonstrating the applicability of invariant confluence, this study exposes a surprising practitioner trend away from transactions towards using invariants via validations.

In all, these results highlight a widespread potential for coordination-avoiding database system design within both classic and emerging database-backed applications. While coordination cannot always be avoided, in many common scenarios, we find it is possible to guarantee application safety within the database while also providing coordination-free execution. Our resulting database system prototypes and their regular order-of-magnitude speedups compared to conventional approaches evidence the power of this latent potential for coordination-avoiding execution.

1.3 Outline and Previously Published Work

The remainder of this dissertation proceeds as follows. Chapter 2 provides background on coordination and defines our system model. Chapter 3 presents the Invariant Confluence property. Chapters 4, 5,and 6 examine the coordination requirements and coordination-free implementations of transaction isolation, database functionality such as indexes, and constraints. Chapter 7 discusses related work and Chapter 8 concludes with a discussion of lessons learned, topics for future work, and closing thoughts.

Chapter 2 includes material from several previous publications [32, 34, 38, 40, 42–44]. Chapter 3 revises material from [34]. Chapter 4 revises [32] and [36]. Chapter 5 revises [37] and includes material from [35]. Chapter 6 revises material from [34] and [33] and includes material from [39]. Chapter 8 includes material from [86, 122].

Chapter 2

Coordination: Concepts and Costs

In this chapter, we further examine the concept of coordination and why we seek to avoid it. We discuss traditional uses of coordination to maintain correct data in database systems and measure its costs in modern distributed environment, which we will attempt to circumvent in the remainder of this thesis. We also present our formal system model for replicated databases.

2.1 Coordination and Correctness in Database Systems

As repositories for application state, databases are traditionally tasked with maintaining, informally, "correct" data—that is, data that obey some semantic guarantees about their integrity—on behalf of users. Thus, during concurrent access to data, a database ensuring data correctness must therefore decide which user operations can execute simultaneously and which, if any, cannot. Informally, we say that two operations within a database must *coordinate* if they cannot execute concurrently on independent copies of the database state (Section 2.3 provides a more formal treatment).

By example. Consider a database-backed payroll application that maintains information about employees and departments within a small business. In the application, a.) each employee is assigned a unique ID number and b.) each employee belongs to exactly one department. A database ensuring correctness must maintain these application-level semantic guarantees (or data *invariants*) on behalf of the application (i.e., without application-level intervention). In our payroll application, this is non-trivial: for example, if the application attempts to simultaneously create two employees by examining the set of currently assigned IDs and choosing an unassigned ID for each new employee, then the database must ensure the employees are assigned distinct IDs.

Serializability and conflicts. The classic answer to maintaining application-level invariants is to use serializable isolation: execute each user's ordered sequence of operations, or *transactions*, such that the end result is equivalent to some sequential execution [53, 123, 215]. If each transaction preserves correctness in isolation, composition via serializable execution ensures correctness. In our payroll example, the database would execute the two employee creation transactions such that one transaction appears to execute after the other. The second transaction would observe the ID that the first transaction chose, thus avoiding duplicate ID assignment.

While serializability is a powerful abstraction, it comes with a cost: for arbitrary transactions (and for all implementations of serializability's more conservative variant—conflict serializability), any two operations to the same item—at least one of which is a write—will result in a *read/write conflict*. Under serializability, these conflicts require coordination: to provide a serial ordering, conflicts must be totally ordered across transactions, and so transactions cannot proceed entirely independently [53]. As a canonical example, given initial database state containing two variables x and y, where $\{x = \bot, y = \bot\}$, if transaction T_1 reads from y and writes $x = 1$ and and T_2 reads from x and writes $y = 1$, then a database cannot both execute T_1 and T_2 independently on separate copies of state while maintaining serializability [32, 92].[1]

Because serializable semantics require coordination, *all* database implementations that provide serializability will coordinate. For example, a database could use two-phase locking [125] to provide serializability: in a simplified protocol, the first time a transaction accesses a data item x, the transaction can acquire an exclusive lock on x; once the transaction has completed all of its operations on the database, it can release all of its locks. In this protocol, locks form a point of coordination between concurrent transactions: while one transaction holds an exclusive lock on an item, other transactions that wish to operate on the same item cannot make progress.

Semantics, Sufficiency, and Necessity. It is often convenient to reason about semantic guarantees instead of concrete implementations of those guarantees. Instead of examining concurrency control mechanisms one-by-one (e.g., multi-version concurrency control, optimistic concurrency control, pre-scheduling, and so on), we can unequivocally determine—as in the case of serializability—that all implementations of a given semantics require coordination to enforce.

[1] This read-write pattern might arise in our ID assignment scenario: T_1 attempts to reserve ID 1 for its user, x, while T_2 attempts to reserve ID 1 for its user, y. If the two transactions run concurrently on separate copies of the database, neither T_1 nor T_2 would observe the others's updates. We present this example in terms of reads and writes because it is standard and to highlight the fact that serializability enforces concurrency by examining read/write access to variables, not by examining the program semantics.

However, the converse does not hold: just because a given implementation of a guarantee uses coordination does not mean that the guarantee necessarily requires coordination for enforcement. For example, even though a database may employ serializability to enforce an invariant, the invariant may not *require* coordination for correct enforcement. There may or may not be ways to enforce the invariants without coordination. In general, we can always coordinate. The core question is whether coordination is necessary for a given invariant or semantic property.

2.2 Understanding the Costs of Coordination

Why worry about coordination? Peter Deutsch starts his classic list of "Fallacies of Distributed Computing" with two concerns fundamental to distributed database systems: "*1.)* The network is reliable. *2.)* Latency is zero" [97]. In a distributed setting, network failures may prevent database servers from communicating, and, in the absence of failures, communication is slowed by factors like physical distance, network congestion, and routing. Thus, as we discuss here, the costs of coordination can be observed across three primary dimensions: increased latency, decreased throughput, and, in the event of partial failures, unavailability. In this section, we examine these costs in detail.

2.2.1 Latency

Even with fault-free networks, distributed systems face the challenge of network communication latency. If two operations running on two separate servers must coordinate, then the latency experienced by the operations will be bounded from below by the amount of time required to exchange information between the sites. In this section, we quantify network latencies of modern cloud computing environments. These are often large and may exceed hundreds of milliseconds in a geo-replicated, multi-datacenter context. Fundamentally, the speed at which two servers can communicate is (according to modern physics) bounded by the speed of light. In the best case, two servers on opposite sides of the Earth communicating via a hypothetical link through the planet's core would require a minimum 85.1 ms round-trip time (RTT; 133.7 ms if sent at surface level). As services are replicated to multiple, geographically distinct sites, the cost of communication between replicas increases.

In actual server deployments, messages travel slower than the speed of light due to routing, congestion, and computational overheads. To illustrate the behavior of intra-datacenter, inter-datacenter, and inter-planetary networks, we performed a measurement study of network behavior on Amazon's Elastic Compute Cloud (EC2), a widely used pub-

	H2	H3
H1	0.55	0.56
H2		0.50

(a) Within us-east-b availability zone

	C	D
B	1.08	3.12
C		3.57

(b) Across us-east availability zones

	OR	VA	TO	IR	SY	SP	SI
CA	22.5	84.5	143.7	169.8	179.1	185.9	186.9
OR		82.9	135.1	170.6	200.6	207.8	234.4
VA			202.4	107.9	265.6	163.4	253.5
TO				278.3	144.2	301.4	90.6
IR					346.2	239.8	234.1
SY						333.6	243.1
SP							362.8

(c) Cross-region (CA: California, OR: Oregon, VA: Virginia, TO: Tokyo, IR: Ireland, SY: Sydney, SP: São Paulo, SI: Singapore)

Table 2.1: Mean RTT times on EC2 (min and max highlighted)

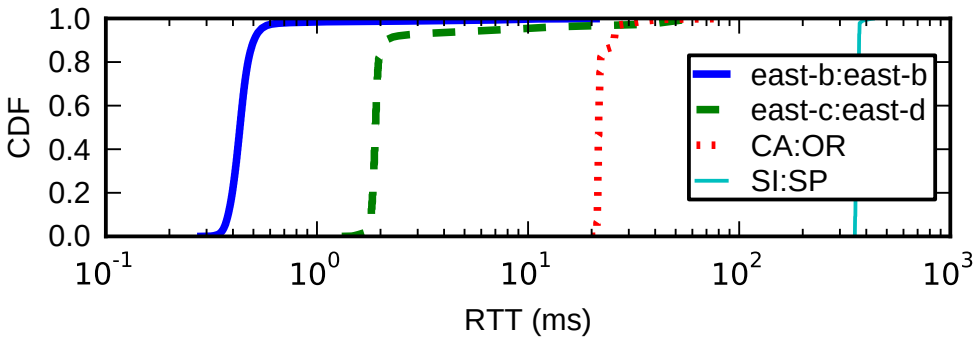


Figure 2.1: CDF of round-trip times for slowest inter- and intra- availability zone links compared to cross-region links.

lic compute cloud. We measured one week of ping times (i.e., round-trip times, or RTTs) between all seven EC2 geographic "regions," across three "availability zones" (closely co-located datacenters), and within a single "availability zone" (datacenter), at a granularity of 1s. We summarize the results of our network measurement study in Table 2.1. On average, intra-datacenter communication (Table 2.1a) is between 1.82 and 6.38 times faster than across geographically co-located datacenters (Table 2.1b) and between 40 and 647 times faster than across geographically distributed datacenters (Table 2.1c). The cost of wide-area communication exceeds the speed of light: for example, while a speed-of-light RTT

from São Paulo to Singapore RTT is 106.7 ms, ping packets incur an average 362.8 ms RTT (95th percentile: 649 ms). As shown in Figure 2.1, the distribution of latencies varies between links, but the overall trend is clear: coordination may lead to substantial delays. Quantifying and minimizing communication delays is also an active area of research in the networking community [230].

2.2.2 Throughput and Scalability

Coordination also affects throughput. If a transaction takes d seconds to execute, the maximum throughput of coordinating transactions operating on the same items under a general-purpose (i.e., interactive, non-batched) transaction model is limited by $\frac{1}{d}$. Any operations that arrive in excess of this limit will also have to wait. Within a single server, delays can be small, permitting tens to hundreds of thousands of conflicting transactions per item per second. In a partitioned database system, where different items are located on different servers, or in a replicated database system, where the same item is located (and is available for operations) on multiple servers, the cost increases: delay is lower-bounded by network latency. On a local area network, delay may vary from several microseconds (e.g., via Infiniband or RDMA) to several milliseconds on today's cloud infrastructure, permitting anywhere from a few hundred transactions to a few hundred thousand transactions per second. However, as we have seen, a wide-area network, delay is lower-bounded by the speed of light (worst-case on Earth, around 75 ms, or about 13 operations per second [32]). Under network partitions [41], as delay tends towards infinity, these penalties lead to unavailability [32, 118]. In contrast, operations executing without coordination can proceed concurrently and will not incur these penalties.

To further understand the costs of coordination, we performed two sets of measurements—one using a database prototype and one using traces from prior studies. We first compared the throughput of a set of coordinated and coordination-free transaction execution. Our basic workload is simple: a set of transactions read and increment a set of integers stored on separate servers.

First, we implemented two coordinated algorithms: traditional two-phase locking and an optimized variant of two-phase locking, both on in-memory data. In two-phase locking, each client acquires locks one at a time, requiring a full round trip time (RTT) for every lock request. For an N item transaction, locks are held for $2N + 1$ message delays (the $+1$ is due to broadcasting the unlock/commit command to the participating servers). Our optimized two-phase locking only uses one message delay (half RTT) to perform each lock request: the client specifies the entire set of items it wishes to modify at the start of the transaction (in our implementation, the number of items in the transaction and the starting item ID),

and, once a server has updated its respective item, the server forwards the remainder of the transaction to the server responsible for the next write in the transaction (similar to linear commit protocols [53]). For an N-item transaction, locks are only held for N message delays (the final server both broadcasts the unlock request to all other servers and also notifies the client), while a 1-item transaction does not require distributed locking.

To avoid deadlock (which we found was otherwise common in this high-contention microbenchmark), our implementation totally orders any lock requests according to item and executes them sequentially (e.g., lock item 1 then lock item 2 and so on). Our implementation also piggybacks operation commands along with lock requests, further avoiding message delays. Since we are only locking one item per server, our microbenchmark code does not use a dynamic lock manager and instead associates a single lock with each item; this further lowers locking overheads.

Our coordination-free transaction implementation is simpler: it uses no locks and simply performs the increment operation across each transaction in parallel, without acquiring locks.

We partitioned eight in-memory items (integers) across eight `cr1.8xlarge` Amazon EC2 instances with clients located on a separate set of `cr1.8xlarge` instances. Figure 2.2 reported in depicts results for the coordination-free implementation and the optimized two-phase locking case. Unsurprisingly, two-phase locking performs worse than optimized two-phase locking, but both incur substantial penalties due to coordination delay over the network.

With single-item, non-distributed transactions, the coordination-free implementation achieves, in aggregate, over 12M transactions per second and bottlenecks on *physical resources*—namely, CPU cycles. In contrast, the lock-based implementation achieves approximately 1.1M transactions per second: it is unable to fully utilize all 32 multi-core processor contexts due to lock contention. For distributed transactions, coordination-free throughput decreases linearly (as an N-item transaction performs N writes), while the throughput of coordinating transactions drops by over three orders of magnitude.

While the above microbenchmark demonstrates the costs of a particular *implementation* of coordination, we also studied the effect of more fundamental, implementation-independent overheads (i.e., also applicable to optimistic and scheduling-based concurrency control mechanisms). We determined the maximum attainable throughput for coordinated execution within a single datacenter (based on data from [230]) and across multiple datacenters (based on data from [32]) due to blocking coordination during atomic commitment [53].

We simulate traditional two-phase commit [53] and decentralized two-phase commit [124] using network models derived from existing studies. For an N-server transaction, classic

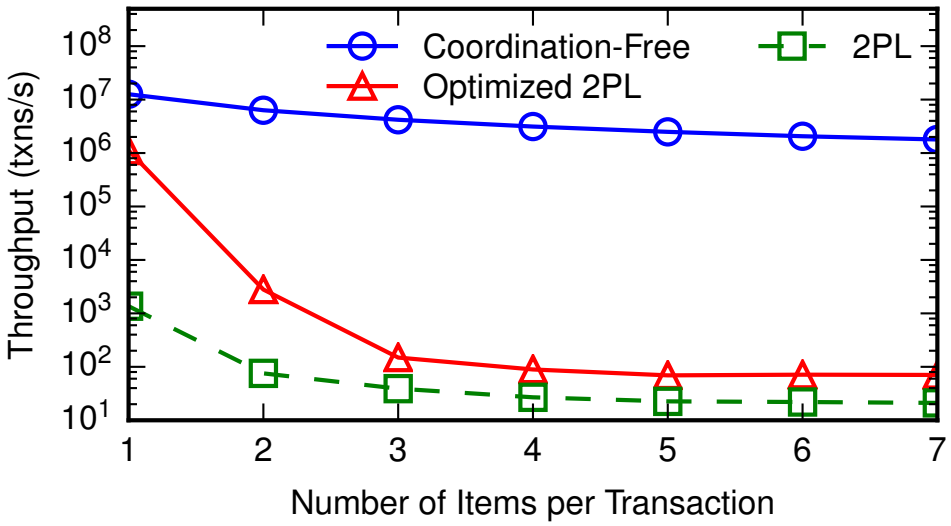


Figure 2.2: Microbenchmark performance of coordinated and coordination-free execution of transactions of varying size writing to eight items located on eight separate multi-core servers.

two-phase commit (`C-2PC`) requires N (parallel) coordinator to server RTTs, while decentralized two-phase commit (`D-2PC`) requires N (parallel) server to server broadcasts, or N^2 messages. Our simulation is straightforward, but we make several optimizations to improve the throughput of each algorithm. First, we assume that transactions are pipelined, so that each server can `prepare` immediately after it has `committed` the prior transaction. Second, our pipelines are ideal in that we do not consider deadlock: only one transaction `prepares` at a given time. Third, we do not consider the cost of local processing of each transaction: throughput is determined entirely by communication delay.

Figure 2.3 shows that, in the local area, with only two servers (e.g., two replicas or two coordinating operations on items residing on different servers), throughput is bounded by 1125 transactions per second (via `D-2PC`; 668 transactions per second via `C-2PC`). Across eight servers, `D-2PC` throughput drops to 173 transactions per second (respectively 321 for `C-2PC`) due to long-tailed latency distributions. In the wide area, the effects are more severe: if coordinating from Virginia to Oregon, `D-2PC` message delays are 83 ms per commit, allowing 12 operations per second. If coordinating between all eight EC2 availability zones, throughput drops to slightly over 2 transactions per second in both algorithms.

These results should also be unsurprising: coordinating—especially over the network—can incur serious throughput penalties. In contrast, coordination-free operations can execute without incurring these costs. The costs of actual workloads can vary: if coordinating

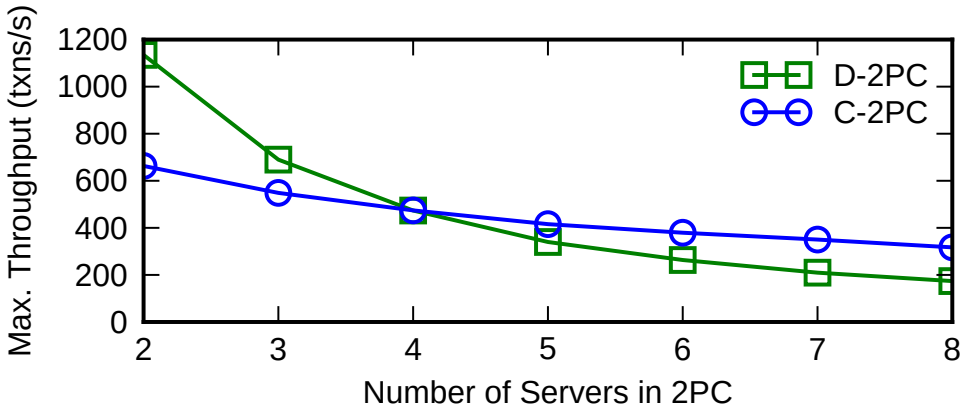
operations are rare, concurrency control will not be a bottleneck. For example, a serializable database executing transactions with disjoint read and write sets can perform as well as a non-serializable database without compromising correctness [140]. However, as these results demonstrate, minimizing the amount of coordination and its degree of distribution can therefore have a tangible impact on performance, latency, and availability [8, 32, 118]. While we study real applications in Section 6.3, these measurements highlight the worst of coordination costs on modern hardware.

While this study is based solely on reported latencies, deployment reports corroborate our findings. For example, Google's F1 uses optimistic concurrency control via WAN with commit latencies of 50 to 150 ms. As the authors discuss, this limits throughput to between 6 to 20 transactions per second per data item [207]. Megastore's average write latencies of 100 to 400 ms suggest similar throughputs to those that we have predicted [45]. Again, *aggregate* throughput may be greater as multiple 2PC rounds for disjoint sets of data items may safely proceed in parallel. However, *worst-case* access patterns—in effect, serial access to data items—will greatly limit throughput and scalability. Adding more servers will not assist; parallel processing is ineffective for workloads that must proceed serially.

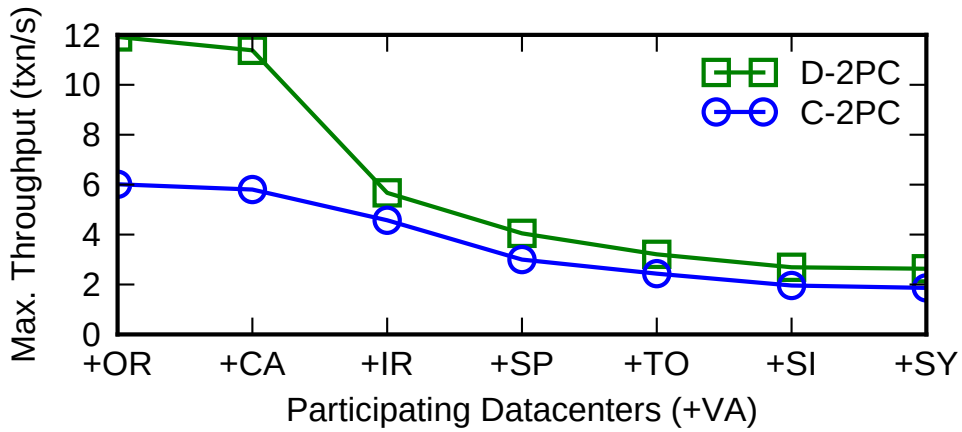
2.2.3 Availability and Failures

According to James Hamilton, Vice President and Distinguished Engineer on the Amazon Web Services team, "network partitions should be rare but net gear continues to cause more issues than it should" [128]. Anecdotal evidence confirms Hamilton's assertion. In April 2011, a network misconfiguration led to a twelve-hour series of outages across the Amazon EC2 and RDS services [29]. Subsequent misconfigurations and partial failures such as another EC2 outage in October 2012 have led to full site disruptions for popular web services like Reddit, Foursquare, and Heroku [99]. At global scale, hardware failures—like the 2011 outages in Internet backbones in North America and Europe due a router bug [204]—and misconfigurations like the BGP faults in 2008 [176] and 2010 [178] can cause widespread partitioning behavior.

Many of our discussions with practitioners—especially those operating on public cloud infrastructure—as well as reports from large-scale operators like Google [93] confirm that partition management is an important consideration for service operators today. System designs that do not account for partition behavior may prove difficult to operate at scale: for example, less than one year after its announcement, Yahoo!'s PNUTS developers explicitly added support for weaker, highly available operation. The engineers explained that "strict adherence [to strong consistency] leads to difficult situations under network partitioning or server failures...in many circumstances, applications need a relaxed approach" [181].



a.) Maximum serializable transaction throughput over local-area network in [230]



b.) Maximum serializable transaction throughput over wide-area network in [32] with transactions originating from a coordinator in Virginia (VA; OR: Oregon, CA: California, IR: Ireland, SP: São Paulo, TO: Tokyo, SI: Singapore, SY: Sydney)

Figure 2.3: Atomic commitment latency as an upper bound on conflicting serializable transaction throughput over local-area and wide-area networks.

Several recent studies rigorously quantify partition behavior. A 2011 study of several Microsoft datacenters observed over 13,300 network failures with end-user impact, with an estimated median 59,000 packets lost per failure. The study found a mean of 40.8 network link failures per day (95th percentile: 136), with a median time to repair of around five minutes (and up to one week). Perhaps surprisingly, provisioning redundant networks only reduces impact of failures by up to 40%, meaning network providers cannot easily curtail partition behavior [119]. A 2010 study of over 200 wide-area routers found an average of 16.2–302.0 failures per link per year with an average annual downtime of 24–497 minutes per link per year (95th percentile at least 34 hours) [223]. In HP's managed enterprise networks, WAN, LAN, and connectivity problems account for 28.1% of all customer support tickets while 39% of tickets relate to network hardware. The median incident duration for highest priority tickets ranges from 114–188 minutes and up to a full day for all tickets [222]. Other studies confirm these results, showing median time between connectivity failures over a WAN network of approximately 3000 seconds with a median time to repair between 2 and 1000 seconds [175] as well as frequent path routing failures on the Internet [150]. A recent, informal report by Kingsbury and Bailis catalogs a host of additional practitioner reports [40]. Not surprisingly, isolating, quantifying, and accounting for these network failures is an area of active research in networking community [165].

2.2.4 Summary: Costs

Coordination is expensive. Empirical measurements on existing infrastructure confirm its latency and throughput costs, and a host of reports describe the difficulty of dealing with failures. Given an absence of quantitative failure data, we focus primarily on the performance-related aspects of coordination in this dissertation. However, our pursuit of coordination avoidance benefits all three dimensions.

2.2.5 Outcome: NoSQL, Historical Context, Safety and Liveness

The above costs have become especially pressing over the past decade, leading to a schism among distributed database systems designers. An increasing number of modern applications demand low latency and available operation at unprecedented scale. For example, Facebook's RocksDB reportedly handles nine billion queries per second [182], while increased latency may have a marked impact on web application engagement and revenue [162, 163, 203]. As a result, the database market has been inundated by a large number of data stores promising coordination-free execution (see Chapter 7). This market shift is perhaps the most significant development in transaction processing over the past decade.

While these stores, often collectively labeled "NoSQL," provide admirable scalability and behavioral properties, they infrequently provide useful semantics for developers. That is, the common denominator among these semantics is a particular property called *eventual consistency*: informally, if no additional updates are made to a given data item, all reads to that item will eventually return the same value [224]. While this is a useful property, it leaves some unfortunate holes. First, what is the eventual state of the database? A database always returning the value 42 is eventually consistent, even if 42 was never written. The database can eventually choose (i.e., *converge to*) an arbitrary value. Second, what values can be returned before the eventual state of the database is reached? If replicas have not yet converged, what guarantees can be made about the data returned?

To more precisely explain why eventual consistency is not strong enough, we consider two concepts from distributed systems. A *safety* property guarantees that "nothing bad happens": for example, every value that is read was, at some point in time, written to the database. A *liveness* property guarantees that "something good eventually happens": for example, all requests eventually receive a response [17]. The difficulty with eventual consistency is that it makes no safety guarantees—eventual consistency is purely a liveness property. Something good eventually happens—eventually all reads return the same value—but there are no guarantees with respect to what value is eventually returned, and any value can be returned out in the meantime. For truly meaningful guarantees, safety and liveness properties need to be taken together: without one or the other, systems can provide trivial implementations that provide less-than-satisfactory results.

In practice, eventually consistent systems often provide "strongly consistent" (e.g., linearizable) behavior with frequency; for example, using production latency data from LinkedIn's eventually consistent database clusters, we found that, under common deployment settings, 99.9% of reads delivered the last completed write within 45.5 ms of the write's completion [42–44]. However, given the possibility of "inconsistent" behavior, programmers must either explicitly account for this possibility or otherwise "code around" these anomalies (Chapter 7). Here, we wish to *guarantee* safety under all circumstances.

Our goal in this work is to preserve convergence and coordination-free execution while *simultaneously* preserving safety guarantees as found in modern applications and databases. Thus, we attempt to restore safety to this class of scalable databases without compromising their benefits. In the next section, we formally define these concepts.

Property	Informal Effect
Global validity	Invariants hold over committed states
Transactional availability	Non-trivial response guaranteed
Convergence	Updates are eventually reflected in shared state
Coordination-freedom	No synchronous coordination

Table 2.2: Key properties of the system model and their informal effects.

2.3 System Model

In this section, we present a more formal model for transaction execution and define our desirable criteria for transaction execution, including coordination-free execution. We begin with an informal description of our model and provide more formal definitions in the remainder of the section.

Informally, in our model, transactions operate over logical replicas, or independent snapshots of database state. Transaction writes are applied at one or more replicas initially when the transaction commits and then are integrated into other replicas asynchronously via a "merge" operator that incorporates those changes into the snapshot's state. Given a set of invariants describing valid database states, as Table 2.2 outlines, we seek to understand when it is possible to ensure invariants are always satisfied (global validity) while guaranteeing a response (transactional availability) and the existence of an eventually agreed upon common state shared between replicas (convergence), all without communication during transaction execution (coordination-freedom). This model need not directly correspond to a given implementation and may not even correspond to the operation of a distributed system, as it can be implemented via multi-versioning. Rather, it serves as a useful abstraction. The remainder of this section defines these concepts.

Databases. We represent a state of a database as a set D of unique *versions* of data items located on an arbitrary set of database servers, where each version is located on at least one server. Thus, each server contains a *local* database that is a subset of the database located on all servers (the *global* database). We will denote version i of item x as x_i and use $\mathcal{D}$ to denote the set of possible database states—that is, the set of sets of versions. The global database is initially populated by an initial state D_0 (typically but not necessarily empty).

Transactions, Replicas, and Merging. Application clients submit requests to the database in the form of transactions, or ordered groups of operations on data items. Each transaction operates on a logical *replica*, or set of versions of the items mentioned in the transaction. At the beginning of the transaction, the replica reflects a subset of the global database state and is formed from all of the versions of the relevant items from the local databases

of one or more physical servers that are contacted during transaction execution. As the transaction executes, it may add versions (of items in its writeset) to its replica. Thus, we define a transaction T as a transformation on a replica: $T : \mathcal{D} \to \mathcal{D}$. We treat transactions as opaque transformations that can contain writes (which add new versions to the replica's set of versions) or reads (which return a specific set of versions from the replica).

Upon completion, each transaction can *commit*, signaling success, or *abort*, signaling failure. Upon commit, the replica state is subsequently *merged* ($\sqcup$:$\mathcal{D} \times \mathcal{D} \to \mathcal{D}$) into the local database of at least one server. We require that the merged effects of a committed transaction will eventually become *visible* to other transactions—that is, its versions will be present within those transactions' replicas—that later begin execution on the same server.[2] Over time, effects eventually propagate to other servers, again through the use of the merge operator. Though not strictly necessary (see "Alternative Merge" below), we assume the merge operator is commutative, associative, and idempotent [19, 205] and that, for all states D_i, $D_0 \sqcup D_i = D_i$. In our initial model, we define merge as set union of the versions contained at different servers. For example, if server $R_x = \{v\}$ and $R_y = \{w\}$, then $R_x \sqcup R_y = \{v, w\}$.

In effect, each transaction can modify its replica state without modifying any other concurrently executing transactions' replica state. Replicas therefore provide transactions with partial "snapshot" views of the global database (that we will use to simulate concurrent executions, similar to revision diagrams [68]). Importantly, two transactions' replicas do not necessarily correspond to two physically separate servers; rather, a replica is simply a partial "view" over the global state of the database. For now, we assume advance knowledge of all transactions to be submitted to the system.

Invariants. To determine whether a database state is valid according to application correctness criteria, we reason about a set of declared *invariants*, or predicates over databases: $I : \mathcal{D} \to \{\texttt{true}, \texttt{false}\}$ [105]. Thus, our invariants capture safety of data stored in the database. In our payroll example, we could specify an invariant that only one user in a database has a given ID. This invariant—as well as almost all invariants we consider—is naturally expressed as a part of the database schema (e.g., via DDL); however, our approach allows us to reason about invariants even if they are known to the developer but not declared to the system. Invariants directly capture the notion of ACID Consistency [53, 123], and we say that a database state is *valid* under an invariant I (or I-valid) if it satisfies the predicate:

Definition 1. *A replica state* $R \in \mathcal{D}$ *is* I*-valid iff* $I(R) = \texttt{true}$.

[2] This implicitly disallows servers from always returning the initial database state when they have newer writes on hand. This is a relatively pragmatic assumption but also simplifies our later reasoning about admissible executions.

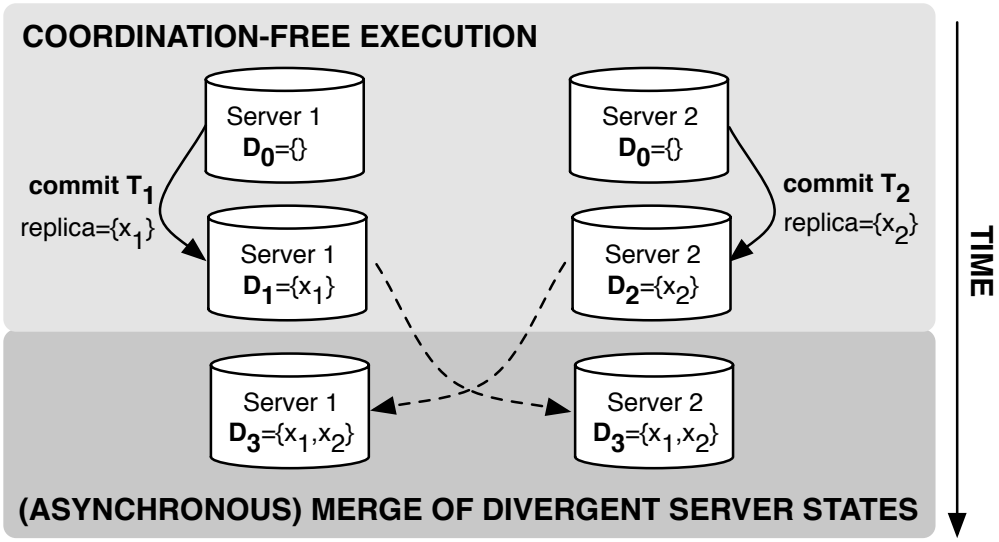


Figure 2.4: An example coordination-free execution of two transactions, T_1 and T_2, on two servers. Each transaction writes to its local replica, then, after commit, the servers asynchronously exchange state and converge to a common state (D_3).

We require that D_0 be valid under invariants. Section 3.3 provides additional discussion regarding our use of invariants.

Availability. To ensure each transaction receives a non-trivial response, we adopt the following definition of *availability* [32]:

Definition 2. *A system provides transactionally available execution iff, whenever a client executing a transaction* T *can access servers containing one or more versions of each item in* T*, then* T *eventually commits unless* T *aborts due to an explicit abort operation in* T *or if committing* T *would violate a declared invariant over* T*'s replica state.*

Under the above definition, a transaction can only abort if it explicitly chooses to abort itself or if committing would violate invariants over the transaction's replica state.[3]

Convergence. Transactional availability allows replicas to maintain valid state *independently*, but it is vacuously possible to maintain "consistent" database states by letting replicas diverge (contain different state) forever. This guarantees safety but not liveness [202]. To force state sharing, we adopt the following definition:

[3]This basic definition precludes fault tolerance (i.e., durability) guarantees beyond a single server failure [32]. We can relax this requirement and allow communication with a fixed number of servers (e.g., $F + 1$ servers for F-fault tolerance; F is often small [95]) without affecting our results. This does not affect scalability because, as more replicas are added, the communication overhead required for durability remains constant.

Definition 3. *A system is convergent iff, for each pair of servers, in the absence of new writes to the servers and in the absence of indefinite communication delays between the servers, the servers eventually contain the same versions for any item they both store.*

To capture the process of reconciling divergent states, we use the previously introduced merge operator: given two divergent server states, we apply the merge operator to produce convergent state. We assume the effects of merge are atomically visible: either all effects of a merge are visible or none are. This assumption is not always necessary but it simplifies our discussion and, as we later discuss, is maintainable without coordination [32, 37].

Our treatment of convergence uses a pair-wise definition (i.e., each pair converges) [171] rather than a system-wide definition (i.e., all nodes converge). This is more restrictive than system-wide convergence but allows us to make guarantees on progress despite partitions between subsets of the servers. This also precludes the use of protocols such as background consensus, which can stall indefinitely in the presence of partitions. Like many of the other decisions in our model, this too could be relaxed if system-wide convergence is sufficient.

Maintaining validity. To make sure that both divergent and convergent database states are valid and, therefore, that transactions never observe invalid states, we introduce the following property:

Definition 4. *A system is globally* I*-valid iff all replicas always contain* I*-valid state.*

Coordination. Our system model is missing one final constraint on coordination between concurrent transaction execution:

Definition 5. *A system provides coordination-free execution for a set of transactions* T *iff the progress of executing each* $\mathtt{t} \in \mathsf{T}$ *is only dependent on* $\mathtt{t}$*'s replica's state (i.e., the versions of the items* $\mathtt{t}$ *reads).*

That is, in a coordination-free execution, each transaction's progress towards commit/abort is independent of other operations (e.g., writes, locking, validations) being performed on behalf of other transactions. Thus, transaction execution cannot rely on blocking synchronization or communication with other concurrently running transactions. This coordination-free execution corresponds to availability under the asynchronous network model used in distributed computing [118]: progress is guaranteed despite the possibility of indefinite delays between servers.

A note on partial replication. We have explicitly considered a replicated model. This was a natural in allowing us to reason about distributed and multi-versioned databases. Our model captures both fully replicated databases, where all data items are located on all

servers, and partially replicated databases, where data items are located on a proper subset of servers. That is, as long as clients can access *some* copy of the data items they are looking for, they are allowed to proceed. This allows us to reason about properties over the entire set of data items (i.e., the abstraction of a replica is a copy of the whole database), without having to describe data placement. The distinction is not relevant from the perspective of reasoning about coordination requirements: if two operations can proceed independently, irrespective of any side-effects, timing, or other information produced (or not) by the other, then the question of whether the data they operate under is stored on multiple servers or one server is irrelevant.

To illustrate this point, a coordination-free algorithm designed for a partially replicated system will trivially execute on a fully replicated system. A coordination-free algorithm designed for a fully replicated system can execute on a partially replicated system by having each client operate over its own independent set of versions until writes quiesce. The latter is not practical but illustrates our point. In practice, building efficient algorithms for partitioned databases is challenging; for example, Chapter 5 is devoted to a suite of algorithms for enforcing a common semantics in partially-replicated databases.

However, in a partially replicated system implementation, checking whether an invariant is satisfied over a transaction's logical replica state prior to transaction commit may require transactions to access servers containing data that they did not directly mention in their program text. In effect, each transaction must end with an inline invariant check over any data it modifies and any data referenced by invariants over the data it modifies to avoid the transaction committing a non-I-valid state. Thus, in a partially replicated system, this checking can incur communication overhead (e.g., to ensure that the opposite end of a foreign key relation is present)—although not always (e.g., to check for a null value upon insertion). This cost is fundamental to general-purpose invariant verification in partially replicated systems and has well-studied relatives in active database systems [14,228] but is—as a consequence of our decision not to distinguish fully-replicated and partially-replicated stores—not transparent in our model.

A note on convergence. We have chosen to implement convergence using anti-entropy [96, 200] via the merge operator. In our model, servers exchange versions by shipping the *side effects* of transactions rather than the transactions themselves. Alternatives, such as shipping closures containing the transactions [189] are possible but are less common in practice [38,95]. Our goal here is that, under a merge function like set union, transaction effects are propagated, and transactions are not "rewritten" as in alternative extended transaction models such as compensating actions (Chapter 7).

Alternative Merges. As discussed above, merge need not necessarily be associative, commu-

tative, and idempotent. For example, in Bayou [189], users write arbitrary merge functions (that may not be commutative, associative, or idempotent) and the server processes determine a total ordering on merge operations in the background. As long as there is eventually connectivity between all servers, Bayou guarantees convergence. Thus, the system is convergent insofar as all servers replay all relevant merge operations even though the merges themselves are not. Nevertheless, due to their practical implementation, we focus on associative, commutative, and idempotent merges here.

By example. Figure 2.4 illustrates a coordination-free execution of two transactions T_1 and T_2 on two separate, fully-replicated physical servers. Each transaction commits on its local replica, and the result of each transaction is reflected in the transaction's local server state. After the transactions have completed, the servers exchange state and, after applying the merge operator, converge to the same state. Any transactions executing later on either server will obtain a replica that includes the effects of both transactions.

Chapter 3

Invariant Confluence and Coordination

With a system model and goals in hand, we now address the question: when do applications require coordination for correctness? The answer depends not just on an application's transactions or on an application's invariants. Rather, the answer depends on the *combination* of the two under consideration. Our contribution in this section is to formulate a criterion that will answer this question for specific combinations in an implementation-agnostic manner.

In this section, we focus almost exclusively on providing a general answer to this question. The remainder of this thesis is devoted to practical interpretation and application of these results.

3.1 Invariant Confluence: Criteria Defined

To begin, we introduce the central property (adapted from the constraint programming literature [100]) in our main result: invariant confluence. Applied in a transactional context, the invariant confluence property informally ensures that divergent but I-valid database states can be merged into a valid database state—that is, the set of valid states reachable by executing transactions and merging their results is closed (w.r.t. validity) under merge. In the next sub-section, we show that invariant confluence analysis directly determines the potential for safe, coordination-free execution.

We say that a database D_i is a I-T-*reachable state* if, given an invariant I and set of transactions T (with merge function $\sqcup$), there exists a partially ordered set of transaction and merge function invocations that yields D_i, and each intermediate state produced by transaction execution or merge invocation is also I-valid. We call these previous states *ancestor states*. Each ancestor state is either the initial state D_0 or is I-T-reachable from D_0.

We can now formalize the invariant confluence property:

Definition 6 (Invariant Confluence)**.** *A set of transactions* T *is invariant confluent with respect to invariant* I *if, for all* I-T*-reachable states* D_i, D_j *with a common ancestor state,* $D_i \sqcup D_j$ *is* I*-valid.*

Figure 3.1 depicts a invariant confluent merge of two I-T-reachable states, each starting from a shared, I-T-reachable state D_s. Two sequences of transactions $t_{in} \dots t_{i1}$ and $t_{jm} \dots t_{j1}$ each independently modify D_s. Under invariant confluence, the states produced by these sequences (D_{in} and D_{jm}) must be valid under merge.

We require our merged states in the invariant confluence formulation to have a common ancestor to rule out the possibility of merging states that could not have arisen from transaction execution (e.g., even if no transaction assigns IDs, merging two states that each have unique but overlapping sets of IDs could be invalid). Moreover, in practice, every non-invariant confluent set of transactions and invariants we encountered had a counterexample execution consisting of a divergent execution consisting of a single pair of transactions. However, we admit the possiblity that more exotic transactions and merge functions might lead to complex behavior in non-single-step divergence, so we consider arbitrary histories here. Precisely characterizing the difference in expressive power between invariant confluent transactions under single-transaction divergence versus multi-transaction divergence is an interesting question for future work.

Invariant confluence holds for specific combinations of invariants and transactions. In our payroll database example from Section 2.1, removing a user from the database is invariant confluent with respect to the invariant that user IDs are unique. However, two transactions that remove two different users from the database are not invariant confluent with respect to the invariant that there exists at least one user in the database at all times. Later chapters discuss actual combinations of transactions and invariants (and with greater precision).

3.2 Invariant Confluence and Coordination-Free Execution

We now apply invariant confluence to our goals from Section 2.3:

Theorem 1. *A globally* I*-valid system can execute a set of transactions* T *with coordination-freedom, transactional availability, and convergence if and only if* T *is invariant confluent with respect to* I.

Theorem 1 establishes invariant confluence as a necessary and sufficient condition for invariant-preserving, coordination-free execution. If invariant confluence holds, there exists

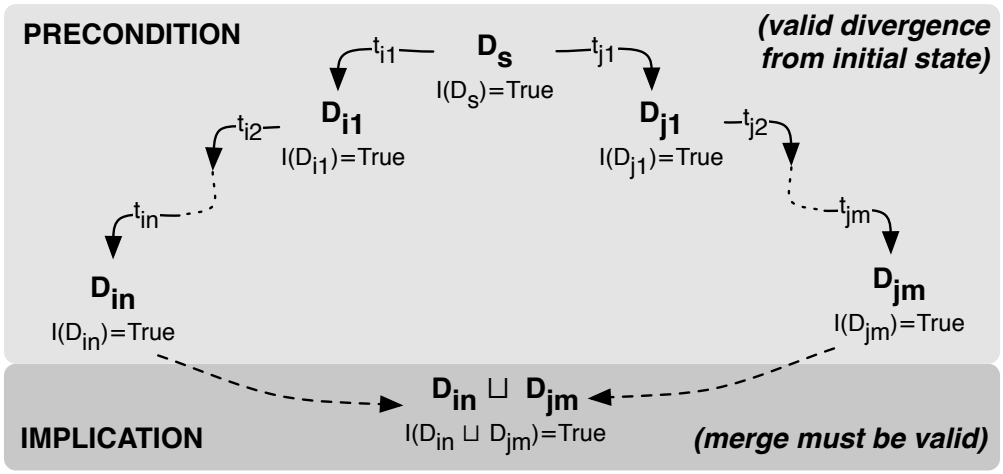


Figure 3.1: A invariant confluent execution illustrated via a diamond diagram. If a set of transactions T is invariant confluent, then all database states reachable by executing and merging transactions in T starting with a common ancestor (D_s) must be mergeable ($\sqcup$) into an I-valid database state.

a correct, coordination-free execution strategy for the transactions; if not, no possible implementation can guarantee these properties for the provided invariants and transactions. That is, if invariant confluence does not hold, there exists at least one execution of transactions on separate replicas that will violate the given invariants when servers converge. To prevent invalid states from occurring, at least one of the transaction sequences will have to forego availability or coordination-freedom, or the system will have to forego convergence. Invariant confluence analysis is independent of any given implementation, and effectively "lifts" prior discussions of scalability, availability, and low latency [8,32,118] to the level of application (i.e., not "I/O" [20]) correctness. This provides a useful handle on the implications of coordination-free execution without requiring reasoning about low-level properties such as physical data location and the number of servers.

We provide a full proof of Theorem 1 below but first provide a sketch. The backward direction is by construction: if invariant confluence holds, each replica can check each transaction's modifications locally and replicas can merge independent modifications to guarantee convergence to a valid state. The forwards direction uses a partitioning argument [118] to derive a contradiction: we construct a scenario under which a system cannot determine whether a non-invariant confluent transaction should commit without violating one of our desired properties (either compromising validity or availability, diverging forever, or coordinating). The structure of our argument is not novel, but the ability to discuss the effects of operations without discussing the underlying system behavior (e.g., the presence of network partitions) is useful.

To begin, we demonstrate the possibility of forcing a coordination-free system into any I-T-reachable database state via a carefully crafted sequence of partitioning behavior.

Lemma 1. *Given a set of transactions* T *and invariants* I*, a globally* I*-valid, coordination-free, transactionally available, and convergent system is able to produce any* I-T*-reachable state* S_i.

Proof Lemma 1. Let α_i represent a partially ordered sequence of transactions T_i and merge procedure invocations M_i (call this a *history*) starting from S_0 that produces S_i.

We REPLAY the history α on a set of servers as follows. Starting from the initial state S_0, we traverse the partial order according to a topological sort. Initially, we mark all operations (transactions or merges) in α as *not done*. We begin by executing all transactions T_i in α_i that have no predeceding operations in α. For each transaction $t \in T_i$, we execute t on a server that is unable to communicate with any other server.[1] Upon transaction commit, we merge each replica's modifications into the server. (Recall that, because S_i is I-T-reachable, each transaction in α is an I-valid transformation and must either eventually commit or abort itself to preserve transactional availability, and, due to coordination-freedom, the result of the execution is dependent solely on its input—in this case, S_0.) We subsequently mark each executed transaction t as *done* and denote the server that executed t as s_t.[2]

Next, we repeatedly select an operation o_i from α that is marked as *not done* but whose preceding operations are all marked as *done*.

If o_i is a transaction with preceding operation o_j, performed corresponding server s_j, we *partition* s_j, and a second server s_i, a server containing state S_0, such that s_j and s_i can communicate with each other but cannot communicate with any other server. Under convergent execution, s_j and s_i must eventually contain the same state (given that $s_j \sqcup S_0$ is defined in our model to be s_j). Following convergence, we partition s_j and s_i so they can no longer communicate.[3] We subsequently execute o_i on s_i. Again, o_i must either commit or abort itself to preserve transactional availability, and its behavior is solely dependent on its input due to coordination-freedom. Once o_i is completed, we mark it as *done*.

[1] Without loss of generality, we discuss replicated databases, where each server contains the entire set of items referenced in the history. It is trivial to extend the REPLAY procedure to a partially replicated environment by replacing each server with a set of servers that contains all data items necessary for each operation.

[2] Recall from Section 2.3 that we consider arbitrary groups of servers. Thus, we simply execute each operation in the history on a new server. We could be more parsimonious with our use of servers in this procedure but choose not to do so for simplicity.

[3] In the event that o_i is the only event that immediately follows o_j in α, we could simply execute o_i on s_j. In the event that multiple operations immediately follow o_j in α, we need to ensure that each operation proceeds independently. Thus, we are conservative and give every operation its own server that contains the effects of the preceding operations in α.

If o_i is a merge procedure with preceding operations o_j and o_k on corresponding servers s_j and s_k, we produce servers $s_{j'}$ and $s_{k'}$ containing the same contents as s_j and s_k, respectively, as above, by partitioning $s_{j'}$ and s_j and, respectively, s_k and $s_{k'}$, waiting until convergence, then repartitioning each. Subsequently, we place $s_{j'}$ and $s_{k'}$ in the same network partition, forcing the merge (o_i) of these states via the convergence requirement. We subsequently mark o_i as *done*.

When all operations in α are marked as *done*, the final operation we have performed will produce server containing state S_i. We have effectively (serially) traversed the history, inducing the partially ordered sequence of transactions and merges by triggering partitions; we force transaction commits due to transactional availability and merges due to our pairwise convergence requirement. □

Given this possibility, we proceed to prove Theorem 1 from Section 3.2.

Proof Theorem 1. (⇐) We begin with the simpler proof, which is by construction. Assume a set of transactions T are invariant confluent with respect to an invariant I. Consider a system in which each server executes the transactions it receives against a replica of its current state and checks whether or not the resulting state is I-valid. If the resulting state is I-valid, the replica commits the transaction and its mutations to the state. If not, the replica aborts the transaction. Servers opportunistically exchange copies of their local states and merge them. No individual replica will install an invalid state upon executing transactions, and, because T is invariant confluent under I, the merge of any two I-valid replica states from individual servers (i.e., I-T-reachable states) as constructed above is I-valid. Therefore, the converged database state will be I-valid. Transactional availability, convergence, and global I-validity are all maintained via coordination-free execution.

(⇒) Assume a system M guarantees globally I-valid operation for set of transactions T and invariant I with coordination-freedom, transactional availability, and convergence, but T is not I-confluent. Then there exist two I-T-reachable states S_a and S_b with common ancestor I-T-reachable state S_o such that, by definition, $I(S_a)$ and $I(S_b)$ are true, but $I(S_a \sqcup S_b)$ is false. Call the history that corresponds to S_a α_a and the thisory that corresponds to S_b α_b.[4]

Consider two executions of system M corresponding to histories α_a and α_b. In each execution, we begin by forcing M to produce a server containing S_o (via invoking REPLAY from Lemma 1 on M). In α_a, we subsequently REPLAY the history from S_o using M. In α_b, we subsequently REPLAY the history α_b starting from S_c. Call T_{fa} and T_{fb} the final (set of) transactions that produced each of S_a and S_b (that is, the set of transactions in each

[4]We may be able to apply Newman's lemma and only consider single-transaction divergence (in the case of convergent and therefore "terminating" executions) [100, 144], but this is not necessary for our results.

execution that are not followed by any other transaction). During the execution of α_a and α_b, all transactions in each of T_{fa} and T_{fb} will have committed to maintain transactional availability, their end result will be equivalent to the result in S_a and S_b, which are both I-valid, by assumption.

We now consider a third execution, α_c. α_c produces a server containing S_o by performing REPLAY using M, as above. Then, α_c independently performs REPLAY for α_1 and α_2 but does not execute or proceed further in either history than the first element of T_{fa} or T_{fb}. Instead, we consider these specially:

First, note that, if we independently REPLAY the transactions in T_{fa} and T_{fb}, and M commits them, then we will have completed all operations in α_a and α_b. In this case, M will produce two servers s_a and s_b containing states S_a and S_b. If we partition servers s_a and s_b such that they can communicate with each other but cannot communicate with any other servers, s_i and s_j must eventually converge. When s_a and s_b eventually converge, they will converge to $S_a \sqcup S_b$. However, $I(S_a \sqcup S_b)$ is false, which would violate global I-validity. Thus, M cannot commit all of the transactions in T_{fa} and T_{fb}.

On the other hand, suppose we independently REPLAY the the transactions in T_{fa} and T_{fb} and M aborts one of the transactions t_a in T_{fa}. In this case, the server that aborts t_a (call this server s_{ac}) will have exactly the same information (set of versions) available to it as the server that executed t_a in the execution corresponding to α_a above (call this second server s_{aa}). However, in the execution corresponding to α_a, s_{aa} committed t_a. Thus, in one execution (α_a), M commits t_a (on s_{ac}) and, in another execution (α_c), M aborts t_a (on s_{aa}), even though the contents of the servers are identical. Thus, despite the fact that the configurations are indistinguishable, M performs a different operation, a contradiction. If we REPLAY the transactions in T_{fa} and T_{fb} and M aborts one of the transactions t_b in T_{fb}, we will similarly observe a contradiction: a server in the execution corresponding to α_c will have aborted a transaction despite having the same information available to it as another server in the execution corresponding to α_b, which committed the same transaction. Thus, to the servers executing T_{fb}, α_a is indistinguishable from α_c, and, to the servers executing T_{fb}, α_b is indistinguishable from α_c.

Therefore, to preserve transactional availability in the execution corresponding to α_c, M must sacrifice one of global validity (by allowing the merge of S_a and S_b, resulting in an invalid state), convergence (by never merging the contents of the servers containing S_a and S_b), or coordination-freedom (by requiring communication between servers during REPLAY). □

3.3 Discussion and Limitations

Invariant Confluence captures a simple, informal rule: ***coordination can only be avoided if all local commit decisions are globally valid***. (Alternatively, commit decisions are composable.) If two independent decisions to commit can result in invalid merged (converged) state, then replicas must coordinate in order to ensure that only one of the decisions is to commit. Given the existence of an unsafe execution and the inability to detect the unsafe execution using only local information, a globally valid system *must* coordinate in order to prevent the invalid execution from arising.

Use of invariants. Our use of invariants in invariant confluence is key to achieving a *necessary* and not simply sufficient condition. By directly capturing correctness criteria via invariants, invariant confluence analysis only identifies "true" conflicts. This allows invariant confluence analysis to perform a more accurate assessment of whether coordination is needed compared to related conditions such as commutativity (Chapter 7).

However, the reliance on invariants also has drawbacks. Invariant confluence analysis only guards against violations of any specified invariants. If invariants are incorrectly or incompletely specified, a invariant confluent database system may violate correctness. If users cannot guarantee the correctness and completeness of their invariants and operations, they should opt for a more conservative analysis or mechanism such as employing serializable transactions. In fact, without such a correctness specification, for arbitrary transaction schedules, serializability is—in a sense—the "optimal" strategy [149]. Alternatively, restricting the programming language—for example, to allow only invariant confluent operations or operations that exhibit other, possibly more conservative properties like monotonicity or commutativity (Chapter 7)—can also assist here. In either case, by casting correctness in terms of admissible application states instead of (simply) a property of read-write executions over distributed replicas, we achieve a more precise statement of coordination overheads. Finally, when full application invariants are unavailable, individual, high-value transactions may be amenable to optimization via invariant confluence coordination analysis. Accordingly, our development of invariant confluence analysis provides developers with a powerful option—but only if used correctly. If used incorrectly, invariant confluence allows incorrect results, or, if not used at all, developers must resort to existing alternatives.

This final point raises several questions: can we specify invariants in real-world use cases? Classic database concurrency control models assume that "the [set of application invariants] is generally not known to the system but is embodied in the structure of the transaction" [105, 220]. Nevertheless, since 1976, databases have introduced support for a finite set of invariants [52, 115, 120, 126, 145] in the form of primary key, foreign key,

uniqueness, and row-level "check" constraints [161]. We can (and, in the remainder of this dissertation, do) analyze these invariants, which can—like many program analyses [80]—lead to new insights about execution strategies. We have found the process of invariant specification to be non-trivial but feasible in practice. In many cases, specifications for certain invariants already exist but have not been examined in the context of coordination-free implementation.

Physical and logical replication. We have used the concept of replicas to reason about concurrent transaction execution. However, as previously noted, our use of replicas is simply a formal device used to capture concurrency in the system implementation, independent of the actual concurrency control mechanisms at work. Specifically, reasoning about replicas allows us to separate the *analysis* of transactions from their *implementation*: just because a transaction is executed with (or without) coordination does not mean that all query plans or implementations require (or do not require) coordination [32]. Simply because an application is invariant confluent does not mean that all implementations will be coordination-free. Rather, invariant confluence ensures that a coordination-free implementation exists.

(Non-)determinism. Invariant confluence analysis effectively captures points of *unsafe non-determinism* [20] in transaction execution. As we have seen in many of our examples thus far, total non-determinism under concurrent execution can compromise application-level consistency [19, 144]. But not all non-determinism is bad: many desirable properties (e.g., classical distributed consensus among processes) involve forms of acceptable non-determinism (e.g., any proposed outcome is acceptable as long as all processes agree) [124]. In many cases, maximizing concurrency requires admitting the possibility of non-determinism in outcomes.

Invariant confluence analysis allows this non-deterministic divergence of database states but makes two useful guarantees about those states. First, the requirement for global validity ensures safety (in the form of invariants). Second, the requirement for convergence ensures liveness (in the form of convergence). Accordingly, via its use of invariants, invariant confluence allows users to scope non-determinism while only permitting systems to produce "acceptable" states.

3.4 Summary

In this chapter, we developed a framework for determining whether a given safety property can be maintained without coordination while still guaranteeing availability of operations and convergence of data. This invariant confluence generalizes partitioning arguments from distributed systems and allows us to reason about semantic properties at the applica-

tion level. In effect, if the set of database states reachable by applying program operations and merge invocations is closed with respect to declared invariants, a coordination-free implementation of those operations and invariants is possible. In the remainder of this dissertation, we will apply this invariant confluence to several semantic properties found in real database systems.

Chapter 4

Coordination Avoidance and Weak Isolation

In this section, we begin to apply the invariant confluence property to the semantics found in today's database systems. We examine the potential for coordination-free execution of transaction under a number of *weak isolation* guarantees. These non-serializable transaction semantics are widespread use in today's database engines and therefore provide an attractive target for invariant confluence analysis. They are also among the lowest-level semantics we will investigate in this thesis; that is, these guarantees pertain to the admissible interleavings of individual reads and writes to opaque variables, rather than whole program behavior or invariants over database states. Thus, our primary motivation in this section is to examine a range of widely-deployed—if not widely understood—guarantees. Subsequently, we will move upwards in levels of abstraction.

We provide background on the use of weak isolation in Section 4.1. We examine the invariant confluence of these guarantees and provide several coordination-free implementations in Section 4.2. In Section 4.3, we experimentally evaluate their benefits. Section 4.4 presents a more formal model for these guarantees.

4.1 ACID in the Wild

Even within a single-node database, the coordination penalties associated with serializability can be severe. In this context, coordination manifests itself in the form of decreased concurrency, performance degradation, multi-core scalability limitations, and, sometimes, aborts due to deadlock or contention [125]. Accordingly, since the early 1970s, database systems have offered a range of ACID properties weaker than serializability: the host of

so-called *weak isolation* models describe varying restrictions on the space of schedules that are allowable by the system [9, 22, 49]. None of these weak isolation models guarantees serializability, but, as we see below, their benefits to concurrency are frequently considered by database administrators and application developers to outweigh costs of possible consistency anomalies that might arise from their use.

To better understand the prevalence of weak isolation, we surveyed the default and maximum isolation guarantees provided by 18 databases, often claiming to provide "ACID" or "NewSQL" functionality [36]. As shown in Table 4.1, only three out of 18 databases provided serializability by default, and eight did not provide serializability as an option at all. This is particularly surprising when we consider the widespread deployment of many of these non-serializable databases, like Oracle, which are known to power major businesses and product functionality. While we have established that serializability is unachievable in coordination-free systems, the widespread usage of these alternative, weak models indicates that this inability may be of limited importance to applications built upon database systems today. If application writers and database vendors have already decided that the benefits of weak isolation outweigh potential application inconsistencies, then, in a coordination-free system that that prohibits serializability, similar decisions may be tenable.

It has been unknown *which* of these common isolation levels can be provided with coordination-free execution. Existing algorithms for providing weak isolation are often designed for a single-node context and often rely on coordination-based concurrency control mechanisms like locking or mututal exclusion. Moreover, we are not aware of any prior literature that provides guidance as to the relationship between weak isolation and coordination-free execution: prior work has examined the relationship between serializability and coordination-freedom [92] and has studied several variants of weak isolation [9, 49, 125] but not weak isolation and coordination-free execution together.

4.2 Invariant Confluence Analysis: Isolation Levels

In this section, we determine the invariant confluence of a number of these weak isolation guarantees. We also provide a treatment of several distributed consistency models that are complementary to these isolation guarantees. As Brewer states, "systems and database communities are separate but overlapping (with distinct vocabulary)" [63]. With this challenge in mind, we build on existing properties and definitions from the database and distributed systems literature, providing an informal explanation and example for each guarantee.[1] The database isolation guarantees require particular care, since different DBMSs

[1]For clarity, we call these "distributed consistency" guarantees "isolation models" as well.

Database	Default	Maximum
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read		

Table 4.1: Default and maximum isolation levels for ACID and NewSQL databases.

often use the same terminology for different mechanisms and may provide additional guarantees in addition to our implementation-agnostic definitions. We draw largely on Adya's dissertation [9] and somewhat on its predecessor work: the ANSI SQL specification [22] and Berenson et al.'s subsequent critique [49].

In this section, we provide a short proof sketch of each guarantee and provide a more comprehensive treatment in Section 4.4. For each invariant confluent guarantee, we also offer proof-of-concept coordination-free algorithms. These are not necessarily optimal or even efficient: the goal is to illustrate the existence of algorithms. However, we will investigate performance implications in Section 4.3.

Informal model. In our examples, per Adya [9], we exclusively consider read and write operations, denoting a write of version v_i with unique timestamp i drawn from a totally ordered domain (e.g., integers) to data item d as $w_d(v_i)$ and a read from data item d returning v_i as $r_d(v)$. We assume that all data items have the null value, $\perp$, at database initialization, and, unless otherwise specified, all transactions in the examples commit.

In our invariant confluence analysis, we reason about *read-write histories* (or simply *histories*: sets of transactions consisting of read and write operations and their return values. Each history can be represented as a graph of operations with various edges that we describe below; thus, in our invariant confluence model, we reason about the results of transaction executions in the database as histories—or, in effect, traces of transaction execution. In this section, we examine invariants over histories, and the set-oriented nature of histories lends itself naturally to a set-oriented merge. Compared to later analyses (e.g., those in Chapter 6), these are relatively straightforward, and we keep their discussion brief.

Our use of read/write histories is somewhat awkward given our data-centric expression of invariants. However, this model is necessary for compatibility with existing descriptions of weak isolation guarantees. One consolation is that this formalism highlights the unintuitive nature of these guarantees. We study them because they are in widespread use today, but we find them difficult to reason about. As we move towards higher layers of abstraction in later chapters, the higher level specifications are—in our experience—much more intuitive and natural to reason about.

We also assume in this section that the final versions written to each data item within a transaction are assigned the same timestamp. This practice is standard in treatments of multi-version serializability theory [53] but is not actually enforced by Adya's original model. It does not affect the generality of our results[2] but makes several of them clearer.

We begin with invariant confluent guarantees and then present non-invariant confluent guarantees.

4.2.1 Invariant Confluent Isolation Guarantees

To begin, Adya captures **Read Uncommitted** isolation as *PL-1*. In this model, writes to each object are totally ordered, corresponding to the order in which they are installed in the database. In a distributed database, different replicas may receive writes to their local copies of data at different times but should handle concurrent updates (i.e., overwrites) in accordance with the total order for each item. *PL-1* requires that writes to different objects be ordered consistently across transactions, prohibiting Adya's phenomenon G0 (also called "Dirty Writes" [49]). If we build a graph of transactions with edges from one transaction to another when the former overwrites the latter's write to the same object then, under Read

[2]Namely, if we draw timestamps from an infinite domain, we can partition the ID space among replicas. In an implementation, this may require new servers joining a cluster to obtain a subset of the ID space from at least one existing cluster member. In a practical implementation, this could require a substantial number of bits for timestamp allocation. However, at an extreme, 256 bits support extreme transaction volumes.

Uncommitted, the graph should not contain cycles [9]. Consider the following example:

$$T_1 : w_x(1)\ w_y(1)$$
$$T_2 : w_x(2)\ w_y(2)$$

In this example, under Read Uncommitted, it is unacceptable for the database to both order T_1's $w_x(1)$ before T_2's $w_x(2)$ and order T_2's $w_y(2)$ before T_1's $w_y(1)$.

Read Uncommitted is invariant confluent under read and write operations; we provide a sketch here. The G0 phenomenon described above only pertains to write operations, so we do not consider read operations. With unique and totally ordered timestamps, G0 graphs are acyclic by construction. To show this, consider any two transactions T_i and T_j appearing in a G0 graph G, and the final versions of each item produced by T_i (timestamped k) and the final versions of each item produced by T_j (timestamped l, either $k < l$ or $l < k$. In either case, G is acyclic. Merging two acyclic graphs does not produce new transactions, and all transactions within the two graphs will similarly be totally ordered.

Because Read Uncommitted is invariant confluent, we can find a coordination-free implementation. (However, note that traditional implementations such as the lock-based implementation due to Gray in the original formulation of Read Uncommitted [125]), do require coordinate.) Read Uncommitted is easily achieved by marking each of a transaction's writes with the same timestamp (unique across transactions; e.g., combining a client's ID with a sequence number) and applying a "last writer wins" conflict reconciliation policy at each replica. Later properties will strengthen Read Uncommitted.

Read Committed isolation is particularly important in practice as it is the default isolation level of many DBMSs (Section 4.1). Centralized implementations differ, with some based on long-duration exclusive locks and short-duration read locks [125] and others based on multiple versions. These implementations often provide properties beyond what is implied by the name "Read Committed" and what is captured by the implementation-agnostic definition. However, under the implementation-independent definition of Read Committed, transactions should not access uncommitted or intermediate (i.e., non-final) versions of data items. This prohibits both "Dirty Writes", as above, and also "Dirty Reads" phenomena. This isolation is Adya's *PL-2* and is formalized by prohibiting Adya's *G1{a-c}* (or ANSI's P1, or "broad" P1 [2.2] from Berenson et al.). For instance, in the example below, T_3 should never read $a = 1$, and, if T_2 aborts, T_3 should not read $a = 3$:

$$T_1 : w_x(1)\ w_x(2)$$
$$T_2 : w_x(3)$$
$$T_3 : r_x(a)$$

Read Committed is also invariant confluent for read and write operations. The basic idea is relatively straightforward: if two histories do not contain reads of uncommitted or intermediate versions of data items, unioning them will not introduce additional reads of uncommitted or intermediate versions, nor will unioning them change any of the values of that any of the reads returned (preventing G1a and G1b). Adya's G1c is prevented by the fact that G0 graphs are acyclic for writes (as above) and transactions can only read-depend on other transactions that appear in their histories; thus, unioning two histories that do not exhibit G1c cannot introduce new edges to incur G1c.

Because Read Committed is invariant confluent, we can find a coordination-free implementation: if no client ever writes uncommitted data to shared copies of data, then transactions will never read each others' dirty data. As a simple solution, clients can buffer their writes until they commit, or, alternatively, can send them to servers, who will not deliver their value to other readers until notified that the writes have been committed. Unlike a lock-based implementation, this implementation does not guarantee that readers observe the most recent write to a data item, but it the implementation-agnostic definition.

Several different properties have been labeled **Repeatable Read** isolation. As we will show in Section 4.2.3, some of these are not achievable in a coordination-free system. However, the ANSI-standard, implementation-agnostic definition [22] *is* achievable and directly captures the spirit of the term: if a transaction reads the same data more than once, it sees the same value each time (preventing "Fuzzy Read," or P2). In this paper, to disambiguate between other definitions of "Repeatable Read," we will call this property "cut isolation," since each transaction reads from a non-changing cut, or snapshot, over the data items. If this property holds over reads from a set of individual data items, we call it **Item Cut Isolation**, and, if we also expect a cut over predicate-based reads (e.g., `SELECT WHERE`; preventing Phantoms [125], or Berenson et al.'s P3/A3), we have the stronger property of **Predicate Cut-Isolation**. In the example below, under both levels of cut isolation, T_3 must read $a = 1$:

$$T_1 : w_x(1)$$
$$T_2 : w_x(2)$$
$$T_3 : r_x(1)\ r_x(a)$$

Item Cut Isolation is invariant confluent for reasoning similar to Read Committed: if two histories are valid under Item Cut Isolation, unioning them will not change the return values of reads.

Because Item Cut Isolation is invariant confluent, we can find a coordination-free implementation: we can have transactions store a copy of any read data at the client such that the value that they read for each item never changes unless they overwrite it themselves.

These stored values can be discarded at the end of each transaction and can alternatively be accomplished on servers via multi-versioning. Predicate Cut Isolation is also achievable in coordination-free systems via similar caching middleware or multi-versioning that tracks entire logical ranges of predicates in addition to item based reads.

ACID Atomicity Guarantees

Atomicity, informally guaranteeing that either all or none of transactions' effects should succeed, is core to ACID guarantees. Although, at least by the ACID acronym, atomicity is not an "isolation" property, atomicity properties also restrict the updates visible to other transactions. Accordingly, here, we consider the *isolation* effects of atomicity, which we call **Monotonic Atomic View** (**MAV**) isolation. Under MAV, once some of the effects of a transaction T_i are observed by another transaction T_j, thereafter, all effects of T_i are observed by T_j. That is, if a transaction T_j reads a version of an object that transaction T_i wrote, then a later read by T_j cannot return a value whose later version is installed by T_i. Together with item cut isolation, MAV prevents Read Skew anomalies (Berenson et al.'s A5A) and is useful in several contexts such as maintaining foreign key constraints, consistent global secondary indexing, and maintenance of derived data. In the example below, under MAV, because T_2 has read T_1's write to y, T_2 must observe $b = c = 1$ (or later versions for each key):

$$T_1 : w_x(1)\ w_y(1)\ w_z(1)$$
$$T_2 : r_x(a)\ r_y(1)\ r_x(b)\ r_z(c)$$

T_2 can also observe $a = \bot$, $a = 1$, or a later version of x. In the hierarchy of existing isolation properties, we place MAV below Adya's *PL-2L* (as it does not necessarily enforce transitive read-write dependencies) but above Read Committed ($PL - 2$). Notably, MAV requires disallows reading intermediate writes (Adya's G1b): observing all effects of a transaction implicitly requires observing the final (committed) effects of the transaction as well.

Perplexingly, discussions of MAV are absent from existing treatments of weak isolation. This is perhaps again due to the single-node context in which prior work was developed: on a single server (or a fully replicated database), MAV is achievable via lightweight locking and/or local concurrency control over data items [90,143]. In contrast, in a distributed environment, MAV over arbitrary groups of non-co-located items is considerably more difficult to achieve with coordination-free execution.

MAV is invariant confluent for read and write operations. The reasoning is similar to Read Committed and Item Cut Isolation: if two histories obey MAV, then their union does

not change the effects of the reads in each history, and therefore every transaction T in the unioned history reads from the same transactions in the original history in which T was originally found, and T will not miss the effects of transactions it depends upon.

Because MAV is invariant confluent, we can find a coordination-free implementation. Due to its utility, we develop an advanced set of implementations of MAV in Chapter 5.

Session Guarantees

A useful class of safety guarantees refer to real-time or client-centric ordering within a *session*, "an abstraction for the sequence of...operations performed during the execution of an application" [216]. These "session guarantees" have been explored in the distributed systems literature [216, 224] and sometimes in the database literature [91]. For us, a session describes a context that should persist between transactions: for example, on a social networking site, all of a user's transactions submitted between "log in" and "log out" operations might form a session. Session guarantees are often expressed in a non-transactional context, and there are several ways to extend them to transactions. Per [91], we examine them in terms of ordering across transactions.

Several session guarantees can be made with coordination-free execution. We describe them informally below:

The **monotonic reads** guarantee requires that, within a session, subsequent reads to a given object "never return any previous values"; reads from each item progress according to a total order (e.g., the order from Read Uncommitted).

The **monotonic writes** guarantee requires that each session's writes become accessible to other transactions in the order they were committed. Any order on transactions (as in Read Uncommitted isolation) should also be consistent with any precedence (e.g., Adya's ordering on versions of each item) that a global oracle would observe.

The **writes follow reads** guarantee requires that, if a session observes an effect of transaction T_1 and subsequently commits transaction T_2, then another session can only observe effects of T_2 if it can also observe T_1's effects (or later values that supersede T_1's); this corresponds to Lamport's "happens-before" relation [157]. Any order on transactions should respect this transitive order.

The above guarantees are invariant confluent for reads and writes and can be achieved by forcing servers to wait to reveal new writes (say, by buffering them in separate local storage) until each write's respective dependencies are visible on all replicas. This mechanism effectively ensures that all clients read from a globally agreed upon lower bound on the

versions written. This is coordination-free because a client will never block due to inability to find a server with a sufficiently up-to-date version of a data item. However, it does not imply that transactions within a session will observe previous updates within the session or, in the presence of partitions, make forward progress through the version history. The problem is that under our standard definition of transactional availability, a system must handle the possibility that, under a partition, an unfortunate client will be forced to issue its next requests against a partitioned, out-of-date server.

4.2.2 Sticky Availability

To address the above concern, we can introduce the concept of "stickiness": clients can ensure continuity between operations (e.g., reading their prior updates to a data item) by maintaining affinity or "stickiness" with a server or set of servers [224]. In a fully replicated system, where all servers are replicas for all data items, stickiness is simple: a client can maintain stickiness by contacting the same server for each of its requests. However, to stay "sticky" in a partially-replicated system, where servers are replicas for subsets of the set of data items (which we consider in this paper), a client must maintain stickiness with a single *logical* copy of the database, which may consist of multiple physical servers. We say that a system provides **sticky availability** if, whenever a client's transactions is executed against a copy of database state that reflects all of the client's prior operations, it eventually receives a response, even in the presence of indefinitely long partitions (where "reflects" is dependent on semantics). A client may choose to become sticky available by acting as a server itself; for example, a client might cache its reads and writes [39,216,233]. Any guarantee achievable in a coordination-free system is achievable in a sticky coordination-free system but not vice-versa. In the above example, if the client remains sticky with the server that executed T_1, then the client can read its writes. While sticky availability is implicit in prior work, we believe this is one of the first instances where it is discussed in detail.

Sticky availability permits three additional guarantees, which we first define and then show are unavailable in a generic coordination-free system:

Read your writes requires that whenever a session reads a given data item d after writing a version d_i to it, the read returns the d_i or another version d_j, where $j > i$.

PRAM (Pipelined Random Access Memory) provides the illusion of serializing each of the operations (both reads and writes) within each session and is the combination of monotonic reads, monotonic writes, and read your writes [134].

Causal consistency [13] results from the combination of all session guarantees [67] (i.e., PRAM with writes-follow-reads) and is also referred to by Adya as *PL-2L* isolation [9]).

Read your writes is not achievable in a coordination-free system. Consider a client that executes the following two transactions:

$$T_1 : w_x(1)$$
$$T_2 : r_x(a)$$

If the client executes T_1 against a server that is partitioned from the rest of the other servers, then, for transactional availability, the server must allow T_1 to commit. If the same client subsequently executes T_2 against the same (partitioned) server in the same session, then it will be able to read its writes. However, if the network topology changes and the client can only execute T_2 on a different replica that is partitioned from the replica that executed T_1, then the system will have to either stall indefinitely to allow the client to read her writes (violating transactional availability) or will have to sacrifice read your writes guarantees.

Accordingly, read your writes, and, by proxy, causal consistency and PRAM require stickiness. Read your writes is provided by default in a sticky system. Causality and PRAM guarantees can be accomplished with well-known variants [13, 39, 168, 216, 233] of the prior session guarantee algorithms we presented earlier: only reveal new writes to clients when their (respective, model-specific) dependencies have been revealed.

4.2.3 Non-Invariant Confluent Semantics

At this point, we have considered most of the previously defined (and useful) isolation guarantee that are available to coordination-free systems. Before summarizing our possibility results, we will present impossibility results, also defined in terms of previously identified isolation anomalies. Most notably, it is impossible to prevent Lost Update or Write Skew in a coordination-free system.

Unachievable ACID Isolation

In this section, we demonstrate that preventing Lost Update and Write Skew—and therefore providing Snapshot Isolation, Repeatable Read, and one-copy serializability—inherently requires foregoing high availability guarantees.

Berenson et al. define *Lost Update* as when one transaction T1 reads a given data item, a second transaction T2 updates the same data item, then T1 modifies the data item based on its original read of the data item, "missing" or "losing" T2's newer update. Consider a

database containing only the following transactions:

$$T_1 : r_x(a)\ w_x(a+2)$$
$$T_2 : w_x(2)$$

If T_1 reads $a = 1$ but T_2's write to x precedes T_1's write operation, then the database will end up with $a = 3$, a state that could not have resulted in a serial execution due to T_2's "Lost Update."

It is impossible to prevent Lost Update in a highly available environment. Consider two clients who submit the following T_1 and T_2 as part of two separate histories H_1 and H_2.

$$T_1 : r_x(100)\ w_x(100 + 20 = 120)$$
$$T_2 : r_x(100)\ w_x(100 + 30 = 130)$$

Regardless of whether $x = 120$ or $x = 130$ is chosen by a replica, the database state could not have arisen from a serial execution of H_1 and H_2.[3] To prevent this, either T_1 or T_2 should not have committed. Each client's respective server might try to detect that another write occurred, but this requires knowing the version of the latest write to x. In our example, this reduces to a requirement for linearizability, which is, via Gilbert and Lynch's proof of the CAP Theorem, provably at odds with coordination-free execution [118].

Write Skew is a generalization of Lost Update to multiple keys. It occurs when one transaction T1 reads a given data item x, a second transaction T2 reads a different data item y, then T1 writes to y and commits and T2 writes to x and commits. As an example of Write Skew, consider the following two transactions:

$$T_1 : r_y(0)\ w_x(1)$$
$$T_2 : r_x(0)\ w_y(1)$$

As Berenson et al. describe, if there was an integrity constraint between x and y such that only one of x or y should have value 1 at any given time, then this write skew would violate the constraint (which is preserved in serializable executions). Write skew is a somewhat esoteric anomaly—for example, it does not appear in TPC-C [110]—but can result in improper behavior in many unexpected scenarios, such as transfers from one bank account to another [110]. As a generalization of Lost Update, Write Skew is also unavailable to coordination-free systems.

[3] In this example, we assume that, as is standard in modern databases, databases accept values as they are written (i.e., register semantics). However, if we replace the write operation with a richer semantics, like "increment," we can avoid this issue. However, the problem persists in the general case.

Consistent Read, Snapshot Isolation (including Parallel Snapshot Isolation [210]), and Cursor Stability guarantees are all unavailable because they require preventing Lost Update phenomena. Repeatable Read (defined by Gray [125], Berenson et al. [49], and Adya [9]) and One-Copy Serializability [26] need to prevent both Lost Update and Write Skew. Their prevention requirements mean that these guarantees are inherently unachievable in a coordination-free system.

Unachievable Recency Guarantees

Distributed data storage systems often make various recency guarantees on reads of data items. Unfortunately, an indefinitely long partition can force an available system to violate any recency bound, so recency bounds are not enforceable by coordination-free systems [118]. One of the most famous of these guarantees is linearizability [134], which states that reads will return the last completed write to a data item. There are also several other (weaker) variants such as safe and regular register semantics. When applied to transactional semantics, the combination of one-copy serializability and linearizability is called *strong (or strict) one-copy serializability* [9] (e.g., Spanner [85]). It is also common, particularly in systems that allow reading from masters and slaves, to provide a guarantee such as "read a version that is no more than five seconds out of date" or similar. None of these guarantees are invariant confluent.

Durability

A client requiring that its transactions' effects survive F server faults requires that the client be able to contact at least $F + 1$ non-failing replicas before committing. This affects availability and, according to the model we have adopted, $F > 1$ fault tolerance is not achievable in an invariant confluent system. However, with minor modifications to the model, we can ensure durability at a penalty to operation availability. Namely, if we modify the definition of transactional availability (Definition 2) such that a transaction terminates if it can access at least $F+1$ servers for each item its transaction, the remainder of our results hold for systems with at least $F + 1$ physical servers. This is a moderately subtle concept, but an important consequence is that communication overhead associated with ensuring durability is a function of F and not of the physical cluster size. Thus, if a cluster doubles in size from fifty to one hundred physical servers, the overheads due to durability are constant, whereas the overheads for, say, a majority consensus algorithm will double (as the majority is a function of the number of nodes).

4.2.4 Summary

As we summarize in Table 4.2, a wide range of isolation levels are achievable in coordination-free systems. With sticky availability, a system can achieve read your writes guarantees, PRAM, and causal consistency. However, many other prominent semantics, such as Snapshot Isolation, One-Copy Serializability, and Strong Serializability cannot be achieved due to the inability to prevent Lost Update and Write Skew phenomena.

We illustrate the hierarchy of invariant confluent, sticky, and non-invariant confluent isolation models we have discussed in Figure 4.1. Many models are simultaneously achievable, but we find several particularly compelling. If we combine all coordination-free and sticky guarantees, we have transactional, causally consistent snapshot reads (i.e., Causal Transactional Predicate Cut Isolation). If we combine MAV and P-CI, we have transactional snapshot reads (see Chapter 5). We can achieve RC, MR, and RYW by simply sticking clients to servers. We can also combine unavailable models—for example, an unavailable system might provide PRAM and One-Copy Serializability [91].

To the best of our knowledge, this is the first unification of transactional isolation, distributed consistency, and session guarantee models. Interestingly, strong one-copy serializability subsumes all other models, while considering the (large) power set of all compatible models (e.g., the diagram depicts 144 possible coordination-free combinations) hints at the vast expanse of consistency models found in the literature. This taxonomy is not exhaustive, but we believe it lends substantial clarity to the relationships between a large subset of the prominent ACID and distributed consistency models.

In light the of current practice of deploying weak isolation levels (Section 4.1), it is perhaps surprising that so many weak isolation levels are achievable with coordination-freedom. Indeed, isolation levels such as Read Committed expose and are defined in terms of end-user anomalies that could not arise during serializable execution. However, the widespread usage of these models suggests that, in many cases, applications can tolerate these their associated anomalies. In turn, our results suggest that–despite idiosyncrasies relating to concurrent updates and data recency–coordination-free database systems can provide sufficiently strong semantics for many applications. For non-invariant confluent semantics, coordination-free databases may expose more anomalies (e.g., linearizability violations) than a single-site database (particularly during network partitions). However, for, invariant confluent isolation levels, users of single-site databases are subject to the same (worst-case) application-level anomalies as a coordination-free implementation. The necessary (indefinite) visibility penalties (i.e., the right side of Figure 4.1) and lack of support for preventing concurrent updates (via the upper left half of Figure 4.1) mean coordination-free systems are *not* well-suited for all applications (see Section 4.3): these limitations are fundamental. However, common practices such as ad-hoc, user-level compensation

Invariant Confluent	Read Uncommitted (RU), Read Committed (RC), Monotonic Atomic View (MAV), Item Cut Isolation (I-CI), Predicate Cut Isolation (P-CI), Writes Follow Reads (WFR), Monotonic Reads (MR), Monotonic Writes (MW)
Sticky	Read Your Writes (RYW), PRAM, Causal
Unavailable	Cursor Stability (CS)†, Snapshot Isolation (SI)†, Repeatable Read (RR)†‡, One-Copy Serializability (1SR)†‡, Recency$^{\oplus}$, Safe$^{\oplus}$, Regular$^{\oplus}$, Linearizability$^{\oplus}$, Strong 1SR$^{\dagger\ddagger\oplus}$

Table 4.2: Summary of invariant confluent, sticky, and non-invariant confluent models considered in this paper. Non-invariant confluent models are labeled by cause: preventing lost update†, preventing write skew‡, and requiring recency guarantees$^{\oplus}$.

and per-statement isolation "upgrades" (e.g., `SELECT FOR UPDATE` under weak isolation)—commonly used to augment weak isolation—are also applicable in coordination-free systems (although they may in turn compromise availability). That is, if an application already selectively and explicitly opts-in to coordination via SQL keywords like `SELECT FOR UPDATE`, a coordination-avoiding system can similarly use these hints as a basis for understanding when coordination is required.

4.3 Implications: Existing Algorithms and Empirical Impact

Given our understanding of which isolation models are invariant confluent, in this section, we analyze the implications of these results for existing systems and study coordination-free implementations on public cloud infrastructure. Specifically, we revisit traditional database concurrency control with a focus on coordination costs and on coordination-free execution. We also perform a experimental evaluation of coordination-free versus non-coordination-free properties on public cloud infrastructure.

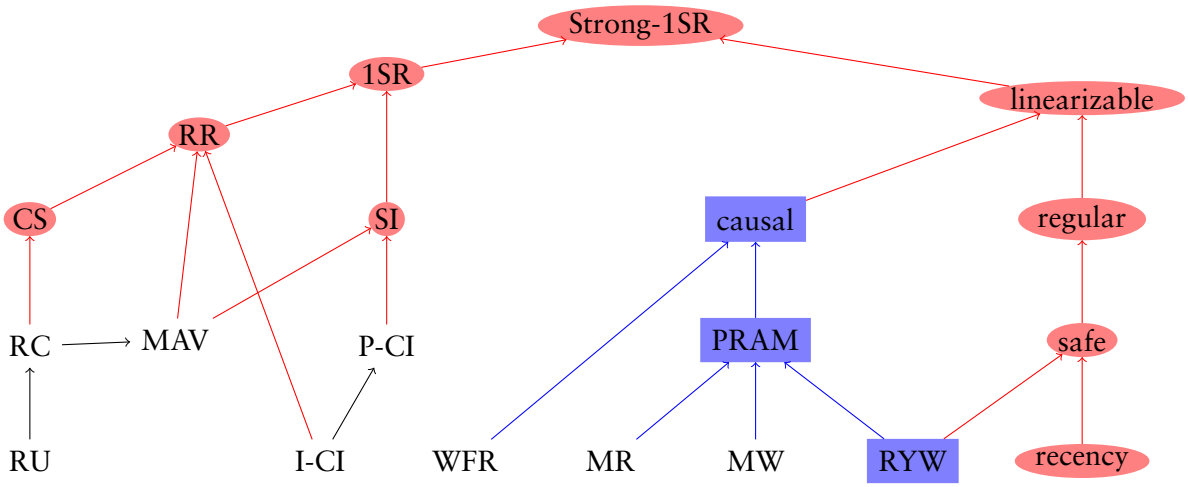


Figure 4.1: Partial ordering of invariant confluent, sticky (in boxes), and non-invariant confluent models (circled) from Table 4.2. Directed edges represent ordering by model strength. Incomparable models can be simultaneously achieved, and the availability of a combination of models has the availability of the least available individual model.

4.3.1 Existing Algorithms

While we have shown that the semantics of many database isolation levels are achievable with coordination-freedom, many traditional concurrency control mechanisms do not provide coordination-free execution—even for invariant confluent isolation levels. Existing mechanisms often presume (or are adapted from) single-server, non-partitioned deployments or are otherwise adapted from mechanisms that enforce serializability as a primary use case.

Most existing implementations of weak isolation are not coordination-free. Lock-based mechanisms such as those in Gray's original proposal [125] do not degrade gracefully in the presence of partial failures. (Note, however, that lock-based protocols *do* offer the benefit of recency guarantees.) While multi-versioned storage systems allow for a variety of transactional guarantees, few offer traditional weak isolation (e.g., non-"tentative update" schemes) in this context. Chan and Gray's read-only transactions have item-cut isolation with causal consistency and MAV (session *PL-2L* [9]) but are unavailable in the presence of coordinator failure and assume serializable update transactions [73]; this is similar to read-only and write-only transactions more recently proposed by Eiger [168]. Brantner's S3 database [62] and Bayou [216] can all provide variants of session *PL-2L* with coordination-free execution, but none provide this coordination-free functionality without substantial modification. Accordingly, it is possible to implement many guarantees weaker than serializability—including coordination-free semantics—and still not achieve a

coordination-free implementation. Thus, coordination-free execution must often be explicitly considered in concurrency control designs.

4.3.2 Empirical Impact: Isolation Guarantees

To investigate the performance implications of coordination-free weak isolation implementation in a real-world environment, we implemented a database prototype of several of the guarantees in this chapter. We verify that, as Chapter 2's measurements suggested, "strongly consistent" algorithms incur substantial latency penalties (over WAN, 10 to 100 times higher than their coordination-free counterparts) compared to coordination-free-compliant algorithms, which scale linearly. Our goal is *not* a complete performance analysis of coordination-free semantics but instead a demonstration of coordination-free designs on real-world infrastructure.

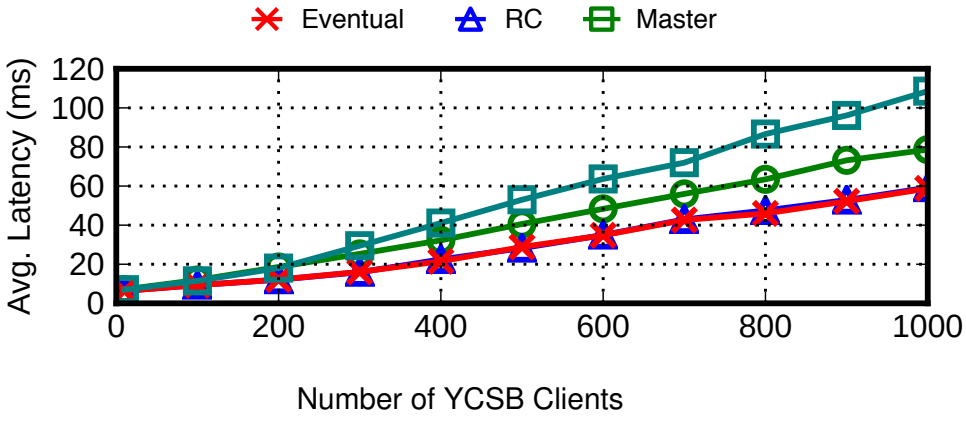
Implementation. Our prototype database is a partially replicated (hash-based partitioned) key-value backed by LevelDB and implemented in Java using Apache Thrift. It supports eventual consistency (hereafter, `eventual`) using a last-writer-wins reconciliation policy for concurrent writes, effectively providing Read Uncommitted replication, via standard all-to-all anti-entropy between replicas. We support non-coordination-free operation whereby all operations for a given key are routed to a (randomly) designated `master` replica for each key (guaranteeing single-key linearizability, as in Gilbert and Lynch's CAP Theorem proof [118] and in PNUTS [83]'s "read latest" operation; hereafter, `master`) as well as distributed two-phase locking. Servers are durable: they synchronously write to LevelDB before responding to client requests.

Configuration. We deploy the database in *clusters*—disjoint sets of database servers that each contain a single, fully replicated copy of the data—across one or more datacenters and stick all clients within a datacenter to their respective cluster (trivially providing read-your-writes and monotonic reads guarantees). By default, we deploy 5 Amazon EC2 `m1.xlarge` instances (15GB RAM, with 4 cores comprising 8 "EC2 Compute Units") as servers in each cluster. For our workload, we link our client library to the YCSB benchmark [84], which is well suited to LevelDB's key-value schema, grouping every eight YCSB operations from the default workload (50% reads, 50% writes) to form a transaction. We increase the number of keys in the workload from the default 1,000 to 100,000 with uniform random key access, keeping the default value size of 1KB, and running YCSB for 180 seconds per configuration.

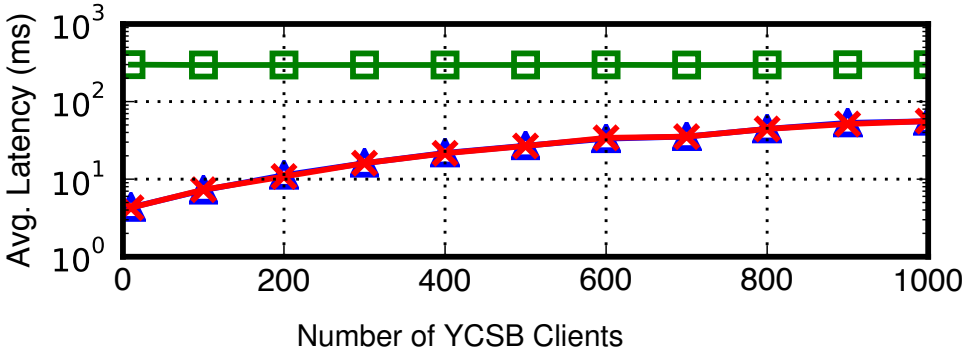
Geo-replication. We first deploy the database prototype across an increasing number of datacenters. Figures 4.2A and 4.3A shows that, when operating two clusters within a

single datacenter, mastering each data item results in approximately half the throughput and double the latency of `eventual`. This is because coordination-free models are able to utilize replicas in both clusters instead of always contacting the (single) master. `RC`—essentially `eventual` with buffering—is almost identical to `eventual`. Latency increases linearly with the number of YCSB clients.

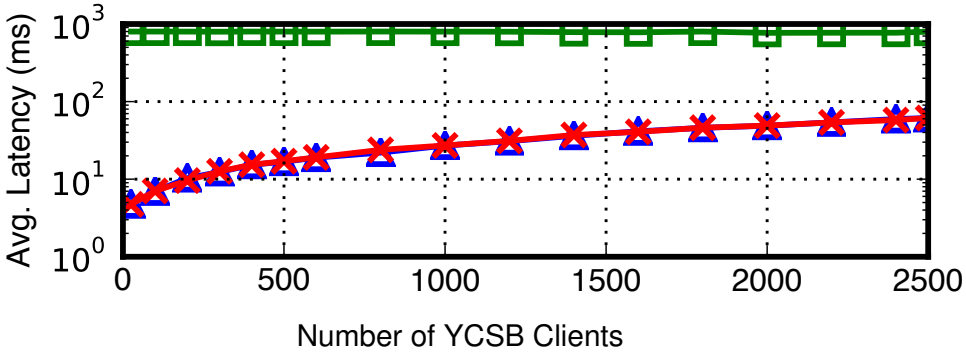
In contrast, when the two clusters are deployed across the continental United States (Figures 4.2B and 4.3B), the average latency of `master` increases to 300ms (a 278–4257% latency increase; average 37ms latency per operation). For the same number of YCSB client threads, `master` has substantially lower throughput than the coordination-free configurations. Increasing the number of YCSB clients *does* increase the throughput of `master`, but our Thrift-based server-side connection processing did not gracefully handle more than several thousand concurrent connections. In contrast, across two datacenters, the performance of eventual and RC are near identical to a single-datacenter deployment.



A.) Within us-east (VA)



B.) Between us-east (CA) and us-west-2 (OR)



C.) Between us-east (VA), us-west-1 (CA),
us-west-2 (OR), eu-west (IR), ap-northeast (SI)

Figure 4.2: YCSB latency for two clusters of five servers each deployed within a single datacenter and cross-datacenters (note log scale for multi-datacenter deployment).

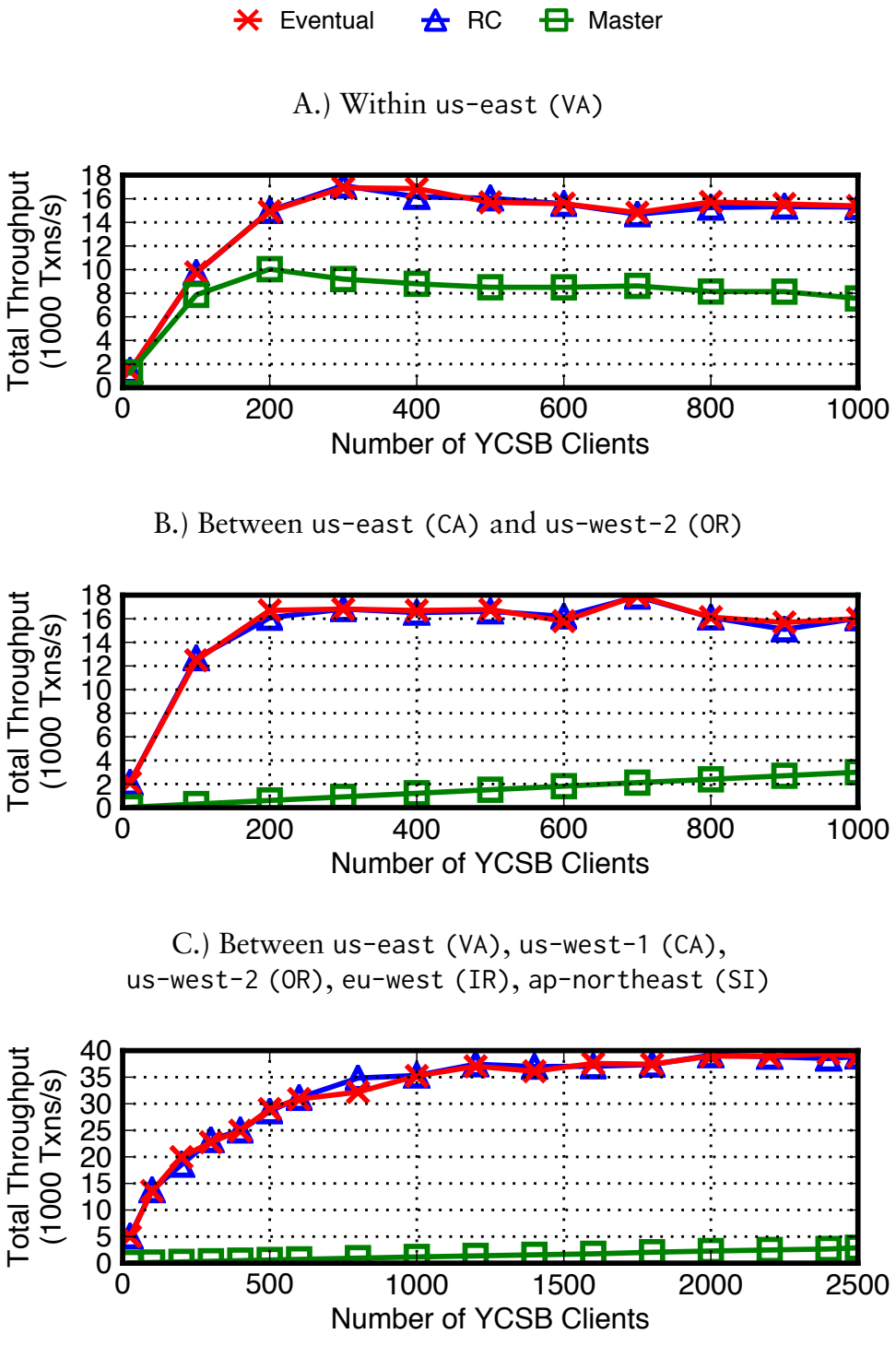


Figure 4.3: YCSB throughput for two clusters of five servers each deployed within a single datacenter and cross-datacenters.

When five clusters (as opposed to two, as before) are deployed across the five EC2 datacenters with lowest communication cost (Figures 4.2C and 4.3C), the trend continues: `master` latency increases to nearly 800ms per transaction. As an attempt at reducing this overhead, we implemented and benchmarked a variant of quorum-based replication as in Dynamo [95], in which clients sent requests to all replicas, which completed as soon as a majority of servers responded (guaranteeing regular semantics [134]). This strategy (not pictured) did not substantially improve performance due to the network topology and because worst-case server load was unaffected.

Because `master` performed far better than our textbook implementation, we have omitted performance data for two-phase locking. In addition to incurring the same WAN round-trip latency, locking also incurred substantial overheads due to mutual exclusion. While techniques such as those recently proposed in Calvin [217] can reduce the overhead of serializable transactions by avoiding locking, our mastered implementation and the data from Section 2.2.1 are reasonable lower bounds on latency.

Read proportion. Our default (equal) proportion of reads and writes is fairly pessimistic: for example, Facebook reports 99.8% reads for their workload [168]. As Figure 4.4 demonstrates, increasing the proportion of reads increases throughput; this is due to the decreased cost of read operations on each node, and `RC` and `eventual` stay matched in terms of throughput.

Scale-out. One of the key benefits of our coordination-free algorithms is that they are shared-nothing [214], meaning they should not compromise scalability. Figure 6.2 shows that varying the number of servers across two clusters in Virginia and Oregon (with 15 YCSB clients per server) results in linear scale-out for `eventual` and `RC`. `RC` and `eventual` scale linearly: increasing the number of servers per cluster from 5 to 25 yields an approximately 5x throughput increase.

Summary. Our experimental prototype confirms our earlier analytical intuition. Coordination-free systems can provide useful semantics without substantial performance penalties. Our coordination-free algorithms circumvent high WAN latencies inevitable with non-coordination-free implementations. Our results highlight Deutsch's observation that ignoring factors such as latency can "cause big trouble and painful learning experiences" [97]—in a single-site context, paying the cost of coordination may be tenable, but, especially as services are geo-replicated, costs increase. In the next chapter, we examine a more sophisticated set of implementations.

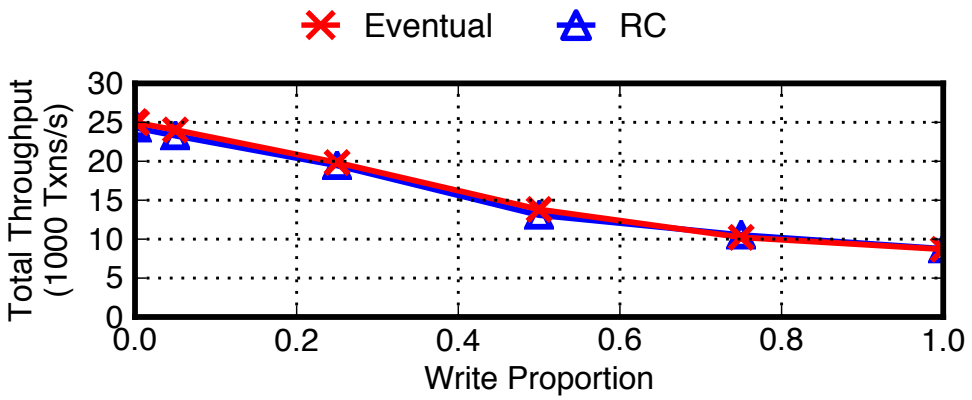


Figure 4.4: Proportion of reads and writes versus throughput.

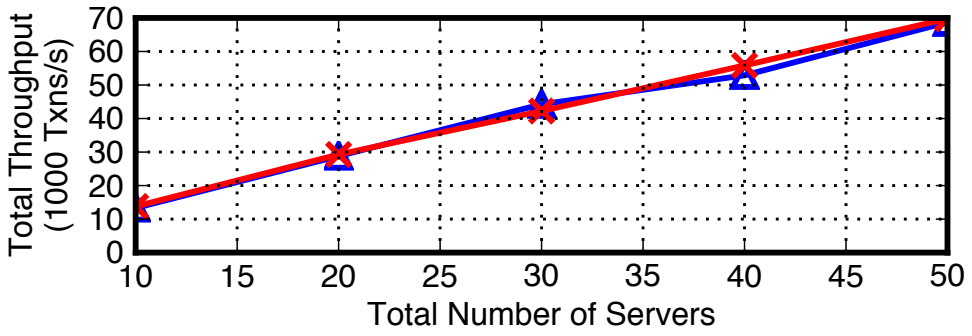


Figure 4.5: Scale-out of Eventual and RC.

4.4 Isolation Models

In this section, we formally define coordination-free transactional semantics as a reference for the previous section. Our formalism is based on that of Adya [9]. For the reader familiar with his formalism, this is a mostly-straightforward exercise combining transactional models with distributed systems semantics. While the novel aspects of this work largely pertain to *using* these definitions, we believe it is instructive to accompany them by appropriate definitions as well.

To begin, we describe our model of weakly isolated transactions. This is, with the exception of sessions, identical to that of Adya [9]. We omit a full duplication of his formalism here but highlight several salient criteria. We refer the interested reader to Adya's Ph.D. thesis, Section 3.1 (pp. 33–43).

Users submit transactions to a database system that contains a set of objects, each of

which is represented by multiple distinct versions. Each transaction is composed of writes, which create new versions of an object, and reads, which return a written version or the initial version of the object. A transaction's last operation is either `commit` or `abort`, and hence there is exactly one invocation of either these two operations per transaction. Transactions can either read individual items or read based on predicates (or logical "ranges") of data items.

Definition 7 (Version set of a predicate-based operation). *When a transaction executes a read or write based on a predicate P, the system selects a version for each tuple in PâĂŹs relations. The set of selected versions is called the Version set of this predicate-based operation and is denoted by Vset(P).*

A history over transactions has two parts: first, a partial order of events, comprised of several different types of edges that we describe below, reflects the ordering of operations with respect to each transaction, and, second, a order ($\ll$) on the committed versions of each object. The history forms a graph, with nodes corresponding to either transactions or operations within a transaction, and edges constructed as we describe below.

As a departure from Adya's formalism, to capture the use of session guarantees, we allow transactions to be grouped into sessions. We represent sessions as a partial ordering on committed transactions such that each transaction in the history appears in at most one session.

Framework

To reason about isolation anomalies, we use Adya's concept of a conflict graph, which is composed of dependencies between transaction. The definitions in this section are directly from Adya, with two differences. First, we expand Adya's formalism to deal with *per-item* dependencies. Second, we define session dependencies (Definition 20).

Definition 8 (Change the Matches of a Predicate-Based Read). *A transaction* T_i *changes the matches of a predicate-based read* r_j*(P:Vset(P)) if* T_i *installs* x_i, x_h *immediately precedes* x_i *in the version order, and* x_h *matches P whereas* x_i *does not or vice-versa. In this case, we also say that* x_i *changes the matches of the predicate-based read. The above definition identifies* T_i *to be a transaction where a change occurs for the matched set of* r_j *(P: Vset(P)).*

Definition 9 (Directly item-read-depends by x). T_j *directly item-read-depends on transaction* T_i *if* T_i *installs some object version* x_i *and* T_j *reads* x_i.

Definition 10 (Directly item-read-depends). T_j *directly item-read-depends on transaction* T_i *if* T_j *directly item-read-depends by* x *on* T_i *for some data item* x.

Definition 11 (Directly predicate-read-depends by P). *Transaction* T_j *directly predicate-read-depends by* P *on transaction* T_i *if* T_j *performs an operation* r_j*(P: Vset(P))*, $x_k \in$ *Vset(P),* $i = k$ *or* $x_i \ll x_k$ *, and* x_i *changes the matches of* r_j *(P: Vset(P)).*

Definition 12 (Directly Read-Depends [9, Definition 2]). T_j *directly read-depends on transaction* T_i *if it directly item-read-depends or directly predicate-read-depends on* T_i.

Definition 13 (Directly predicate-read-depends). T_j *directly predicate-read-depends on* T_i *if* T_j *directly predicate-read-depends by* P *on* T_i *for some predicate* P.

Definition 14 (Directly item-anti-depends by x). T_j *directly item-anti-depends by* x *on transaction* T_i *if* T_i *reads some object version* x_k *and* T_j *installs* x*'s next version (after* x_k*) in the version order. Note that the transaction that wrote the later version directly item-anti-depends on the transaction that read the earlier version.*

Definition 15 (Directly item-anti-depends). T_j *directly item-anti-depends on transaction* T_i *if* T_j *directly item-anti-depends on transaction* T_i.

Definition 16 (Directly predicate-anti-depends by P). T_j *directly predicate-anti-depends by* P *on transaction* T_i *if* T_j *overwrites an operation* r_i(P : *Vset(P)). That is, if* T_j *installs a later version of some object that changes the matches of a predicate based read performed by* T_i.

Definition 17 (Directly Anti-Depends [9, Definition 4]). *Transaction* T_j *directly anti-depends on transaction* T_i *if it directly item-anti-depends or directly predicate-anti-depends on* T_i.

Definition 18 (Directly Write-Depends by x). *A transaction* T_j *directly write-depends by* x *on transaction* T_i *if* T_i *installs a version* x_i *and* T_j *installs* x*'s next version (after* x_i*) in the version order.*

Definition 19 (Directly Write-Depends [9, Definition 5]). *A transaction* T_j *directly write-depends on transaction* T_i *if* T_i *directly write-depends by* x *on* T_j *for some item* x.

Definition 20 (Session-Depends). *A transaction* T_j *session-depends on transaction* T_i *if* T_i *and* T_j *occur in the same session and* T_i *precedes* T_j *in the session commit order.*

The dependencies for a history H form a graph called its Directed Serialization Graph (DSG(H)). If T_j directly write-depends on T_i by x, we draw $T_i \xrightarrow{ww_x} T_j$. If T_j read-depends on T_i by x, we draw $T_i \xrightarrow{wr_x} T_j$. If T_j directly anti-depends on transaction T_j by x, we draw $T_i \dashrightarrow^{rw_x} T_j$. If T_j session-depends on T_i in session S, we draw $T_i \xrightarrow{S} T_j$ [9, Definition 8].

We also consider the Unfolded Serialization Graph (USG(H)) that is a variation of the DSG. The USG is specified for the transaction of interest, T_i, and a history, H, and is denoted by USG(H, T_i). For the USG, we retain all nodes and edges of the DSG except for

T_i and the edges incident on it. Instead, we split the node for T_i into multiple nodes—one node for every read/write event in T_i. The edges are now incident on the corresponding operation of of T_i.

$USG(H, T_i)$ is obtained by transforming $DSG(H)$ as follows:. For each node p ($p \neq T_i$) in $DSG(H)$, we add a node to $USG(H, T_i)$. For each edge from node p to node q in $DSG(H)$, where p and q are different from T_i, we draw a corresponding edge in $USG(H, T_i)$. Now we add a node corresponding to every read and write performed by T_i. Any edge that was incident on T_i in the DSG is now incident on the corresponding read or write operation on T_i in the USG. Finally, consecutive events in T_i are connected by *order edges*, e.g., if an action (e.g., SQL statement) reads object y_j and immediately follows a write on object x in transaction T_i, we add an order-edge from $w_i(x_i)$ to $r_i(y_j)$ [9, Section 4.2.1]. Note that creating a graph with "supernodes" replacing each set of read and write operations for each T_i in $USG(H)$ yields $DSG(H)$.

Transactional Anomalies and Isolation Levels

Following Adya, we define isolation levels according to possible *anomalies*–typically represented by cycles in the serialization graphs. Definitions 27–43 are not found in Adya but are found (albeit not in this formalism) in Berenson et al. [49] and the literature on session guarantees [216, 224].

Definition 21 (Write Cycles (G0)). *A history* H *exhibits phenomenon G0 if* $DSG(H)$ *contains a directed cycle consisting entirely of write-dependency edges.*

Definition 22 (Read Uncommitted). *A system that provides Read Uncommitted isolation prohibits phenomenon G0.*

Definition 23 (Aborted Reads (G1a)). *A history* H *exhibits phenomenon G1a if it contains an aborted transaction* T_1 *and a committed transaction* T_2 *such that* T_2 *has read some object (possibly via a predicate) modified by* T_1.

Definition 24 (Intermediate Reads (G1b)). *A history* H *exhibits phenomenon G1b if it contains a committed transaction* T_2 *that has read a version of object* x *(possibly via a predicate) written by transaction* T_1 *that was not* T_1*'s final modification of* x.

Definition 25 (Circular Information Flow (G1c)). *A history* H *exhibits phenomenon G1c if* $DSG(H)$ *contains a directed cycle consisting entirely of dependency edges.*

Definition 26 (Read Committed). *A system that provides Read Committed isolation prohibits phenomenon G0, G1a, G1b, and G1c.*

Definition 27 (Item-Many-Preceders (IMP)). *A history* H *exhibits phenomenon IMP if* $DSG(H)$ *contains a transaction* T_i *such that* T_i *directly item-read-depends by* x *on more than one other transaction.*

$$T_1 : w_x(1)$$
$$T_2 : w_x(2)$$
$$T_3 : r_x(1)\ r_x(2)$$

Figure 4.6: Example of *IMP* anomaly.

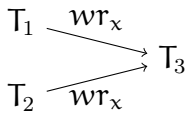


Figure 4.7: DSG for Figure 4.6.

Definition 28 (Item Cut Isolation (I-CI)). *A system that provides Item Cut Isolation prohibits phenomenon IMP.*

Definition 29 (Predicate-Many-Preceders (PMP)). *A history* H *exhibits phenomenon PMP if, for all predicate-based reads* $r_i(P_i : Vset(P_i))$ *and* $r_j(P_j : Vset(P_j)$ *in* T_k *such that the logical ranges of* P_i *and* P_j *overlap (call it* P_o*), the set of transactions that change the matches of* P_o *for* r_i *and* r_j *differ.*

Definition 30 (Predicate Cut Isolation (P-CI)). *A system that provides Predicate Cut Isolation prohibits phenomenon PMP.*

Definition 31 (Observed Transaction Vanishes (OTV)). *A history* H *exhibits phenomenon OTV if* $USG(H)$ *contains a directed cycle consisting of exactly one read-dependency edge by* x *from* T_j *to* T_i *and a set of edges by* y *containing at least one anti-dependency edge from* T_i *to* T_j *and* T_j*'s read from* y *precedes its read from* x.

$$T_1 : w_x(1)\ w_y(1)$$
$$T_2 : w_x(2)\ w_y(2)$$
$$T_3 : r_x(2)\ r_y(1)$$

Figure 4.8: Example of *OTV* anomaly.

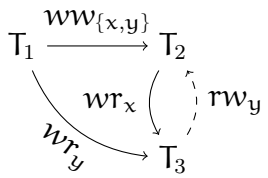


Figure 4.9: DSG for Figure 4.8.

Definition 32 (Monotonic Atomic View (MAV))**.** *A system that provides Monotonic Atomic View isolation prohibits phenomenon OTV in addition to providing Read Committed isolation.*

The following session guarantees are directly adapted from Terry et al.'s original definitions [216]:

Definition 33 (Non-monotonic Reads (N-MR))**.** *A history* H *exhibits phenomenon N-MR if* DSG(H) *contains a directed cycle consisting of a transitive session-dependency between transactions* T_j *and* T_i *with an anti-dependency edge by* i *from* T_j *and a read-dependency edge by* i *into* T_i*.*

Definition 34 (Monotonic Reads (MR))**.** *A system provides Monotonic Reads if it prohibits phenomenon N-MR.*

$$T_1 : w_x(1)$$
$$T_2 : w_x(2)$$
$$T_3 : r_x(2)$$
$$T_4 : r_x(1)$$

Figure 4.10: Example of *N-MR* violation when $w_x(1) \ll w_x(2)$ and T_4 directly session-depends on T_3.

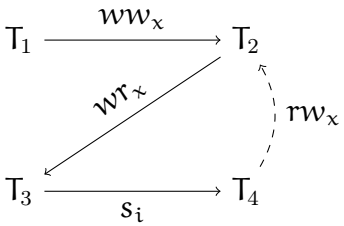


Figure 4.11: DSG for Figure 4.10. wr_x dependency from T_1 to T_4 omitted.

Definition 35 (Non-monotonic Writes (N-MW)). *A history* H *exhibits phenomenon N-MW if* DSG(H) *contains a directed cycle consisting of a transitive session-dependency between transactions* T_j *and* T_i *and at least one write-dependency edge.*

$$T_1 : w_x(1)$$
$$T_2 : w_y(1)$$
$$T_3 : r_y(1)\ r_x(0)$$

Figure 4.12: Example of *N-MW* anomaly if T_2 directly session-depends on T_1.

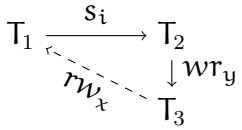


Figure 4.13: DSG for Figure 4.12.

Definition 36 (Monotonic Writes (MW)). *A system provides Monotonic Writes if it prohibits phenomenon N-MW.*

Definition 37 (Missing Read-Write Dependency (MRWD)). *A history* H *exhibits phenomenon MRWD if, in* DSG(H), *for all committed transactions* T_1, T_2, T_3 *such that* T_2 *read-depends on* T_1 *and* T_3 *read-depends on* T_2, T_3 *does not directly anti-depend on* T_1.

$$T_1 : w_x(1)$$
$$T_2 : r_x(1)\ w_y(1)$$
$$T_3 : r_y(1)\ r_x(0)$$

Figure 4.14: Example of *MRWD* anomaly.

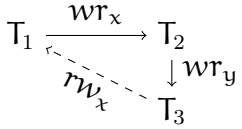


Figure 4.15: DSG for Figure 4.14.

Definition 38 (Writes Follow Reads (WFR)). *A system provides Writes Follow Reads if it prohibits phenomenon MWRD.*

Definition 39 (Missing Your Writes (MYR)). *A history* H *exhibits phenomenon MYR if* $DSG(H)$ *contains a directed cycle consisting of a transitive session-dependency between transactions* T_j *and* T_i*, at least one anti-dependency edge, and the remainder anti-dependency or write-dependency edges.*

$$T_1 : w_x(1)$$
$$T_2 : r_x(0)$$

Figure 4.16: Example of *MYR* anomaly if T_2 directly session-depends on T_1.

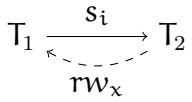


Figure 4.17: DSG for Figure 4.14.

Definition 40 (Read Your Writes (RYW)). *A system provides Read Your Writes if it prohibits phenomenon MYR.*

Definition 41 (PRAM Consistency). *A system provides PRAM Consistency if it prohibits phenomenon N-MR, N-MW, and MYR.*

Definition 42 (Causal Consistency). *A system provides Causal Consistency if it provides PRAM Consistency and prohibits phenomenon MWRD.*

Definition 43 (Lost Update). *A history* H *exhibits phenomenon Lost if* $DSG(H)$ *contains a directed cycle having one or more item-antidependency edges and all edges are by the same data item* x*.*

Definition 44 (Write Skew (Adya G2-item)). *A history* H *exhibits phenomenon Write Skew if* $DSG(H)$ *contains a directed cycle having one or more item-antidependency edges.*

For Snapshot Isolation, we depart from Adya's recency-based definition (see Adya Section 4.3). Nonetheless, implementations of this definition will still be unavailable due to reliance of preventing Lost Update.

Definition 45 (Snapshot Isolation). *A system that provides Snapshot Isolation prevents phenomena G0, G1a, G1b, G1c, PMP, OTV, and Lost Update.*

For Repeatable Read, we return to Adya.

Definition 46 (Repeatable Read). *A system that provides Repeatable Read Isolation prohibits phenomena G0, G1a, G1b, G1c, and Write Skew.*

4.5 Summary

In this chapter, we have shown that many previously defined isolation and data consistency models from the database and distributed systems communities are invariant confluent and can be implemented in a coordination-free manner. While traditional implementations of several of these semantics employ coordination, for those that we prove invariant confluent, this is not strictly necessary. Thus, existing applications that are built using one of these existing models may enjoy the benefits of coordination-free execution.

Chapter 5

Coordination Avoidance and RAMP Transactions

In the previous chapter, we identified several existing isolation and distributed consistency guarantees as coordination-free. Our goal was proof-of-concept algorithms and systems support for these levels. In this section, we go further. First, we develop a *new* isolation model that is tailored to a set of existing use cases for which there is no existing, sufficiently powerful invariant confluent semantics. Second, we develop high performance algorithms for enforcing those semantics.

Specifically, in this chapter, we address a largely underserved class of applications requiring multi-partition, atomically visible[1] *transactional* access: cases where all or none of each transaction's effects should be visible. In fact, this access corresponds to two semantics from the previous chapter: MAV combined with Cut Isolation. The status quo for these multi-partition atomic transactions provides an uncomfortable choice between algorithms that are fast but deliver inconsistent results and algorithms that deliver consistent results but are often slow and unavailable under failure. Many of the largest modern, real-world systems opt for protocols that guarantee fast and scalable operation but provide few—if any—transactional semantics for operations on arbitrary sets of data items [65, 74, 83, 95, 138, 192, 227]. This may lead to anomalous behavior for several use cases requiring atomic visibility, including secondary indexing, foreign key constraint enforcement, and materialized view maintenance (Section 6.4.1). In contrast, many traditional transactional mechanisms correctly ensure atomicity of updates [53, 85, 217]. However, these algorithms—such as two-phase locking and variants of optimistic concurrency

[1] Our use of "atomic" (specifically, Read Atomic isolation) concerns all-or-nothing *visibility* of updates (i.e., the ACID isolation effects of ACID atomicity; Section 5.3). This differs from uses of "atomicity" to denote serializability [53] or linearizability [28].

control—are often coordination-intensive, slow, and, under failure, unavailable in a distributed environment [32, 89, 141, 187]. This specific dichotomy between scalability and atomic visibility has been described as "a fact of life in the big cruel world of huge systems" [133].

Our contribution in this chapter is to demonstrate that atomically visible transactions on partitioned databases are *not* at odds with scalability. We provide high-performance implementations of a new, non-serializable isolation model called Read Atomic (RA) isolation, corresponding to MAV with Cut Isolation. RA ensures that all or none of each transaction's updates are visible to others and that each transaction reads from an atomic snapshot of database state (Section 5.3)—this is useful in the applications we target. We subsequently develop three new, scalable algorithms for achieving RA isolation that we collectively title Read Atomic Multi-Partition (RAMP) transactions (Section **??**). RAMP transactions guarantee scalability and outperform existing atomic algorithms because they satisfy two key scalability constraints. First, RAMP transactions guarantee coordination-free execution: per Chapter 2, one client's transactions cannot cause another client's transactions to stall or fail. Second, RAMP transactions guarantee *partition independence*: clients only contact partitions that their transactions directly reference (i.e., there is no central master, coordinator, or scheduler). Together, these properties ensure guaranteed completion, limited coordination across partitions, and horizontal scalability for multi-partition access.

RAMP transactions are scalable because they appropriately control the visibility of updates without inhibiting concurrency. Rather than force concurrent reads and writes to stall, RAMP transactions allow reads to "race" writes: RAMP transactions can autonomously detect the presence of non-atomic (partial) reads and, if necessary, repair them via a second round of communication with servers. To accomplish this, RAMP writers attach metadata to each write and use limited multi-versioning to prevent readers from stalling. The three algorithms we present offer a trade-off between the size of this metadata and performance. `RAMP-Small` transactions require constant space (a timestamp per write) and two round trip time delays (RTTs) for reads and writes. `RAMP-Fast` transactions require metadata size that is linear in the number of writes in the transaction but only require one RTT for reads in the common case and two in the worst case. `RAMP-Hybrid` transactions employ Bloom filters [59] to provide an intermediate solution. Traditional techniques like locking couple atomic visibility and mutual exclusion; RAMP transactions provide the benefits of the former without incurring the scalability, availability, or latency penalties of the latter.

In addition to providing a theoretical analysis and proofs of correctness, we demonstrate that RAMP transactions deliver in practice. Our RAMP implementation achieves linear scalability to over 7 million operations per second on a 100 server cluster (at overhead below 5% for a workload of 95% reads). Moreover, across a range of workload

configurations, RAMP transactions incur limited overhead compared to other techniques and achieve higher performance than existing approaches to atomic visibility (Section 5.5).

While the literature contains an abundance of isolation models, we believe that the large number of modern applications requiring RA isolation and the excellent scalability of RAMP transactions justify the addition of yet another model. RA isolation is too weak for some applications, but, for the many that it can serve, RAMP transactions offer substantial benefits.

The remainder of this article proceeds as follows: Section 5.1 presents an overview of RAMP transactions and describes key use cases based on industry reports. Section 5.3 defines Read Atomic isolation, presents both a detailed comparison with existing isolation guarantees and a syntactic condition, the Read-Subset-Writes property, that guarantees equivalence to serializable isolation, and defines two key scalability criteria for RAMP algorithms to provide. Section 5.4 presents and analyzes three RAMP algorithms, which we experimentally evaluate in Section 5.5. Section 5.6 presents modifications of the RAMP protocols to better support multi-datacenter deployments and to enforce transitive dependencies. Section 5.10 concludes with a discussion of extensions to the protocols presented here.

5.1 Overview

In this chapter, we consider the problem of making transactional updates atomically visible to readers—a requirement that, as we outline in this section, is found in several prominent use cases today. The basic property we provide is fairly simple: either all or none of each transaction's updates should be visible to other transactions. For example, if x and y are initially $\mathtt{null}$ and a transaction T_1 writes $x = 1$ and $y = 1$, then another transaction T_2 should not read $x = 1$ and $y = \mathtt{null}$. Instead, T_2 should either read $x = 1$ and $y = 1$ or, possibly, $x = \mathtt{null}$ and $y = \mathtt{null}$. Informally, each transaction reads from an unchanging snapshot of database state that is aligned along transactional boundaries. We call this property *atomic visibility* and formalize it via the Read Atomic isolation guarantee in Section 5.3.

The classic strategy for providing atomic visibility is to ensure mutual exclusion between readers and writers. For example, if a transaction like T_1 above wants to update data items x and y, it can acquire exclusive locks for each of x and y, update both items, then release the locks. No other transactions will observe partial updates to x and y, ensuring atomic visibility. However, this solution has a drawback: while one transaction holds exclusive locks on x and y, no other transactions can access x and y for either reads or writes. By

using mutual exclusion to enforce the atomic visibility of updates, we have also limited concurrency. In our example, if x and y are located on different servers, concurrent readers and writers will be unable to perform useful work during communication delays. These communication delays form an upper bound on throughput: effectively, $\frac{1}{\text{message delay}}$ operations per second.

To avoid this upper bound, we separate the problem of providing atomic visibility from the mechanism of mutual exclusion. By achieving the former but avoiding the latter, the algorithms we develop in this paper are not subject to the scalability penalties of many prior approaches. To ensure that all servers successfully execute a transaction (or that none do), our algorithms employ an atomic commitment protocol (ACP). When coupled with a coordinating concurrency control mechanism such as locking, ACPs are harmful to scalability and availability: arbitrary failures can (provably) cause any ACP implementation to stall [53]. We instead use ACPs with non-blocking concurrency control mechanisms; this means that individual transactions can stall due to failures or communication delays without forcing other transactions to stall. In a departure from traditional concurrency control, we allow multiple ACP rounds to proceed in parallel over the same data.

The end result—what we call RAMP transactions—provide excellent scalability and performance under contention (e.g., in the event of write hotspots) and are robust to partial failure. RAMP transactions' non-blocking behavior means that they cannot provide certain guarantees like preventing concurrent updates. However, the applications we highlight—for which Read Atomic isolation is sufficient to maintain correctness—will benefit from our algorithms. The remainder of this section identifies several relevant use cases from industry that require atomic visibility for correctness.

5.2 Read Atomic Isolation in the Wild

As a simple example, consider a social networking application: if two users, Sam and Mary, become "friends" (a bi-directional relationship), other users should never see that Sam is a friend of Mary but Mary is not a friend of Sam: either both relationships should be visible, or neither should be. A transaction under Read Atomic isolation would correctly enforce this behavior. We can further classify three general use cases for Read Atomic isolation:

1. Foreign key constraints. Many database schemas contain information about relationships between records in the form of foreign key constraints. For example, Facebook's TAO [65], LinkedIn's Espresso [192], and Yahoo! PNUTS [83] store information about business entities such as users, photos, and status updates as well as relationships between

them (e.g., the friend relationships above). Their data models often represent bi-directional edges as two distinct uni-directional relationships. For example, in TAO, a user performing a "like" action on a Facebook page produces updates to both the `LIKES` and `LIKED_BY` associations [65]. PNUTS's authors describe an identical scenario [83]. These applications require foreign key maintenance and often, due to their unidirectional relationships, multi-entity update and access. Violations of atomic visibility surface as broken bi-directional relationships (as with Sam and Mary above) and dangling or incorrect references. For example, clients should never observe that Frank is an `employee` of `department.id=5`, but no such `department` exists in the `department` table.

Under RA isolation, when inserting new entities, applications can bundle relevant entities from each side of a foreign key constraint into a transaction. When deleting associations, users can avoid dangling pointers by creating a "tombstone" at the opposite end of the association (i.e., delete any entries with associations via a special record that signifies deletion) [234].

2. Secondary indexing. Data is typically partitioned across servers according to a primary key (e.g., user ID). This allows fast location and retrieval of data via primary key lookups but makes access by secondary attributes challenging (e.g., indexing by birth date). There are two dominant strategies for distributed secondary indexing. First, the *local secondary index* approach co-locates secondary indexes and primary data, so each server contains a secondary index that only references and indexes data stored on its server [45, 192]. This allows easy, single-server updates but requires contacting every partition for secondary attribute lookups (write-one, read-all), compromising scalability for read-heavy workloads [65, 85, 192]. Alternatively, the *global secondary index* approach locates secondary indexes (which may be partitioned, but by a secondary attribute) separately from primary data [45, 83]. This alternative allows fast secondary lookups (read-one) but requires multi-partition update (at least write-two).

Real-world services employ either local secondary indexing (e.g., Espresso [192], Cassandra, and Google Megastore's local indexes [45]) or non-atomic (incorrect) global secondary indexing (e.g., Espresso and Megastore's global indexes, Yahoo! PNUTS's proposed secondary indexes [83]). The former uses coordination and limits the workloads that are scalable but is correct. The latter does not use coordination and is scalable for a range of workloads but is incorrect. For example, in a database partitioned by `id` with an incorrectly-maintained global secondary index on `salary`, the query `'SELECT id, salary WHERE salary > 60,000'` might return records with `salary` less than $60,000 and omit some records with `salary` greater than $60,000.

Under RA isolation, the secondary index entry for a given attribute can be updated atomically with base data. For example, suppose a secondary index is stored as a mapping

from secondary attribute values to sets of item-versions matching the secondary attribute (e.g., the secondary index entry for users with blue hair would contain a list of user IDs and last-modified timestamps corresponding to all of the users with attribute `hair-color=blue`). Insertions of new primary data require additions to the corresponding index entry, deletions require removals, and updates require a "tombstone" deletion from one entry and an insertion into another.

3. Materialized view maintenance. Many applications precompute (i.e., materialize) queries over data, as in Twitter's Rainbird service [227], Google's Percolator [188], and LinkedIn's Espresso systems [192]. As a simple example, Espresso stores a mailbox of messages for each user along with statistics about the mailbox messages: for Espresso's read-mostly workload, it is more efficient to maintain (i.e., pre-materialize) a count of unread messages rather than scan all messages every time a user accesses her mailbox [192]. In this case, any unread message indicators should remain in sync with the messages in the mailbox. However, atomicity violations will allow materialized views to diverge from the base data (e.g., Susan's mailbox displays a notification that she has `unread` messages but all 60 messages in her `inbox` are marked as `read`).

With RAMP transactions, base data and views can be updated atomically. The maintenance of a view depends on its specification [58, 79, 139], but RAMP transactions provide appropriate concurrency control primitives for ensuring that changes are delivered to the materialized view partition. For select-project views, a simple solution is to treat the view as a separate table and perform maintenance as needed: new rows can be inserted/deleted according to the specification, and, if necessary, the view can be (re-)computed on demand (i.e., lazy view maintenance [238]). For more complex views, such as counters, users can execute RAMP transactions over specialized data structures such as the CRDT G-Counter [205].

Status Quo. Despite application requirements for Read Atomic isolation, few large-scale production systems provide it. For example, the authors of Tao, Espresso, and PNUTS describe several classes of atomicity anomalies exposed by their systems, ranging from dangling pointers to the exposure of intermediate states and incorrect secondary index lookups, often highlighting these cases as areas for future research and design [65, 83, 192]. These systems are not exceptions: data stores like Bigtable [74], Dynamo [95], and many popular "NoSQL" [179] and even some "NewSQL" [32] stores do not provide transactional guarantees for multi-item operations. Unless users are willing to sacrifice scalability by opting for serializable semantics [85], they are often left without transactional semantics.

The designers of these Internet-scale, real-world systems have made a conscious decision to provide scalability at the expense of multi-partition transactional semantics. Our goal

with RAMP transactions is to preserve this scalability but deliver atomically visible behavior that is sufficient to maintain key consistency criteria for the use cases we have described.

5.3 Semantics and System Model

In this section, we formalize Read Atomic isolation and, to capture scalability, formulate a pair of strict scalability criteria: coordination-free execution and partition independence. Readers more interested in RAMP algorithms may wish to proceed to Section 5.4.

5.3.1 RA Isolation: Formal Specification

To formalize RA isolation, as is standard [9, 53] (and as in Chapter 4), we consider ordered sequences of reads and writes to arbitrary sets of items, or transactions. We call the set of items a transaction reads from and writes to its *item read set* and *item write set*. Each write creates a *version* of an item and we identify versions of items by a *timestamp* taken from a totally ordered set (e.g., natural numbers) that is unique across all versions of each item. Timestamps therefore induce a total order on versions of each item, and we denote version i of item x as x_i. All items have an initial version $\bot$ that is located at the start of each order of versions for each item and is produced by an initial transaction $T_\bot$. Each transaction ends in a *commit* or an *abort* operation; we call a transaction that commits a *committed* transaction and a transaction that aborts a *aborted* transaction. In our model, we consider *histories* comprised of a set of transactions along with their read and write operations, versions read and written, and commit or abort operations. In our example histories, all transactions commit unless otherwise noted.

Definition 47 (Fractured Reads). *A transaction* T_j *exhibits the fractured reads phenomenon if transaction* T_i *writes versions* x_a *and* y_b *(in any order, where* x *and* y *may or may not be distinct items),* T_j *reads version* x_a *and version* y_c*, and* $c < b$.

We also define Read Atomic isolation to prevent transactions from reading uncommitted or aborted writes. This is needed to capture the notion that, under RA isolation, readers only observe the final output of a given transaction that has been accepted by the database. To do so, we draw on existing definitions from the literature on weak isolation.

Our RAMP protocols provide this property by assigning the final write to each item in each transaction the same timestamp. However, to avoid further confusion between the standard practice of assigning each final write in a serializable multi-version history the same timestamp [53] and the flexibility of timestamp assignment admitted in Adya's formulation of weak isolation, we continue with the above definitions.

These criteria prevent readers from observing *uncommitted* versions (i.e., those produced by a transaction that has not committed or aborted), *aborted* versions (i.e., those produced by a transaction that has aborted), or *intermediate* versions (i.e., those produced by a transaction but were later overwritten by writes to the same items by the same transaction).

We can finally define Read Atomic isolation:

Definition 48 (Read Atomic)**.** *A system provides Read Atomic isolation (RA) if it prevents fractured reads phenomena and also prevents transactions from reading uncommitted, aborted, or intermediate versions (i.e., Adya's G0, G1a, G1b, G1c).*

Thus, RA informally provides transactions with a "snapshot" view of the database that respects transaction boundaries (see Sections 5.3.3 and 5.3.4 for more details, including a discussion of transitivity). RA is simply a restriction on read *visibility*—if the ACID "Atomicity" property requires that all or none of a transaction's updates are performed, RA requires that all or none of a transaction's updates are visible to other transactions.

Importantly, RA is invariant confluent: if two read/write histories each independently do not have fractured reads, composing them will not change the values returned by any read operations. This means that there is at least one coordination-free implementation of RA, which we will develop in Section 5.4.

5.3.2 RA Implications and Limitations

As outlined in Section 5.2, RA isolation matches several common use cases. However, RA is *not* sufficient for all applications. RA does not prevent concurrent updates or provide serial access to data items; that is, under RA, two transactions are never prevented from both producing different versions of the same data items. For example, RA is an incorrect choice for an application that wishes to maintain positive bank account balances in the event of withdrawals. RA is a better fit for our "friend" operation because the operation is write-only and correct execution (i.e., inserting both records) is not conditional on concurrent updates.

From a programmer's perspective, we have found RA isolation to be most easily understandable (at least initially) with read-only and write-only transactions; after all, because RA allows concurrent writes, any values that are read might be changed at any time. However, read-write transactions are indeed well defined under RA.

To handle conflicting operations, RA isolation benefits from the use of commutative and associative merge functions. The default behavior we present here is a "last write wins" policy, with ties broken according to version. However, more sophisticated datatypes such

as commutative replicated sets, counters, and maps [205] are also useful, especially for data structures such as index entries.

To illustrate these points, in Section 5.3.3, we describe RA's relation to other formally defined isolation levels, and, in Section 5.3.4, we discuss when RA provides serializable outcomes.

5.3.3 RA Compared to Other Isolation Models

In this section, we illustrate RA's relationship to alternative weak isolation models by both example and reference to particular isolation phenomena drawn from [9] and [32]. Formal definitions of the models below can be found in in Section 4.4

RA is stronger than Read Committed as Read Committed does not prevent fractured reads. History 5.1 does not respect RA isolation. After T_1 commits, both T_2 and T_3 could both commit but, to prevent fractured reads, T_4 and T_5 must abort. History 5.1 respects RC isolation and all transactions can safely commit.

$$\begin{array}{ll} T_1 & w(x_1); w(y_1) \\ T_2 & r(x_\perp); r(y_\perp) \\ T_3 & r(x_1); r(y_1) \\ T_4 & r(x_\perp); r(y_1) \\ T_5 & r(x_1); r(y_\perp) \end{array} \tag{5.1}$$

Lost Updates. Lost Updates phenomena informally occur when two transactions simultaneously attempt to make conditional modifications to the same data item(s).

RA does not prevent Lost Updates phenomena. History 5.2 exhibits the Lost Updates phenomenon but is valid under RA. That is, T_1 and T_2 can both commit under RA isolation.

$$\begin{array}{ll} T_1 & r(x_\perp); w(x_1) \\ T_2 & r(x_\perp); w(x_2) \end{array} \tag{5.2}$$

History 5.2 is invalid under a stronger isolation model that prevents Lost Updates phenomena, such as Snapshot Isolation or Cursor Isolation. Under either of these models, the system would abort T_1, T_2, or both. However, Cursor Stability does not prevent fractured

reads phenomena, so RA and Cursor Stability are incomparable.

Write Skew. RA does not prevent Write Skew phenomena. History 5.3 exhibits the Write Skew phenomenon (Adya's G2) but is valid under RA. That is, T_1 and T_2 can both commit under RA isolation.

$$\begin{aligned} T_1 \quad & r(y_\perp); w(x_1) \\ T_2 \quad & r(x_\perp); w(y_2) \end{aligned} \tag{5.3}$$

History 5.3 is invalid under a stronger isolation model that prevents Write Skew phenomena. One stronger model is Repeatable Read. Under Repeatable Read isolation, the system would abort either T_1, T_2, or both. (Importantly, Adya's formulation of Repeatable Read is considerably stronger than the ANSI SQL standard specification; RA is stronger than the Cut Isolation we consider in Chapter 4.)

Missing dependencies. Notably, RA does not—on its own—prevent missing dependencies—in effect, missing transitive updates. We again reproduce Adya's definitions below:

Definition 49 (Missing Transaction Updates). *A transaction* T_j *misses the effects of a transaction* T_i *if* T_i *writes* x_i *and commits and another transaction* T_j *reads another version* x_k *such that* $k < i$*; i.e.,* T_j *reads a version of* x *that is older than the version that was committed by* T_i.

Adya subsequently defines a criteria that prohibits missing transaction updates across all types of dependency edges:

Definition 50 (No-Depend-Misses). *If transaction* T_j *depends on transaction* T_i, T_j *does not miss the effects of* T_i.

History 5.4 does not exhibit the No-Depend-Misses phenomenon but is still valid under RA. That is, T_1, T_2, and T_3 can all commit under RA isolation. Thus, fractured reads prevention is similar to No-Depend-Misses but only applies to immediate read dependencies (rather than all transitive dependencies).

$$\begin{aligned} T_1 \quad & w(x_1); w(y_1) \\ T_2 \quad & r(y_1); w(z_2) \\ T_3 \quad & r(x_\perp); r(z_2) \end{aligned} \tag{5.4}$$

History 5.4 is invalid under a stronger isolation model that prevents missing dependencies phenomena, such as standard semantics for Snapshot Isolation (notably, not Parallel Snapshot Isolation [210]) and Repeatable Read isolation. Under one of these models, the system would abort either T_3 or all of T_1, T_2, and T_3.

This behavior is particularly important to the use cases that we discuss in Sections 5.3.2 and 5.3.4: writes that should be read together should be written together.

We further discuss the benefits and enforcements of transitivity in Section 5.6.3.

OTV and Many-Preceders. As noted in Section **??**, the Fractured Reads phenomenon subsumes the Observed Transaction Vanishes and Many-Preceders phenomena from Chapter 4. To illustrate:

If T_j exhibits the OTV phenomenon reads x_a produced by T_i in the definition of Fractured Reads above then there is a read-dependency edge by x from T_j to T_i in $USG(H)$; however, if T_j also reads y_c and $c < b$, then T_i must anti-depend on T_j, resulting in OTV. Thus, every fractured read is an instance of OTV. However, not every fractured read is an instance of OTV. That is, in our example, if T_j reads y_b and then reads y_c, fractured reads have occurred, but OTV has not (due to the clause that "T_j's read from y precedes its read from x" in Definition 31).

Fractured reads also subsumes the many-preceders phenomenon (in the item-specific case, Definition 27). If T_i exhibits the IMP phenomenon, it directly item-depends by x on more than one transaction—say, T_j and T_k—then T_i read versions x_i and x_k produced by each of T_i and T_k. However, by definition, $i < k$ or $k < i$, and thus T_j also has fractured reads. Again, not every fractured read is an instance of IMP. Consider the following history:

$$\begin{array}{ll} T_1 & w(x_1); w(y_1) \\ T_2 & r(x_\perp); r(y_1) \end{array} \tag{5.5}$$

T_2 exhibits fractured reads but does not exhibit IMP.

Predicates. Thus far, we have not extensively discussed the use of predicate-based reads. As Adya notes [9] and we describe above, predicate-based isolation guarantees can be cast as an extension of item-based isolation guarantees (see also Adya's *PL-2L*, which closely resembles RA). RA isolation is no exception to this rule. We can extend each RA definition to include predicates using Adya's predicate-based formalism.

Relating to Additional Guarantees. RA isolation subsumes several other useful guarantees. RA prohibits Item-Many-Preceders and Observed Transaction Vanishes phenomena; RA also guarantees Item Cut Isolation, and with predicate support, RA subsumes Predicate

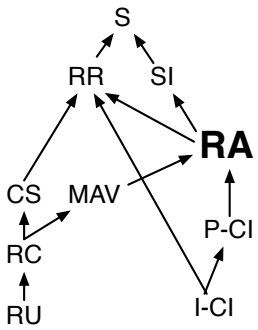


Figure 5.1: Comparison of RA with isolation levels from [9, 32]. RU: Read Uncommitted, RC: Read Committed, CS: Cursor Stability, MAV: Monotonic Atomic View, ICI: Item Cut Isolation, PCI: Predicate Cut Isolation, RA: Read Atomic, SI: Snapshot Isolation, RR: Repeatable Read (Adya *PL-2.99*), S: Serializable.

Cut Isolation Thus, it is a combination of Monotonic Atomic View and Item Cut Isolation (Section 4.4). **Summary.** Figure 5.1 relates RA isolation to several existing models. RA is stronger than Read Committed, Monotonic Atomic View, and Cut Isolation, weaker than Snapshot Isolation, Repeatable Read, and Serializability, and incomparable to Cursor Stability.

5.3.4 RA and Serializability

When we began this work, we started by examining the use cases outlined in Section 6.4.1 and deriving a weak isolation guarantee that would be sufficient to ensure their correct execution. For general-purpose read-write transactions, RA isolation may indeed lead to non-serializable (and possibly incorrect) database states and transaction outcomes. Yet, as Section 5.3.2 hints, there appears to be a broader "natural" pattern for which RA isolation appears to provide an intuitive (even "correct") semantics. In this section, we show that for transactions with a particular property of their item read and item write sets, RA is, in fact, serializable. We define this property, called the *read-subset-items-written (RSIW) property*, prove that transactions obeying the RSIW property lead to serializable outcomes, and discuss the implications of the RSIW property for the applications outlined in Section 6.4.1.

Because our system model operates on multiple versions, we must make a small refinement to our use of the term "serializability"—namely, we draw a distinction between serial and one-copy serializable schedules, per Bernstein et al. [53]. First, we say that two histories H_1 and H_2 are *view equivalent* if they contain the same set of committed transactions and have the same operations and $DSG(H_1)$ and $DSG(H_2)$ have the same direct read de-

pendencies. For consistency with prior work, we say that T_i *reads from* T_j if T_i directly read-depends on T_j. We say that a transaction is *read-only* if it does not contain write operations and that a transaction is *write-only* if it does not contain read operations. In this section, we concern ourselves with *one-copy serializability* [53], which we define using the previous definition of view equivalence.

Definition 51 (One-Copy Serializability). *A history is one-copy serializable if it is view equivalent to a serial execution of the transactions over a single logical copy of the database.*

The basic intuition behind the RSIW property is straightforward: under RA isolation, if application developers use a transaction to bundle a set of writes that should be observed together, any transactions that read from the items that were written will, in fact, behave "properly"—or one-copy serializably. That is, for read-only and write-only transactions, if each reading transaction only reads a subset of the items that another write-only transaction wrote, then RA isolation is equivalent to one-copy serializable isolation. Before proving that this behavior is one-copy serializable, we can more precisely characterize this condition as follows:

Definition 52 (Read-Subset-Items-Written). *A read-only transaction* T_r *exhibits the Read-Subset-Items-Written property if, whenever* T_r *reads a version produced by a write-only transaction* T_w*,* T_r *only reads items written to by* T_w*.*

For example, consider the following History 5.6:

$$\begin{aligned} T_1 \quad & w(x_1); w(y_1) \\ T_2 \quad & r(x_1); w(y_1) \\ T_3 \quad & r(x_1); r(z_\perp) \end{aligned} \tag{5.6}$$

Under History 5.6, T_2 exhibits the RSIW property because it reads a version produced by transaction T_1 and its item read set ($\{x, y\}$) is a subset of T_1's item write set ($\{x, y\}$). However, T_3 does not exhibit the RSIW property because i.) T_3 reads from T_1 but T_3's read set ($\{x, z\}$) is not a subset of T_1's write set ($\{x, y\}$) and ii.), perhaps more subtly, T_3 reads from both T_1 and $T_\perp$.

We say that a history H containing read-only and write-only transactions exhibits the RSIW property (or *has RSIW*) if every read-only transaction in H exhibits the RSIW property.

This brings us to our main result in this section:

Theorem 2. *If a history* H *containing read-only and write-only transactions has RSIW and is valid under RA isolation, then* H *is one-copy serializable.*

The proof of Theorem 2 is by construction: given a history H has RSIW and is valid under RA isolation, we describe how to derive an equivalent one-copy serial execution of the transactions in H. We begin with the construction procedure, provide examples of how to apply the procedure, then prove that the procedure converts RSIW histories to their one-copy serial equivalents. We provide the proof in Section 5.7

Utility. Theorem 2 is helpful because it provides a simple syntactic condition for understanding when RA will provide one-copy serializable access. For example, we can apply this theorem to our use cases from Section 6.4.1. In the case of multi-entity update and read, if clients issue read-only and write-only transactions that obey the RSIW property, their result sets will be one-copy serializable. The RSIW property holds for equality-based lookup of single records from an index (e.g., fetch from the index and subsequently fetch the corresponding base tuple, each of which was written in the same transaction (e.g., was auto-generated upon insertion of the tuple into the base relation). However, the RSIW property does not in the event of multi-tuple reads, leading to less intuitive behavior. Specifically, if two different clients trigger two separate updates to an index entry, some clients may observe one update but not the other, and other clients may observe the opposite behavior. In this case, the RAMP protocols still provide a snapshot view of the database according to the index(es)—that is, clients will never observe base data that is inconsistent with the index entries—but nevertheless surface non-serializable database states. Finally, for more general materialized view accesses, point queries and bulk insertions have RSIW.

As discussed in Section 6.4.1, in the case of indexes and views, it is helpful to view each physical data structure (e.g., a CRDT [205] used to represent an index entry) as a *collection* of versions. In this case, the RSIW property applies only if clients make modifications to the entire collection at once (e.g., as in a `DELETE CASCADE` operation).

Coupled with an appropriate algorithm ensuring RA isolation, we can ensure one-copy serializable isolation. This addresses a long-standing concern with our work: why is RA somehow "natural" for these use cases (but not necessarily all use cases)? We have encountered applications that do not require one-copy serializable access—such as the mailbox unread message maintenance from Section 6.4.1 and, in some cases, index maintenance for non-read-modify-write workloads—and therefore may safely violate RSIW. However, we believe the RSIW property is a handy principle (or, at the least, rule of thumb) for reasoning about applications of RA isolation and the RAMP protocols.

Finally, the RSIW property is only a *sufficient* condition for one-copy serializable behavior under RA isolation. It is not necessary—for example, there are several alternative sufficient conditions to consider. As a natural extension, while RSIW only pertains to pairs of read-only and write-only transactions, one might consider allowing readers to observe

multiple write transactions. For example, consider the following history:

$$\begin{array}{ll} T_1 & w(x_1); w(y_1) \\ T_2 & w(u_2); w(z_2) \\ T_3 & r(x_1); r(z_2) \end{array} \tag{5.7}$$

History 5.7 is valid under RA and is also one-copy serializable but does not have RSIW: T_3 reads from *two* transactions' write sets. However, consider the following history:

$$\begin{array}{ll} T_1: & w(x_1); w(y_1) \\ T_2: & w(u_2); w(z_2) \\ T_3: & r(x_1); r(z_\perp) \\ T_4: & r(x_\perp); r(z_2) \end{array} \tag{5.8}$$

History 5.8 is valid under RA, consists only of read-only and write-only transactions, yet is no longer one-copy serializable. T_3 observes, in effect, a one-copy serializable prefix beginning with $T_\perp; T_1$ while T_4 observes a prefix beginning with $T_\perp; T_2$. Neither T_3 nor T_4 observes the prefixes $T_\perp; T_1; T_2$ or $T_\perp; T_2; T_1$.

Thus, while there may indeed be useful criteria beyond the RSIW property that we might consider as a basis for one-copy serializable execution under RA, we have observed RSIW to be the most intuitive and useful thus far. One clear criteria is to search for schedules or restrictions under RA with an acyclic Directed Serialization Graph (from Appendix 5.7). The reason why RSIW is so simple for read-only and write-only transactions is that each read-only transaction only reads from one other transaction and does not induce any additional anti-dependencies. Combining reads and writes complicates reasoning about the acyclicity of the graph.

This exercise touches upon an important lesson in the design and use of weakly isolated systems: by restricting the set of operations accessible to a user (e.g., RSIW read-only and write-only transactions), one can often achieve more scalable implementations (e.g., using weaker semantics) *without* necessarily violating existing abstractions (e.g., one-copy serializable isolation). While prior work often focuses on restricting only operations (e.g., to read-only or write-only transactions [11, 168], or stored procedures [141, 142, 217], or single-site transactions [45]) or only semantics (e.g., weak isolation guarantees [32, 35, 168]), we see considerable promise in better understanding the intersection between and combinations of the two. This is often subtle and almost always challenging, but the results—as we found here—may be surprising.

5.3.5 System Model and Scalability

We consider databases that are partitioned, with the set of items in the database spread over multiple servers. Each item has a single logical copy, stored on a server—called the item's *partition*—whose identity can be calculated using the item. Clients forward operations on each item to the item's partition, where they are executed. In our examples, all data items have the null value ($\perp$) at database initialization. In this section, we do not model replication of data items within a partition; this can happen at a lower level of the system than our discussion (see Section 5.4.5) as long as operations on each item are linearizable [28].

Scalability criteria. As we discussed in Section 5.1, large-scale deployments often eschew transactional functionality on the premise that it would be too expensive or unstable in the presence of failure and degraded operating modes [57, 65, 74, 83, 95, 133, 138, 192, 227]. Our goal in this paper is to provide robust and scalable transactional functionality, and, so we first define criteria for "scalability":

Per Section 2.3, *Coordination-free execution* ensures that one client's transactions cannot cause another client's to block and that, if a client can contact the partition responsible for each item in its transaction, the transaction will eventually commit (or abort of its own volition). This prevents one transaction from causing another to abort—which is particularly important in the presence of partial failures—and guarantees that each client is able to make useful progress. Note that "strong" isolation models like serializability and Snapshot Isolation require coordination and thus limit scalability. Locking is an example of a non-coordination-free implementation mechanism.

Many applications can limit their data accesses to a single partition via explicit data modeling [45, 90, 133, 192] or planning [89, 187]. However, this is not always possible. In the case of secondary indexing, there is a cost associated with requiring single-partition updates (scatter-gather reads), while, in social networks like Facebook and large-scale hierarchical access patterns as in Rainbird [227], perfect partitioning of data accesses is near-impossible. Accordingly:

Partition independence ensures that, in order to execute a transaction, a client only contacts partitions for data items that its transaction directly accesses. Thus, a partition failure only affects transactions that access items contained on the partition. This also reduces load on servers not directly involved in a transaction's execution. In the literature, partition independence for replicated data is also called *replica availability* [32] or *genuine partial replication* [201]. Using a centralized validator or scheduler for transactions is an example of a non-partition-independent implementation mechanism.

In addition to the above requirements, we limit the *metadata overhead* of algorithms. There are many potential solutions for providing atomic visibility that rely on storing prohibitive amounts of state. We will attempt to minimize the *metadata*—that is, data that the transaction did not itself write but which is required for correct execution. In our algorithms, we will specifically provide constant-factor metadata overheads (RAMP-S, RAMP-H) or else overhead linear in transaction size (but independent of data size; RAMP-F). As an example of a solution using prohibitive amounts of metadata, each transaction could send copies of all of its writes to every partition it accesses so that readers observe all of its writes by reading a single item. This provides RA isolation but requires considerable storage. Other solutions may require extra data storage proportional to the number of servers in the cluster or, worse, the database size, which we discuss in Related Work (Chapter 7).

5.4 RAMP Transaction Algorithms

Given specifications for RA isolation and scalability, we present algorithms for achieving both. For ease of understanding, we first focus on providing read-only and write-only transactions with a "last writer wins" overwrite policy, then subsequently discuss how to perform read/write transactions. Our focus in this section is on intuition and understanding; we defer all correctness and scalability proofs to Section 5.8, providing salient details inline.

At a high level, RAMP transactions allow reads and writes to proceed concurrently. This provides excellent performance but, in turn, introduces certain race conditions that could cause undesirable anomalies: one transaction might only read a subset of another transaction's writes, violating RA (i.e., fractured reads might occur). Instead of preventing this race (hampering scalability), RAMP readers *autonomously* detect the race (using metadata attached to each data item) and fetch any missing, in-flight writes from their respective partitions. To make sure that readers never have to wait for writes to arrive at a partition, writers use a two-phase (atomic commitment) protocol that ensures that once a write is visible to readers on one partition, any other writes in the transaction are present on and, if appropriately identified by version, readable from their respective partitions.

In this section, we present three algorithms that provide a trade-off between the amount of metadata required and the expected number of extra reads to fetch missing writes. As discussed in Section 6.4.1, while techniques like distributed locking couple mutual exclusion with atomic visibility of writes, RAMP transactions correctly control visibility but allow concurrent and scalable execution.

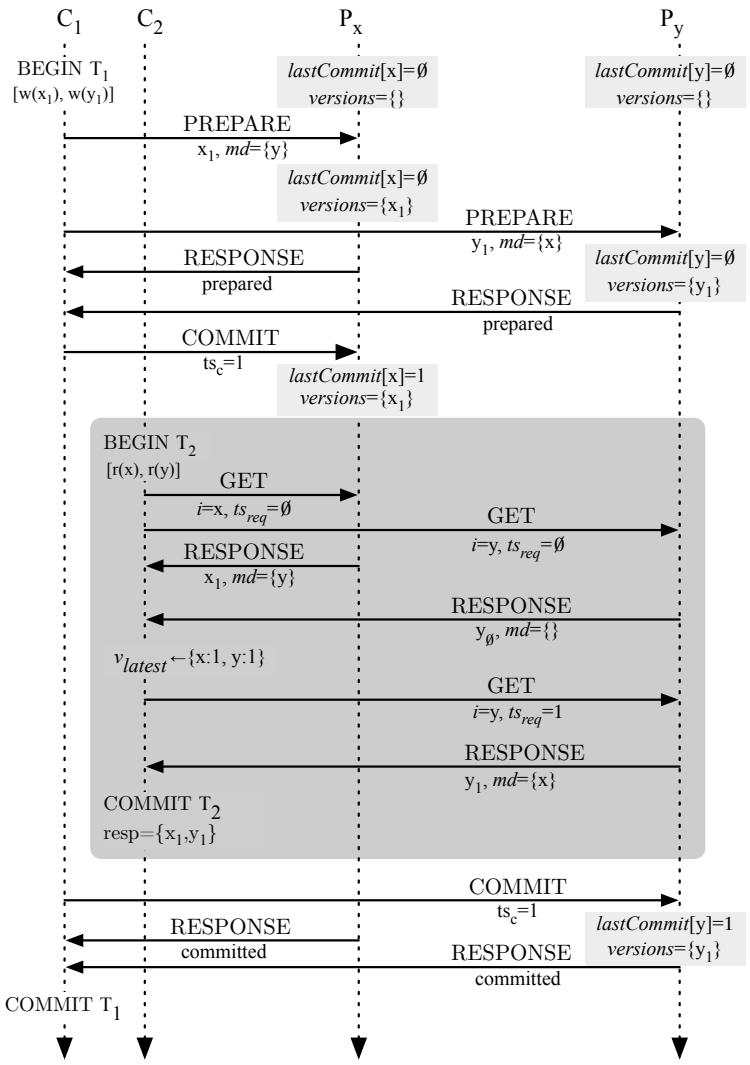


Figure 5.2: Space-time diagram for RAMP-F execution for two transactions T_1 and T_2 performed by clients C_1 and C_2 on partitions P_x and P_y. Lightly-shaded boxes represent current partition state (lastCommit and versions), while the single darker box encapsulates all messages exchanged during C_2's execution of transaction T_2. Because T_1 overlaps with T_2, T_2 must perform a second round of reads to repair the fractured read between x and y. T_1's writes are assigned timestamp 1. In our depiction, each item does not appear in its list of writes (e.g., P_x sees $\{y\}$ only and not $\{x, y\}$).

5.4.1 RAMP-Fast

To begin, we present a RAMP algorithm that, in the race-free case, requires one RTT for reads and two RTTs for writes, called RAMP-Fast (abbreviated RAMP-F; Algorithm 1).

RAMP-F stores metadata in the form of write sets (overhead linear in transaction size).

Overview. Each write in RAMP-F (lines 14–23) contains a timestamp (line 15) that uniquely identifies the writing transaction as well as a set of items written in the transaction (line 16). For now, combining a unique client ID and client-local sequence number is sufficient for timestamp generation (see also Section 5.4.4).

RAMP-F write transactions proceed in two phases: a first round of communication places each timestamped write on its respective partition. In this PREPARE phase, each partition adds the write to its local database (*versions*, lines 1, 17–20). A second round of communication (lines 21–23) marks versions as committed. In this COMMIT phase, each partition updates an index containing the highest-timestamped committed version of each item (lastCommit, lines 2, 6–8).

RAMP-F read transactions begin by first fetching the last (highest-timestamped) committed version for each item from its respective partition (lines 25–33). Using the results from this first round of reads, each reader can calculate whether it is "missing" any versions (that is, versions that were prepared but not yet committed on their partitions). The reader calculates a mapping from each item i to the highest timestamped version of i that appears in the metadata of any version (of i or of any other item) in the first-round read set (lines 29–32). If the reader has read a version of an item that has a lower timestamp than indicated in the mapping for that item, the reader issues a second read to fetch the missing version (by timestamp) from its partition (lines 33–36). Once all missing versions are fetched (which can be done in parallel), the client can return the resulting set of versions—the first-round reads, with any missing versions replaced by the optional, second round of reads.

By example. Consider the RAMP-F execution depicted in Figure 5.2. T_1 writes to both x and y, performing the two-round write protocol on two partitions, P_x and P_y. However, T_2 reads from x and y while T_1 is concurrently writing. Specifically, T_2 reads from P_x *after* P_x has committed T_1's write to x, but T_2 reads from P_y *before* P_y has committed T_1's write to y. Therefore, T_2's first-round reads return $x = x_1$ and $y = \bot$, and returning this set of reads would violate RA. Using the metadata attached to its first-round reads, T_2 determines that it is missing y_1 (since $v_{latest}[y] = 1$ and $1 > \bot$) and so T_2 subsequently issues a second read from P_y to fetch y_1 by version. After completing its second-round read, T_2 can safely return its result set. T_1's progress is unaffected by T_2, and T_1 subsequently completes by committing y_1 on P_y.

Why it works. RAMP-F writers use metadata as a record of intent: a reader can detect if it has raced with an in-progress commit round and use the metadata stored by the writer to fetch the missing data. Accordingly, RAMP-F readers only issue a second round of reads in the event that they read from a partially-committed write transaction (where some but not all

partitions have committed a write). In this event, readers will fetch the appropriate writes from the not-yet-committed partitions. Most importantly, RAMP-F readers never have to stall waiting for a write that has not yet arrived at a partition: the two-round RAMP-F write protocol guarantees that, if a partition commits a write, all of the corresponding writes in the transaction are present on their respective partitions (though possibly not committed locally). As long as a reader can identify the corresponding version by timestamp, the reader can fetch the version from the respective partition's set of pending writes without waiting. To enable this, RAMP-F writes contain metadata linear in the size of the writing transaction's write set (plus a timestamp per write).

RAMP-F requires two RTTs for writes: one for PREPARE and one for COMMIT. For reads, RAMP-F requires one RTT in the absence of concurrent writes and two RTTs otherwise.

RAMP timestamps are only used to identify specific versions and in ordering concurrent writes to the same item; RAMP-F transactions do not require a "global" timestamp authority. For example, if $\mathit{lastCommit}[k] = 2$, there is no requirement that a transaction with timestamp 1 has committed or even that such a transaction exists.

5.4.2 RAMP-Small: Trading Metadata for RTTs

While RAMP-F requires metadata size linear in write set size but provides best-case one RTT for reads, RAMP-Small (RAMP-S) uses constant metadata but always requires two RTT for reads (Algorithm 2). RAMP-S and RAMP-F writes are identical, but, instead of attaching the entire write set to each write, RAMP-S writers only store the transaction timestamp (line 7). Unlike RAMP-F, RAMP-S readers issue a first round of reads to fetch the highest committed timestamp for each item from its respective partition (lines 3, 9–12). Then the readers send the entire set of timestamps they received to the partitions in a second round of communication (lines 14–16). For each item in the read request, RAMP-S servers return the highest-timestamped version of the item that also appears in the supplied set of timestamps (lines 5–6). Readers subsequently return the results from the mandatory second round of requests.

By example. In Figure 5.3, under RAMP-S, P_x and P_y respectively return the sets $\{1\}$ and $\{\bot\}$ in response to T_2's first round of reads. T_2 would subsequently send the set $\{1, \bot\}$ to both P_x and P_y, which would return x_1 and y_1. (Including $\bot$ in the second-round request is unnecessary, but we leave it in for ease of understanding.)

Why it works. In RAMP-S, if a transaction has committed on some but not all partitions, the transaction timestamp will be returned in the first round of any concurrent read transaction accessing the committed partitions' items. In the (required) second round of read requests,

```
    Server-side Data Structures
 1: versions: set of versions ⟨item, value, timestamp ts_v, metadata md⟩
 2: lastCommit[i]: last committed timestamp for item i
    Server-side Methods
 3: procedure PREPARE(v : version)
 4:     versions.add(v)
 5:     return
 6: procedure COMMIT(ts_c : timestamp)
 7:     I_ts ← {w.item | w ∈ versions ∧ w.ts_v = ts_c}
 8:     ∀i ∈ I_ts, lastCommit[i] ← max(lastCommit[i], ts_c)
 9: procedure GET(i : item, ts_req : timestamp)
10:     if ts_req = ∅ then
11:         return v ∈ versions : v.item = i ∧ v.ts_v = lastCommit[item]
12:     else
13:         return v ∈ versions : v.item = i ∧ v.ts_v = ts_req
    ---------------------------------------------------------------
    Client-side Methods
14: procedure PUT_ALL(W : set of ⟨item, value⟩)
15:     ts_tx ← generate new timestamp
16:     I_tx ← set of items in W
17:     parfor ⟨i, v⟩ ∈ W do
18:         v ← ⟨item = i, value = v, ts_v = ts_tx, md = (I_tx − {i})⟩
19:             invoke PREPARE(v) on respective server (i.e., partition)
20:
21:     parfor server s : s contains an item in W do
22:         invoke COMMIT(ts_tx) on s
23:
24: procedure GET_ALL(I : set of items)
25:     ret ← {}
26:     parfor i ∈ I do
27:         ret[i] ← GET(i, ∅)
28:
29:     v_latest ← {} (default value: −1)
30:     for response r ∈ ret do
31:         for i_tx ∈ r.md do
32:             v_latest[i_tx] ← max(v_latest[i_tx], r.ts_v)
33:     parfor item i ∈ I do
34:         if v_latest[i] > ret[i].ts_v then
35:             ret[i] ← GET(i, v_latest[i])
36:
37:     return ret
```

Algorithm 1: RAMP-Fast

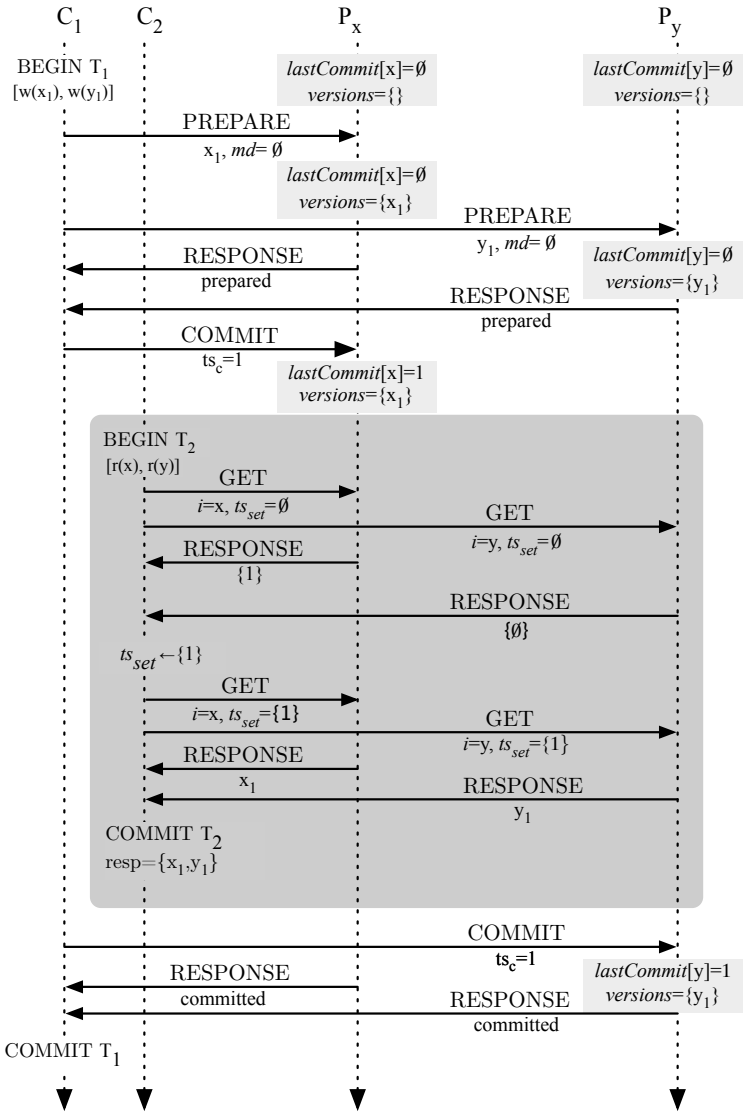


Figure 5.3: Space-time diagram for RAMP-S execution for two transactions T_1 and T_2 performed by clients C_1 and C_2 on partitions P_x and P_y. Lightly-shaded boxes represent current partition state (lastCommit and versions), while the single darker box encapsulates all messages exchanged during C_2's execution of transaction T_2. T_1 first fetches the highest committed timestamp from each partition, then fetches the corresponding version. In this depiction, partitions only return timestamps instead of actual versions in response to first-round reads.

any prepared-but-not-committed partitions will find the committed timestamp in the reader-provided set and return the appropriate version. In contrast with RAMP-F, where readers explicitly provide partitions with a specific version to return in the (optional) second round,

Server-side Data Structures
same as in RAMP-F (Algorithm 1)

Server-side Methods
PREPARE, COMMIT same as in RAMP-F

```
1:  procedure GET(i : item, ts_set : set of timestamps)
2:      if ts_set = ∅ then
3:          return v ∈ versions : v.item = i ∧ v.ts_v = lastCommit[k]
4:      else
5:          ts_match = {t | t ∈ ts_set ∧ ∃v ∈ versions : v.item = i ∧ v.t_v = t}
6:          return v ∈ versions : v.item = i ∧ v.ts_v = max(ts_match)
```

Client-side Methods

```
7:  procedure PUT_ALL(W : set of ⟨item, value⟩)
        same as RAMP-F PUT_ALL but do not instantiate md on line 18
8:  procedure GET_ALL(I : set of items)
9:      ts_set ← {}
10:     parfor i ∈ I do
11:         ts_set.add(GET(i, ∅).ts_v)
12:
13:     ret ← {}
14:     parfor item i ∈ I do
15:         ret[i] ← GET(i, ts_set)
16:
17:     return ret
```

Algorithm 2: RAMP-Small

RAMP-S readers defer the decision of which version to return to the partition, which uses the reader-provided set to decide. This saves metadata but increases RTTs, and the size of the parameters of each second-round GET request is (worst-case) linear in the read set size. Unlike RAMP-F, there is no requirement to return the value of the last committed version in the first round (returning the version, lastCommit[k], suffices in line 3).

5.4.3 RAMP-Hybrid: An Intermediate Solution

RAMP-Hybrid (RAMP-H; Algorithm 3) strikes a compromise between RAMP-F and RAMP-S. RAMP-H and RAMP-S write protocols are identical, but, instead of storing the entire write set (as in RAMP-F), RAMP-H writers store a Bloom filter [59] representing the transaction write set (line 1). RAMP-H readers proceed as in RAMP-F, with a first round of communication to fetch the last-committed version of each item from its partition (lines 3–6). Given this set of versions, RAMP-H readers subsequently compute a list of *potentially* higher-timestamped

writes for each item (lines 8–11). Any potentially missing versions are fetched in a second round of reads (lines 14).

By example. In Figure 5.2, under RAMP-H, x_1 would contain a Bloom filter with positives for x and y and $y_\perp$ would contain an empty Bloom filter. T_2 would check for the presence of y in x_1's Bloom filter (since x_1's version is 1 and $1 > \perp$) and, finding a match, conclude that it is potentially missing a write (y_1). T_2 would subsequently fetch y_1 from P_y.

Why it works. RAMP-H is effectively a hybrid between RAMP-F and RAMP-S. If the Bloom filter has no false positives, RAMP-H reads behave like RAMP-F reads. If the Bloom filter has all false positives, RAMP-H reads behave like RAMP-S reads. Accordingly, the number of (unnecessary) second-round reads (i.e., which would not be performed by RAMP-F) is controlled by the Bloom filter false positive rate, which is in turn (in expectation) proportional to the size of the Bloom filter. Any second-round GET requests are accompanied by a set of timestamps that is also proportional in size to the false positive rate. Therefore, RAMP-H exposes a trade-off between metadata size and expected performance. To understand why RAMP-H is safe, we simply have to show that any false positives (second-round reads) will not compromise the integrity of the result set; with unique timestamps, any reads due to false positives will return null.

5.4.4 Summary and Additional Details

The RAMP algorithms allow readers to safely race writers without requiring either to stall. The metadata attached to each write allows readers in all three algorithms to safely handle concurrent and/or partial writes and in turn allows a trade-off between metadata size and performance (Table 5.1): RAMP-F is optimized for fast reads, RAMP-S is optimized for small metadata, and RAMP-H is, as the name suggests, a middle ground. RAMP-F requires metadata linear in transaction size, while RAMP-S and RAMP-H require constant metadata. However, RAMP-S and RAMP-H require more RTTs for reads compared to RAMP-F when there is no race between readers and writers. When reads and writes race, in the worst case, all algorithms require two RTTs for reads. Writes always require two RTTs to prevent readers from stalling due to missing, unprepared writes.

RAMP algorithms are scalable because clients only contact partitions directly accessed by their transactions (partition independence), and clients cannot stall one another (are coordination-free). More specifically, readers do not interfere with other readers, writers do not interfere with other writers, and readers and writers can proceed concurrently. When a reader races a writer to the same items, the writer's new versions will only become visible to the reader (i.e., be committed) once it is guaranteed that the reader will be able to fetch

Server-side Data Structures
same as in RAMP-F (Algorithm 1)

Server-side Methods
PREPARE, COMMIT same as in RAMP-F
GET same as in RAMP-S

Client-side Methods

```
1:  procedure PUT_ALL(W : set of ⟨item, value⟩)
        same as RAMP-F PUT_ALL but instantiate md on line 18
        with Bloom filter containing I_tx

2:  procedure GET_ALL(I : set of items)
3:      ret ← {}
4:      parfor i ∈ I do
5:          ret[i] ← GET(i, ∅)
6:
7:      v_fetch ← {}
8:      for version v ∈ ret do
9:          for version v' ∈ ret : v' ≠ v do
10:             if v.ts_v > v'.ts_v ∧ v.md.lookup(v'.item) → True then
11:                 v_fetch[v'.item].add(v.ts_v)
12:     parfor item i ∈ v_fetch do
13:         ret[i] ← GET(k, v_fetch[i]) if GET(k, v_fetch[i]) ≠ ⊥
14:
15:     return ret
```

Algorithm 3: RAMP-Hybrid

all of them (possibly via a second round of communication). A reader will *never* have to stall waiting for writes to arrive at a partition (for details, see Invariant 1 in the Appendix); however, the reader may have to contact the servers twice in order to fetch any versions that were missing from its first set of reads.

Below, we discuss relevant implementation details.

Multi-versioning and garbage collection. RAMP transactions rely on multi-versioning to allow readers to access versions that have not yet committed and/or have been overwritten. In our pseudocode, we have presented an implementation based on multi-versioned storage; in practice, multi-versioning can be implemented by using a single-versioned storage engine for retaining the last committed version of each item and using a "look-aside" store for access to both prepared-but-not-yet-committed writes and (temporarily) any overwritten versions. The look-aside store should make prepared versions durable but can—at the risk of aborting transactions in the event of a server failure—simply store any overwritten versions in memory. Thus, with some work, RAMP algorithms are portable to non-multi-

Algorithm	RTTs/transaction			Metadata (+stamp)	
	W	R (stable)	R (O)	Stored	Per-Request
RAMP-F	2	1	2	txn items	-
RAMP-S	2	2	2	-	stamp/item
RAMP-H	2	$1+\epsilon$	2	Bloom filter	stamp/item

Table 5.1: Comparison of basic algorithms: RTTs required for writes (W), reads (R) without concurrent writes and in the worst case (O), stored metadata and metadata attached to read requests (in addition to a timestamp for each).

versioned storage systems.

In both architectures, each partition's data will grow without bound if old versions are not removed. If a committed version of an item is not the highest-timestamped committed version (i.e., a committed version v of item k where $v < \texttt{lastCommit}[k]$), it can be safely discarded (i.e., garbage collected, or GCed) as long as no readers will attempt to access it in the future (via second-round GET requests). It is easiest to simply limit the running time of read transactions and GC overwritten versions after a fixed amount of real time has elapsed. Any read transactions that take longer than this GC window can be restarted [167, 168]. Therefore, the maximum number of versions retained for each item is bounded by the item's update rate, and servers can reject any client GET requests for versions that have been GCed (and the read transaction can be restarted). This violates availability under asynchronous network behavior, so, as a fallback and a more principled solution, partitions can also gossip the timestamps of items that have been overwritten and have not been returned in the first round of any ongoing read transactions. Under RAMP-F, if a second-round read request arrives a server and the server does not have that version due to garbage collection, it can safely ignore the request or signal failure.

Read-write transactions. Until now, we have focused on read-only and write-only transactions. However, we can extend our algorithms to provide read-write transactions. If transactions pre-declare the data items they wish to read, then the client can execute a GET_ALL transaction at the start of transaction execution to pre-fetch all items; subsequent accesses to those items can be served from this pre-fetched set. Clients can buffer any writes and, upon transaction commit, send all new versions to servers (in parallel) via a PUT_ALL request. As in Section 5.3, this may result in anomalies due to concurrent update but does not violate RA isolation. Given the benefits of pre-declared read/write sets [89, 187, 217] and write buffering [85, 207], we believe this is a reasonable strategy. For secondary index lookups, clients can first look up secondary index entries then subsequently (within the same transaction) read primary data (specifying versions from index entries as appropriate).

Timestamps. Timestamps should be unique across transactions, and, for "session" consistency (Appendix), increase on a per-client basis. Given unique client IDs, a client ID and sequence number form unique transaction timestamps without coordination. Without unique client IDs, servers can assign unique timestamps with high probability using UUIDs and by hashing transaction contents.

Overwrites. In our algorithms, versions are overwritten according to a highest-timestamp-wins policy. In practice, and, for commutative updates, users may wish to employ a different policy upon COMMIT: for example, perform set union. In this case, `lastCommit[k]` contains an abstract data type (e.g., set of versions) that can be updated with a `merge` operation [95, 216] (instead of `updateIfGreater`) upon commit. This treats each committed record as a set of versions, requiring additional metadata (that can be GCed as in Section 5.4.7).

5.4.5 Distribution and Fault Tolerance

RAMP transactions operate in a distributed setting, which poses challenges due to latency, partial failure, and network partitions. Under coordination-free execution, failed clients do not cause other clients to fail, while partition independence ensures that clients only have to contact partitions for items in their transactions. This provides fault tolerance and availability as long as clients can access relevant partitions. In this section, we address incident concerns. First, replication can be used to increase the number of servers hosting a partition, thus increasing availability. Second, we describe the RAMP protocol behavior when clients are unable to contact servers.

Replication. RAMP protocols can benefit from a variety of mechanisms including traditional database master-slave replication with failover, quorum-based protocols, and state machine replication, which increase the number of physical servers that host a given data item [53]. To improve durability, RAMP clients can wait until the effects of their operations (e.g., modifications to *versions* and *lastCommit*) are persisted to multiple physical servers before returning from PUT_ALL calls (either via master-to-slave replication or via quorum replication and by performing two-phase commit across multiple active servers). Notably, because RAMP transactions can safely overlap in time, replicas can process different transactions' PREPARE and COMMIT requests in parallel. Availability can also benefit in many protocols, such as quorum replication. We discuss more advanced replication techniques in Section 5.6.1.

Stalled Operations. RAMP writes use a two-phase atomic commitment protocol that ensures readers never block waiting for writes to arrive. As discussed in Section 6.4.1,

every ACP may block during failures [53]. However, under coordination-free execution, a blocked transaction (due to failed clients, failed servers, or network partitions) cannot cause other transactions to block. Stalled writes act only as "resource leaks" on partitions: partitions will retain prepared versions indefinitely unless action is taken.

To "free" these leaks, RAMP servers can use the Cooperative Termination Protocol (CTP) described in [53]. CTP can always complete the transaction except when every partition has performed PREPARE but no partition has performed COMMIT. In CTP, if a server S_p has performed PREPARE for transaction T but times out waiting for a COMMIT, S_p can check the status of T on any other partitions for items in T's write set. If another server S_c has received COMMIT for T, then S_p can COMMIT T. If S_a, a server responsible for an item in T, has not received PREPARE for T, S_a and S_p can promise never to PREPARE or COMMIT T in the future and S_p can safely discard its versions. Under CTP, if a client blocks mid-COMMIT, the servers will ensure that the writes will eventually COMMIT and therefore become visible on all partitions. A client recovering from a failure can read from the servers to determine if they unblocked its write.

CTP only runs when writes block (or time-outs fire) and runs *asynchronously* with respect to other operations. CTP requires that PREPARE messages contain a list of servers involved in the transaction (a subset of `RAMP-F` metadata but a superset of `RAMP-H` and `RAMP-S`) and that servers remember when they COMMIT and "abort" writes (e.g., in a log file). Compared to alternatives (e.g., replicating clients [124]), we have found CTP to be both lightweight and effective. We evaluate CTP in Section 5.5.3.

5.4.6 Additional Semantics

While our RAMP transactions provide RA isolation, they also provide a number of additional useful guarantees. With linearizable servers, once a user's operation completes, all other users will observe its effects (regular register semantics, applied at the transaction level); this provides a notion of real-time recency. This also ensures that each user's operations are visible in the order in which they are committed. Our RAMP implementations provide a variant of PRAM consistency, where, for each item, each user's writes are serialized [164] (i.e., "session" ordering [91]). For example, if a user updates her privacy settings and subsequently posts a new photo, the photo cannot be read without the privacy setting change [83]. However, PRAM does not respect the *happens-before* relation [157] across users (or missing dependencies, as discussed in Section 5.3.3). If Sam reads Mary's comment and replies to it, other users may read Sam's comment without Mary's comment. We further discuss this issue in Section 5.6.3.

5.4.7 Further Optimizations

RAMP algorithms also allow several possible optimizations:

Faster commit detection. If a server returns a version in response to a GET request, then the transaction that created the version must have issued a COMMIT on at least one server. In this case, the server can safely mark the version as committed and update lastCommit. This means that the transaction commit will be reflected in any subsequent GET requests that read from lastCommit for this item—even though the COMMIT message from the client may yet be delayed. The net effect is that the later GET requests may not have to issue second-round reads to fetch the versions that otherwise would not have been marked as committed. This scenario will occur when all partitions have performed PREPARE and at least one server but not all partitions have performed COMMIT (as in CTP). This allows faster updates to lastCommit (and therefore fewer expected RAMP-F and RAMP-H RTTs).

Metadata garbage collection. Once all of transaction T's writes are committed on each respective partition (i.e., are reflected in lastCommit), readers are guaranteed to read T's writes (or later writes). Therefore, non-timestamp metadata for T's writes stored in RAMP-F and RAMP-H (write sets and Bloom filters) can be discarded. Detecting that all servers have performed COMMIT can be performed asynchronously via a third round of communication performed by either clients or servers.

One-phase writes. We have considered two-phase writes, but, if a user does not wish to read her writes (thereby sacrificing session guarantees outlined in Section 5.4.6), the client can return after issuing its PREPARE round (without sacrificing durability). The client can subsequently execute the COMMIT phase asynchronously, or, similar to optimizations presented in Paxos Commit [124], the servers can exchange PREPARE acknowledgments with one another and decide to COMMIT autonomously. This optimization is safe because multiple PREPARE phases can safely overlap. We leverage a similar observation in Section 5.6.1.

5.5 Experimental Evaluation

We proceed to experimentally demonstrate RAMP transaction scalability as compared to existing transactional and non-transactional mechanisms. RAMP-F, RAMP-H, and often RAMP-S outperform existing solutions across a range of workload conditions while exhibiting overheads typically within 8% and no more than 48% of peak throughput. As expected from our theoretical analysis, the performance of our RAMP algorithms does not degrade substantially under contention and scales linearly to over 7.1 million operations per second on 100 servers. These outcomes validate our goal of coordination-free design.

5.5.1 Experimental Setup

To demonstrate the effect of concurrency control on performance and scalability, we implemented several concurrency control algorithms in a partitioned, multi-versioned, main-memory database prototype. Our prototype is in Java and employs a custom RPC system for serialization. Servers are arranged as a distributed hash table [211] with partition placement determined by random hashing of keys to servers. As in stores like Dynamo [95], clients can connect to any server to execute operations, which the server will perform on their behalf (i.e., each server acts as a client in our RAMP pseudocode). We implemented `RAMP-F`, `RAMP-S`, and `RAMP-H` and configure a wall-clock GC window of 5 seconds as described in Section 5.4.4. `RAMP-H` uses a 256-bit Bloom filter based on an implementation of MurmurHash2.0 [23], with four hashes per entry; to demonstrate the effects of filter saturation, we do not modify these parameters in our experiments. Our prototype utilizes the faster commit detection optimization from Section 5.4.4. We chose not to employ metadata garbage collection and one-phase writes in order to preserve session guarantees and because metadata overheads were generally minor.

Algorithms for comparison. As a baseline, we do not employ any concurrency control (denoted `NWNR`, for no write and no read locks); reads and writes take one RTT and are executed in parallel.

We also consider three lock-based mechanisms [125]: long write locks and long read locks, providing Repeatable Read isolation (*PL-2.99*; denoted `LWLR`), long write locks with short read locks, providing Read Committed isolation (*PL-2L*; denoted `LWSR`; does not provide RA), and long write locks with no read locks, providing Read Uncommitted isolation (`LWNR`; also does not provide RA). While only `LWLR` provides RA, `LWSR` and `LWNR` provide a useful basis for comparison, particularly in measuring concurrency-related locking overheads. To avoid deadlocks, the system lexicographically orders lock requests by item and performs them sequentially. When locks are not used (as for reads in `LWNR` and reads and writes for `NWNR`), the system parallelizes operations.

We also consider an algorithm where, for each transaction, designated "coordinator" servers enforce RA isolation—effectively, the Eiger system's 2PC-PCI mechanism [168] (denoted `E-PCI`; Chapter 7). Writes proceed via prepare and commit rounds, but any reads that arrive at a partition and while a write transaction to the same item is pending must contact a (randomly chosen, per-write-transaction) "coordinator" partition to determine whether the coordinator's prepared writes have been committed. Writes require two RTTs, while reads require one RTT during quiescence and two RTTs in the presence of concurrent updates (to a variable number of coordinator partitions—linear in the number of concurrent writes to the item). Using a coordinator violates partition independence but—in this case—

is still coordination-free. We optimize 2PC-PCI reads by having clients determine a read timestamp for each transaction (eliminating an RTT) and do not include happens-before metadata.

This range of lock-based strategies (LWNR, LWSR, LWNR), recent comparable approach (E-PCI), and best-case (NWNR; no concurrency control) baseline provides a spectrum of strategies for comparison.

Environment and benchmark. We evaluate each algorithm using the YCSB benchmark [84] and deploy variably-sized sets of servers on public cloud infrastructure. We employ `cr1.8xlarge` instances on Amazon EC2 and, by default, deploy five partitions on five servers. We group sets of reads and sets of writes into read-only and write-only transactions (default size: 4 operations), and use the default YCSB workload (`workloada`, with Zipfian distributed item accesses) but with a 95% read and 5% write proportion, reflecting read-heavy applications (Section 6.4.1, [65,168,227]; e.g., Tao's 500 to 1 reads-to-writes [65,168], Espresso's 1000 to 1 Mailbox application [192], and Spanner's 3396 to 1 advertising application [85]).

By default, use 5000 YCSB clients distributed across 5 separate EC2 instances. As in stock YCSB, each client makes a sequence of synchronous requests to the database. When we later vary the number of clients, we keep the number of servers hosting the clients fixed. To fully expose our metadata overheads, use a value size of 1 byte per write. We found that lock-based algorithms were highly inefficient for YCSB's default 1000 item database, so we increased the database size to one million items by default to decrease contention. Each version contains a timestamp (64 bits), and, with YCSB keys (i.e., item IDs) of size 11 bytes and a transaction length L, RAMP-F requires 11L bytes of metadata per version, while RAMP-H requires 32 bytes. We successively vary several parameters, including number of clients, read proportion, transaction length, value size, database size, and number of servers and report the average of three sixty-second trials.

5.5.2 Experimental Results: Comparison

Our first set of experiments focuses on two metrics: performance compared to baseline and performance compared to existing techniques. The overhead of RAMP algorithms is typically less than 8% compared to baseline (NWNR) throughput, is sometimes zero, and is never greater than 50%. RAMP-F and RAMP-H always outperform the lock-based and E-PCI techniques, while RAMP-S outperforms lock-based techniques and often outperforms E-PCI. We proceed to demonstrate this behavior over a variety of conditions:

Number of clients. RAMP performance scales well with increased load and incurs little overhead (Figure 6.1). With few concurrent clients, there are few concurrent updates and

therefore few second-round reads; performance for RAMP-F and RAMP-H is close to or even matches that of NWNR. At peak throughput with 10,000 clients, RAMP-F and RAMP-H pay a throughput overhead of 4.2% compared to NWNR. RAMP-F and RAMP-H exhibit near-identical performance; the RAMP-H Bloom filter triggers few false positives and therefore few extra RTTs compared to RAMP-F. RAMP-S incurs greater overhead and peaks at almost 60% of the throughput of NWNR. Its guaranteed two-round trip reads are expensive and it acts as an effective lower bound on RAMP-F and RAMP-H performance. In all configurations, the algorithms achieve low latency. RAMP-F, RAMP-H, NWNR less than 35 ms on average and less than 10 ms at 5,000 clients; RAMP-S less than 53 ms, 14.3 ms at 5,000 clients.

In comparison, the remaining algorithms perform less favorably. In contrast with the RAMP algorithms, E-PCI servers must check a coordinator server for each in-flight write transaction to determine whether to reveal writes to clients. For modest load, the overhead of these commit checks places E-PCI performance between that of RAMP-S and RAMP-H. Under YCSB's Zipfian workload, there is a high probability that the several "hot" keys in the workload have a pending write, requiring a E-PCI commit check. The number of in-flight writes further increases with load, increasing the number of E-PCI commit checks. This in turn decreases throughput, and, with 10,000 concurrent clients, E-PCI performs so many commit checks per read that it underperforms the LWNR lock-based scheme. Under this configuration, more than 20% of reads trigger a commit check, and, on servers with hot items, each commit check requires indirected coordinator checks for an average of 9.84 transactions. Meanwhile, multi-partition locking is expensive [187]: with 10,000 clients, the most efficient algorithm, LWNR, attains only 28.6% of the throughput of NWNR, while the least efficient, LWLR, attains only 1.6% and peaks at 3,412 transactions per second.

We subsequently varied several other workload parameters, which we discuss below and plot in Figure 5.5:

Read proportion. Increased write activity leads to a greater number of races between reads and writes and therefore additional second-round RTTs for RAMP-F and RAMP-H reads. With all write transactions, all RAMP algorithms are equivalent (two RTT) and achieve approximately 65% of the throughput of NWNR. With all reads, RAMP-F, RAMP-S, NWNR, and E-PCI are identical, with a single RTT. Between these extremes, RAMP-F and RAMP-S scale near-linearly with the write proportion. In contrast, lock-based protocols fare poorly as contention increases, while E-PCI again incurs penalties due to commit checks.

Transaction length. Increased transaction lengths have variable impact on the relative performance of RAMP algorithms. Coordination-free execution ensures long-running transactions are not penalized, but, with longer transactions, metadata overheads increase. RAMP-F relative throughput decreases due to additional metadata (linear in transaction length) and

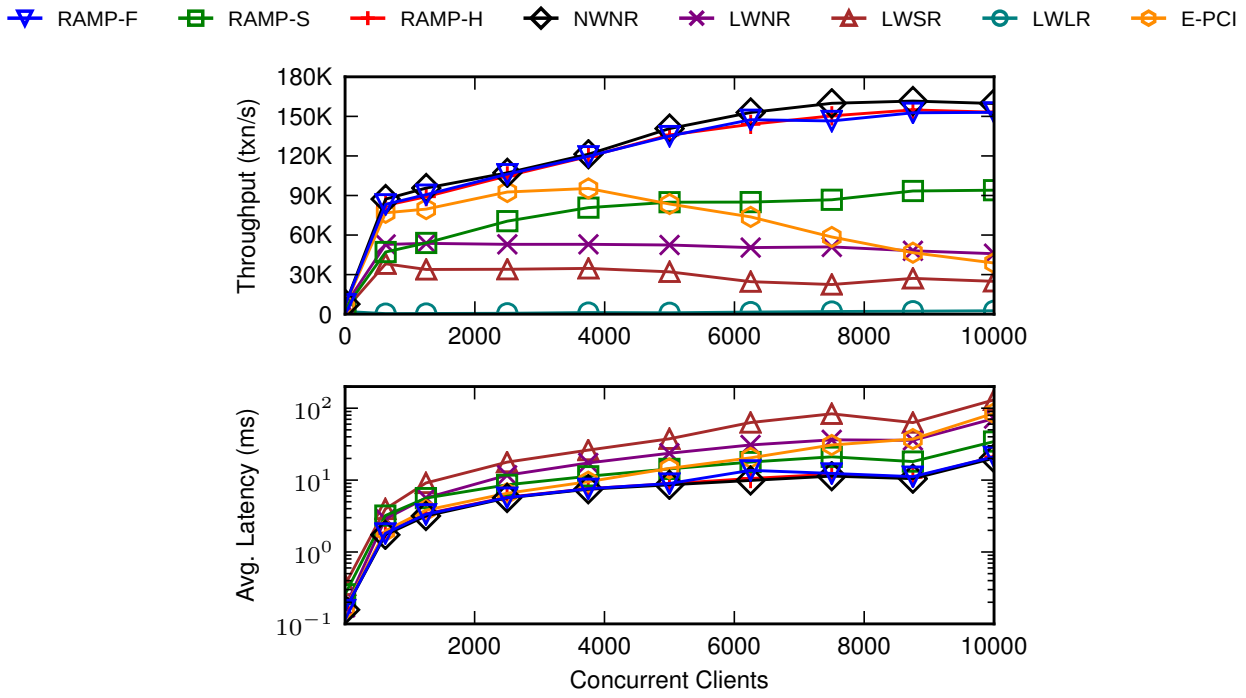


Figure 5.4: Throughput and latency under varying client load. We omit latencies for LWLR, which peaked at over 1.5s.

RAMP-H relative performance also decreases as its Bloom filters saturate. (However, YCSB's Zipfian-distributed access patterns result in a non-linear relationship between length and throughput.) As discussed above, we explicitly decided not to tune RAMP-H Bloom filter size, but a logarithmic increase in filter size could improve RAMP-H performance for large transaction lengths (e.g., 1024 bit filters should lower the false positive rate for transactions of length 256 from over 92% to slightly over 2%).

Value size. Value size similarly does not seriously impact relative throughput. At a value size of 1B, RAMP-F is within 2.3% of NWNR. However, at a value size of 100KB, RAMP-F performance nearly matches that of NWNR: the overhead due to metadata decreases, and write request rates slow, decreasing concurrent writes (and subsequently second-round RTTs). Nonetheless, absolute throughput drops by a factor of 24 as value sizes moves from 1B to 100KB.

Database size. RAMP algorithms are robust to high contention for a small set of items: with only 1000 items in the database, RAMP-F achieves throughput within 3.1% of NWNR. RAMP algorithms are largely agnostic to read/write contention, although, with fewer items in the database, the probability of races between readers and in-progress writers increases,

resulting in additional second-round reads for RAMP-F and RAMP-H. In contrast, lock-based algorithms fare poorly under high contention, while E-PCI indirected commit checks again incurred additional overhead. By relying on clients (rather than additional partitions) to repair fractured writes, RAMP-F, RAMP-H, and RAMP-S performance is less affected by hot items.

Overall, RAMP-F and RAMP-H exhibit performance close to that of no concurrency control due to their independence properties and guaranteed worst-case performance. As the proportion of writes increases, an increasing proportion of RAMP-F and RAMP-H operations take two RTTs and performance trends towards that of RAMP-S, which provides a constant two RTT overhead. In contrast, lock-based protocols perform poorly under contention while E-PCI triggers more commit checks than RAMP-F and RAMP-H trigger second round reads (but still performs well without contention and for particularly read-heavy workloads). The ability to allow clients to independently verify read sets enables good performance despite a range of (sometimes adverse) conditions (e.g., high contention).

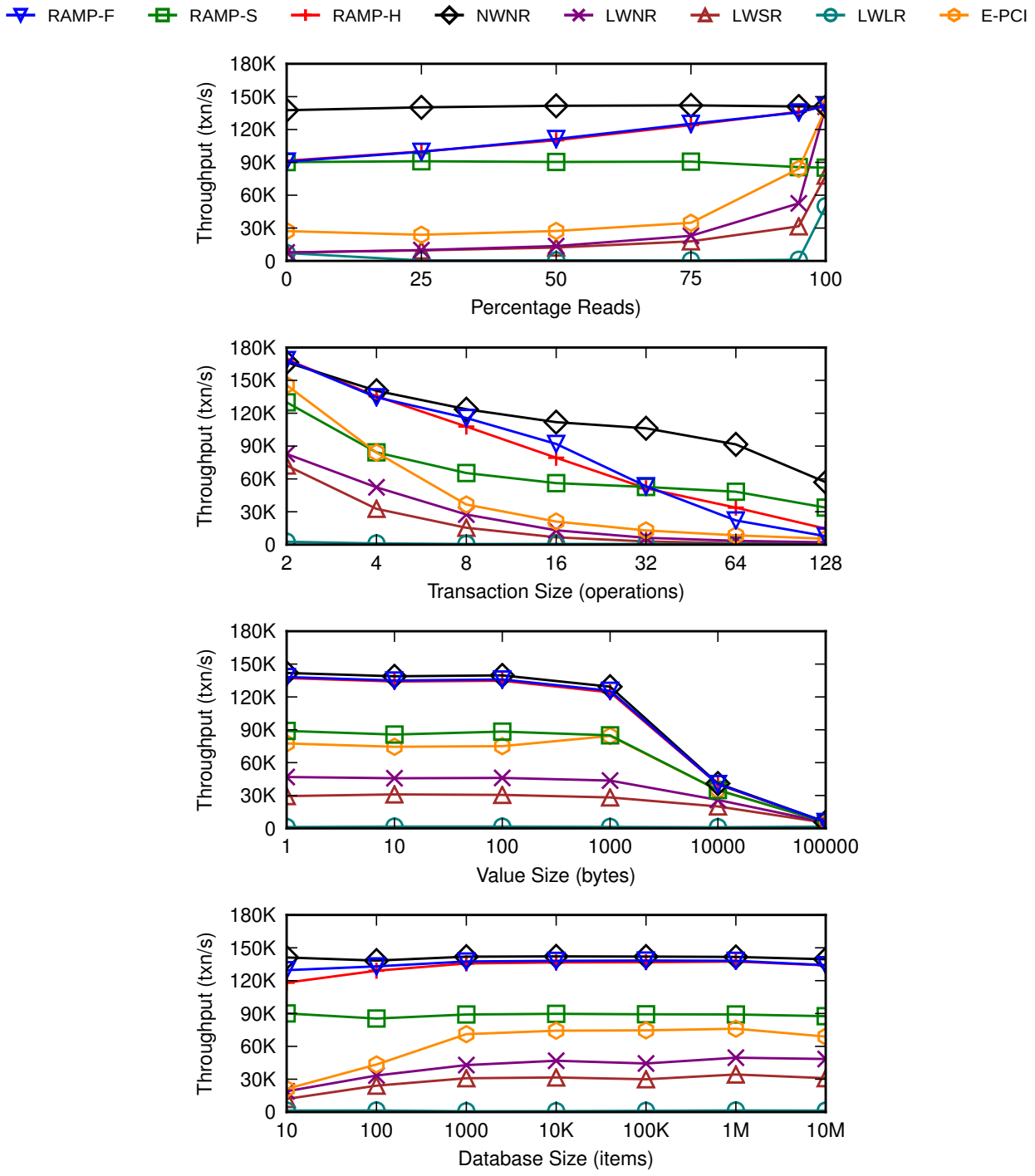


Figure 5.5: Algorithm performance across varying workload conditions. `RAMP-F` and `RAMP-H` exhibit similar performance to `NWNR` baseline, while `RAMP-S`'s 2 RTT reads incur a greater performance penalty across almost all configurations. RAMP transactions consistently outperform RA isolated alternatives.

5.5.3 Experimental Results: CTP Overhead

We also evaluated the overhead of blocked writes in our implementation of the Cooperative Termination Protocol discussed in Section 5.4.5. To simulate blocked writes, we artificially dropped a percentage of COMMIT commands in PUT_ALL calls such that clients returned from writes early and partitions were forced to complete the commit via CTP. This behavior is worse than expected because "blocked" clients continue to issue new operations. The table below reports the throughput reduction as the proportion of blocked writes increases (compared to no blocked writes) for a workload of 100% `RAMP-F` write transactions:

Blocked %	0.01%	0.1%	25%	50%
Throughput	No change	99.86%	77.53%	67.92%

As these results demonstrate, CTP can reduce throughput because each commit check consumes resources (namely, network and CPU capacity). However, CTP only performs commit checks in the event of blocked writes (or time-outs; set to 5s in our experiments), so a modest failure rate of 1 in 1000 writes has a limited effect. The higher failure rates produce a near-linear throughput reduction but, in practice, a blocking rate of even a few percent is likely indicative of larger systemic failures. As Figure 5.5 hints, the effect of additional metadata for the participant list in `RAMP-H` and `RAMP-S` is limited, and, for our default workload of 5% writes, we observe similar trends but with throughput degradation of 10% or less across the above configurations. This validates our initial motivation behind the choice of CTP: average-case overheads are small.

5.5.4 Experimental Results: Scalability

We finally validate our chosen scalability criteria by demonstrating linear scalability of RAMP transactions to 100 servers. We deployed an increasing number of servers within the `us-west-2` EC2 region and, to mitigate the effects of hot items during scaling, configured uniform random access to items. We were unable to include more than 20 instances in an EC2 "placement group," which guarantees 10 GbE connections between instances, so, past 20 servers, servers communicated over a degraded network. At around 40 servers, we exhausted the `us-west-2b` "availability zone" (datacenter) capacity and had to allocate our instances across the remaining zones, further degrading network performance. To avoid bottlenecks on the client, we deploy as many instances to host YCSB clients as we do to host prototype servers.However, as shown in Figure 6.2, each RAMP algorithm scales linearly. In expectation, at 100 servers, almost all transactions span multiple servers: all but one in 100M transactions is a multi-partition operation, highlighting the importance of partition

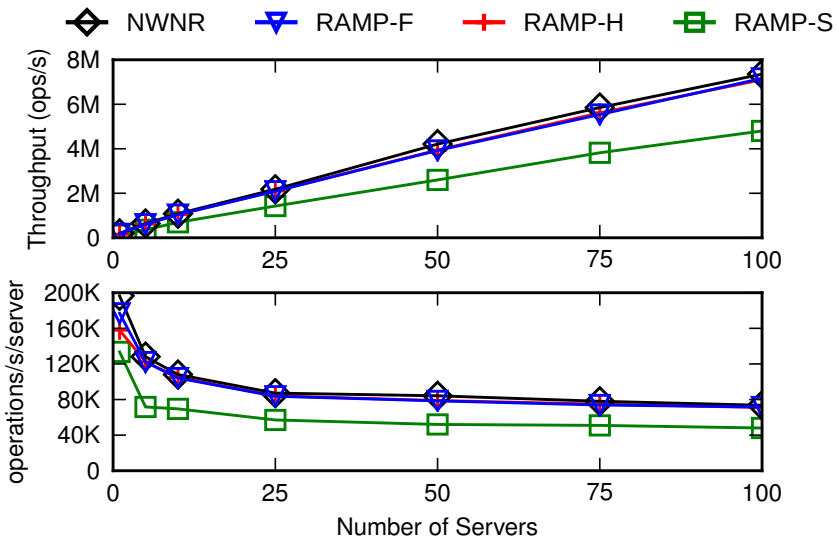


Figure 5.6: RAMP transactions scale linearly to over 7 million operations/s with comparable performance to NWNR baseline.

independence. With 100 servers, RAMP-F achieves slightly under 7.1 million operations per second, or 1.79 million transactions per second on a set of 100 servers (71,635 operations per partition per second). At all scales, RAMP-F throughput was always within 10% of NWNR. With 100 servers, RAMP-F was within 2.6%, RAMP-S within 3.4%, and RAMP-S was within 45% of NWNR. In light of our scalability criteria, this behavior is unsurprising.

5.6 Applying and Modifying the RAMP Protocols

In this section, we discuss modifications to RAMP to enable multi-datacenter and efficient quorum replication as well as causally consistent operation. Our goals here are two-fold. First, we believe this section will be beneficial to systems implementers integrating RAMP protocols into databases such as Cassandra [153] that support wide-area and quorum-replicated deployments. Indeed, its inclusion is a reflection on many helpful reader comments asking for clarification on this topic. Second, we believe this material is a useful inclusion for readers who are familiar with existing and recent work on both multi-datacenter and causally consistent replication. Namely, RAMP is compatible with many of these replication scenarios, and, in some cases, enables new optimizations.

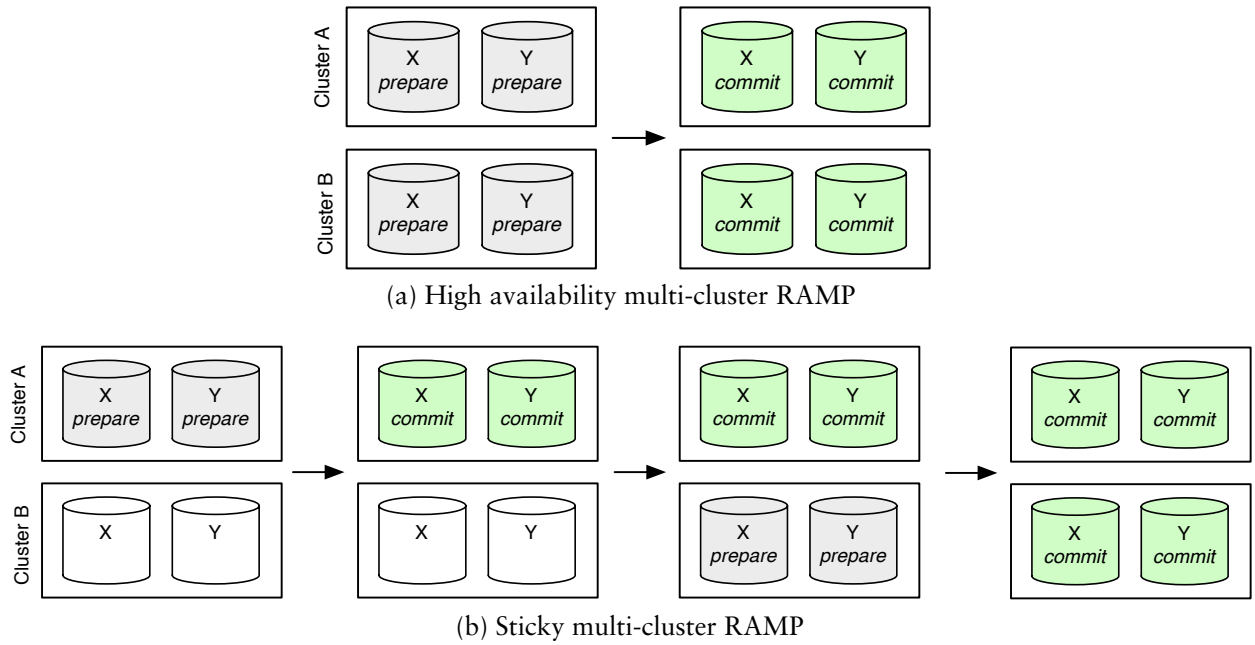
5.6.1 Multi-Datacenter RAMP

The RAMP algorithms presented in this work have assumed linearizable server operation. Hence, if RAMP is used in a system where data items are replicated, then a linearizable replication mechanism must be used, such as a primary-backup or other replicated state machine approach. While this has simplified our discussion and results in reasonable performance in many environments, the cost of linearizability is often be expensive, particularly in geo-replicated environments where latency is lower-bounded by the speed of light [8, 32]. While the RAMP algorithms' lack of coordination mitigates throughput penalties due to, for example, stalls during contended multi-partition access, actually accessing the partitions themselves may take time and increase the latency of individual operations. Moreover, in the event of partial failures, it is often beneficial to provide greater availability guarantees.

In this section, we discuss strategies for lowering the latency and improving the availability of operations. Our primary target in this setting is a multi-datacenter, geo-replicated context, where servers are located in separate clusters in possibly geographically remote regions. This setting has received considerable attention in recent research and, increasingly, in some of the largest production data management systems [85, 167, 168, 210]. The actual porting of concurrency control algorithms to this context is not terribly difficult, but any inefficiencies due to synchronization and coordination are magnified in this setting, making it an idea candidate for practical study. Thus, we couch our discussion in the context of fully-replicated, geo-distributed clusters (i.e., groups of replicas of each partition).

The key challenge in achieving higher availability and lower latency in RAMP is ensuring that partially committed writes can still be completed. In the standard RAMP algorithms, this is accomplished by waiting to commit until after all partitions have prepared. Yet, in a replicated context, this waiting is potentially expensive; over wide-area networks, this can take hundreds of milliseconds. There are two straightforward ways to circumvent these overheads: deferring the commit operation and maintaining stickiness.

Prepare-F HA RAMP. The first strategy is easier to understand but perhaps less practical. A client specifies a minimum durability for its write operations, measured in terms of number of failures it wishes to survive, F. When writing, the client issues a prepare request to all clusters and waits until it receives a successful response from $F + 1$ servers. This ensures that the client's write is durable, and the client knows its intent has been logged on at least $F + 1$ servers. The client transaction subsequently returns success (Figure 5.7a). Once all servers have received the prepare request (which is detectable via either via server-server communication as in the CTP protocol or via an asynchronous callback on the client), the servers can begin to commit the client's writes autonomously. This preserves RA isolation—but at a cost. Namely, there is no guarantee of *visibility* of writes: a client is not guaranteed



(a) High availability multi-cluster RAMP

(b) Sticky multi-cluster RAMP

Figure 5.7: Control flow for operations under multi-datacenter RAMP strategies with client in Cluster A writing to partitions X and Y. In the high availability RAMP strategy (Figure 5.7a), a write must be prepared on $F + 1$ servers (here, $F = 3$) before is committed. In the sticky RAMP strategy, a write can be prepared and committed within a single datacenter and asynchronously propagated to other datacenters, where it is subsequently prepared and committed (Figure 5.7b). The sticky strategy requires that clients maintain affinity with a single cluster in order to guarantee available and correctly isolated behavior.

to read its own writes. Moreover, if a single server is offline, the servers will not begin the commit step, and clients will not observe the effects of the prepared transactions for an indefinite period of time. By ensuring availability of writes (i.e., clients return early), we have sacrificed visibility in the form of ensuring that writes are accessible to readers. Thus, clients will not enjoy session guarantees [216] such as Read Your Writes. Given the importance of these session guarantees for many of the industrial users we have encountered (e.g., see Facebook's TAO geo-replication [65]), we currently do not favor this approach.

Sticky HA RAMP. The second strategy is to ensure a degree of stickiness, or affinity, between clients and servers within a datacenter [32]. Each client is assigned its own datacenter. Instead of having a client issue its writes to the entire database replica set, the client can instead issue its prepare and commit operations to its assigned datacenter (or local replica group) and subsequently forward the writes to be prepared and committed autonomously in separate clusters (Figure 5.7b). That is, once a writer has performed the appropriate

RAMP protocol in its assigned datacenter, it can return. In an N-datacenter deployment, each full write protocol is performed N separate times—once per datacenter. If the same timestamp is assigned to each write, the end state of each datacenter will be equivalent. As long as clients remain connected to the *same* datacenter (i.e., is "sticky" with respect to its database connections), it will read its writes.

The total number of prepare and commit operations are the same as in the first strategy, but the commit point is staggered—each cluster reaches a commit point independently, at different times. Moreover, clusters operate independently, so throughput is not improved—only latency—because each cluster must replay every other cluster's writes [35]. This is the basic strategy espoused by traditional log shipping approaches [151] as well as more recent proposals such as the COPS [167] and Eiger [168] systems.

However, this stickiness has an often-neglected penalty: a client can no longer connect to arbitrary servers and expect to read its own writes. If a server is down in a client's local datacenter, the client must—in the worst case—locate an entire other replica set to which the client can connect. This negatively affects availability: the *Prepare-F* strategy can utilize all servers at once, but the sticky strategy requires clients to maintain affinity for availability. In cases when this "sticky availability" [32] is acceptable (e.g., each datacenter contains a set of application servers that issue the RAMP protocols against another datacenter-local set of storage servers), this may be a reasonable compromise.

5.6.2 Quorum-Replicated RAMP Operation

While RAMP *Prepare-F* and *Sticky HA* are best suited for multi-datacenter deployments, in quorum-replicated systems such as Dynamo and Cassandra [95, 153], there are several optimizations that can be used to further improve availability, even within a single datacenter.

Our key observation here is that, to guarantee maximum of two-round trips for reads, only PREPARE and second-round GET requests need to intersect on a given set of replicas. Recall that second-round GET requests are issued in order to "repair" any fractured reads from the first round of read results. In the event of these fractured reads, a reader *must* have access to versions corresponding to fractured reads that have been prepared but were not committed at the time of the first-round read. However, assembling the first round of committed versions can run under partial (i.e., non-intersecting) quorum operation [172] with respect to commit messages.

This means that COMMIT and first-round GET operations can proceed on effectively any server in a set of replicas, enabling two key optimizations. In these optimizations, we assume that readers issue second-round read requests and writers issue PREPARE operations

using a quorum system [180] of replicas (e.g., majority quorums).

First, first-round read requests can be served from any replica of a given item. Then, if a client detects a race (in RAMP-F or RAMP-H), it can issue the optional second round of requests to a quorum of servers. RAMP-S will always issue the second round of requests. This optimization improves the latency of the first round of reads and also enables speculative retry [94]. It also decreases the load on the servers and increases availability for first-round read operations.

Second, commit operations can be performed on any replica of a given item. Similar to the optimization proposed in *Prepare-F* RAMP, servers can propagate commit messages between themselves asynchronously, possibly piggybacking on anti-entropy messages as in systems like Cassandra and Dynamo. This optimization improves the latency of commit. However, because all servers must commit the transaction eventually, it does not necessarily decrease the load on servers.

To quantify the potential latency improvements achievable using these optimizations, we draw on latency distributions from a recent study of Dynamo-style operation [44]. According to latency data from a Dynamo-style quorum-replicated database running on spinning disks at LinkedIn, moving from waiting for two replicas of three to respond to waiting for one replica of three to respond to a write request decreased latency from 21.0ms to 11.0ms at the 99.9th percentile; 1.63ms to 0.66ms for reads. For a similar database at Yammer, the gains for writes are 427ms to 10.8ms and the gains for reads are 32.6ms to 5.6ms—an even more impressive gain. Over a wide-area network with latency of 75ms, the gains are as much as 148ms. Thus, in practice, these simple optimizations may prove worthwhile.

5.6.3 RAMP, Transitive Dependencies, and Causal Consistency

In Section 5.3.3, we discussed how RA isolation does not enforce transitive read-write dependencies across transactions. For example, if T_a read-depends on T_b (i.e., T_a reads a version that T_b created), another transaction T_c might read-depend on T_a (i.e., T_c reads a version that T_a created) but anti-depend on T_b (i.e., T_b overwrites a version that T_a read). In this section, we discuss why we made this design decision as well as alternatives for enforcing dependencies and their costs.

The primary challenges in enforcing transitive dependencies come in limiting metadata while preserving availability and partition independence. In the extreme, if we limited ourselves to serial access to database state, we could easily preserve information about dependencies using a single scalar: any transactions would observe versions with lower scalar values, similar to classic serializable multi-version concurrency control. However, if we wish to preserve available and coordination-free operation (and therefore concurrent

creation of versions), then we must admit a partial ordering of versions. To avoid fractured reads as in RA isolation while preserving dependency information, we either need to find a way to capture this partial order or otherwise limit the degree of availability in the system.

Full causality tracking. The former approach—tracking "cuts" in a system with partially ordered events—is well-studied. As a first approximation, we can consider the problem of capturing RA with dependency tracking as an instance of capturing causality in a distributed system, with each event corresponding to a transaction commit and dependencies due to reads (i.e., a causal memory with atomically visible, multi-register reads). In line with this approximation, we could replace each timestamp in the RAMP algorithms with a suitable mechanism for tracking causality; for example, instead of storing a scalar timestamp, we could store a vector clock, with one entry per client in the system. Subsequently, clients could maintain a vector clock containing the highest-committed writes they had seen, and, upon reading from servers, ensure that the server commits any writes that happen-before the client's current vector. Thus, we can use vector clocks to track dependencies across transactions.

The problem with the above approach is in the size of the metadata required. Primarily, with N concurrent clients, each vector will require $O(N)$ space, which is potentially prohibitive in practice. Moreover, the distributed systems literature strongly suggests that, with N concurrent clients, $O(N)$ space is *required* to capture full causal lineage as above [75]. Thus, while using vector clocks to enforce transitive dependencies is a correct approach, it incurs considerable overheads that we do not wish to pay and have yet to be proven viable at scale in practical settings [35].[2]

The latter approach—limiting availability—is also viable, at the cost of undercutting our scalability goals from Section 5.3.5.

Bounding writer concurrency. One simple approach—as we hinted above—is to limit the concurrency of writing clients: we can bound the overhead of vector clocks to an arbitrarily small amount by limiting the amount of concurrency in the system. For example, if we allow five clients to perform writes at a given time, we only need a vector of size five. This requires coordination between writers (but not readers). As Section 5.5.2 demonstrated, RAMP transaction performance degrades gracefully under write contention; under the decreased concurrency strategy, performance would effectively hit a cliff. Latency would increase due to queuing delays and write contention, and, for a workload like YCSB with a fixed proportion of read to write operations, throughput would be limited. Specifically, for a workload with p writers ($p = 0.05$ in our default configuration), if W writers were permit-

[2]Another alternative that uses additional metadata is the strawman from Section 5.3.5, in which clients send all of the writes in their transaction to all of the partitions responsible for at least one write in the transaction. This uses even more metadata than the vector-based approach.

ted at a given time, the effective number of active YCSB clients in the system would become $\frac{W}{p}$. Despite these limits, this is perhaps the most viable solution we have encountered and, moreover, does not affect read performance under read-heavy workloads. However, this solution has considerable coordination overheads, and managing which servers are able to perform writes (e.g., using distributed leases) requires potentially heavyweight synchronization protocols.

Sacrificing partition independence. Another approach to improving availability is to sacrifice partition independence. As we discuss and evaluate in Sections 5.5.1 and 5.5.2, it is possible to preserve transaction dependencies by electing special coordinator servers as points of rendezvous for concurrently executing transactions. If extended to a non-partition-independent context, the RAMP protocols begin to more closely resemble traditional multi-version concurrency control solutions, in particular, Chan and Gray's read-only transactions [73]. More recently, the 2PC-PCI mechanism [168] we evaluated is an elegant means of achieving this behavior if partition independence is unimportant. Nevertheless, as our experimental evaluation shows, sacrificing this partition independence can be costly under some workloads.

Sacrificing causality. A final approach to limiting the overhead of dependency tracking is to limit the number of dependencies to track. Several prior systems have used limited forms of causality, for example, application-supplied dependency information [39,151], as a basis for dependency tracking. In this strategy, applications inform the system about what versions should precede a given write; in [35] (see also Section ??), we show that, for many modern web applications, these histories can be rather small (often one item, with a power-law distribution over sizes). In this case, we could encode the causal history in its entirety along with each write or exploit otherwise latent information within the data such as comment `reply-to` fields to mine this data automatically. This strategy breaks the current RAMP API. However, it is the only known strategy for circumventing the O(N) upper-bound on dependency tracking in causally consistent storage systems ensuring availability of both readers and writers.

Experiences with system operators. While causal consistency provides a number of useful guarantees, in practice, we perceive a lack of interest in maintaining full causal consistency; database operators and users are anecdotally often unwilling to pay the metadata and implementation costs of full causality tracking. As we have seen in Section 6.4.1, many of these real-world operators exhibit an aversion to synchronization at scale, so maintaining availability is paramount to either their software offerings or business operation. In fact, we have anecdotally found coordination-free execution and partition independence to be valuable selling points for the RAMP algorithms presented in this work. Instead, we have found

many users instead favor guarantees such as Read Your Writes (provided by the RAMP algorithms) rather than full dependency tracking, opting for variants of explicit causality (e.g., via foreign key constraints or explicit dependencies) or restricted, per-item causality tracking (e.g., version vectors [95]). Despite this mild pessimism, we view further reduction of causality overhead to be an interesting area for future work—including a more conclusive answer to the availability-metadata trade-off surfaced by [75].

5.7 RSIW Proof

To begin, we first show that there exists a well-defined total ordering of write-only transactions in a history that is valid under RA isolation. This will be useful in ordering write transactions in our one-copy equivalent execution.

Lemma 2 (Well-defined Total Order on Writes). *Given a history* H *containing read-only and write-only transactions that is valid under RA isolation,* $DSG(H)$ *does not contain any directed cycles consisting entirely of write-dependency edges.*

Proof. H is valid under RA isolation and therefore does not exhibit phenomenon *G1c*. Thus, H does not contain any directed cycles consisting entirely of dependency edges. Therefore, H does not contain any directed cycles consisting entirely of write-dependency edges. □

We will also need to place read-only transactions in our history. To do so, we show that, under RA isolation and the RSIW property (i.e., the preconditions of Theorem 2), each read-only transaction will only read from one write-only transaction.

Lemma 3 (Single Read Dependency). *Given a history* H *containing read-only and write-only transactions that obeys the RSIW property and is valid under RA isolation, each node in* $DSG(H)$ *contains at most one direct read-dependency edge.*

Proof. Consider a history H containing read-only and write-only transactions that has RSIW and is valid under RA isolation. Write-only transactions have no reads, so they have no read-dependency edges in $DSG(H)$. However, suppose $DSG(H)$ contained a node corresponding to a read-only transaction containing more than one direct read-dependency edge; call this read-only transaction T_r. For two read-dependency edges to exist, T_r must have read versions produced by at least two different write-only transactions; pick any two and call them T_i and T_j, corresponding to read versions x_a and y_d.

If x and y are the same item, then $a < d$ or $d < a$. In either case, T_r exhibits the fractured reads phenomenon and H is not valid under RA isolation, a contradiction.

Therefore, x and y must be distinct items. Because H obeys the RSIW property, T_r must also obey the RSIW property. By the definition of the RSIW property, T_r must have only read items written to by T_i and items also written to by T_j; this implies that T_i and T_j each wrote to items x and y. We can label T_i's write to y as y_b and T_j's write to x as x_c. Per Lemma 2, T_i's writes to x and y must either both come before or both follow T_j's corresponding writes to x and y in the version order for each of x and y; that is, either both $a < b$ and $c < d$ or both $b < a$ and $d < c$.

If $a < b$ and $c < d$, then T_r exhibits the fractured reads phenomenon: T_r read x_a and y_d but T_j, which wrote y_d also wrote x_b, and $a < b$. If $b < a$ and $d < c$, then T_r again exhibits the fractured reads phenomenon: T_r read x_a and y_d but T_i, which wrote x_a, also wrote y_c, and $d < c$. In either case, H is not valid under RA isolation, a contradiction.

Therefore, each node in $DSG(H)$ must not contain more than one read-dependency edge. □

We now use this ordering on reads and writes to construct a total ordering on transactions in a history:

Procedure 1 (Transform). *Given a history H containing read-only and write-only transactions that has RSIW and is valid under RA isolation, construct a total ordering O of the transactions in H as follows:*

1. *Perform a topological sorting in O of committed write-only transactions in H ordered by the write-dependency edges in $DSG(H)$. That is, for each pair of write-only transactions (T_1, T_2) in H that performed at least one write to the same item, place the transaction that wrote the higher-versioned item later in O. Lemma 2 ensures such a total ordering exists.*

2. *For each committed read-only transaction T_r in H, place T_r in O after the write-only transaction T_w whose writes T_r read (i.e., after the write-only transaction that T_r directly read-depends on) but before the next write-only transaction $T_{w'}$ in O (or, if no such transaction exists, at the end of O). By Lemma 3, each committed read-only transaction read-depends on only one write-only transaction, so this ordering is similarly well defined.*

Return O.

As an example, consider the following history:

$$\begin{array}{ll} T_1 & w(x_1); w(y_1) \\ T_2 & w(x_2); w(y_2) \\ T_3 & r(x_1); r(y_1) \\ T_4 & r(y_2); \end{array} \tag{5.9}$$

History 5.9 obeys the RSIW property and is also valid under RA isolation. Applying procedure TRANSFORM to History 5.9, in the first step, we first order our write-only transactions T_1 and T_2. Both T_1 and T_2 write to x and y, but T_2's writes have a later version number than T_1's, so, according to Step 1 of TRANSFORM, we have $O = T_1; T_2$. Now, in Step 2 of TRANSFORM, we consider the read-only transactions T_3 and T_4. We place T_3 after the transaction that it read from (T_1) and before the next write transaction in the sequence (T_2), yielding $O = T_1; T_3; T_2$. We place T_4 after the transaction that it read from (T_2) and, because there is no later write-only transaction following T_2 in O, place T_4 at the end of O, yielding $O = T_1; T_3; T_2; T_4$. In this case, we observe that, as Theorem 2 suggests, it is possible to TRANSFORM an RSIW and RA history into a one-copy serial equivalent and that O is in fact a one-copy serializable execution.

Now we can prove Theorem 2. We demonstrate that executing the transactions of H in the order resulting from applying TRANSFORM to H on a single-copy database yields an equivalent history (i.e., read values and final database state) as H. Because O is a total order, H must be one-copy serializable.

Theorem 2. Consider history H containing read-only and write-only transactions that has RSIW and is valid under RA isolation. We begin by applying TRANSFORM to H to produce an ordering O.

We create a new history H_o by considering an empty one-copy database and examining each transaction T_h in O in serial order as follows: if T_h is a write-only transaction, execute a new transaction T_{ow} that writes each version produced by T_h to the one-copy database and commits. If T_h is a read-only transaction, execute a new transaction T_{or} that reads from the sam items as T_h and commits. H_o is the result of a serial execution of transactions over a single logical copy of the database, H_o is one-copy serializable.

We now show that H and H_o are equivalent. First, all committed transactions and operations in H also appear in H_o because TRANSFORM operates on all transactions in H and all transactions and their operations (with single-copy operations substituted for multi-version operations) appear in the total order O used to produce H_o. Second, $DSG(H)$ and $DSG(H_o)$ have the same direct read dependencies. This is a straightforward consequence of

Step 2 of TRANSFORM: in O, each read-only transaction T_r appears immediately following the write-only transaction T_w upon which T_r read-depends. When the corresponding read transaction is executed against the single-copy database in H_o, the serially preceding write-only transaction will produce the same values that the read transaction read in H. Therefore, H and H_o are equivalent.

Because H_o is one-copy serializable and H_o is equivalent to H, H must also be one-copy serializable. □

We have opted for the above proof technique because we believe the TRANSFORM procedure provides clarity into *how* executions satisfying both RA isolation and the RSIW property relate to their serializable counterparts. An alternative and elegant proof approach leverages the work on multi-version serializability theory [53], which we sketch here. Given a history H that exhibits RA isolation and has RSIW, we show that MVSG(H) is acyclic. By an argument resembling Lemma 3, the in-degree for read-only transactions in SG(H) (i.e., Adya's direct read dependencies) is one. By an argument resembling Lemma 2, the edges between write-only transactions in MVSG(H) produced by the first condition of the MVSG construction ($x_i \ll x_j$ in the definition of the MVSG [53, p. 152]; i.e., Adya's write dependencies) are acyclic. Therefore, any cycles in the MVSG must include at least one of the second kind of edges in the MVSG(H) ($x_j \ll x_i$; i.e., Adya's direct anti-dependencies). But, for such a cycle to exist, a read-only transaction T_r must anti-depend on a write-only transaction T_{wi} that in turn write-depends on another write-only transaction T_{wj} upon which T_r read-depends. Under the RSIW property, T_{wi} and T_{wj} will have written to at least one of the same items, and the presence of a write-dependency cycle will indicate a fractured reads anomaly in T_r.

5.8 RAMP Correctness and Independence

RAMP-F Correctness. To prove RAMP-F provides RA isolation, we show that the two-round read protocol returns a transactionally atomic set of versions. To do so, we formalize criteria for atomic (read) sets of versions in the form of *companion sets*. We will call the set of versions produced by a transaction *sibling versions* and say that x is a sibling item to a write y_j if there exists a version x_k that was written in the same transaction as y_j.

Given two versions x_i and y_j, we say that x_i is a *companion version* to y_j if x_i is a transactional sibling of y_j or x is a sibling item of y_j and $i > j$. We say that a set of versions V is a *companion set* if, for every pair (x_i, y_j) of versions in V where x is a sibling item of y_j, x_i is a companion to y_j. In Figure 5.2, the versions returned by T_2's first round of reads ($\{x_1, y_\perp\}$) do not comprise a companion set because $y_\perp$ has a lower timestamp than

x_1's sibling version of y (that is, x_1 has sibling version y_1 and but $\perp < 1$ so $y_\perp$ has too low of a timestamp). Subsets of companion sets are also companion sets and companion sets also have a useful property for RA isolation:

Claim 1 (Companion sets are atomic)**.** *In the absence of G1c phenomena, companion sets do not contain fractured reads.*

Claim 1 follows from the definitions of companion sets and fractured reads.

Proof. If V is a companion set, then every version $x_i \in V$ is a companion to every other version $y_j \in V$ where v_j contains x in its sibling items. If V contained fractured reads, V would contain two versions x_i, y_j such that the transaction that wrote y_j also wrote a version x_k, $i < k$. However, in this case, x_i would not be a companion to y_j, a contradiction. Therefore, V cannot contain fractured reads. □

To provide RA, RAMP-F clients assemble a companion set for the requested items (in v_{latest}), which we prove below:

Claim 2. RAMP-F *provides Read Atomic isolation.*

Proof. Each write in RAMP-F contains information regarding its siblings, which can be identified by item and timestamp. Given a set of RAMP-F versions, recording the highest timestamped version of each item (as recorded either in the version itself or via sibling metadata) yields a companion set of item-timestamp pairs: if a client reads two versions x_i and y_j such that x is in y_j's sibling items but $i < j$, then $v_{latest}[x]$ will contain j and not i. Accordingly, given the versions returned by the first round of RAMP-F reads, clients calculate a companion set containing versions of the requested items. Given this companion set, clients check the first-round versions against this set by timestamp and issue a second round of reads to fetch any companions that were not returned in the first round. The resulting set of versions will be a subset of the computed companion set and will therefore also be a companion set. This ensures that the returned results do not contain fractured reads. RAMP-F first-round reads access lastCommit, so each transaction corresponding to a first-round version is committed, and, therefore, any siblings requested in the (optional) second round of reads are also committed. Accordingly, RAMP-F never reads aborted or non-final (intermediate) writes. Moreover, RAMP-F timestamps are assigned on a per-transaction basis, preventing write-dependency cycles and therefore *G1c*. This establishes that RAMP-F provides RA. □

RAMP-F Scalability and Independence. RAMP-F also provides the independence guarantees from Section 5.3.5. The following invariant over lastCommit is core to RAMP-F GET request completion:

Invariant 1 (Companions present). *If a version* x_i *is referenced by* lastCommit *(that is,* lastCommit[x] $= i$*), then each of* x_i*'s sibling versions are present in* versions *on their respective partitions.*

Invariant 1 is maintained by RAMP-F's two-phase write protocol. lastCommit is only updated once a transaction's writes have been placed into versions by a first round of PREPARE messages. Siblings will be present in versions (but not necessarily lastCommit).

Claim 3. RAMP-F *is coordination-free.*

Recall from Section 5.3.5 that coordination-free execution ensures that one client's transactions cannot cause another client's to block and that, if a client can contact the partition responsible for each item in its transaction, the transaction will eventually commit (or abort of its own volition).

Proof. Clients in RAMP-F do not communicate or coordinate with one another and only contact servers. Accordingly, to show that RAMP-F provides coordination-free execution, it suffices to show that server-side operations always terminate. PREPARE and COMMIT methods only access data stored on the local partition and do not block due to external coordination or other method invocations; therefore, they complete. GET requests issued in the first round of reads have $ts_{req} = \bot$ and therefore will return the version corresponding to lastCommit[k], which was placed into versions in a previously completed PREPARE round. GET requests issued in the second round of client reads have ts_{req} set to the client's calculated v_{latest}[k]. v_{latest}[k] is a sibling of a version returned from lastCommit in the first round, so, due to Invariant 1, the requested version will be present in versions. Therefore, GET invocations are guaranteed access to their requested version and can return without waiting. The success of RAMP-F operations does not depend on the success or failure of other clients' RAMP-F operations. □

Claim 4. RAMP-F *provides partition independence.*

Proof. RAMP-F transactions do not access partitions that are unrelated to each transaction's specified data items and servers do not contact other servers in order to provide a safe response for operations. □

RAMP-S Correctness. RAMP-S writes and first-round reads proceed identically to RAMP-F writes, but the metadata written and returned is different. Therefore, the proof is similar to RAMP-F, with a slight modification for the second round of reads.

Claim 5. RAMP-S *provides Read Atomic isolation.*

Proof. To show that RAMP-S provides RA, it suffices to show that RAMP-S second-round reads (resp) are a companion set. Given two versions $x_i, y_j \in resp$ such that $x \neq y$, if x is a sibling item of y_j, then x_i must be a companion to y_j. If x_i were not a companion to y_j, then it would imply that x is not a sibling item of y_j (so we are done) or that $j > i$. If $j > i$, then, due to Invariant 1 (which also holds for RAMP-S writes due to identical write protocols), y_j's sibling is present in *versions* on the partition for x and would have been returned by the server (line 6), a contradiction. Each second-round GET request returns only one version, so the final set of reads does not exhibit fractured reads. □

RAMP-S Scalability and Independence. RAMP-S ensures coordination-free execution and partition independence. The proofs closely resemble those of RAMP-F: Invariant 1 ensures that incomplete commits do not stall readers, and all server-side operations are guaranteed to complete.

RAMP-H Correctness. The probabilistic behavior of the RAMP-H Bloom filter admits false positives. However, given unique transaction timestamps (Section 5.4.4), requesting false siblings by timestamp and item does not affect correctness:

Claim 6. RAMP-H *provides Read Atomic isolation.*

Proof. To show that RAMP-H provides Read Atomic isolation, it suffices to show that any versions requested by RAMP-H second-round reads that would not have been requested by RAMP-F second-round reads (call this set v_{false}) do not compromise the validity of RAMP-H's returned companion set. Any versions in v_{false} do not exist: timestamps are unique, so, for each version x_i, there are no versions x_j of non-sibling items with the same timestamp as x_i (i.e., where $i = j$). Therefore, requesting versions in v_{false} do not change the set of results collected in the second round. □

RAMP-H Scalability and Independence. RAMP-H provides coordination-free execution and partition independence. We omit full proofs, which closely resemble those of RAMP-F. The only significant difference from RAMP-F is that second-round GET requests may return $\perp$, but, as we showed above, these empty responses correspond to false positives in the Bloom filter and therefore do not affect correctness.

5.9 Discussion

Given our experiences designing and evaluating the RAMP transaction protocols, we believe there are a number of interesting extensions that merit further examination.

First, RAMP metadata effectively encodes a limited form of transaction *intent*: readers and servers are only able to repair fractured reads because the metadata encodes the remainder of the work required to complete the transaction. We believe it would be interesting to generalize this concept to arbitrary program logic: for example, in a model such as lazy transactions [106] or eventual serializability [109], with transactions expressed as stored procedures, multiple, otherwise conflicting/coordinating clients could instead cooperate in order to complete one anothers' transactions in the event of a failure—without resorting to the use of a centralized master (e.g., for pre-scheduling or validating transaction execution). This programming model is largely incompatible with traditional interactive transaction execution but is nevertheless exciting to consider as an extension of these protocols.

Second, and more concretely, we see several opportunities to extend RAMP to more specialized use cases. The RAMP protocol family is currently not well-suited to large scans and, as we have discussed, does not enforce transitive dependencies across transactions. We view restricting the concurrency of writers (but not readers) to be a useful step forward in this area, with predictable impact on writer performance. This strikes a middle ground between traditional MVCC and the current RAMP protocols.

Finally, as we noted in Section 5.3.4, efficient transaction processing often focuses on weakening semantics (e.g., weak isolation) or changing the programming model (e.g., stored procedures as above). As our investigation of the RSW property demonstrates, there may exist compelling combinations of the two that yield more intuitive, high-performance, or scalable results than examining semantics or programming models in isolation. Addressing this question is especially salient for the many users of weak isolation models in practice today [32], as it can help understand when applications require stronger semantics and when, in fact, weak isolation is not simply fast but is also "correct."

5.10 Summary

This chapter described how to achieve atomically visible multi-partition transactions without incurring the performance and availability penalties of traditional algorithms. We first identified a new isolation level—Read Atomic isolation—that provides atomic visibility and matches the requirements of a large class of real-world applications. We subsequently achieved RA isolation via scalable, contention-agnostic RAMP transactions. In contrast with techniques that use inconsistent but fast updates, RAMP transactions provide correct semantics for applications requiring secondary indexing, foreign key constraints, and materialized view maintenance while maintaining scalability and performance. By leveraging multi-versioning with a variable but small (and, in two of three algorithms, constant) amount of metadata per write, RAMP transactions allow clients to detect and assemble

atomic sets of versions in one to two rounds of communication with servers (depending on the RAMP implementation). The choice of coordination-free and partition independent algorithms allowed us to achieve near-baseline performance across a variety of workload configurations and scale linearly to 100 servers. While RAMP transactions are not appropriate for all applications, the many applications for which they are appropriate will benefit significantly.

Chapter 6

Coordination Avoidance for Database Constraints

Thus far, we have considered semantics expressed in terms of isolation models and application programming patterns such as indexing. Both of these specifications were relatively low-level. The first relied on admissible read-write interleavings of transactions. The latter relied on a bespoke isolation level that we introduced to exactly capture the semantics of several existing scenarios that arise in databases and applications. In this chapter, we raise the level of abstraction further and consider the use of *constraints*, or arbitrary user-defined predicates over database state. We focus on constraints found in contemporary SQL and Ruby on Rails applications.

As we discuss, these constraints are found in many database-backed applications today. The constraints are not necessarily a full specification of program correctness yet are frequently found in modern applications (and are typically enforced by coordination). As candidates for study, we draw upon popular constraints found in languages like SQL as well as a range of constraints found in open source applications, which we subsequently analyze for invariant confluence. As before, we find that many are invariant confluent and provide coordination-avoiding implementations of each.

6.1 Invariant Confluence of SQL Constraints

We begin our study of practical constraints by considering several features of SQL, ending with abstract data types. We will apply these results to full applications in Section 6.3.

In this section, we focus on providing intuition and informal explanations of our invariant confluence analysis. Interested readers can find a more formal analysis in Section 6.2.

Invariant	Operation	invariant confluent?	Proof #
Attribute Equality	Any	Yes	1
Attribute Inequality	Any	Yes	2
Uniqueness	Choose specific value	No	3
Uniqueness	Choose some value	Yes	4
`AUTO_INCREMENT`	Insert	No	5
Foreign Key	Insert	Yes	6
Foreign Key	Delete	No	7
Foreign Key	Cascading Delete	Yes	8
Secondary Indexing	Update	Yes	9
Materialized Views	Update	Yes	10
`>`	Increment [Counter]	Yes	11
`<`	Increment [Counter]	No	12
`>`	Decrement [Counter]	No	13
`<`	Decrement [Counter]	Yes	14
`[NOT] CONTAINS`	Any [Set, List, Map]	Yes	15, 16
`SIZE=`	Mutation [Set, List, Map]	No	17

Table 6.1: Example SQL (top) and ADT invariant confluence along with references to formal proofs in Section 6.2.

including discussion of invariants not presented here. For convenience, we reference specific proofs from Section 6.2 inline.

6.1.1 Invariant Confluence for SQL Relations

We begin by considering several constraints found in SQL.

Equality. As a warm-up, what if an application wants to prevent a particular value from appearing in a database? For example, our payroll application from Section 6.4.1 might require that every user have a last name, marking the `LNAME` column with a `NOT NULL` constraint. While not particularly exciting, we can apply invariant confluence analysis to insertions and updates of databases with (in-)equality constraints (Claims 1, 2 in Section 6.2). Per-record inequality invariants are invariant confluent, which we can show by contradiction: assume two database states S_1 and S_2 are each I-T-reachable under per-record inequality invariant I_e but that $I_e(S_1 \sqcup S_2)$ is false. Then there must be a $r \in S_1 \sqcup S_2$ that violates I_e (i.e., r has the forbidden value). r must appear in S_1, S_2, or both. But, that would imply that one of S_1 or S_2 is not I-valid under I_e, a contradiction.

Uniqueness. We can also consider common uniqueness invariants (e.g., `PRIMARY KEY` and `UNIQUE` constraints). For example, in our payroll example, we wanted user IDs to be

unique. In fact, our earlier discussion in Section 6.4.1 already provided a counterexample showing that arbitrary insertion of users is not invariant confluent under these invariants: {Stan:5} and {Mary:5} are both I-T-reachable states (Section 3.1) that can be created by a sequence of insertions (starting at $S_0 = \{\}$), but their merge—{Stan:5, Mary:5}—is not I-valid. Therefore, uniqueness is not invariant confluent for inserts of unique values (Claim 3). However, reads and deletions are both invariant confluent under uniqueness invariants: reading and removing items cannot introduce duplicates.

Can the database safely *choose* unique values on behalf of users (e.g., assign a new user an ID)? In this case, we can achieve uniqueness without coordination—as long as we have a notion of replica membership (e.g., server or replica IDs). The difference is subtle ("grant this record this specific, unique ID" versus "grant this record some unique ID"), but, in a system model with membership (as is practical in many contexts), is powerful. If replicas assign unique IDs within their respective portion of the ID namespace, then merging locally valid states will also be globally valid (Claim 4).

Foreign Keys. We can consider more complex invariants, such as foreign key constraints. In our payroll example, each employee belongs to a department, so the application could specify a constraint via a schema declaration to capture this relationship (e.g., `EMP.D_ID FOREIGN KEY REFERENCES DEPT.ID`).

Are foreign key constraints maintainable without coordination? Again, the answer depends on the actions of transactions modifying the data governed by the invariant. Insertions under foreign key constraints *are* invariant confluent (Claim 6). To show this, we again attempt to find two I-T-reachable states that, when merged, result in invalid state. Under foreign key constraints, an invalid state will contain a record with a "dangling pointer"—a record missing a corresponding record on the opposite side of the association. If we assume there exists some invalid state $S_1 \sqcup S_2$ containing a record r with an invalid foreign key to record f, but S_1 and S_2 are both valid, then r must appear in S_1, S_2, or both. But, since S_1 and S_2 are both valid, r must have a corresponding foreign key record (f) that "disappeared" during merge. Merge (in the current model) does not remove versions, so this is impossible.

From the perspective of invariant confluence analysis, foreign key constraints concern the *visibility* of related updates: if individual database states maintain referential integrity, a non-destructive merge function such as set union cannot cause tuples to "disappear" and compromise the constraint. This also explains why models such as read committed [9] and read atomic [9] isolation as well as causal consistency [32] are also achievable without coordination: simply restricting the visibility of updates in a given transaction's read set does not require coordination between concurrent operations.

Deletions and modifications under foreign key constraints are more challenging. Arbitrary deletion of records is unsafe: a user might be added to a department that was concurrently deleted (Claim 7). However, performing cascading deletions (e.g., SQL `DELETE CASCADE`), where the deletion of a record also deletes *all* matching records on the opposite end of the association, is invariant confluent under foreign key constraints (Claim 8). We can generalize this discussion to updates (and cascading updates).

Materialized Views. Applications often pre-compute results to speed query performance via a materialized view [215] (e.g., `UNREAD_CNT` as `SELECT COUNT(*) FROM emails WHERE read_date = NULL`). We can consider a class of invariants that specify that materialized views reflect primary data; when a transaction (or merge invocation) modifies data, any relevant materialized views should be updated as well. This requires installing updates at the same time as the changes to the primary data are installed (a problem related to maintaining foreign key constraints). However, given that a view only reflects primary data, there are no "conflicts." Thus, materialized view maintenance updates are invariant confluent (Claim 10).

6.1.2 Invariant Confluence for SQL Data Types

So far, we have considered databases that store growing sets of immutable versions. We have used this model to analyze several useful constraints, but, in practice, databases do not (often) provide these semantics, leading to a variety of interesting anomalies. For example, if we implement a user's account balance using a "last writer wins" merge policy [205], then performing two concurrent withdrawal transactions might result in a database state reflecting only one transaction (a classic example of the Lost Update anomaly) [9, 32]. To avoid variants of these anomalies, many optimistic, coordination-free database designs have proposed the use of *abstract data types* (ADTs), providing merge functions for a variety of uses such as counters, sets, and maps [81, 170, 205, 226] that ensure that all updates are reflected in final database state. For example, a database can represent a simple counter ADT by recording the number of times each transaction performs an `increment` operation on the counter [205].

Invariant confluence analysis is also applicable to these ADTs and their associated invariants. For example, a row-level "greater-than" (>) threshold invariant is invariant confluent for counter `increment` and `assign` (←) but not `decrement` (Claims 11, 13), while a row-level "less-than" (<) threshold invariant is invariant confluent for counter `decrement` and `assign` but not `increment` (Claims 12, 14). This means that, in our payroll example, we can provide coordination-free support for concurrent salary increments but not concurrent salary decrements. ADTs (including lists, sets, and maps) can be combined with standard re-

lational constraints like materialized view maintenance (e.g., the "total salary" row should contain the sum of employee salaries in the `employee` table). This analysis presumes user program explicitly use ADTs, and, as with our generic set-union merge, invariant confluence ADT analysis requires a specification of the ADT merge behavior (Section 6.2 provides several examples).

6.1.3 SQL Discussion and Limitations

We have analyzed a number of combinations of invariants and operations (shown in Table 6.1). These results are by no means comprehensive, but they are expressive for many applications (Section 6.3). In this section, we discuss lessons from this classification process.

Analysis mechanisms. We have manually analyzed particular invariant and constraint combinations, demonstrating each to be invariant confluent or not. To study actual SQL-based applications, we can apply these labels via simple static analysis. Specifically, given invariants (e.g., captured via SQL DDL) and transactions (e.g., expressed as stored procedures), we can examine each invariant and each operation within each transaction and identify pairs that we have labeled as invariant confluent or non-invariant confluent. Any pairs labeled as invariant confluent can be marked as safe, while, for soundness (but not completeness), any unrecognized operations or invariants can be flagged as potentially non-invariant confluent. Despite its simplicity (both conceptually and in terms of implementation), this technique—coupled with the results of Table 6.1—is sufficiently powerful to automatically characterize the I-confluence of the applications we consider in Section 6.3 when expressed in SQL (with support for multi-row aggregates like Invariant 8 in Table 6.2).

By growing our recognized list of invariant confluent pairs on an as-needed basis (via manual analysis of the pair), the above technique has proven useful—due in large part to the common re-use of invariants like foreign key constraints. However, one could use more complex forms of program analysis. For example, one might analyze the invariant confluence of *arbitrary* invariants, leaving the task of proving or disproving invariant confluence to an automated model checker or SMT solver. While invariant confluence—like monotonicity and commutativity (Chapter 7)—is undecidable for arbitrary programs, others have recently shown this alternative approach (e.g., in commutativity analysis [80, 160] and in invariant generation for view serializable transactions [198]) to be fruitful for restricted languages. We view language design and more automated analysis as an interesting area for future work (Section 8.3).

Recency and session support. Our proposed invariants are declarative, but a class of useful semantics—recency, or real-time guarantees on reads and writes—describe properties of op-

erations (i.e., they pertain to transaction execution rather than the state(s) of the database). For example, users often wish to read data that is up-to-date as of a given point in time (e.g., "read latest" [83] or linearizable [118] semantics). While traditional isolation models do not directly address these recency guarantees [9], they are often important to programmers. Are these models invariant confluent? We can attempt to simulate recency guarantees in invariant confluence analysis by logging the result of all reads and any writes with a timestamp and requiring that all logged timestamps respect their recency guarantees (thus treating recency guarantees as invariants over recorded read/write execution traces). However, this is a somewhat pointless exercise: it is well known (and we have already discussed) that recency guarantees are unachievable with transactional availability [32,92,118]. Thus, if application reads face these requirements, coordination is required. Indeed, when application "consistency" means "recency," systems cannot circumvent speed-of-light delays.

If users wish to "read their writes" or desire stronger "session" guarantees [189] (e.g., maintaining recency on a per-user or per-session basis), they must maintain affinity or "stickiness" [32] with a given (set of) replicas. These guarantees are also expressible in the invariant confluence model and do not require coordination between different users' or sessions' transactions.

6.2 More Formal Invariant Confluence Analysis of SQL Constraints

In this section, we more formally demonstrate the invariant confluence of invariants and operations discussed in Section 6.1. Our goals in this section are two-fold. First, we have found the experience of formally proving invariant confluence to be instructive in understanding these combinations (beyond less formal arguments made in the body text for brevity and intuition). Second, we have found invariant confluence proofs to take on two general structure typically take one of two forms:

- To show a set of transactions are not invariant confluent with respect to an invariant I, we use proof by counterexample: we present two I-T-reachable states with a common ancestor that, when merged, are not I-valid.
- To show a set of transactions are invariant confluent with respect to an invariant I, we use proof by contradiction: we show that, if a state S is not I-valid, merging two I-T-reachable states S_1 and S_2 with a common ancestor state to produce S implies either one or both of S_1 or S_2 must not be I-valid.

These results are not exhaustive, and there are literally infinite combinations of invariants and operations to consider. Rather, the seventeen examples below serve as a demonstration of what can be accomplished via invariant confluence analysis.

Notably, the negative results below use fairly simple histories consisting of a single transaction divergence. Nevertheless, we decided to preserve the more general formulation of invariant confluence (accounting for arbitrary I-T-reachable states) to account for more pathological (perhaps less realistic, or, if these results are any indication, less commonly encountered) behaviors that only arise during more complex divergence patterns.

We introduce additional formalism as necessary. To start, unless otherwise specified, we use the set union merge operator. We denote version i of item x as x_i and a write of version x_i with value v as $w(x_i = v)$. For now, we assume each version has an integer value.

Claim 7 (Writes are invariant confluent with respect to per-item equality constraints)**.** *Assume writes are not invariant confluent with respect to some per-item equality constraint* $i = c$*, where* i *is an item and* c *is a constant. By definition, there must exist two* I-T*-reachable states* S_1 *and* S_2 *with common ancestor state such that* $I(S_1) \rightarrow \texttt{true}$ *and* $I(S_2) \rightarrow \texttt{true}$ *but* $I(S_1 \sqcup S_2) \rightarrow \texttt{false}$*; therefore, there exists a version* i_n *in* $S_1 \sqcup S_2$ *such that* $i_n \neq c$*, and, under set union,* $i_n \in S_1$*,* $i_n \in S_2$*, or both. However, this would imply that* $I(S_1) \rightarrow \texttt{false}$ *or* $I(S_2) \rightarrow \texttt{false}$ *(or both), a contradiction.*

Claim 8 (Writes are invariant confluent with respect to per-item inequality constraints)**.** *The proof follows almost identically to the proof of Claim 7, but for an invariant of the form* $i \neq c$*.*

Claim 9 (Writing arbitrary values is not invariant confluent with respect to multi-item uniqueness constraints)**.** *Consider the following transactions:*

$$T_{1u} \coloneqq w(x_a = v);\ \texttt{commit}$$
$$T_{2u} \coloneqq w(x_b = v);\ \texttt{commit}$$

and uniqueness constraint on records:

$$I_u(D) = \{\textit{values of versions in } D \textit{ are unique}\}$$

Now, an empty database trivially does not violate uniqueness constraints ($I_u(D_s = \{\}) \rightarrow \texttt{true}$*), and adding individual versions to the separate empty databases is also valid:*

$$T_{1u}(\{\}) = \{x_a = v\},\ I_u(\{x_a = v\}) \rightarrow \texttt{true}$$
$$T_{2u}(\{\}) = \{x_b = v\},\ I_u(\{x_b = v\}) \rightarrow \texttt{true}$$

However, merging these states results in invalid state:

$$I_u(\{x_a = v\} \sqcup \{x_b = v\} = \{x_a = v, x_b = v\}) \rightarrow \texttt{false}$$

Therefore, $\{T_{1u}, T_{2u}\}$ *is not invariant confluent under* I_s.

For the next proof, we consider a model as suggested in Section 6.1 where replicas are able to generate unique (but not arbitrary (!)) IDs (in the main text, we suggested the use of a replica ID and sequence number). In the following proof, to account for this non-deterministic choice of unique ID, we introduce a special $\texttt{nonce}()$ function and require that, $\texttt{nonce}()$ return unique values for each replica; this can be achieved by assigning each replica a unique ID and implementing nonce by returning the ID along with a local sequence number. That is, $\sqcup$ is not defined for replicas on which independent invocations of $\texttt{nonce}()$ return the same value.

Claim 10 (Assigning values by $\texttt{nonce}()$ is invariant confluent with respect to multi-item uniqueness constraints)**.** *Assume that assigning values by* $\texttt{nonce}()$ *is not invariant confluent with respect to some multi-item uniqueness invariant:*

$$I(D) = \forall c \in \texttt{dom}(D), \{|\{x \in D \mid x = c\}| \leqslant 1\}$$

By definition, there must exist two I-T*-reachable states with a common ancestor reached by executing nonce-generating transactions (of the form* $T_i = [w(x_i = \texttt{nonce}())]$*),* S_1 *and* S_2 *such that* $I(S_1) \rightarrow \texttt{true}$ *and* $I(S_2) \rightarrow \texttt{true}$ *but* $I(S_1 \sqcup S_2) \rightarrow \texttt{false}$.

Therefore, there exist two versions i_a, i_b *in* $S_1 \sqcup S_2$ *such that* i_a *and* i_b *(both generated by* $\texttt{nonce}()$*) are equal in value. Under set union, this means* $i_a \in S_1$ *and* $i_b \in S_2$ *(*i_a *and* i_b *both cannot appear in* S_1 *or* S_2 *since it would violate those states'* I*-validity). Because replica states grow monotonically under set merge and* S_1 *and* S_2 *differ, they must be different replicas. But* $\texttt{nonce}()$ *cannot generate the same values on different replicas, a contradiction.*

Claim 11 (Writing arbitrary values are not invariant confluent with respect to sequentiality constraints)**.** *Consider the following transactions:*

$$T_{1s} \coloneqq w(x_a = 1);\ \texttt{commit}$$
$$T_{2s} \coloneqq w(x_b = 3);\ \texttt{commit}$$

and the sequentiality constraint on records:

$$I_s(D) = \{\texttt{max}(r \in D) - \texttt{min}(r \in D) = |D| + 1\} \vee \{|D| = 0\}$$

Now, I_s *holds over the empty database (*$I_s(\{\}) \rightarrow \texttt{true}$*), while inserting sequential new*

records into independent, empty replicas is also valid:

$$T_{1s}(\{\}) = \{x_a = 1\},\ I_u(\{x_a = 1\}) \to \texttt{true}$$
$$T_{2s}(\{\}) = \{x_b = 3\},\ I_u(\{x_b = 3\}) \to \texttt{true}$$

However, merging these states results in invalid state:

$$I_s(\{x_a = 1\} \sqcup \{x_b = 3\} = \{x_a = 1, x_b = 3\}) \to \texttt{false}$$

Therefore, $\{T_{1s}, T_{2s}\}$ *is not invariant confluent under* I_s.

To discuss foreign key constraints, we need some way to *refer* to other records within the database. There are a number of ways of formalizing this. Here, refer to a field f within a given version x_i as $x_i.f$. In the following discussion, recall that invariant confluence analysis is performed on fully replicated sets of data. While there is considerable *implementation* complexity involved in actually performing multi-partition foreign key constraint maintenance (and indexing; as in RAMP) this is not germane to our model. As a simple strawman solution, we can defer all calculation of tombstoned records until writes have quiesced, guaranteeing convergence. As a slightly more advanced strawman, we can calculate tombstoned records according to a global watermark of writes that is advanced via background consensus.

Claim 12 (Insertion is invariant confluent with respect to foreign key constraints). *Assume that inserting new records is not invariant confluent with respect to some foreign key constraint* $I(D) = \{\forall r_f \in D$ *such that* $r_f.g \neq \texttt{null}, \exists r_t \in D$ *such that* $r_f.g = r_t.h\}$ *(there exists a foreign key reference between fields* g *and* h*). By definition, there must exist two* I-T*-reachable states* S_1 *and* S_2 *with a common ancestor reachable by executing transactions performing insertions such that* $I(S_1) \to \texttt{true}$ *and* $I(S_2) \to \texttt{true}$ *but* $I(S_1 \sqcup S_2) \to \texttt{false}$*; therefore, there exists some version* $r_1 \in S_1 \sqcup S_2$ *such that* $r_1.g \neq \texttt{null}$ *but* $\nexists r_2 \in S_1 \sqcup S_2$ *such that* $r_1.g = r_2.h$*. Under set union,* r_1 *must appear in either* S_1 *or* S_2 *(or both), and, for each set of versions in which it appears, because* S_1 *and* S_2 *are both* I*-valid, they must contain an* r_3 *such that* $r_1.f = r_3.h$*. But, under set union,* $r_3.h$ *should also appear in* $S_1 \sqcup S_2$*, a contradiction.*

For simplicity, in the following proof, we assume that deleted elements remain deleted under merge. In practice, this can be accomplished by tombstoning records and, if required, using counters to record the number of deletions and additions [205]. We represent a deleted version x_d by $\neg x_b$.

Claim 13 (Concurrent deletion and insertion is not invariant confluent with respect to foreign key constraints)**.** *Consider the following transactions:*

$$T_{1f} \coloneqq w(x_a.g = 1);\ \texttt{commit}$$
$$T_{2f} \coloneqq \texttt{delete}(x_b);\ \texttt{commit}$$

and the foreign key constraint:

$$I_f(D) = \{\forall r_f \in D, r_f.g \neq \texttt{null},\ \exists r_t \in D \textit{ s.t. } \neg r_t \notin D \textit{ and } r_f.g = r_t.h\}$$

Foreign key constraints hold over the initial database $S_i = \{x_b.h = 1\}$ *(*$I_u(S_i) \rightarrow \texttt{true}$*) and on independent execution of* T_a *and* T_b*:*

$$T_{1f}(\{x_b.h = 1\}) = \{x_a.g = 1, x_b.h = 1\},\ I_f(\{x_a = 1\}) \rightarrow \texttt{true}$$
$$T_{2f}(\{x_b.h = 1\}) = \{x_b.h = 1, \neg x_b\}\ I_f(\{x_b.h = 1, \neg x_b\}) \rightarrow \texttt{true}$$

However, merging these states results in invalid state:

$$I_f(\{x_a.g = 1\} \sqcup \{x_b.h = 1, \neg x_b\}) \rightarrow \texttt{false}$$

Therefore, $\{T_{1f}, T_{2f}\}$ *is not invariant confluent under* I_f*.*

We denote a cascading delete of all records that reference field f with value v (v a constant) as $\texttt{cascade}(f = v)$.

Claim 14 (Cascading deletion and insertion are invariant confluent with respect to foreign key constraints)**.** *Assume that cascading deletion and insertion of new records are not invariant confluent with respect to some foreign key constraint* $I(D) = \{\forall r_f \in D$ *such that* $r_f.g \neq \texttt{null}$*,* $\exists r_t \in D$ *such that* $r_f.g = r_t.h$ *if* $\texttt{cascade}(h = r_f.g \neq v)\}$ *(there exists a foreign key reference between fields* g *and* h *and the corresponding value for field* h *has not been deleted-by-cascade). By definition, there must exist two* I-T*-reachable states* S_1 *and* S_2 *with common ancestor reachable by performing insertions such that* $I(S_1) \rightarrow \texttt{true}$ *and* $I(S_2) \rightarrow \texttt{true}$ *but* $I(S_1 \sqcup S_2) \rightarrow \texttt{false}$*; therefore, there exists some version* $r_1 \in S_1 \sqcup S_2$ *such that* $r_1.f \neq \texttt{null}$ *but* $\nexists r_2 \in S_1 \sqcup S_2$ *such that* $r_1.g = r_2.h$*. From the proof of Claim 12, we know that insertion is invariant confluent, so the absence of* r_2 *must be due to some cascading deletion. Under set union,* r_1 *must appear in exactly one of* S_1 *or* S_2 *(if* r_1 *appeared in both, there would be no deletion, a contradiction since we know insertion is invariant confluent). For the state* S_j *in which* r_1 *does not appear (either* S_1 *or* S_2*),* S_j *must include* $\texttt{cascade}(h = r_1.g)$*. But, if* $\texttt{cascade}(h = r_1.g) \in S_j$*,* $\texttt{cascade}(h = r_1.g)$ *must also be in* $S_i \sqcup S_j$*, a contradiction and so* $S_i \sqcup S_j \rightarrow \texttt{true}$*, a contradiction.*

We define a "consistent" secondary index invariant as requiring that, when a record is visible, its secondary index entry should also be visible. This is similar to the guarantees provided by Read Atomic isolation [37]. For simplicity, in the following proof, we only consider updates to a single indexed attribute $\mathtt{attr}$, but the proof is easily generalizable to multiple index entries, insertions, and deletion via tombstones. We use last-writer wins for index entries.

Claim 15 (Updates are invariant confluent with respect to consistent secondary indexing). *Assume that updates to records are not invariant confluent with respect a secondary index constraint on attribute* $\mathtt{attr}$*:*

$I(D) = \{\forall r_f \in D$ *such that* $r_f.\mathtt{attr} \neq \mathtt{null}$ *and* f *is the highest version of* $r \in D$*,* $\exists r_{idx} \in D$ *such that* $r_f \in r_{idx}.\mathtt{entries}$ *(all entries with non-null* $\mathtt{attr}$ *are reflected in the secondary index entry for* $\mathtt{attr}$*).*

Represent an update to record r_x *as* $\{w(r_x)$ *and, if* $r_x.\mathtt{attr} \neq \mathtt{null}$*, also* $r_{idx}.\mathtt{entries.add}(r_x)$*, else* $r_{idx}.\mathtt{entries.delete}(r_x)\}$*.*

By definition, there must exist two I-T*-reachable states* S_1 *and* S_2 *with common ancestors reachable by performing insertions* S_1 *and* S_2 *such that* $I(S_1) \to \mathtt{true}$ *and* $I(S_2) \to \mathtt{true}$ *but* $I(S_1 \sqcup S_2) \to \mathtt{false}$*; therefore, there exists some version* $r_1 \in S_1 \sqcup S_2$ *such that* $r_1.\mathtt{attr} \neq \mathtt{null}$ *but* $\nexists r_{idx} \in S_1 \sqcup S_2$ *or* $\exists r_{idx} \in S_1 \sqcup S_2$ *but* $r_1 \notin r_{idx}.\mathtt{entries}$*. In the former case,* $r_{idx} \notin S_1$ *or* S_2*, but* $r_1 \in S_1$ *or* $r_1 \in S_2$*, a contradiction. The latter case also produces a contradiction: if* $r_1 \in S_1$ *or* $r_1 \in S_2$*, it must appear in* r_{idx}*, a contradiction.*

In our formalism, we can treat materialized views as functions over database state $f(D) \to c$.

Claim 16 (Updates are invariant confluent with respect to materialized view maintenance). *The proof is relatively straightforward if we treat the materialized view record(s)* r *as having a foreign key relationship to any records in the domain of the function (as in the proof of Claim 14 and recompute the function on update, cascading delete, and* $\sqcup$*.*

For our proofs over counter ADTs, we represent increments of a counter c by $\mathtt{inc}_i(c)$, where i is a distinct invocation, decrements of c by $\mathtt{dec}_i(c)$, and the value of c in database D as $\mathtt{val}(c, D) = |\{j \mid \mathtt{inc}_j(c) \in D\}| - |\{k \mid \mathtt{dec}_k(c) \in D\}|$.

Claim 17 (Counter ADT increments are invariant confluent with respect to greater-than constraints). *Assume increments are not invariant confluent with respect to some per-counter greater-than constraint* $I(D) = \mathtt{val}(c, D) > k$*, where* k *is a constant. By definition, there must exist two* I-T*-reachable states* S_1 *and* S_2 *with common ancestor reachable by executing write transactions such that* $I(S_1) \to \mathtt{true}$ *and* $I(S_2) \to \mathtt{true}$ *but* $I(S_1 \sqcup S_2) \to \mathtt{false}$*;*

therefore, $\mathtt{val}(c, S_1 \sqcup S_2) \leqslant k$. *However, this implies that* $\mathtt{val}(c, S_1) \leqslant k$), $\mathtt{val}(c, S_2$, *or both, a contradiction.*

Claim 18 (Counter ADT increments are not invariant confluent with respect to less-than constraints)**.** *Consider the following transactions:*

$$T_{1i} \coloneqq \mathtt{inc}_1(c);\ \mathtt{commit}$$
$$T_{2i} \coloneqq \mathtt{inc}_2(c);\ \mathtt{commit}$$

and the less-than inequality constraint:

$$I_i(D) = \{\mathtt{val}(c, D) < 2\}$$

I_i *holds over the empty database state (*$I_i(\{\}) \rightarrow \mathtt{true}$*) and when* T_a *and* T_b *are independently executed:*

$$T_{1i}(\{\}) = \{\mathtt{inc}_1(c) = 1\},\ I_i(\{\mathtt{inc}_1(c) = 1\}) \rightarrow \mathtt{true}$$
$$T_{2i}(\{\}) = \{\mathtt{inc}_2(c)\},\ I_i(\{\mathtt{inc}_2(c)\}) \rightarrow \mathtt{true}$$

However, merging these states results in invalid state:

$$I_u(\{\mathtt{inc}_1(c)\} \sqcup \{\mathtt{inc}_2(c)\}) \rightarrow \mathtt{false}$$

Therefore, $\{T_{1i}, T_{2i}\}$ *is not invariant confluent under* I_u.

Claim 19 (Counter ADT decrements are not invariant confluent with respect to greater-than constraints)**.** *The proof is similar to the proof of Claim 20; substitute* dec *for* inc *and choose* $I_i(D) = \{\mathtt{val}(c, D) > -2\}$.

Claim 20 (Counter ADT decrements are invariant confluent with respect to less-than constraints)**.** *Unsurprisingly, the proof is almost identical to the proof of Claim 17, but with* $<$ *instead of* $>$ *and* dec *instead of* inc.

We provide proofs for ADT lists; the remainder are remarkably similar. Our implementation of ADT lists in these proofs uses a lexicographic sorting of values to determine list order. Transactions add a version v to list l via $\mathtt{add}(v, l)$ and remove it via $\mathtt{del}(v, l)$ (where an item is considered contained in the list if it has been added more times than it has been deleted) and access the length of l in database D via $\mathtt{size}(l) = |\{k \mid \mathtt{add}(k, l) \in D\}| - |\{m \mid \mathtt{del}(m, l) \in D\}|$ (note that size of a non-existent list is zero).

Claim 21 (Modifying a list ADT is invariant confluent with respect to containment constraints)**.** *Assume ADT list modifications are not invariant confluent with respect to some*

equality constraint $I(D) = \{\mathtt{add}(k, l) \in D \wedge \mathtt{del}(k, l) \notin D\}$ *for some constant* k*. By definition, there must exist two* I-T*-reachable states* S_1 *and* S_2 *with common ancestor reachable by list modifications such that* $I(S_1) \to \mathtt{true}$ *and* $I(S_2) \to \mathtt{true}$ *but* $I(S_1 \sqcup S_2) \to \mathtt{false}$*; therefore,* $\mathtt{add}(k, l) \notin \{S_1 \sqcup S_2\}$ *or* $\mathtt{del}(k, l) \in \{S_1 \sqcup S_2\}$*. In the former case, neither* S_1 *nor* S_2 *contain* $\mathtt{add}(k, l)$ *a contradiction. In the latter case, if either of* S_1 *or* S_2 *contains* $\mathtt{del}(k, l)$*, it will be invalid, a contradiction.*

Claim 22 (Modifying a list ADT is invariant confluent with respect to non-containment constraints)**.** *Assume ADT list modifications are not invariant confluent with respect to some non-containment constraint* $I(D) = \{\mathtt{add}(k, l) \notin D \wedge \mathtt{del}(k, l) \in D\}$ *for some constant* k*. By definition, there must exist two* I-T*-reachable states* S_1 *and* S_2 *with common ancestor reachable via list modifications such that* $I(S_1) \to \mathtt{true}$ *and* $I(S_2) \to \mathtt{true}$ *but* $I(S_1 \sqcup S_2) \to \mathtt{false}$*; therefore,* $\mathtt{add}(k, l) \in \{S_1 \sqcup S_2\}$ *and* $\mathtt{del}(k, l) \notin \{S_1 \sqcup S_2\}$*. But this would imply that* $\mathtt{add}(k, l) \in S_1$*,* $\mathtt{add}(k, l) \in S_2$*, or both (while* $\mathtt{del}(k, l)$ *is in neither), a contradiction.*

Claim 23 (Arbitrary modifications to a list ADT are not invariant confluent with respect to equality constraints on the size of the list)**.** *Consider the following transactions:*

$$T_{1l} \coloneqq \mathtt{del}(x_i, l);\ \mathtt{add}(x_a, l);\ \mathtt{commit}$$
$$T_{2l} \coloneqq \mathtt{del}(x_i, l);\ \mathtt{add}(x_b, l);\ \mathtt{commit}$$

and the list size invariant:

$$I_l(D) = \{\mathtt{size}(l) = 1\}$$

Now, the size invariant holds on a list of size one ($I_u(\{\mathtt{add}(x_i, l)\}) \to \mathtt{true}$*) and on independent state modifications:*

$$T_{1l}(\{\mathtt{add}(x_i, l)\}) = \{\mathtt{add}(x_i, l),\ \mathtt{del}(x_i, l),\ \mathtt{add}(x_a, l)\}$$
$$T_{2l}(\{\mathtt{add}(x_i, l)\}) = \{\mathtt{add}(x_i, l),\ \mathtt{del}(x_i, l),\ \mathtt{add}(x_b, 1)\}$$

However, merging these states result in an invalid state:

$$I_l(\{\mathtt{add}(x_i, l),\ \mathtt{del}(x_i, l),\ \mathtt{add}(x_a, l)\}$$
$$\sqcup \{\mathtt{add}(x_i, l),\ \mathtt{del}(x_i, l),\ \mathtt{add}(x_b, l)\}) \to \mathtt{false}$$

Therefore, $\{T_{1l}, T_{2l}\}$ *is not invariant confluent under* I_u*.*

Note that, in our above list ADT, modifying the list is invariant confluent with respect to constraints on the head and tail of the list but not intermediate elements of the list! That is, the head (resp. tail) of the merged list will be the head (tail) of one of the un-merged lists. However, the second element may come from either of the two lists.

6.3 Empirical Impact: SQL-Based Constraints

When achievable, coordination-free execution enables scalability limited to that of available hardware. This is powerful: a invariant confluent application can scale out without sacrificing correctness, latency, or availability. In Section 6.1, we saw combinations of invariants and transactions that were invariant confluent and others that were not. In this section, we apply these combinations to the workloads of the OLTP-Bench suite [98], with a focus on the TPC-C benchmark. Our focus is on the coordination required in order to correctly execute each and the resulting, coordination-related performance costs.

6.3.1 TPC-C Invariants and Execution

The TPC-C benchmark is the gold standard for database concurrency control [98] both in research and in industry [219], and in recent years has been used as a yardstick for distributed database concurrency control performance [142, 217, 221]. How much coordination does TPC-C actually require a compliant execution?

The TPC-C workload is designed to be representative of a wholesale supplier's transaction processing requirements. The workload has a number of application-level correctness criteria that represent basic business needs (e.g., order IDs must be unique) as formulated by the TPC-C Council and which must be maintained in a compliant run. We can interpret these well-defined "consistency criteria" as invariants and subsequently use invariant confluence analysis to determine which transactions require coordination and which do not.

Table 6.2 summarizes the twelve invariants found in TPC-C as well as their invariant confluence analysis results as determined by Table 6.1. We classify the invariants into three broad categories: materialized view maintenance, foreign key constraint maintenance, and unique ID assignment. As we discussed in Section 6.1, the first two categories are invariant confluent (and therefore maintainable without coordination) because they only regulate the *visibility* of updates to multiple records. Because these (10 of 12) invariants are invariant confluent under the workload transactions, there exists some execution strategy that does not use coordination. However, simply because these invariants are invariant confluent does not mean that *all* execution strategies will scale well: for example, using locking would *not* be coordination-free.

As one coordination-free execution strategy (which we implement in Section 6.3.2) that respects the foreign key and materialized view invariants, we can use RAMP transactions from Chapter 5, which provide atomically visible transactional updates across servers without relying on coordination for correctness. As a reminder, RAMP transactions employ limited multi-versioning and metadata to ensure that readers and writers can always pro-

#	Informal Invariant Description	Type	Txns	$\mathcal{I}$-C
1	YTD wh sales = sum(YTD district sales)	MV	P	Yes
2	Per-district order IDs are sequential	S_{ID}+FK	N, D	No
3	New order IDs are sequentially assigned	S_{ID}	N, D	No
4	Per-district, item order count = roll-up	MV	N	Yes
5	Order carrier is set iff order is pending	FK	N, D	Yes
6	Per-order item count = line item roll-up	MV	N	Yes
7	Delivery date set iff carrier ID set	FK	D	Yes
8	YTD wh = sum(historical wh)	MV	D	Yes
9	YTD district = sum(historical district)	MV	P	Yes
10	Customer balance matches expenditures	MV	P, D	Yes
11	Orders reference New-Orders table	FK	N	Yes
12	Per-customer balance = cust. expenditures	MV	P, D	Yes

Table 6.2: TPC-C Declared "Consistency Conditions" (3.3.2.x) and invariant confluence analysis results with respect to the workload transactions (Invariant type: MV: materialized view, S_{ID}: sequential ID assignment, FK: foreign key; Transactions: N: New-Order, P: Payment, D: Delivery).

ceed concurrently: any client whose reads overlap with another client's writes to the same item(s) can use metadata stored in the items to fetch any "missing" writes from the respective servers. A standard RAMP transaction over data items suffices to enforce foreign key constraints, while a RAMP transaction over commutative counters as described in [37] is sufficient to enforce the TPC-C materialized view constraints.

Two of TPC-C's invariants are not invariant confluent with respect to the workload transactions and therefore *do* require coordination. On a per-district basis, order IDs should be assigned sequentially (both uniquely and sequentially, in the New-Order transaction) and orders should be processed sequentially (in the Delivery transaction). If the database is partitioned by warehouse (as is standard [142, 217, 221]), the former is a distributed transaction (by default, 10% of New-Order transactions span multiple warehouses). The benchmark specification allows the latter to be run asynchronously and in batch mode on a per-warehouse (non-distributed) basis, so we, like others [217, 221], focus on New-Order. Including additional transactions like the read-only Order-Status in the workload mix would increase performance due to the transactions' lack of distributed coordination and (often considerably) smaller read/write footprints.

Avoiding New-Order Coordination. New-Order is not invariant confluent with respect to the TPC-C invariants, so we can always fall back to using serializable isolation. However, the per-district ID assignment records (10 per warehouse) would become a point of contention, limiting our throughput to effectively $\frac{100W}{\text{RTT}}$ for a W-warehouse TPC-C bench-

mark with the expected 10% distributed transactions. Others [221] (including us, in prior work [32]) have suggested disregarding consistency criteria 3.3.2.3 and 3.3.2.4, instead opting for unique but non-sequential ID assignment: this allows inconsistency and violates the benchmark compliance criteria.

During a compliant run, New-Order transactions must coordinate. However, as discussed above, only the ID assignment operation is non-invariant confluent; the remainder of the operations in the transaction can execute coordination-free. With some effort, we can avoid *distributed* coordination. A naïve implementation might grab a lock on the appropriate district's "next ID" record, perform (possibly remote) remaining reads and writes, then release the lock at commit time. Instead, as a more efficient solution, New-Order can defer ID assignment until commit time by introducing a layer of indirection. New-Order transactions can generate a temporary, unique, but non-sequential ID (`tmpID`) and perform updates using this ID using a RAMP transaction (which, in turn, handles the foreign key constraints) [37]. Immediately prior to transaction commit, the New-Order transaction can assign a "real" ID by atomically incrementing the current district's local"next ID" record (yielding `realID`) and recording the `[tmpID, realID]` mapping in a special ID lookup table. Any read requests for the `ID` column of the Order, New-Order, or Order-Line tables can be safely satisfied (transparently to the end user) by joining with the ID lookup table on `tmpID`. In effect, the New-Order ID assignment can use a nested atomic transaction [170] upon commit, and all coordination between any two transactions is confined to a single server.

6.3.2 Evaluating TPC-C New-Order

We subsequently implemented the above execution strategy in a distributed database prototype to quantify the overheads associated with coordination in TPC-C New-Order. Most notably, the coordination-avoiding query plan scales linearly to over 12.7M transactions per second on 200 servers while substantially outperforming distributed two-phase locking. Our goal here is to demonstrate—beyond the microbenchmarks of Section 6.4.1—that safe but judicious use of coordination can have meaningful positive effect on performance.

Implementation and Deployment. We employ a multi-versioned storage manager, with RAMP-Fast transactions for snapshot reads and atomically visible writes/"merge" (providing a variant of regular register semantics, with writes visible to later transactions after commit) [37] and implement the nested atomic transaction for ID assignment as a subprocedure inside RAMP-Fast's server-side commit procedure (using spinlocks). We implement transactions as stored procedures and fulfill the TPC-C "Isolation Requirements" by using read and write buffering as proposed in [32]. As is common [141, 142, 195, 217],

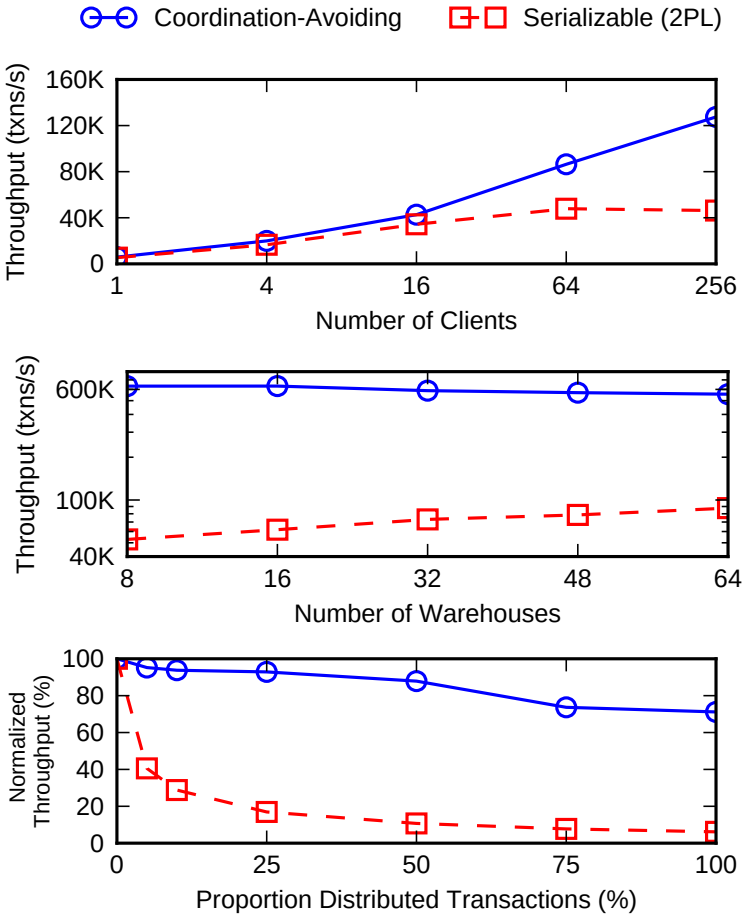


Figure 6.1: TPC-C New-Order throughput across eight servers.

we disregard per-warehouse client limits and "think time" to increase load per warehouse. In all, our base prototype architecture is similar to that of [37]: a JVM-based partitioned, main-memory, mastered database.

For an apples-to-apples comparison with a coordination-intensive technique within the same system, we also implemented textbook two-phase locking (2PL) [53], which provides serializability but also requires distributed coordination. We totally order lock requests across servers to avoid deadlock, batching lock requests to each server and piggybacking read and write requests on lock request RPC. As a validation of our implementation, our 2PL prototype achieves per-warehouse (and sometimes aggregate) throughput similar to (and often in excess of) several recent serializable database implementations (of both 2PL and other approaches) [141, 142, 195, 217].

By default, we deploy our prototype on eight EC2 `cr1.8xlarge` instances (32 cores comprising 88 Amazon Elastic Compute units, each with 244GB RAM) in the Amazon EC2 `us-west-2` region with non-co-located clients and one warehouse per server (recall there are 10 "hot" district ID records per warehouse) and report the average of three 120 second runs.

Basic behavior. Figure 6.1 shows performance across a variety of configurations, which we detail below. Overall, the coordination-avoiding query plan far outperforms the serializable execution. The coordination-avoiding query plan performs some coordination, but, because coordination points are not distributed (unlike 2PL), physical resources (and not coordination) are the bottleneck.

Varying load. As we increase the number of clients, the coordination-avoiding query plan throughput increases linearly, while 2PL throughput increases to 40K transactions per second, then levels off. As in our microbenchmarks in Section 2.2.2, the former utilizes available hardware resources (bottlenecking on CPU cycles at 640K transactions per second), while the latter bottlenecks on logical contention.

Physical resource consumption. To understand the overheads of each component in the coordination-avoiding query plan, we used JVM profiling tools to sample thread execution while running at peak throughput, attributing time spent in functions to relevant modules within the database implementation (where possible):

Code Path	Cycles
Storage Manager (Insert, Update, Read)	45.3%
Stored Procedure Execution	14.4%
RPC and Networking	13.2%
Serialization	12.6%
ID Assignment Synchronization (spinlock contention)	0.19%
Other	14.3%

The coordination-avoiding prototype spends a large portion of execution in the storage manager, performing B-tree modifications and lookups and result set creation, and in RPC/serialization. In contrast to 2PL, the prototype spends less than 0.2% of time coordinating, in the form of waiting for locks in the New-Order ID assignment; the (single-site) assignment is fast (a linearizable integer increment and store, followed by a write and fence instruction on the spinlock), so this should not be surprising. We observed large throughput penalties due to garbage collection (GC) overheads (up to 40%)—an unfortunate cost of our highly compact (several thousand lines of Scala), JVM-based implementation. However, even in this current prototype, physical resources are the bottleneck—not coordination.

Varying contention. We subsequently varied the number of "hot," or contended items by increasing the number of warehouses on each server. Unsurprisingly, 2PL benefits from a decreased contention, rising to over 87K transactions per second with 64 warehouses. In contrast, our coordination-avoiding implementation is largely unaffected (and, at 64 warehouses, is even negatively impacted by increased GC pressure). The coordination-avoiding query plan is effectively agnostic to read/write contention.

Varying distribution. We also varied the percentage of distributed transactions. The coordination-avoiding query plan incurred a 29% overhead moving from no distributed transactions to all distributed transactions due to increased serialization overheads and less efficient batching of RPCs. However, the 2PL implementation decreased in throughput by over 90% (in line with prior results [195, 217], albeit exaggerated here due to higher contention) as more requests stalled due to coordination with remote servers.

Scaling out. Finally, we examined our prototype's scalability, again deploying one warehouse per server. As Figure 6.2 demonstrates, our prototype scales linearly, to over 12.74 million transactions per second on 200 servers (in light of our earlier results, and, for economic reasons, we do not run 2PL at this scale). Per-server throughput is largely constant after 100 servers, at which point our deployment spanned all three `us-west-2` datacenters and experienced slightly degraded per-server performance. While we make use of application semantics, we are unaware of any other compliant multi-server TPC-C implementation that has achieved greater than 500K New-Order transactions per second [141, 142, 195, 217].

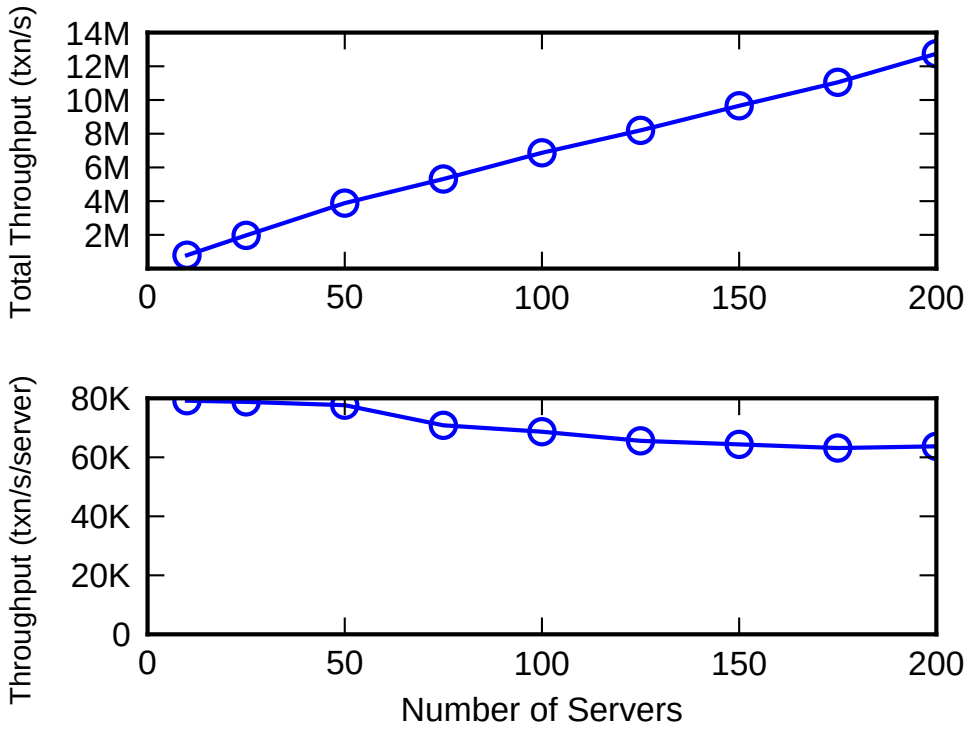


Figure 6.2: Coordination-avoiding New-Order scalability.

Summary. We present these quantitative results as a proof of concept that executing even challenging workloads like TPC-C that contain complex integrity constraints are not necessarily at odds with scalability if implemented in a coordination-avoiding manner. Distributed coordination need not be a bottleneck for all applications, even if conflict serializable execution indicates otherwise. Coordination avoidance ensures that physical resources—and not logical contention—are the system bottleneck whenever possible.

6.3.3 Analyzing Additional Applications

These results begin to quantify the effects of coordination-avoiding concurrency control. If considering *application-level* invariants, databases only have to pay the price of coordination when necessary. We were surprised that the "current industry standard for evaluating the performance of OLTP systems" [98] was so amenable to coordination-avoiding execution—at least for compliant execution as defined by the official TPC-C specification.

For greater variety, we also studied the workloads of the recently assembled OLTP-Bench suite [98], performing a similar analysis to that of Section 6.3.1. We found (and confirmed with an author of [98]) that for nine of fourteen remaining (non-TPC-C) OLTP-Bench applications, the workload transactions did not involve integrity constraints (e.g., did not

modify primary key columns), one (`CH-benCHmark`) matched TPC-C, and two specifications implied (but did not explicitly state) a requirement for unique ID assignment (`AuctionMark`'s `new-purchase` order completion, `SEATS`'s `NewReservation` seat booking; achievable like TPC-C order IDs). The remaining two benchmarks, `sibench` and `smallbank` were specifically designed as research benchmarks for serializable isolation. Finally, the three "consistency conditions" required by the newer TPC-E benchmark are a proper subset of the twelve conditions from TPC-C considered here (and are all materialized counters). It is possible (even likely) that these benchmarks are underspecified, but according to official specifications, TPC-C contains the most coordination-intensive invariants among all but two of the OLTP-Bench workloads.

Anecdotally, our conversations and experiences with real-world application programmers and database developers have not identified invariants that are radically different than those we have studied here. A simple thought experiment identifying the invariants required for a social networking site yields a number of invariants but none that are particularly exotic (e.g., username uniqueness, foreign key constraints between updates, privacy settings [37, 83]). Nonetheless, we examine additional invariants from real-world applications in the remainder of this chapter. The results presented in this section hint at what is possible with coordination-avoidance as well as the costs of coordination if applications are not invariant confluent.

6.4 Constraints from Open Source Applications

In the remainder of this chapter, we examine constraints as found in modern, open source web applications. The rise of "Web 2.0" Internet applications delivering dynamic, highly interactive user experiences has been accompanied by a new generation of programming frameworks [218]. These frameworks simplify common tasks such as content templating and presentation, request handling, and, notably, data storage, allowing developers to focus on "agile" development of their applications. This trend embodies the most recent realization of the larger vision of object-relational mapping (ORM) systems [70], albeit at a unprecedented scale of deployment and programmer adoption.

As a lens for understanding issues of data integrity in modern ORM systems, we study Ruby on Rails (or, simply, "Rails") [117, 199], a central player among modern frameworks powering sites including (at one point) Twitter [82], Airbnb [15], GitHub [191], Hulu [72], Shopify [101], Groupon [177], SoundCloud [69], Twitch [196], Goodreads [1], and Zendesk [229]. From the perspective of database systems research, Rails is interesting for at least two reasons. First, it continues to be a popular means of developing responsive web application front-end and business logic, with an active open source community and user

base. Rails recently celebrated its tenth anniversary and enjoys considerable commercial interest, both in terms of deployment and the availability of hosted "cloud" environments such as Heroku. Thus, Rails programmers represent a large class of consumers of database technology. Second, and perhaps more importantly, Rails is "opinionated software" [102]. That is, Rails embodies the strong personal convictions of its developer community, and, in particular, David Heinemeier Hansson (known as DHH), its creator. Rails is particularly opinionated towards the database systems that it tasks with data storage. To quote DHH:

> "I don't *want* my database to be clever! ...I consider stored procedures and constraints vile and reckless destroyers of coherence. No, Mr. Database, you can not have my business logic. Your procedural ambitions will bear no fruit and you'll have to pry that logic from my dead, cold object-oriented hands ...I want a single layer of cleverness: My domain model." [130]

Thus, in several regards, this wildly successful software framework bears an actively antagonistic relationship to database management systems, echoing a familiar refrain of the "NoSQL" movement: get the database out of the way and let the application do the work.

In this paper, we examine the implications of this impedance mismatch between databases and modern ORM frameworks in the context of application integrity. Rails largely ignores decades of work on native database concurrency control solutions by providing a set of primitives for handling application integrity that are enforced at the application tier—building, from the underlying database system's perspective, a *feral* concurrency control system. Core to feral concurrency control mechanisms is the use of data invariants, as we have studied in this chapter in the context of SQL. We examine the design and use of these feral mechanisms and evaluate their effectiveness in practice by analyzing them and experimentally quantifying data integrity violations in practice. Our goal is to understand how this growing class of applications currently interacts with database systems and how we, as a database systems community, can positively engage with these criticisms to better serve the needs of these developers.

We begin by surveying the state of Rails' application-tier concurrency control primitives and examining their use in 67 open source applications representing a variety of use cases from e-Commerce to Customer Relationship Management and social networking. We find that, these applications overwhelmingly use Rails' built-in support for declarative invariants—*validations* and *associations*—to protect data integrity—instead of application-defined transactions, which are used more than 37 times less frequently. Across the survey, we find over 9950 uses of application-level validations designed to ensure correctness criteria including referential integrity, uniqueness, and adherence to common data formats.

Given this corpus, we subsequently ask: are these feral invariants correctly enforced? Do they work in practice? Rails may execute validation checks in parallel, so we study the

potential for data corruption due to races if validation and update activity does not run within a serializable transaction in the database. This is a real concern, as many DBMS platforms use non-serializable isolation by default and in many cases (despite labeling otherwise) do not provide serializable isolation as an option at all. Accordingly, we apply invariant confluence analysis and show that, in fact, up to 86.9% of Rails validation usage by volume is actually invariant confluent and therefore safe under concurrent execution. However, the remainder—which include uniqueness violations under insertion and foreign key constraint violations under deletion—are not. Therefore, we quantify the impact of concurrency on data corruption for Rails uniqueness and foreign key constraints under both worst-case analysis and via actual Rails deployment. We demonstrate that, for pathological workloads, validations reduce the severity of data corruption by orders of magnitude but nevertheless still permit serious integrity violations.

Given these results, we return to our goal of improving the underlying data management systems that power these applications and present recommendations for the database research community. We expand our study to survey several additional web frameworks and demonstrate that many also provide a notion of feral validations, suggesting an industry-wide trend. While the success of Rails and its ilk—despite (or perhaps due to) their aversion to database technology—are firm evidence of the continued impedance mismatch between object-oriented programming and the relational model, we see considerable opportunity in improving database systems to better serve these communities—via more programmer- and ORM-friendly interfaces that ensure correctness while minimizing impacts on performance and portability.

6.4.1 Background and Context

As a primary focus of our study, we investigate the operational model, database use, and application primitives provided in Rails. In this section, we provide a overview of the Rails programming model and describe standard Rails deployment architectures.

Rails Tenets and MVC

Rails was developed in order to maximize developer productivity. This focus is captured by two core architectural principles [199]. First, Rails adopts a "Don't Repeat Yourself" (DRY) philosophy: "every piece of knowledge should be expressed in just one place" in the code. Data modeling and schema descriptions are relegated to one portion of the system, while presentation and business logic are relegated to two others. Rails attempts to minimize the amount of boilerplate code required to achieve this functionality. Second, Rails

adopts a philosophy of "Convention over Configuration," aiming for sensible defaults and allowing easy deployment without many—if any—modifications to configuration.

A natural corollary to the above principles is that Rails encourages an idiomatic style of programming. The Rails framework authors claim that "somehow, [this style] just seems right" for quickly building responsive web applications [199]. The framework's success hints that its idioms are, in fact, natural to web developers.

More concretely, Rails divides application code into a three-component architecture called Model-View-Controller [114, 147]:

- The **Model** acts as a basic ORM and is responsible for representing and storing business objects, including schemas, querying, and persistence functionality. For example, in a banking application, an account's state could be represented by a model with a numeric owner ID field and a numeric balance field.
- The **View** acts as a presentation layer for application objects, including rendering into browser-ingestible HTML and/or other formats such as JSON. In our banking application, the View would be responsible for rendering the page displaying a user's account balance.
- The **Controller** encapsulates the remainder of the application's business logic, including actual generation of queries and transformations on the Active Record models. In our banking application, we would write logic for orchestrating withdrawal and deposit operations within the Controller.

Actually building a Rails application is a matter of instantiating a collection of models and writing appropriate controller and presentation logic for each.

As we are concerned with how Rails utilizes database back-ends, we largely focus on how Rails applications interact with the Model layer. Rails natively supports a Model implementation called `Active Record`. Rails's Active Record module is an implementation of the Active Record pattern originally proposed by Martin Fowler, a prominent software design consultant [113]. Per Fowler, an Active Record is "an object that wraps a row in a database or view, encapsulates the database access, and adds domain logic on that data" (further references to Active Record will correspond to Rails's implementation). The first two tasks—persistence and database encapsulation—fit squarely in the realm of standard ORM design, and Rails adopts Fowler's recommendation of a one-to-one correlation between object fields and database columns (thus, each declared Active Record class is stored in a separate table in the database). The third component, domain logic, is more complicated. Each Rails model may contain a number of attributes (and must include a special

primary-key-backed `id` field) as well as associated logic including data validation, associations, and other constraints. Fowler suggests that "domain logic that isn't too complex" is well-suited for encapsulation in an Active Record class. We will discuss these in greater depth in the next section.

Databases and Deployment

This otherwise benign separation of data and logic becomes interesting when we consider how Rails servers process concurrent requests. In this section, we describe how, in standard Rails deployments, application logic may be executed concurrently and without synchronization within separate threads or processes.

In Rails, the database is—at least for basic usages—simply a place to store model state and is otherwise divorced from the application logic. All application code is run within the Ruby virtual machine (VM), and Active Record makes appropriate calls to the database in order to materialize collections of models in the VM memory as needed (as well as to persist model state). However, from the database's perspective (and per DHH's passionate declaration in Section 6.4), logic remains in the application layer. Active Record natively provides support for PostgreSQL, MySQL, and SQLite, with extensions for databases including Oracle and is otherwise agnostic to database choice.

Rails deployments typically resemble traditional multi-tier web architectures [16] and consist of an HTTP server such as Apache or Nginx that acts as a proxy for a pool of Ruby VMs running the Rails application stack. Depending on the Ruby VM and Rails implementation, the Rails application may or may not be multi-threaded.[1] Thus, when an end-user makes a HTTP request on a Rails-powered web site, the request is first accepted by a web server and passed to a Rails worker process (or thread within the process). Based on the HTTP headers and destination, Rails subsequently determines the appropriate Controller logic and runs it, including any database calls via Active Record, and renders a response via the View, which is returned to the HTTP server.

Thus, in a Rails application, the *only* coordination between individual application requests occurs within the database system. Controller execution—whether in separate threads or across Ruby VMs (which may be active on different physical servers)—is entirely inde-

[1] Ruby was not traditionally designed for highly concurrent operations: its standard reference VM—Ruby MRI—contains (like Python's CPython) a "Global VM Lock" that prevents multiple OS threads from executing at a given time. While alternative VM implementations provide more concurrent behavior, until Rails 2.2 (released in November 2008), Rails embraced this behavior and was unable to process more than one request at a time (due to state shared state including database connections and logging state) [184]. In practice today, the choice of multi-process, multi-threaded, or multi-process and multi-threaded deployment depends on the actual application server architecture. For example, three popular servers—Phusion Passenger, Puma, and Unicorn —each provide a different configuration.

pendent, save for the rendezvous of queries and modifications within the database tier, as triggered by Active Record operations.

The independent execution of concurrent business logic should give serious pause to disciples of transaction processing systems. Is this execution strategy actually safe? Thus far, we have yet to discuss any mechanisms for maintaining correct application data, such as the use of transactions. In fact, as we will discuss in the next section, Rails has, over its lifetime, introduced several mechanisms for maintaining consistency of application data. In keeping with Rails' focus on maintaining application logic within Rails (and not within the database), this has led to several different proposals. In the remainder of this paper, we examine their use and whether, in fact, they correctly maintain application data.

6.4.2 Feral Mechanisms in Rails

As we discussed in Section 6.4.1, Rails services user requests independently, with the database acting as a point of rendezvous for concurrent operations. Given Rails's design goals of maintaining application logic at the user level, this appears—on its face—a somewhat cavalier proposition with respect to application integrity. In response, Rails has developed a range of concurrency control strategies, two of which operate external to the database, at the application level, which we term *feral concurrency control* mechanisms.

In this section, we outline four major mechanisms for guarding against integrity violations under concurrent execution in Rails. We subsequently begin our study of 67 open source applications to determine which of these mechanisms are used in practice. In the following section, we will determine which are sufficient to maintain correct data—and when they are not.

Rails Concurrency Control Mechanisms

Rails contains four main mechanisms for concurrency control.

1. Rails provides support for **transactions**. By wrapping a sequence of operations within a special `transaction` block, Rails operations will execute transactionally, backed by an actual database transaction. The database transaction either runs at the database's configured default isolation level or, as of Rails 4.0.0, can be configured on a per-transaction basis [158].

2. Rails provides support for both optimistic and pessimistic per-record **locking**. Applications invoke pessimistic locks on an Active Record object by calling its `lock` method,

which invokes a `SELECT FOR UPDATE` statement in the database. Optimistic locking is invoked by declaring a special `lock_version` field in an Active Record model. When a Rails process performs an update to an optimistically locked model, Active Record uses a transaction to atomically check whether the corresponding record's `lock_version` field has changed since the process last read the object. If the record has not changed, Rails transactionally increments `lock_version` and updates the database record; if the record has changed, the update fails.

3. Rails provides support for application-level **validations**. Each Active Record model has a set of zero or more validations, or boolean-valued functions, and a model instance many only be saved to the database if all of its declared validations return true. These validations ensure, for example, that particular fields within a record are not null or are unique within the database. Rails provides a number of built-in validations but also allows arbitrary user-defined validations (we discuss actual validations further in subsequent sections). The framework runs each declared validation sequentially and, if all succeed, the model state is updated in the database; this happens within a database-backed transaction.[2] The validations supported by Rails today include ones that are natively supported by many commercial databases today, as well as others.

4. Rails provides support for application-level **associations**. As the name suggests, "an association is a connection between two Active Record models," effectively acting like a foreign key in an RDBMS. Associations can be declared on one or both sides of a one-to-one or one-to-many relationship, including transitive dependencies (via a `:through` annotation). Declaring an association (e.g., `:belongs_to dept`) produces a special field for the associated record ID within the model (e.g., `dept_id`). Coupling an association with an appropriate validation (e.g., `:presence`) ensures that the association is indeed valid (and is, via the validation, backed by a database transaction). Until the release of Rails 4.2 in December 2014, Rails did not provide native support for database-backed foreign key constraints. In Rails 4.2, foreign keys are supported via manual schema annotations declared separately from each model; declaring an association does not declare a corresponding foreign key constraint and vice-versa.

Overall, these four mechanisms provide a range of options for developers. The first is squarely in the realm of traditional concurrency control. The second is, in effect, a coarse-grained user-level implementation of single-record transactions via database-level

[2]The practice of wrapping validations in a transaction dates to the earliest public Rails commit (albeit, in 2004, transactions were only supported via a per-Ruby VM global lock [129]). However, as late as 2010, updates were only partially protected by transactions [208].

"compare-and-swap" primitives (implemented via `SELECT FOR UPDATE`). However, the latter two—validations and associations—operate, in effect, at the application level. Although some validations like uniqueness validations have analogs in an RDBMS, the semantics of these validations are entirely contained within the Rails code. In effect, from the database's perspective, these validations exist external to the system and are *feral* concurrency control mechanisms.

Rails's feral mechanisms—validations and associations—are a prominent feature of the Active Record model. In contrast, neither transactions nor locks are actually discussed in the official "Rails Guides," and, generally, are not promoted as a means of ensuring data integrity. Instead, the Rails documentation [6] prefers validations as they are "are database agnostic, cannot be bypassed by end users, and are convenient to test and maintain." Moreover, the Rails documentation opines that "database constraints and/or stored procedures make the validation mechanisms database-dependent and can make testing and maintenance more difficult." As we will show shortly, these feral mechanisms accordingly dominate in terms of developer popularity in real applications.

Adoption in Practice

To understand exactly how users interact with these concurrency control mechanisms and determine which deserved more study, we examined their usage in a portfolio of publicly available open source applications. We find that validations and associations are overwhelmingly the most popular forms of concurrency control.

Application corpus. We selected 67 open source applications built using Ruby on Rails and Active Record, representing a variety of application domains, including eCommerce, customer relationship management, retail point of sale, conference management, content management, build management, project management, personal task tracking, community management and forums, commenting, calendaring, file sharing, Git hosting, link aggregation, crowdfunding, social networking, and blogging. We sought projects with substantial code-bases (average: 26,809 lines of Ruby) multiple contributors (average: 69.1), and relative popularity (measured according to GitHub stars) on the site. Table 6.3 provides a detailed overview.

To determine the occurrences and number of models, transactions, locks, validations, and associations in Rails, we wrote a set of analysis scripts that performed a very rudimentary syntactic static analysis . We do not consider the analysis techniques here a contribution; rather, our interest is in the output of the analysis. The syntactic approach proved portable between the many versions of Rails against which each application is linked; otherwise, porting between non-backwards-compatible Rails versions was difficult and, in fact,

unsupported by several of the Rails code analysis tools we considered using as alternatives. The choice to use syntax as a means of distinguishing code constructs led to some ambiguity. To compensate, we introduced custom logic to handle esoteric syntaxes that arose in particular projects (e.g., some projects extend `ActiveRecord::Base` with a separate, project-specific base class, while some validation usages vary between constructs like `:validates_presence` and `:validates_presence_of`).

To determine code authorship, we used the output of git `log` and `blame` and did not attempt any sophisticated entity resolution.

Name	Description	Authors	LoC Ruby	Commits	M	T	PL	OL	V	A	Stars	Githash	Last commit
Canvas LMS	Education	132	309,580	12,853	161	46	12	1	354	837	1,251	3fb8e69	10/16/14
OpenCongress	Congress data	15	30,867	1,884	106	1	0	0	48	357	124	850b602	02/11/13
Fedena	Education management	4	49,297	1,471	104	5	0	0	153	317	262	40cafe3	01/23/13
Discourse	Community discussion	440	72,225	11,480	77	41	0	0	83	266	12,233	1cf4a0d	10/20/14
Spree	eCommerce	677	47,268	14,096	72	6	0	0	92	252	5,582	aa34b3a	10/16/14
Sharetribe	Content management	35	31,164	7,140	68	0	0	0	112	202	127	8e0d382	10/21/14
ROR Ecommerce	eCommerce	19	16,808	1,604	63	2	3	0	219	207	857	c60a675	10/09/14
Diaspora	Social network	388	31,726	14,640	63	2	0	0	66	128	9,571	1913397	10/03/14
Redmine	Project management	10	81,536	11,042	62	11	0	1	131	157	2,264	e23d4d9	10/19/14
ChiliProject	Project management	53	66,683	5,532	61	7	0	1	118	130	623	984c9ff	08/13/13
Spot.us	Community reporting	46	94,705	9,280	58	0	0	0	96	165	343	61b65b6	12/02/13
Jobsworth	Project management	46	24,731	7,890	55	10	0	0	86	225	478	3a1f8e1	09/12/14
OpenProject	Project management	63	84,374	11,185	49	8	1	3	136	227	371	c1e66af	11/21/13
Danbooru	Image board	25	27,857	3,738	47	9	0	0	71	114	238	c082ed1	10/17/14
Salor Retail	Point of Sale	26	18,404	2,259	44	0	0	0	81	309	24	00e1839	10/07/14
Zena	Content management	7	56,430	2,514	44	1	0	0	12	43	172	79576ac	08/18/14
Skyline CMS	Content management	7	10,404	894	40	5	0	0	28	89	127	64b0932	12/09/13
Opal	Project management	6	10,707	474	38	3	0	0	42	96	45	11edf34	01/09/13
OneBody	Church portal	33	20,398	3,973	36	3	0	0	97	140	1,041	2dfbd4d	10/19/14
CommunityEngine	Social networking	67	13,967	1,613	35	3	0	0	92	101	1,073	a4d3ea2	10/16/14
Publify	Blogging	93	16,763	5,067	35	7	0	0	33	50	1,274	4acf86e	10/20/14
Comas	Conference management	5	5,879	435	33	6	0	0	80	45	21	81c25a4	09/09/14
BrowserCMS	Content management	56	21,259	2,503	32	4	0	0	47	77	1,183	d654557	09/30/14
RailsCollab	Project management	25	8,849	865	29	6	0	0	40	122	262	9f6c8c1	02/16/12
OpenGovernment	Government data	15	9,383	2,231	28	4	0	0	22	141	160	fa80204	11/21/13
Tracks	Personal productivity	89	17,419	3,121	27	2	0	0	24	43	639	eb2650c	10/02/14
GitLab	Code management	671	39,094	12,266	24	15	0	0	131	114	14,129	72abe9f	10/20/14
Brevidy	Video sharing	2	7,608	6	24	1	0	0	74	56	167	d0ddb1a	01/18/14
Insoshi	Social network	16	121,552	1,321	24	1	0	0	41	63	1,583	9976cfe	02/24/10
Alchemy	Content management	34	19,329	4,222	23	2	0	0	37	40	240	91d9d08	10/20/14
Teambox	Project management	48	32,844	3,155	22	2	0	0	56	116	1,864	62a8b02	09/20/11
Fat Free CRM	Customer relationship	99	21,284	4,144	21	3	0	0	39	92	2,384	3dd2c62	10/17/14
linuxfr.org	FLOSS community	29	8,123	2,271	20	1	0	0	50	50	86	5d4d6df	10/14/14

Squash	Bug reporting	28	15,776	231	19	6	0	0	87	62	879	c217ac1	09/15/14
Shoppe	eCommerce	14	3,172	349	19	1	0	0	58	34	208	19e60c8	10/18/14
nimbleShop	eCommerce	12	8,041	1,805	19	0	0	0	47	34	47	4254806	02/18/13
Piggybak	eCommerce	16	2,235	383	17	1	0	0	51	35	166	2bed094	09/10/14
wallgig	Wallpaper sharing	6	5,543	350	17	1	0	0	42	45	18	4424d44	03/23/14
Rucksack	Collaboration	7	5,346	445	17	3	0	0	18	79	169	59703d3	10/05/13
Calagator	Online calendar	48	9,061	1,766	16	0	0	0	8	11	196	6e5df08	10/19/14
Amahi Platform	Home media sharing	15	6,244	577	15	2	0	0	38	22	65	5101c8b	08/20/14
Sprint	Project management	5	3,056	71	14	0	0	0	50	45	247	584d887	09/17/14
Citizenry	Community directory	17	8,197	512	13	0	0	0	12	45	138	e314fe4	04/01/14
LovdByLess	Social network	17	30,718	150	12	0	0	0	27	41	568	26e79a7	10/09/09

Table 6.3: Corpus of applications used in analysis (M: Models, T: Transactions, PL: Pessimistic Locking, OL: Optimistic Locking, V: Validations, A: Associations). Stars record number of GitHub Stars as of October 2014.

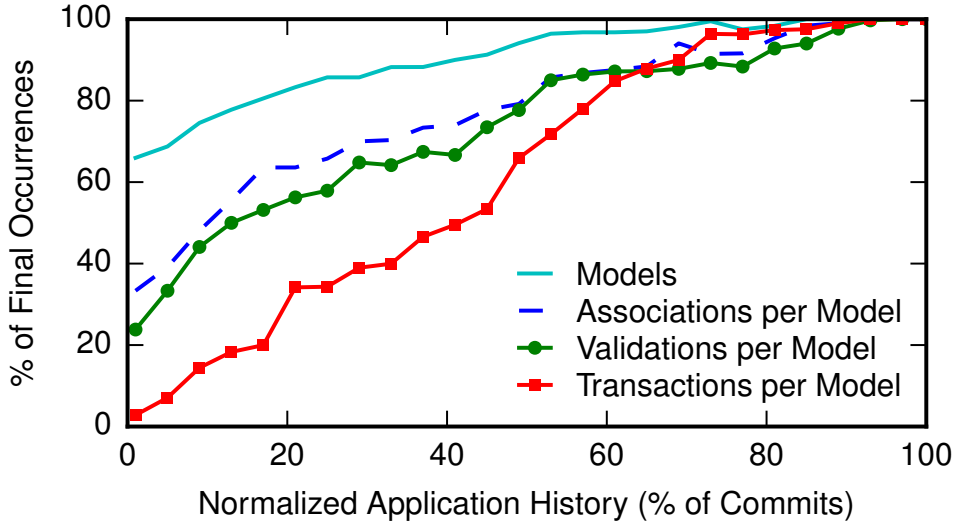


Figure 6.3: Use of mechanisms over each project's history. We plot the median value of each metric across projects and, for each mechanism, omit projects that do not contain any uses of the mechanism (e.g., if a project lacks transactions, the project is omitted from the median calculation for transactions).

While several of these applications are projects undertaken by hobbyists, many are either commercially supported (e.g., Canvas LMS, Discourse, Spree, GitLab) and/or have a large open source community (e.g., Radiant, Comfortable Mexican Sofa, Diaspora). A larger-scale commercial, closed-source Rails application such as Twitter, GitHub, or Airbnb might exhibit different trends than those we observe here. However, in the open source domain, we believe this set of applications contains a diverse selection of Rails use cases and is a reasonably representative sample of popular open source Rails applications as hosted on GitHub.

Mechanism usage. We performed a simple analysis of the applications to determine how each of the concurrency control mechanisms were used.

Overwhelmingly, applications did not use transactions or locks (Figure 6.5 and Table 6.3). On average, applications used 0.13 transactions, 0.01 locks, 1.80 validations, and 3.19 associations per model (with an average of 29.1 models per application). While 46 (68.7%) of applications used transactions, all used some validations or associations. Only six applications used locks. Use of pessimistic locks was over twice as common as the use of optimistic locks.

Perhaps most notable among these general trends, we find that validations and associations are, respectively, 13.6 and 24.2 times more common than transactions and orders of

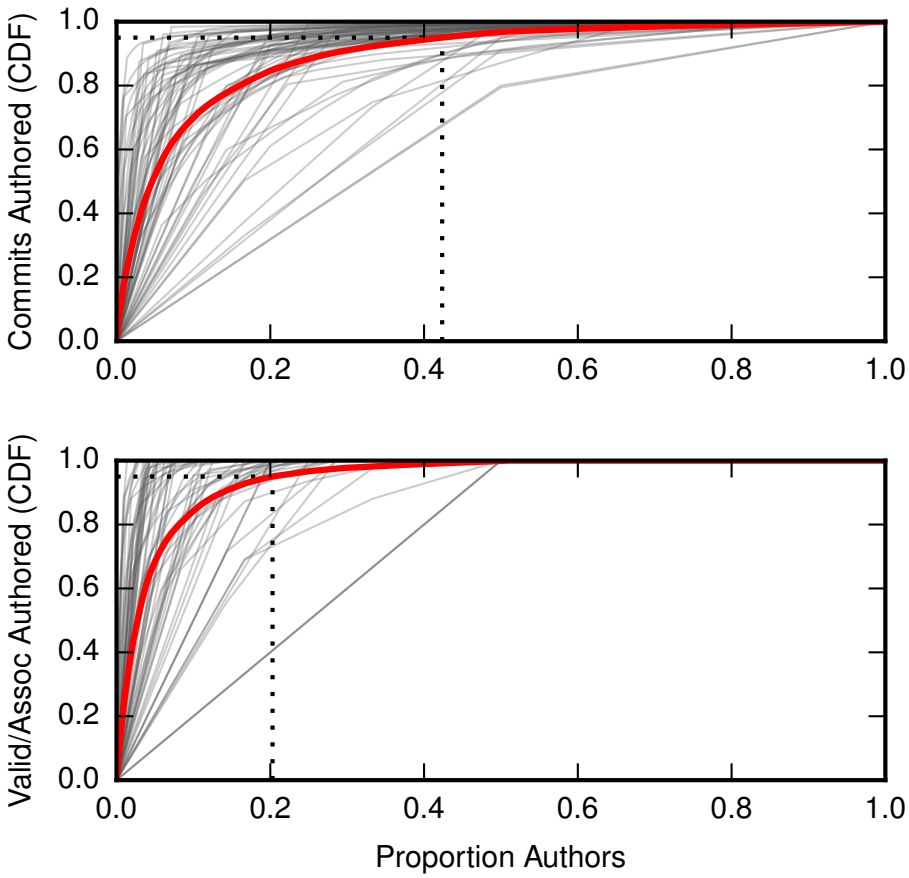


Figure 6.4: CDFs of authorship of invariants (validations plus associations) and commits. Bolded line shows the average CDF across projects, while faint lines show CDFs for individual projects. The dotted line shows the 95th percentile CDF value.

magnitude more common than locking. These feral mechanisms are—in keeping with the Rails philosophy—favored by these application developers. That is, rather than adopting the use of traditional transactional programming primitives, Rails application writers chose to instead specify correctness criteria and have the ORM system enforce the criteria on their behalf. It is unclear and even unlikely that these declarative criteria are a complete specification of program correctness: undoubtedly, some of these programs contain errors and omissions. However, given that these criteria are nevertheless being declared by application writers and represent a departure from traditional, transaction-oriented programming, we devote much of the remainder of this work to examining exactly what they are attempting to preserve (and whether they are actually sufficient to do so).

Understanding specific applications. Over the course of our investigation, we found that application use of mechanisms varied. While our focus is largely on aggregate behavior,

studying individual applications is also interesting. For example, consider Spree, a popular eCommerce application:

Spree uses only six transactions, one for each of 1.) canceling an order, 2.) approving an order (atomically setting the user ID and timestamp), 3.) transferring shipments between fulfillment locations (e.g., warehouses), 4.) transferring items between shipments, 5.) transferring stock between fulfillment locations, and 6.) updating an order's specific inventory status. While this is a reasonable set of locations for transactions, in an eCommerce application, one might expect a larger number of scenarios to require transactions, including order placement and stock adjustment.

In the case of Spree stock adjustment, the inventory count for each item is a potential hotspot for concurrency issues. Manual adjustments of stock that is available ("adjust_count_on_hand") is indeed protected via a pessimistic lock, but simply setting the amount of stock that is available ("set_count_on_hand") is not. It is unclear why one operation necessitates a lock but the other does not, given that both are ostensibly sensitive to concurrent accesses. Meanwhile, the stock level field is wrapped in a validation ensuring non-negative balances, preventing negative balances but not necessarily classic Lost Update anomalies [9].

At one point, Spree's inventory count was protected by an optimistic lock; it was removed due to optimistic lock failure during customer checkouts. On relevant GitHub issue pertaining to this lock removal, a committer notes that "I think we should get rid of the [optimistic lock] if there's no documentation about why it's there...I think we can look at this issue again in a month's time and see if there's been any problems since you turned it off" [88]. This removal has, to our knowledge, not been revisited, despite the potential dangers of removing this point of synchronization.

The remainder of the application corpus contains a number of such fascinating examples, illustrating the often ad-hoc process of deciding upon a concurrency control mechanism. Broadly, the use of each style of concurrency control varies across repositories, but our results demonstrate a clear trend towards feral mechanisms within Rails rather than traditional use of transactions.

Additional metrics. To better understand how programmers used each of these mechanisms, we performed two additional analyses.

First, we analyzed the number of models, transactions, validations, and associations over each project's lifetime. Using each project's Git history, we repeated the above analysis at a fixed set of intervals through the project's lifespan (measured by commits). Figure 6.3 plots the median number of occurrences across all projects. The results show that concurrency control mechanisms (of all forms) tend to be introduced after models are introduced.

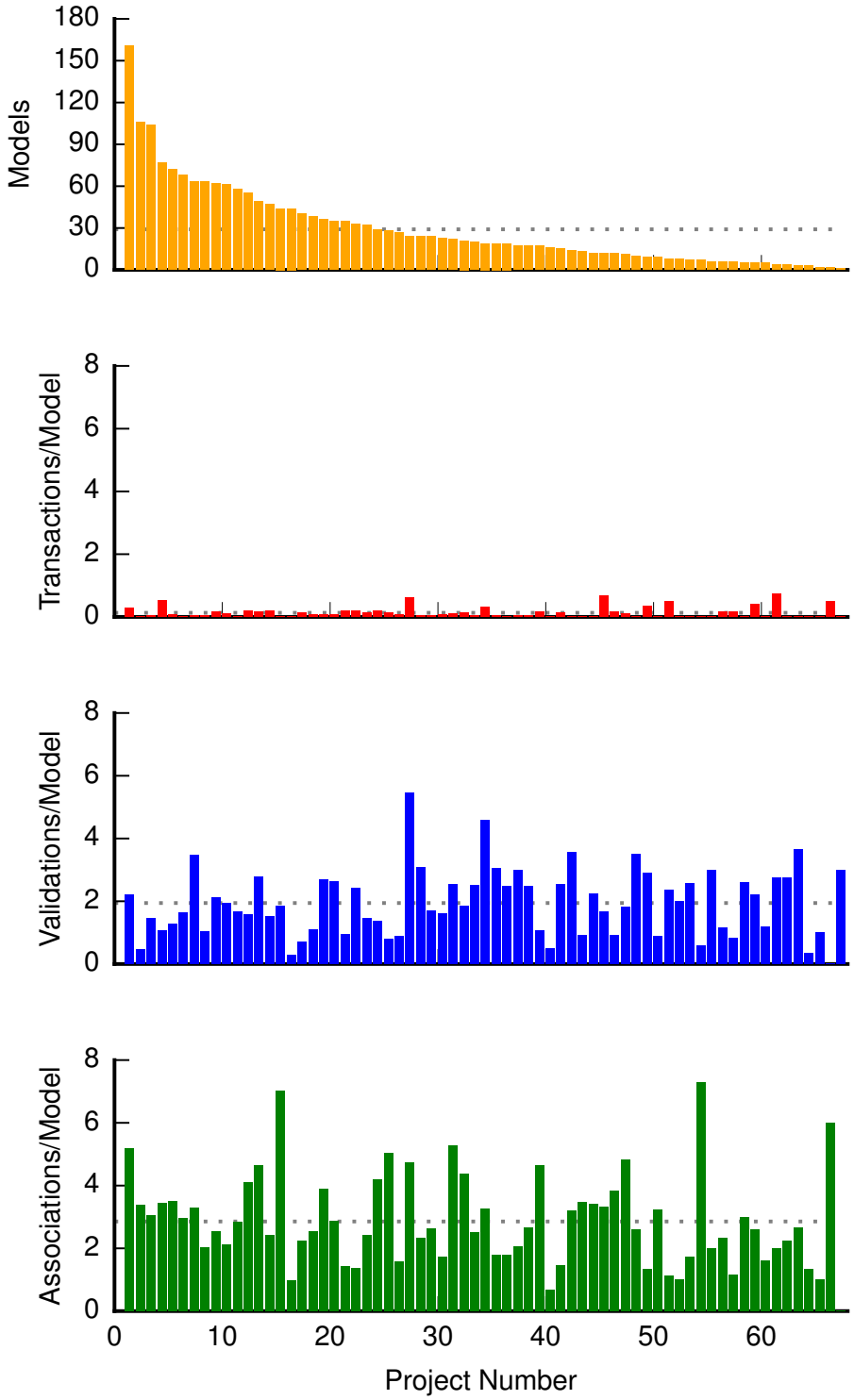


Figure 6.5: Use of concurrency control mechanisms in Rails applications. We maintain the same ordering of applications for each plot (i.e., same x-axis values; identical to Table 6.3) and show the average for each plot using the dotted line.

That is, additions to the data model precede (often by a considerable amount) additional uses of transactions, validations, and associations. It is unclear whether the bulk of concurrency control usage additions are intended to correct concurrency issues or are instead due to natural growth in Controller code and business logic. However, the gap between models and concurrency control usage shrinks over time; thus, the data model appears to stabilize faster than the controller logic, but both eventually stabilize. We view additional longitudinal analysis along these lines as worthwhile future work.

Second, we analyze the distribution of authors to commits compared to the distribution of authors to validations and associations authored.[3] As Figure 6.4 (see Appendix, page 149) demonstrates, 95% of all commits are authored by 42.4% of authors. However, 95% of invariants (validations plus associations) are authored by only 20.3% of authors. This is reminiscent of database schema authorship, where, traditionally, a smaller number of authors (e.g., DBAs) modify the schema than contribute to the actual application code.

Summary and Discussion

Returning to the Rails design philosophy, the applications we have encountered do indeed express their logic at the application layer. There is little actual communication of correctness criteria to the database layer. Part of this is due to limitations within Rails. As we have mentioned, there is no way to actually declare a foreign key constraint in Rails without importing additional third-party modules. Insofar as Rails is an "opinionated" framework encouraging an idiomatic programming style, if our application corpus is any indication, DHH and his co-authors advocating application-level data management appear to have succeeded en masse.

Having observed the relative popularity of these mechanisms, we turn our attention to the question of their correctness. Specifically, do these application-level criteria actually enforce the constraints that they claim to enforce? We restrict ourself to studying declared validations and associations for three reasons. First, as we have seen, these constructs are more widely used in the codebases we have studied. Second, these constructs represent a deviation from standard concurrency control techniques and are therefore perhaps more likely to contain errors. Third, while analyzing latent constraints (e.g., those that might be determined via more sophisticated techniques such as pre- and post-condition invariant mining [160, 198] and/or by interviewing each developer on each project) would be instructive, this is difficult to scale. We view these forms of analysis as highly promising avenues for future research.

[3] We chose to analyze commits authored rather than lines of code written because git tracks large-scale code refactoring commits as an often large set of deletions and insertions. Nevertheless, we observed a close correlation between lines of code and commits authored.

6.4.3 Rails Invariant Confluence Analysis

We now turn our attention to understanding which of Rails' feral validations and associations are actually correct under concurrent execution as described in Section 6.4.1 and which require stronger forms of isolation or synchronization for correct enforcement.

Understanding Validation Behavior

To begin, recall that each sequence of validations (and model update as well, if validations pass) is wrapped within a database-backed transaction, the validation's intended integrity will be preserved provided the database is using serializable isolation. However, relational database engines often default to non-serializable isolation [32]; notably for Rails, PostgreSQL and MySQL actually default to, respectively, the weaker Read Committed and Repeatable Read isolation levels.

We did not encounter evidence that applications changed the isolation level. Rails does not configure the database isolation level for validations, and none of the application code or configurations we encountered change the default isolation level, either (or mention doing so in documentation). Thus, although we cannot prove that this is indeed the case, this data suggests that validations are likely to run at default database isolation in production environments.

Validations with weak isolation. Given that validations are not likely to be perfectly isolated, does this lack of serializable isolation actually affect these invariants? Just because validations effectively run concurrently does not mean that they are necessarily incorrect. To determine exactly which of these invariants are correct under concurrent execution, we employ invariant confluence analysis. In the case of Rails, we wish to determine whether, in the event of concurrent validations and model saves, the result of concurrent model saves will not violate the validation for either model. In the event that two concurrent controllers save objects that are backed by the same database record, only one will be persisted (a some-write-wins "merge"). In the event that two concurrent controllers save different models (i.e., backed by different database records), both will be persisted (a set-based "merge"). In both cases, we must ensure that validations hold after merge.

Per above, our invariant confluence analysis currently relies on a combination of manual proofs and simple static analysis: given a set of invariant and operation pairs classified as providing the invariant confluence property, we can iterate through all operations and declared invariants and check whether or not they appear in the set of invariant confluent pairs. If so, we label the pair as invariant confluent. If not, we can either conservatively label the pair as unsafe under concurrent execution or prove the pair as invariant confluent

or not. (To prove a pair is invariant confluent, we must show that the set of database states reachable by executing operations preserves the invariant under merge, as described above.)

Returning to our task of classifying Rails validations and associations as safe or not, we applied this invariant confluence analysis to the invariants[4] in the corpus. In our analysis, we found that only 60 out of 3505 validations were expressed as user-defined functions. The remainder were drawn from the standard set of validations supported by Rails core.[5] Accordingly, we begin by considering built-in validations, then examine each of the custom validations.

Built-In Validations

We now discuss common, built-in validations and their invariant confluence. Many are invariant confluent and are therefore safe to execute concurrently.

Table 6.4 presents the ten most common built-in validations by usage and their occurrences in our application corpus. The exact coordination requirements depended on their usage.

The most popular invariant, `presence`, serves multiple purposes. Its basic behavior is to simply check for empty values in a model before saving. This is invariant confluent as, in our model, concurrent model saves cannot result in non-null values suddenly becoming null. However, `presence` can also be used to enforce that the opposite end of an association is, in fact, present in the database (i.e., referential integrity). Under insertions, foreign key constraints are invariant confluent [34], but, under deletions, they are not.

The second most popular invariant, concerning record uniqueness, is *not* invariant confluent [34]. That is, if two users concurrently insert or modify records, they can introduce duplicates.

Eight of the next nine invariants are largely concerned with data formatting and are invariant confluent. For example, `numericality` ensures that the field contains a number rather than an alphanumeric string. These invariants are indeed invariant confluent under concurrent update.

Finally, the safety of `associated` (like `presence`) is contingent on whether or not the current updates are both insertions (invariant confluent) or mixed insertions and deletions (not invariant confluent). Thus, correctness depends on the operation.

[4] We focus on validations here as, while associations *do* represent an invariant, it is only when they are coupled with validations that they are enforced.

[5] It is unclear exactly why this is the case. It is possible that, because these invariants are standardized, they are more accessible to users. It is also possible that Rails developers have simply done a good job of codifying common patterns that programmers tend to use.

Name	Occurrences	I-Confluent?
validates_presence_of	1762	Depends
validates_uniqueness_of	440	No
validates_length_of	438	Yes
validates_inclusion_of	201	Yes
validates_numericality_of	133	Yes
validates_associated	39	Depends
validates_email	34	Yes
validates_attachment_content_type	29	Yes
validates_attachment_size	29	Yes
validates_confirmation_of	19	Yes
Other	321	Mixed

Table 6.4: Use of and invariant confluence of built-in validations.

Overall, a large number of built-in validations are safe under concurrent operation. Under insertions, 86.9% of built-in validation occurrences as invariant confluent. Under deletions, only 36.6% of occurrences are invariant confluent. However, associations and multi-record uniqueness are—depending on the workload—not invariant confluent and are therefore likely to cause problems. In the next section, we examine these validations in greater detail.

Custom Validations

We also manually inspected the coordination requirements of the 60 (1.71%) validations (from 17 projects) that were declared as UDFs. 52 of these custom validations were declared inline via Rails's `validates_each` syntax, while 8 were custom classes that implemented Rails's validation interface. 42 of 60 validations were invariant confluent, while the remaining 18 were not. Due to space constraints, we omit a discussion of each validation but discuss several trends and notable examples of custom validations below.

Among the custom validations that were invariant confluent, many consisted of simple format checks or other domain-specific validations, including credit card formatting and static username blacklisting.

The validations that were not invariant confluent took on a range of forms. Three validations performed the equivalent of foreign key checking, which, as we have discussed, is unsafe under deletion. Three validations checked database-backed configuration options including the maximum allowed file upload size and default tax rate; while configuration updates are ostensibly rare, the outcome of each validation could be affected under a configuration change. Two validations were especially interesting. Spree's `AvailabilityValidator`

checks whether an eCommerce inventory has sufficient stock available to fulfill an order; concurrent order placement might result in negative stock. Discourse's `PostValidator` checks whether a user has been spamming the forum; while not necessarily critical, a spammer could technically foil this validation by attempting to simultaneously author many posts.

In summary, again, a large proportion of validations appear safe. Nevertheless, the few non-invariant confluent validations should be cause for concern under concurrent execution.

6.5 Quantifying Integrity Violations in Rails

While many of the validations we encountered were invariant confluent, not all were. In this section, we specifically investigate the effect of concurrent execution on two of the most popular non-invariant confluent validations: uniqueness and foreign key validations.

Uniqueness Constraints and Isolation

To begin, we consider Rails's uniqueness validations: 12.7% of the built-in validation uses we encountered. In this section, we discuss how Rails implements uniqueness and show that this is—at least theoretically—unsafe.

When a model field is declared with a `:validates_uniqueness` annotation, any instance of that model is compared against all other corresponding records in the database to ensure that the field is indeed unique. ActiveRecord accomplishes this by issuing a "`SELECT`" query in SQL and, if no such record is found, Rails updates the instance state in the database (Appendix 6.8.1).

While this user-level uniqueness validation runs within a transaction, the isolation level of the transaction affects its correctness. For correct execution, the `SELECT` query must effectively attain a predicate lock on the validated column for the duration of the transaction. This behavior *is* supported under serializable isolation. However, under Read Committed or Repeatable Read isolation, no such mutual exclusion will be performed, leading to potential inconsistency.[6] Moreover, validation under Snapshot Isolation may similarly result in inconsistencies.[7] Thus, unless the database is configured for serializable isolation, integrity

[6] Using `SELECT FOR UPDATE` under these weaker models would be safe, but Rails does not implement its predicate-based lookups as such (i.e., it instead opts for a simple `SELECT` statement).

[7] The first reference to the potential integrity violations resulting from this implementation in the Rails code that we are aware of dates to December 2007, in Rails v.2.0.0 [146]. In September 2008, another user added additional discussion within the code comments, noting that "this could even happen if you use transactions

violations may result.

As we have discussed, MySQL and PostgreSQL each support serializable isolation but default to weaker isolation. Moreover, in our investigation, we discovered a bug in PostgreSQL's implementation of Serializable Snapshot Isolation that allowed duplicate records to be created under serializable isolation when running a set of transactions derived from the Rails primary key validator. We have confirmed this anomalous behavior with the core PostgreSQL developers[8] and, as of March 2015, the behavior persists. Thus, any discussion of weak isolation levels aside, PostgreSQL's implementation of serializability is non-serializable and is insufficient to provide correct behavior for Rails' uniqueness validations. So-called "serializable" databases such as Oracle 12c that actually provide Snapshot Isolation will similarly fall prey to duplicate validations.

The Rails documentation warns that uniqueness validations may fail and admit duplicate records [6]. Yet, despite the availability of patches that remedy this behavior by the use of an in-database constraint and/or index, Rails provides this incorrect behavior by default. (One patch was rejected; a developer reports "[t]he reasons for it not being incorporated...are lost in the mists of time but I suspect it's to do with backwards compatibility, cross database compatibility and applications varying on how they want/need to handle these kind of errors." [66]).

In another bug report complaining of duplicates due to concurrent uniqueness validation, a commenter asserts "this is not a bug but documented and inherent behavior of validates_uniqueness_of" [197]. A Rails committer follows up, noting that "the only way to handle [uniqueness] properly is at the database layer with a unique constraint on the column," and subsequently closes the issue. The original bug reporter protests that "the problem extends beyond unique constraints and into validations that are unique to a Rails application that can't [sic?!] be enforced on the DB level"; the Rails committer responds that "with the possible exception of [associations,] all of the other validations are constrained by the attribute values currently in memory, so aren't susceptible to similar flaws." This final statement is correct for many of the built-in validations but is not correct for arbitrary user-defined validations. We discuss the user-defined validation issue further in Section 6.7.

Understanding validation behavior. Given that entirely feral mechanisms can introduce duplicates, how many duplicates can be introduced? Once a record is written, any later validations will observe it via `SELECT` calls. However, *while* a record is being validated,

with the 'serializable' isolation level" [152]. The use of "'serializable"' suggests familiarity with the common, erroneous labeling of Snapshot Isolation as "serializable" (as in Oracle 12c documentation and PostgreSQL documentation prior to the introduction of SSI in version 9.1.1 in September 2011).

[8] "BUG #11732: Non-serializable outcomes under serializable isolation" at http://www.postgresql.org/message-id/20141021071458.2678.9080@wrigleys.postgresql.org

any number of concurrent validations can unsafely proceed. In practice, the number of concurrent validations is dependent on the Rails environment. In a Rails deployment permitting P concurrent validations (e.g., a single-threaded, multi-process environment with P processes), each value in the domain of the model field/database column can be inserted no more than P times. Thus, validations—at least theoretically—bound the worst-case number of duplicate records for each unique value in the database table.

Quantifying Uniqueness Anomalies

Given that feral uniqueness validations are acknowledged to be unsafe under non-serializable isolation yet are widely used, we sought to understand exactly how often uniqueness anomalies occur in an experimental deployment. In this section, we demonstrate that uniqueness validations in Rails are indeed unsafe under non-serializable isolation. While they prevent some data integrity errors, we observe—depending on the workload—many duplicate records.

Experimental setup. We developed a Rails 4.1.5 application that performed insertions to a non-indexed string column and compared the incidence of violations both with and without a uniqueness validator (see also Appendix 6.8.3).[9] We deployed this application on two Amazon EC2 `m2.4xlarge` instances, offering 68.4 GB RAM, 8 CPU cores, and 1680GB local storage, running Ubuntu 14.04 LTS. On one instance, we deployed our application, using Nginx 1.6.2 as a web frontend proxied to a set of Unicorn 4.8.3 (Ruby VM pool) workers. Nginx acts as a HTTP frontend and forwards incoming requests to a variably sized pool of Rails VMs (managed by Unicorn, in a multi-process, single-threaded server) that `epoll` on a shared Linux file descriptor. On the other EC2 instance, we deployed PostgreSQL 9.3.5 and configured it to run on the instance local storage. We used a third EC2 instance to direct traffic to the front-end instance and drive load. We plot the average and standard deviation of three runs per experiment.

Stress test. We began our study by issuing a simple stress test that executed a number of concurrent insertion requests against a variable number of Unicorn workers. We repeatedly issued a set of 64 concurrent model creation (SQL insertion) requests, each with the same validated key (e.g., all with field `key` set to value 1) against the Rails application. Across an increasing number of Unicorn workers, we repeated this set of requests 100 times (blocking in-between rounds to ensure that each round is, in fact, a concurrent set of requests),

[9] In our experimental evaluation, we use the custom applications described in Section 6.8 for two reasons. First, these test cases allow us to isolate ActiveRecord behavior to the relevant set of validations as they are deployed by default, independent of any specialized controller logic. Second, this reduces the complexity of automated testing. Many of the applications in our corpus indeed use the same code paths within ActiveRecord, but evaluating these custom applications simplifies programmatic triggering of validation logic.

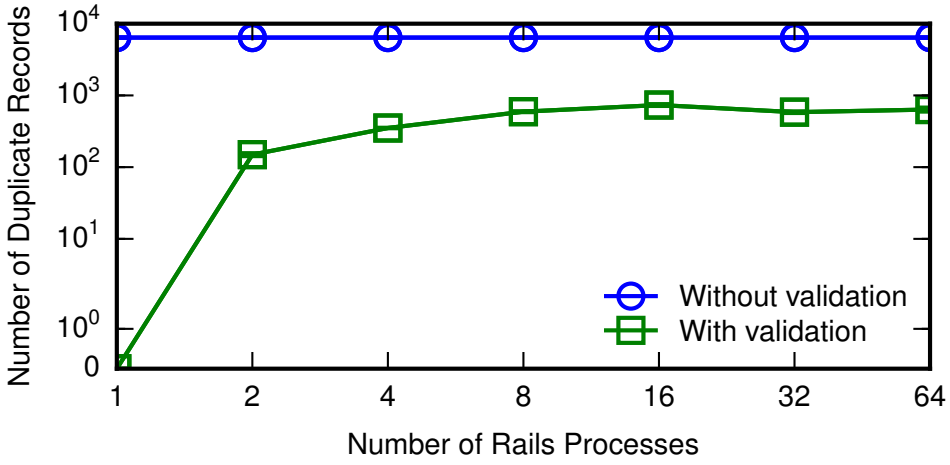


Figure 6.6: Uniqueness stress test integrity violations.

changing the validated key each round (Appendix 6.8.4).

Figure 6.6 shows the results. With no validation, all concurrent requests succeed, resulting in 6300 duplicate records (100 rounds of 64-1 duplicate keys). With validations enabled, the number of violations depends on the degree of concurrency allowed by Unicorn. With only one process, Unicorn performs the validations serially, creating no duplicates. However, with two processes, Unicorn processes race, resulting in 70 duplicate records spread across 70 keys. With three processes, Unicorn produces 249 duplicate records across all 100 keys. The number of duplicates increases with the number of processes, peaking at 16 workers. With additional workers, duplicate counts decrease slightly, which we attribute to thrashing between workers and within PostgreSQL (recall that each instance has only 8 cores). Nevertheless, using validations, the microbenchmark duplicate count remains below 700—nearly an order-of-magnitude fewer duplicates than without using validations. Therefore, even though these validations are incorrectly implemented, they still result in fewer anomalies. However, when we added in in-database unique index on the `key` column[10] and repeated the experiment, we observed no duplicates, as expected.

Actual workloads. The preceding experiment stressed a particularly high-contention workload—in effect, a worst case workload for uniqueness validations. In practice, such a workload is likely rare.[11] Accordingly, we set up another workload meant to capture a

[10]In this case, we added a unique index to the model using Active Record's *database migration*, or manual schema change functionality. Migrations are written separately from the Active Record model declarations. Adding the index was not difficult, but, nevertheless, the index addition logic is separate from the domain model. Without using third-party models, we are unaware of a way to enforce uniqueness within Rails without first declaring an index that is *also* annotated with a special `unique: true` attribute.

[11]In fact, it was in the above workload that we encountered the non-serializable PostgreSQL behavior

less pathological access pattern. We ran another insert-only workload, with key choice distributed among a fixed set of keys. By varying the distribution and number of keys, we were able to both capture more realistic workloads and also control the amount of contention in the workload. As a basis for comparison, we ran four different distributions. First, we considered uniform key access. Second, we used YCSB's Zipfian-distributed accesses from `workloada` [84]. Third and fourth, we used the item distribution access from Facebook's LinkBench workload, which captures MySQL record access when serving Facebook's social graph [25]. Specifically, we used—separately—the insert and update traffic from this benchmark.

For each trial in this workload, we used 64 concurrent clients independently issuing a set of 100 requests each, with a fixed number of 64 Unicorn workers per process (Appendix 6.8.5).

Figure 6.7 illustrates the number of duplicate records observed under each of these workloads. As we increase the number of possible keys, there are two opposing effects. With more keys, the probability of any two operations colliding decreases. However, recall that, once a key is written, all subsequent validators can read it. While the uniform workload observes an average of 2.33 duplicate records with only one possible key, it observes an average of 26 duplicate keys with 1000 possible keys. Nevertheless, with 1 million possible keys, we do not observe any duplicate records.

The actual "production" workloads exhibit different trends. In general, YCSB is an extremely high contention workload, with a Zipfian constant of 0.99, resulting in one very hot key. This decreases the beneficial effect of increasing the number of keys in the database. However, LinkBench has less contention and anomalies decrease more rapidly with increased numbers of keys.

Association Validations and Isolation

Having investigated uniqueness constraints, we turn our attention to association validations. We first, again, discuss how Rails enforces these validations and describe how—at least theoretically—validations might result in integrity errors.

When a model field is declared with an association (e.g., it `:belongs_to` another model) *and* a `:validates_presence` validation, Rails will attempt to ensure that the declared validation is valid before saving the model. Rails accomplishes this by issuing a "`SELECT WHERE`" query in SQL to find an associated record (e.g., to ensure the "one" end of a one-to-many relationship exists) and, if a matching association is found, Rails updates the instance state

under serializable isolation. Under serializable isolation, the number of anomalies is reduced compared to the number under Read Committed isolation (as we report here), but we still detected duplicate records.

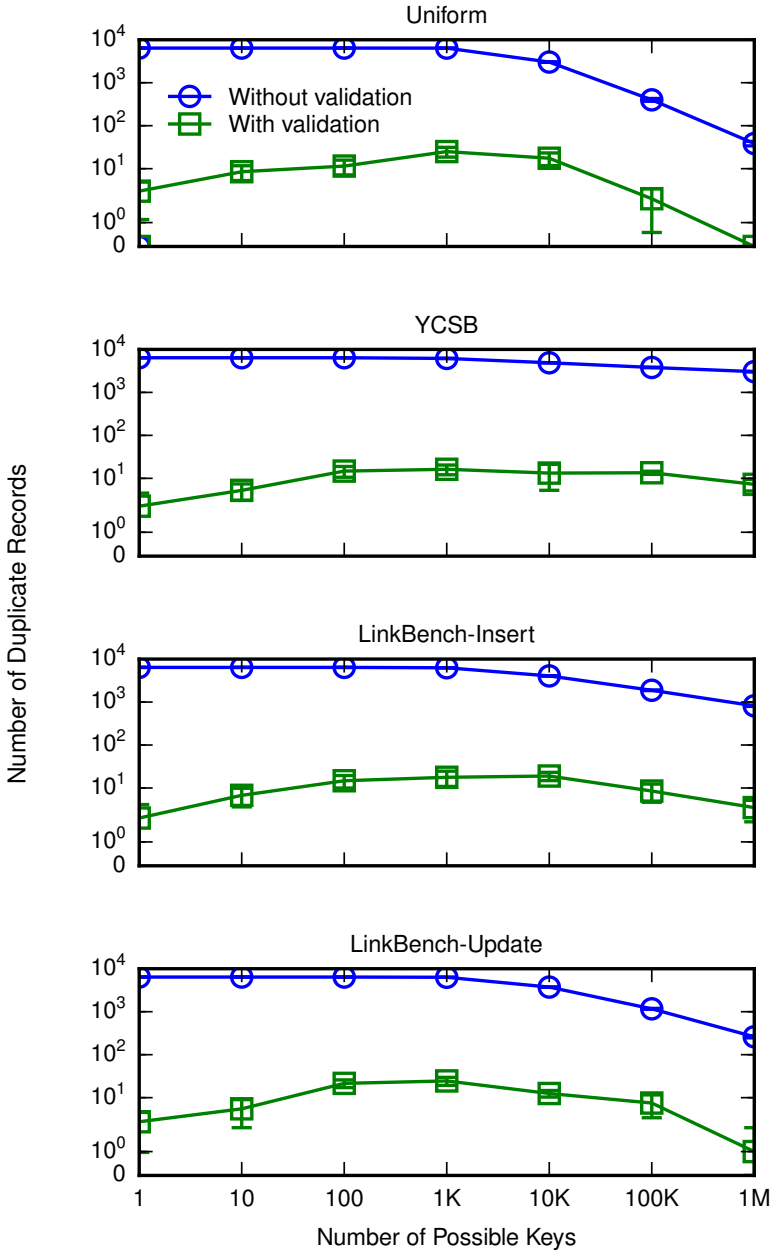


Figure 6.7: Uniqueness workload integrity violations.

in the database (Appendix 6.8.2). On deletion, any models with associations marked with `:dependent => destroy` (or `:dependent => delete`) will have any associated models destroyed (i.e., removed by instantiating in Rails and calling `destroy` on the model) or deleted (i.e., removed by simply calling the database's `DELETE` method).

This feral association validation runs within a transaction, but, again the exact isolation level of the transaction affects its correctness. For correct execution, the `SELECT` query must also attain a predicate lock on the specific value of the validated column for the duration of the transaction. Similar to the uniqueness validator, concurrent deletions and insertions are unsafe under Read Committed, Repeatable Read, and Snapshot Isolation. Thus, unless the database is configured for serializable isolation, inconsistency may result and the feral validation will fail to prevent data corruption.

Unlike uniqueness validations, there is no discussion of associations and concurrency anomalies in the Rails documentation. Moreover, in Rails 4.1, there is no way to natively declare a foreign key constraint;[12] it must be done via a third-party library such as `foreigner` [136] or `schema_plus` [7]. Only two applications (`canvaslms` and `diaspora`) used `foreigner`, and only one application (`juvia`) used `schema_plus`. One application (`jobsworth`) used a custom schema annotation and constraint generator.

Understanding association behavior. Given that entirely feral mechanisms can introduce broken associations, how many dangling records can be introduced? Once a record is deleted, any later validations will observe it via `SELECT` calls. However, in the worst case, the feral cascading deletion on the one side of a one-to-many relation can stall indefinitely, allowing an unlimited number of concurrent insertions to the many side of the relation. Thus, validations—at least theoretically—only reduce the worst-case number of dangling records that were inserted prior to deletion; any number of concurrent insertions may occur during validation, leading to unbounded numbers of dangling records.

Quantifying Association Anomalies

Given this potential for errors, we again set out to quantify integrity errors. We demonstrate that weak isolation can indeed lead to data integrity errors in Rails' implementation of associations.

We performed another set of experiments to test association validation behavior under concurrent insertions and deletions. Using the same Unicorn and PostgreSQL deployment as above, we configured another application to test whether or not Rails validations would correctly enforce association-based integrity constraints. We consider an application with

[12] Rails 4.2 added support for foreign keys via migration annotation (separate from models; similarly to adding a unique index) in December 2014.

two models: Users and Departments. We configure a one-to-many relationship: each user `belongs_to` a department, and each department `has_many` user (Appendix 6.8.6).

As a basic stress test, we initialize the database by creating 100 departments with no users. Subsequently, for each department in the database, we issue a single request to delete the department along with 64 concurrent requests to insert users in that department. To correctly preserve the one-to-many relationship, the database should either reject the deletion operation or perform a cascading deletion of the department and any users (while rejecting any future user creation requests for that department). We can quantify the degree of inconsistency by counting the number of users left in the database who have no corresponding department (Appendix 6.8.7).

With associations declared in Rails, the Rails process performing the deletion will attempt a cascading delete of users upon department deletion. However, this cascade is performed, again, ferally—at the application level. Thus, under non-serializable isolation, any user creation events that are processed while the search for Users to delete is underway will result in Users without departments.

Figure 6.8 shows the number of "orphaned" Users (i.e., Users without a matching Department) as a function of Rails worker processes. With no constraints declared to Rails or to the database, all User creations succeed, resulting in 6400 dangling Users. With constraints declared in Rails (via a mix of validation and association), the degree of inconsistency depends on the degree of parallelism. Under the worst case, with 64 concurrent processes, the validations are almost worthless in preventing integrity errors. In contrast, when we declare a foreign key constraint within the database[13] and run the workload again, we observe no inconsistency.

The above stress test shows that inconsistency due to feral concurrency control occurs only during times of contention—parallel deletions and insertions. We subsequently varied the degree of contention within the workload. We configured the same application and performed a set of insertions and deletions, but spread across a greater number of keys and at random. A set of 64 processes concurrently each issued 100 User creation and Department deletion requests (at a ratio of 10 to 1) to a set of randomly-selected keys (again at a ratio of 10 Users to each Department). By varying the number of Users and Departments, we were able to control the amount of contention within the workload. Under this workload, inconsistency resulted only when a Department deletion proceeded concurrently with a User creation event and the feral cascading deletion "missed" the User creation (Appendix 6.8.8).

Figure 6.9 shows the results. As the number of Departments increases, we observe two trends. First, with only one Department, there is again less chance of inconsistency: all oper-

[13] In this case, we introduced the constraint via SQL using a direct connection to the database. This change was straightforward but—like the unique index addition—was not reflected in the base Active Record models.

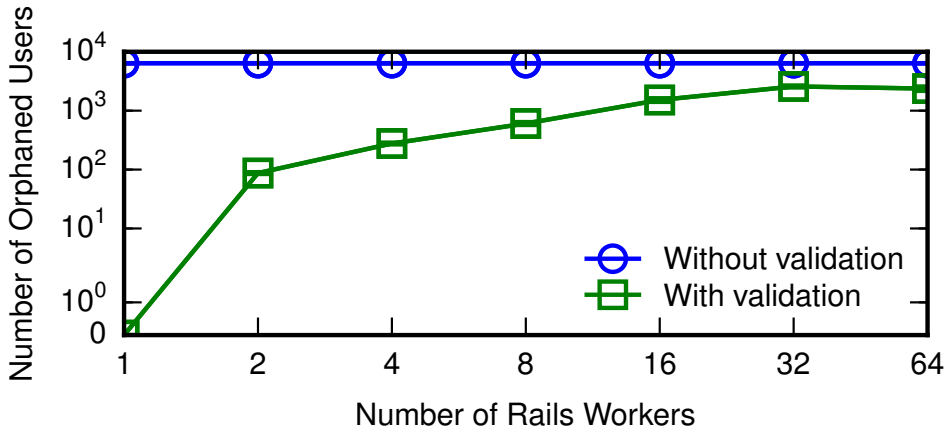


Figure 6.8: Foreign key stress association anomalies.

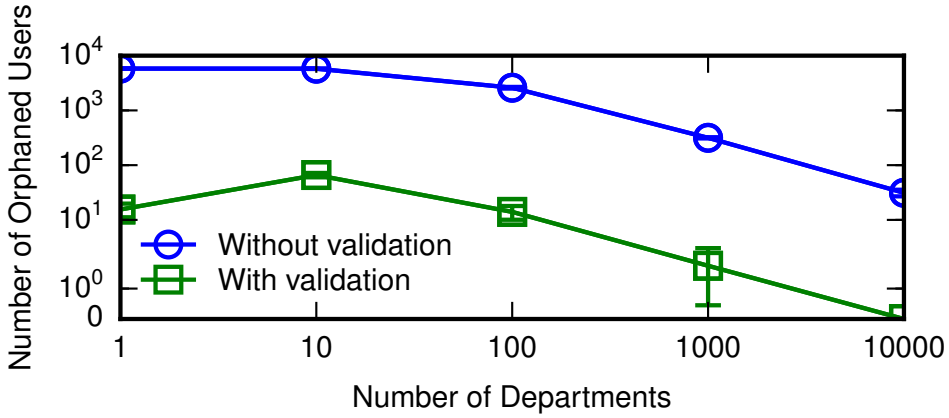


Figure 6.9: Foreign key workload association anomalies.

ations contend on the same data item, so the total number of inconsistent, orphaned users is limited by the number of potentially racing. However, as the number of Departments increases, the chance of concurrent deletions and insertions drops.

Takeaways and Discussion

The preceding experiments demonstrate that, indeed, Active Record is unsafe as deployed by default. Validations are susceptible to data corruption due to sensitivity to weak isolation anomalies.

This raises the question: why declare validations at all? As we observe, validations protect against *some* data corruption. First, they correctly guard against non-concurrency-

related anomalies such as data entry or input errors. For example, if a user attempts to reserve a username that was previously chosen, a validation would succeed. This basic functionality is reasonably left to the ORM engine for implementation. The failures we observe here are solely due to concurrent execution. Without concurrent execution, validations are correct. Second, validations *do* reduce the incidence of inconsistency. Empirically, even under worst-case workloads, these validations result in order-of-magnitude reductions in inconsistency. Under less pathological workloads, they may eliminate it with high probability. It is possible that, in fact, the degree of concurrency and data contention within Rails-backed applications simply does not lead to these concurrency races—that, in some sense, validations are "good enough" for many applications.

Nevertheless, in both cases, Rails's feral mechanisms are a poor substitute for their respective database counterparts—at least in terms of integrity. We re-examine the Rails community's reluctance to embrace these mechanisms in Section 6.7.

6.6 Other Frameworks

While our primary focus in this paper is Rails, we investigated support for uniqueness, foreign key, and custom validations in several other ORM frameworks. We find widespread support for validations and varying susceptibility to integrity errors.

Java Persistence API (JPA; version EE 7) [2] is a standard Java Object persistence interface and supports both uniqueness and primary key constraints in the database via specialized object annotations. Thus, when JPA is used to create a table, it will use the database to enforce these constraints. In 2009, JPA introduced support for UDF validations via a JavaBean interface [51]. Interestingly, both the original (and current) Bean validation specifications specifically address the use of uniqueness validations in their notes:

> "Question: should we add @Unique that would map to @Column(unique=true)? @Unique cannot be tested at the Java level reliably but could generate a database unique constraint generation. @Unique is not part of the [Bean Validation] spec today." [48]

An author of a portion of the code specification notes separately:

> "The reason @Unique is not part of the built-in constraints is the fact that accessing the [database] during a valiation [sic] is opening yourself up for potenital [sic] phantom reads. Think twice before you go for [an application-level] approach." [111]

By default, JPA Validations are run upon model save and run in a transaction at the default isolation level, and therefore, as the developers above hint, are susceptible to the same kinds of integrity violations we study here.

Hibernate (version 4.3.7) [135], a Java ORM based on JPA, does *not* automatically enforce declared foreign key relationships: if a foreign key constraint is declared; a corresponding column is added, but that column is *not* backed by a database foreign key. Instead, for both uniqueness and foreign key constraints, Hibernate relies on JPA schema annotations for correctness. Therefore, without appropriate schema annotations, Hibernate's basic associations may contain dangling references. Hibernate also has an extensive user-level validation framework implementing the JPA Validation Bean specification [112] and is sensitive to weak isolation anomalies, similar to Rails validations.

CakePHP (version 2.5.5) [3], a PHP-based web framework, supports uniqueness, foreign key, and UDF validations. CakePHP does *not* back any of its validation checking with a database transaction and relies on the user to correctly specify any corresponding foreign keys or uniqueness constraints within the database the schema. Thus, while users can declare each of these validations, there is no guarantee that they are actually enforced by the database. Thus, unless users are careful to specify constraints in both their schema and in their validations, validations may lead to integrity violations.

Laravel (version 4.2) [5], another PHP-based web framework, supports the same set of functionality as CakePHP, including application-level uniqueness, foreign key, and UDF validations in the application. Any database-backed constraints must be specified manually in the schema. Per one set of community documentation [104], "database-level validations can efficiently handle some things (such as uniqueness of a column in heavily-used tables) that can be difficult to implement otherwise" but "testing and maintenance is more difficult...[and] your validations would be database- and schema-specific, which makes migrations or switching to another database backend more difficult in the future." In contrast, model-level validations are "the recommended way to ensure that only valid data is saved into your database. They are database agnostic, cannot be bypassed by end users, and are convenient to test and maintain."

Django (version 1.7) [4], a popular Python-based framework, backs declared uniqueness and foreign key constraints with database-level constraints. It also supports custom validations, but these validations are not wrapped in a transaction [30]. Thus, Django also appears problematic, but only for custom validations.

Waterline (version 0.10) [47], the default ORM for Sails.js (a popular MVC framework for Node.js [46]), provides support for in-DB foreign key and uniqueness constraints (when

supported by the database) as well as custom validations (that are *not* supported via transactions; e.g., "TO-DO: This should all be wrapped in a transaction. That's coming next but for the meantime just hope we don't get in a nasty state where the operation fails!" [213]).

Summary. In all, we observe common cross-framework support for feral validation/invariants, with inconsistent use of mechanisms for enforcing them, ranging from the use of in-database constraints to transactions to no ostensible use of concurrency control in either application or database.

6.7 Implications for Databases

In light of this empirical evidence of the continued mismatch between ORM applications and databases, in this section, we reflect on the core database limitations for application writers today and suggest a set of directions for alleviating them.

6.7.1 Summary: Database Shortcomings Today

The use of feral invariants is not well-supported by today's databases. At a high level, today's databases effectively offer two primary options for ORM framework developers and users:

1. **Use ACID transactions.** Serializable transactions are sufficient to correctly enforce arbitrary application invariants, including transaction-backed feral validations. This is core to the transaction concept: isolation is a means towards preserving integrity (i.e., "I" provides "C"). Unfortunately, in practice, for application developers, transactions are problematic. Given serializability's performance and availability overheads [64], developers at scale have largely eschewed the use of serializable transactions (which are anyway not required for correct enforcement of approximately 87% of the invariants we encountered in the Rails corpus). Moreover, many databases offering "ACID" semantics do not provide serializability by default and often, even among industry-standard enterprise offerings, do not offer it as an option at all [32] (to say nothing of implementation difficulties, as in Footnote 8). Instead, developers using these systems today must manually reason about a host of highly technical, often obscure, and poorly understood weak isolation models expressed in terms of low-level read/write anomalies such as Write Skew and Lost Update [9, 20]. We have observed (e.g., Footnote 7) that ORM and expert application developers are familiar with the prevalence of weak isolation, which may also help explain the relative unpopularity of transactions within the web programming community.

2. **Custom, feral enforcement.** Building user-level concurrency control solutions on a per-framework or, worse, per-application basis is an expensive, error-prone, and difficult process that neglects decades of contributions from the database community. While this solution is sufficient to maintain correctness in the approximately 87% (invariant confluent) invariants in our corpus, the remainder can—in many modern ORM implementations—lead to data corruption on behalf of applications.

   However, and perhaps most importantly, this feral approach preserves a key tenet of the Rails philosophy: a recurring insistence on expressing domain logic in the application. This also enables the declaration of invariants that are not among the few natively supported by databases today (e.g., uniqueness constraints).

In summary, application writers today lack a solution that guarantees correctness while maintaining high performance *and* programmability. Serializability is too expensive for some applications, is not widely supported, and is not necessary for many application invariants. Feral concurrency control is often less expensive and is trivially portable but is not sufficient for many other application invariants. In neither case does the database respect and assist with application programmer desires for a clean, idiomatic means of expressing correctness criteria in domain logic. We believe there is an opportunity and pressing need to build systems that provide all three criteria: performance, correctness, and programmability.

6.7.2 Domesticating Feral Mechanisms

Constructively, to properly provide database support and thereby "domesticate" these feral mechanisms, we believe application users and framework authors need a new database interface that will enable them to:

1. *Express correctness criteria in the language of their domain model, with minimal friction, while permitting their automatic enforcement.* Per Section 6.4.1, a core factor behind the success of ORMs like Rails appears to be their promulgation of an idiomatic programming style that "seems right" for web programming. Rails' disregard for advanced database functionality is evidence of a continued impedance mismatch between application domain logic and current database primitives: databases today do not understand the semantics of feral validations.

   We believe any solution to domestication must respect ORM application patterns and programming style, including the ability to specify invariants in each framework's native language. Ideally, database systems could enforce applications' existing feral invariants without modification. This is already feasible for a subset of invariants—like

uniqueness and foreign key constraints—but not all. An ideal solution to domestication would provide universal support with no additional overhead for application writers. ORM authors may be able to meet the database halfway by pushing constraints into the database when possible.

2. *Only pay the price of coordination when necessary.* Per Section 6.4.3, many invariants can be safely executed without coordination, while others cannot. The many that do not need coordination should not be unnecessarily penalized.

   An ideal solution to domestication would enable applications to avoid coordination whenever possible, thus maximizing both performance and operation availability. The database should facilitate this avoidance, thus evading common complaints (especially within the Internet community) about serializable transactions.

3. *Easily deploy to multiple database backends.* ORM frameworks today are deployed across a range of database implementations, and, when deciding which database features to exercise, framework authors often choose the least common denominator for compatibility purposes.

   An ideal solution to domestication would preserve this compatibility, possibly by providing a "bolt on" compatibility layer between ORM systems and databases lacking advanced functionality (effectively, a "blessed" set of mechanisms beneath the application/ORM that correctly enforce feral mechanisms). This "bolt on" layer would act as a proxy for the ORM, implementing concurrency control on the ORM's behalf. We previously implemented such a compatibility layer for providing causal consistency atop eventually consistent data stores [39] and believe similar techniques are promising here.

Fulfilling these design requirements would enable high performance, correct execution, and programmability. However, doing so represents a considerable challenge.

Promise in the literature. The actual vehicle for implementing this interface is an open question, but the literature lends several clues. On the one hand, we do not believe the answer lies in exposing additional read/write isolation or consistency guarantees like Read Committed; these fail our requirement for an abstraction operating the level of domain logic and, as we have noted, are challenging for developers (and researchers) to reason about. On the other hand, more recent proposals for invariant-based concurrency control [34, 160] and a litany of work from prior decades on rule-based [228] and, broadly, semantics-based concurrency control [215] appear immediately applicable and worth (re-)considering. Recent advances in program analysis for extracting invariants [198] and subroutines from imperative code [77] may allow us to programmatically suggest new invariants, perform corre-

spondence checking for existing applications, and apply a range of automated optimizations to legacy code [78, 209]. Finally, clean-slate language design and program analysis obviate the need for explicit invariant declaration (thus alleviating concerns of specification completeness) [18, 19, 231]; while adoption within the ORM community is a challenge, we view this exploration as worthwhile.

Summary. In all, the wide gap between research and current practice is both a pressing concern and an exciting opportunity to revisit many decades of research on alternatives to serializability with an eye towards current operating conditions, application demands, and programmer practices. Our proposal here is demanding, but so are the framework and application writers our databases serve. Given the correct primitives, database systems may yet have a role to play in ensuring application integrity.

6.8 Detailed Validation Behavior, Experimental Workload

In this section, we provide more detail regarding how validations are executed in Rails as well as our workloads from Section 6.5.

6.8.1 Uniqueness Validation Behavior

When a controller attempts to `save` an ActiveRecord model instance i of type M, if M has a declared `:validates_uniqueness` annotation on attribute a, the following actions will be performed:

1. Assuming that instances of M are stored in database table T_M (with attribute a stored in column C_a), Active Record will perform the equivalent of

   $$\texttt{SELECT 1 FROM } T_M \texttt{ where } C_a \texttt{ = i.a LIMIT ONE;}$$

   (`SELECT COUNT(*)` would be sufficient here as well, but this is not how the query is actually implemented).

2. If this result set is empty, the validation succeeds.

3. If this result set is not empty, the validation fails. If the validation was called during `save`, it returns `false`. If the validation was called during `save!`, it raises an `ActiveRecord::RecordInvalid` exception.

This is a classic example of the phantom problem. Changing this `SELECT` call to `SELECT FOR UPDATE` would be sufficient. However, Rails is not implemented this way.

6.8.2 Association Validation Behavior

When a controller attempts to save an ActiveRecord model instance i of type M, if M has a declared :belongs_to annotation on attribute a pointing to attribute b of model N *and* M has a declared :validates_presence annotation on attribute a, the following actions will be performed:

1. Assuming that instances of N are stored in database table T_N (with attribute b stored in column C_b), Active Record will perform the equivalent of

$$\texttt{SELECT 1 FROM } T_N \texttt{ where } C_b \texttt{ = } i.a \texttt{ LIMIT ONE;}$$

2. If this result set is not empty, the validation succeeds.

3. If this result set is empty, the validation fails. If the validation was called during save, it returns false. If the validation was called during save!, it raises an ActiveRecord::RecordInvalid exception.

6.8.3 Uniqueness Validation Schema

In our uniqueness stress test, we declare two models, each containing two attributes: key, a string, and value, also a string. The generated schema for each of the models, which we call SimpleKeyValue and ValidatedKeyValue, is the same. The schema for SimpleKeyValue is as follows:

```
create_table "validated_key_values", force: true do |t|
  t.string "key"
  t.string "value"
  t.datetime "created_at"
  t.datetime "updated_at"
end
```

For the non-uniqueness-validated model, we simply require that the key and value fields are not null via a presence: true annotation. For the ferally validated model, we add an additional uniqueness: true validation to the key field in the Active Record model. The remainder of the application consists of a simple View and Controller logic to allow us to POST, GET, and DELETE each kind of model instance programatically via HTTP.

6.8.4 Uniqueness Stress Test

For the uniqueness stress test (Figure 6.6), we repeatedly attempt to create duplicate records. We issue a set of 64 concurrent requests to create instances with the key field set to an increasing sequence number (k, below) and repeat 100 times. At the end of the run, we count the number of duplicate records in the table:

```
for model m ∈ {SimpleKeyValue, ValidatedKeyValue} do
    for k ← 1 to 100 do
        parfor 1 to 64 do
            via HTTP: create new m with key=k
    dups ←execute(SELECT key, COUNT(key)-1 FROM T_M
                  GROUP BY key HAVING COUNT(key) > 1;)
```

Under correct validation, for each choice of k (i.e., for each key k), all but one of the model creation requests should fail.

6.8.5 Uniqueness Workload Test

For the uniqueness workload test (Figure 6.7), a set of 64 workers sequentially issues a set of 100 operations each. Each operation attempts to create a new model instance with the key field set to a random item generated according to the distributions described in Section 6.5:

```
for model m ∈ {SimpleKeyValue, ValidatedKeyValue} do
    parfor 1 to 64 do
        for 1 to 100 do
            k ← pick new key according to distribution
            via HTTP: create new m with key=k
    dups ←execute(SELECT key, COUNT(key)-1 FROM T_M
                  GROUP BY key HAVING COUNT(key) > 1;)
```

6.8.6 Association Validation Schema

In the association stress test, we declare two sets of models, each containing two models each: a `User` model and a `Departments` model. Each `User` has a(n implicit) `id` (as generated by Rails ActiveRecord) and an integer corresponding `department_i`. Each `Department` has

an `id`. Both models have a timestamp of the last updated and creation time, as is auto-generated by Rails. Aside from the table names, both schemas are equivalent. Below is the schema for the non-validated users and departments:

```
create_table "simple_users", force: true do |t|
    t.integer "simple_department_id"
    t.datetime "created_at"
    t.datetime "updated_at"
end

create_table "simple_departments", force: true do |t|
    t.datetime "created_at"
    t.datetime "updated_at"
end
```

The two pairs of models vary in their validations. One pair of models has no validations or associations. The other pair of models contain validations, including rules for cascading deletions. Specifically, we place an association `has_many :users, :dependent => :destroy` on the department, and, on the user, an association `belongs_to :department` and validation `validates :department, :presence => true` (note that we only delete from Departments in our workload, below). Thus, on deletion of a model of type `ValidatedDepartment`, ActiveRecord will attempt to call `destroy` on each matching `ValidatedUser`.

6.8.7 Association Stress Test

For the association stress test (Figure 6.8), we repeatedly attempt to create orphan users. We issue a set of 64 concurrent requests to create Users belonging to a particular department, while simultaneously deleting that department and repeat 100 times. At the end of the run, we count the number of users with a department that does not exist:

```
for model m ∈ {Simple, Validated} do
    for i ← 1 to 100 do
        via HTTP: create mDepartment with id=i
    for i ← 1 to 100 do
        parfor w ∈ 1 to 65 do
            if w = 1 then
                via HTTP: delete mDepartment with id=i
            else
                via HTTP: create new mUser department_id=i
    orphaned ←execute("SELECT m_department_id,
                       COUNT(*) FROM m_users AS U
                       LEFT OUTER JOIN m_departments AS D
                       ON U.m_department_id = D.id
                       WHERE D.id IS NULL
                       GROUP BY m_department_id
                       HAVING COUNT(*) > 0;")
```

6.8.8 Association Workload Test

For the association workload test (Figure 6.9), we begin by creating a variable number of departments (Figure 6.9 x-axis; D). We next have 64 concurrent clients simultaneously attempt to create users belonging to a random department and delete random departments (in a 10:1 ratio of creations to deletions, for 100 operations each). We end by counting the number of orphaned users, as above.

```
for model m ∈ {Simple, Validated} do
    for d ← 1 to D do
        via HTTP: create mDepartment with id=i
    parfor w ∈ 1 to 64 do
        d ← uniformRandomInt([1, D])
        if uniformRandomDouble([0, 1]) < 1/11 then
            via HTTP: delete mDepartment with id=d
        else
            via HTTP: create new mUser department_id=d
    orphaned ← as above, in stress test
```

6.9 Summary

In the first part of this chapter, we demonstrated that, in fact, many—but not all—common database invariants and integrity constraints are actually achievable without coordination. By applying these results to a range of actual transactional workloads, we demonstrated an opportunity to avoid coordination in many cases that traditional serializable mechanisms would otherwise coordinate. The order-of-magnitude performance improvements we demonstrated via coordination-avoiding concurrency control strategies provide compelling evidence that constraint-based coordination avoidance is a promising approach to meaningfully scaling future data management systems.

In the second part of this chapter, we examined the use of concurrency control mechanisms in a set of 67 open source Ruby on Rails applications and, to a less thorough extent, concurrency control support in a range of other web-oriented ORM frameworks. We found that, in contrast with traditional transaction processing, these applications overwhelmingly prefer to leverage application-level *feral* support for data integrity, typically in the form of declarative (sometimes user-defined) validation and association logic. Despite the popularity of these invariants, we find limited use of in-database support to correctly implement them, leading to a range of quantifiable inconsistencies for Rails' built-in uniqueness and association validations. While many validations are invariant confluent and therefore correct under concurrent execution given standard RDBMS weak isolation and concurrent update semantics, we see considerable opportunity to better support these users and their feral invariants in the future.

Chapter 7

Related Work

In this chapter, we provide a discussion of related work. We begin with a discussion of work related to the general themes in this thesis, then examine specific areas in depth.

Database system designers have long sought to manage the trade-off between consistency and coordination. As we have discussed, serializability and its many implementations (including lock-based, optimistic, and pre-scheduling mechanisms) [53, 55, 105, 123, 142, 215, 217, 221] are sufficient for maintaining application correctness. However, serializability is not always necessary: serializable databases do not allow certain executions that are correct according to application semantics. This has led to a large class of application-level—or semantic—concurrency control models and mechanisms that admit greater concurrency. There are several surveys on this topic, such as [120, 215], and, in our solutions, we integrate many concepts from this literature.

Commutativity. One of the most popular alternatives to serializability is to exploit *commutativity*: if transaction return values (e.g., of reads) and/or final database states are equivalent despite reordering, they can be executed simultaneously [80, 159, 226]. Commutativity is often sufficient for correctness but is not necessary. For example, if an analyst at a wholesaler creates a report on daily cash flows, any concurrent sale transactions will *not* commute with the report (the results will change depending on whether the sale completes before or after the analyst runs her queries). However, the report creation is invariant confluent with respect to, say, the invariant that every sale in the report references a customer from the customers table. [80, 155] provide additional examples of safe non-commutativity.

The CALM Theorem, Monotonicity, and Confluence. Hellerstein's CALM Theorem [21] shows that program outcomes are confluent, or deterministic, under coordination-free execution if and only if the program logic is monotone. CALM is a declarative result: it captures the class of computations that can be implemented deterministically without coor-

dination. CALM can also be used as a program analysis technique: if a particular program implementation uses only monotonic operations (where "program" could include a service and its client code), then that program will be deterministic when executed without coordination; otherwise, coordination should be injected to "protect" non-monotonic operations to ensure determinism. CALM program analysis is natural to apply in logic languages like Bloom [19] where monotonicity can be assessed from syntax. It can also be applied to dataflow systems like Storm [193] with the help of program annotations [18].

CALM's notion of confluence differs from invariant confluence in several ways. First, CALM assesses the confluence, or determinism, of program logic; invariant confluence assesses whether a set of safety properties holds during and following the execution of a set of transactions over replicated or multi-versioned data given a particular merge function. Invariant confluence admits non-deterministic outcomes as long as the outcomes satisfy the provided invariants. Second, CALM does not consider transactions, while invariant confluence analyzes transactions that individually ensure invariant-preserving updates. Third, invariant confluence considers replicated or multi-versioned state (via the use of the replica abstraction). As discussed in Section 2.3, invariant confluent does not distinguish between partially-replicated and fully-replicated systems; under invariant confluence, a transaction is presented with an entire logical snapshot (replica) of the database upon which it can operate. A partially replicated implementation of a set of invariant confluent operations may need to communicate with partitions responsible for items that were not explicitly mentioned in the transaction operations but that are related to invariants over data modified by the transaction. Again, and by Theorem 1, for invariant confluent semantics, this checking can be performed in parallel by concurrently committing transactions over their respective logical replicas. However, in contrast, CALM analysis is agnostic to replication, versioning, and partitioning, which, if desired, are implemented as part of program logic to be analyzed.

CALM and invariant confluence use different mathematical foundations. CALM is based on monotonicity analysis from logic programs. Invariant confluence generalizes classic partitioning arguments from distributed systems to the domain of user-supplied invariants, transactions, and merge functions. For associative, commutative, and idempotent merge functions, an invariant confluent execution effectively defines a join semi-lattice: invariants begin true in D_0 and remain true as the execution progresses. Monotone programs also compute over a join semi-lattice of relations and union. However, the analyses and proof techniques of the two concepts are quite different.

Further understanding the relationship between invariant confluence and CALM is an interesting area for exploration. For example, it is natural to ask if there is an extension of CALM analysis that can, like invariant confluence, incorporate invariants over possibly

non-deterministic outputs. A possible direction here is to view invariants as boolean-valued formulas whose results "start true" and monotonically remain true. In this direction, an invariant is a morphism mapping from potentially monotone relational inputs to a monotone boolean output lattice [81]. Additionally, as CALM is non-transactional and our formulation of invariant confluence is inherently transactional, it is interesting to consider what "transactional CALM" would mean. In our formulation of invariant confluence, transactions that violate invariants when committing to local replica state are aborted; it is unclear how to model abort logic in CALM analysis.

Convergent Data Types. On a related subject, Commutative Replicated Data Type (CRDT) objects [205] similarly ensure convergent outcomes that reflect all updates made to each object. This convergence is a useful *liveness* property [202] (e.g., a converged CRDT OR-Set reflects all concurrent additions and removals) but does not prevent users from observing inconsistent data [160], or *safety* (e.g., the CRDT OR-Set does not—by itself—enforce invariants, such as ensuring that no employee belongs to two departments), and are therefore not sufficient to guarantee correctness for all applications. Here, we use CRDTs to implement many of our merge functions, and we add safety to the intermediate states and final outcomes. Thus, each replica state is, in effect, a CRDT, and our goal is to determine which operations need coordination to ensure variants of safety properties are upheld.

Use of Invariants. A large number of database designs—including, in restricted forms, many commercial databases today—use various forms of application-supplied invariants, constraints, or other semantic descriptions of valid database states as a specification for application correctness (e.g., [10, 52, 92, 107, 115, 120, 126, 127, 145, 159–161, 194, 198]). We draw inspiration and, in particular, our use of invariants from this prior work. However, we are not aware of related work that discusses when coordination is strictly *required* to enforce a given set of invariants. That is, our formulation of coordination-free execution of transactions on separate replicas, which is key to capturing scalability, low latency, and availability properties, is not found in this related work; we, in effect, operate at the junction between this prior work on semantics-based concurrency control from databases and classic analyses from distributed computing [118].

To illustrate why replication is so important to our model, consider the work on relative serializability [10]. In this work, the authors generalize prior efforts, including [107, 115, 145, 170], and re-define conflicting actions within otherwise conflict serializable transaction execution in order to allow greater concurrency. That is, instead of defining conflict as "any two operations on the same item from different transactions, at least one of which is a write" as in conflict serializability, relative serializability allows users to define an abstract atomicity relation to determine conflicts—for example, two increment operations need not necessarily conflict, even if they both update the same counter. Thus, the goal of this work

is to preserve equivalence a serial schedule, defined according to the abstract atomicity relation, and there is still a total order on operations. As a result, in relative serializability and related models [194], the "union" (or combination) of two databases is undefined if two items have different versions (e.g., $\{a_1\} \cup \{a_2\}$), because such databases would correspond to two separate total orders. In contrast, in our invariant confluence analysis, we explicitly consider a partial order on operations, with divergent states reconciled with a merge operator; instead of reasoning about conflicts, we allow arbitrary divergent states that i) are guaranteed to satisfy a user-specified invariant over the data and ii) are reconciled using a user-specified merge function.

Because data is replicated in our model, it is natural to reason about a "merge" function. Insofar as servers must explicitly integrate updates from others in order to guarantee convergence (in contrast with conventional shared-memory systems, where hardware automatically chooses an ordering and conflict resolution policies for updates), merge allows users to specify their own conflict resolution. As we have discussed, the merge operator is itself drawn from the literature on optimistic replication [200] and is relatively popular today in stores including Dynamo [95] and its descendants as well as systems like Git. Thus, while the goals of work on semantics-based concurrency control (including relative serializability) are similar to ours (especially in terms of increasing concurrency), our use of merge leads to a substantially different system and execution model. In effect, we can think of invariant confluence as relative serializability with a special, system-induced compensating action ("merge") to deal with divergent paths in the semantic serializability serialization graph (RSG), if the graph were extended to account for replication.

Thus, our invariant confluence analysis here is inspired by prior work on semantics-based concurrency control and adapts the practice of using application (and database) criteria as the basis of concurrency control to the replicated (and non-serializable, multi-versioned) setting. Moreover, compared to this prior work, our practical focus here is oriented towards invariants found in SQL and in modern applications.

In contrast with many of the conditions above (esp. commutativity and monotonicity), we explicitly require more information from the application in the form of invariants (Kung and Papadimitriou [149] suggest this is information is *required* for general-purpose non-serializable yet safe execution.) In this work, we provide a necessary and sufficient condition for safe, coordination-free execution over replicated and multi-version data. When invariants are unavailable, many of these more conservative approaches may still be applicable. Our use of analysis-as-design-tool is inspired by this literature—in particular, [80].

Coordination costs. In this work, we determine when transactions can run entirely concurrently and without coordination. In contrast, a large number of alternative models (e.g., [14, 27, 115, 127, 145, 161, 169]) assume serializable or linearizable (and therefore

coordinated) updates to shared state. These assumptions are standard (but not universal [68]) in the concurrent programming literature [27, 202]. (Additionally, unlike much of this literature, we only consider a single set of invariants per database rather than per-operation invariants.) For example, transaction chopping [206] and later application-aware extensions [12, 52] decompose transactions into a set of smaller transactions, providing increased concurrency, but in turn require that individual transactions execute in a serializable (or strict serializable) manner. This reliance on coordinated updates is at odds with our goal of coordination-free execution. However, these alternative techniques are useful in reducing the duration and distribution of coordination once it is established that coordination is required.

Term rewriting. In term rewriting systems, invariant confluence guarantees that arbitrary rule application will not violate a given invariant [100], generalizing Church-Rosser confluence [144]. We adapt this concept and effectively treat transactions as rewrite rules, database states as constraint states, and the database merge operator as a special *join* operator (in the term-rewriting sense) defined for all states. Rewriting system concepts—including confluence [14]—have previously been integrated into active database systems [228] (e.g., in triggers, rule processing), but we are not familiar with a concept analogous to invariant confluence in the existing database literature.

Coordination-free algorithms and semantics. Our work is influenced by the distributed systems literature, where coordination-free execution across replicas of a given data item has been captured as "availability" [38, 118]. A large class of systems provides availability via "optimistic replication" (i.e., perform operations locally, then replicate) [200]. We—like others [68]—adopt the use of the merge operator to reconcile divergent database states [189] from this literature. Both traditional database systems [9] and more recent proposals [159, 160] allow the simultaneous use of "weak" and "strong" isolation; we seek to understand *when* strong mechanisms are needed rather than an optimal implementation of either. Unlike "tentative update" models [116], we do not require programmers to specify compensatory actions (beyond merge, which we expect to typically be generic and/or system-supplied) and do not reverse transaction commit decisions. Compensatory actions could be captured under invariant confluence as a specialized merge procedure.

The CAP Theorem [8, 118] recently popularized the tension between strong semantics and coordination and pertains to a specific model (linearizability). In a recent retrospective, Brewer discusses the role of CAP in reasoning about and "repairing" more general invariants, such as those we study here [64]. The relationship between serializability and coordination requirements has also been well documented in the database literature [92]. Our research here addresses when particular database-backed applications require coordination, providing a new property, invariant confluence, for doing so.

Summary. The invariant confluence property is a necessary and sufficient condition for safe, coordination-free execution. Sufficient conditions such as commutativity and monotonicity are useful in reducing coordination overheads but are not always necessary. Here, we explore the fundamental limits of coordination-free execution. To do so, we explicitly consider a model without synchronous communication. This is key to scalability: if, by default, operations must contact a centralized validation service, perform atomic updates to shared state, or otherwise communicate, then scalability will be compromised. Finally, we only consider a single set of invariants for the entire application, reducing programmer overhead without affecting our invariant confluence results.

Isolation and RAMP Transactions

Replicated databases offer a broad spectrum of isolation guarantees at varying costs to performance and availability [53]:

Serializability. At the strong end of the isolation spectrum is serializability, which provides transactions with the equivalent of a serial execution (and therefore also provides RA). A range of techniques can enforce serializability in distributed databases [11, 53], multi-version concurrency control (e.g. [190]), locking (e.g. [166]), and optimistic concurrency control [207]. These useful semantics come with costs in the form of decreased concurrency (e.g., contention and/or failed optimistic operations) and limited availability during partial failure [32, 92]. Many designs [90, 142] exploit cheap serializability within a single partition but face scalability challenges for distributed operations. Recent industrial efforts like F1 [207] and Spanner [85] have improved performance via aggressive hardware advances but, their reported throughput is still limited to 20 and 250 writes per item per second. Multi-partition, multi-datacenter, and, generally, distributed serializable transactions are expensive and, especially under adverse conditions, are likely to remain expensive [89, 141, 187].

Weak isolation. The remainder of the isolation spectrum is more varied. Most real-world databases offer (and often default to) non-serializable isolation models [32, 179]. These "weak isolation" levels allow greater concurrency and fewer system-induced aborts compared to serializable execution but provide weaker semantic guarantees. For example, the popular choice of Snapshot Isolation prevents Lost Update anomalies but not Write Skew anomalies [9]; by preventing Lost Update, concurrency control mechanisms providing Snapshot Isolation require coordination [32]. In recent years, many "NoSQL" designs have avoided cross-partition transactions entirely, effectively providing Read Uncommitted isolation in many industrial databases such PNUTS [83], Dynamo [95], TAO [65], Espresso [192], Rainbird [227], and BigTable [74]. These systems avoid penalties associ-

ated with stronger isolation but in turn sacrifice transactional guarantees (and therefore do not offer RA).

RAMP and related mechanisms. There are several algorithms that are closely related to our choice of RA and RAMP algorithm design.

COPS-GT's two-round read-only transaction protocol [167] is similar to `RAMP-F` reads—client read transactions identify causally inconsistent versions by timestamp and fetch them from servers. While COPS-GT provides causal consistency (requiring additional metadata), it does not support RA isolation for multi-item writes.

Eiger provides its write-only transactions [168] by electing a coordinator server for each write. As discussed in Section 5.5 (`E-PCI`), the number of "commit checks" performed during its read-only transactions is proportional to the number of concurrent writes. Using a coordinator violates partition independence but in turn provides causal consistency. This coordinator election is analogous to G-Store's dynamic key grouping [90] but with weaker isolation guarantees; each coordinator effectively contains a partitioned completed transaction list from [73]. Instead of relying on indirection, RAMP transaction clients autonomously assemble reads and only require constant factor (or, for `RAMP-F`, linear in transaction size) metadata size compared to Eiger's *PL-2L* (worst-case linear in database size).

We are not aware of another concurrency control mechanism for partitioned databases that ensures coordination-free execution, partition independence, and at least RA isolation.

Constraints

There is a large body of related work related to our investigation of specific database and ORM constraints that we consider in three categories: object relational mapping systems, the quantification of isolation behavior, and empirical open source software analysis.

ORMs. Database systems and application programming frameworks have a long history [56, 71, 154]. The "impedance mismatch" between object-oriented programming and the relational model is a perennial problem in data management systems. Ruby on Rails is no exception, and the concurrency control issues we study here are endemic to this mismatch—namely, the disuse of common concurrency control mechanisms like database-backed constraints. Bridging this gap remains an active area of research [173].

The latest wave of web programming frameworks has inspired diverse research spanning databases, verification, and security. StatusQuo uses program analysis and synthesis to transform imperative ORM code into SQL, leveraging the efficiency of database-backed web applications written in the Spring framework [77]. Rails has been the subject of study

in the verification of cross-site scripting attacks [76], errors in data modeling of associations [183], and arbitrary, user-specified (non-validation) invariants [60]. Rails-style ORM validations have been used to improve systems security via client-side execution [137, 209]. Our focus here is on the concurrency control requirements and usages of applications written in Rails.

Quantifying anomalies. A range of research similarly quantifies the effect of non-serializable isolation in a variety of ways.

Perhaps closest to our examination of ORM integrity violations is a study by Fekete et al., which quantitatively analyzed data inconsistencies arising from non-serializable schedules [108]. This study used a hand-crafted benchmark for analysis but is nevertheless one of the only studies of actual application inconsistencies. Here, we focus on open source applications from the Rails community.

A larger body of work examines isolation anomalies at the read-write interface (that is, measures deviations from properties such as serializability or linearizability but *not* the end effect of these deviations on actual application behavior). Wada et al. evaluated the staleness of Amazon's SimpleDB using end-user request tracing [225], while Bermbach and Tai evaluated Amazon S3 [50], each quantifying various forms of non-serializable behavior. Golab et al. provide algorithms for verifying the linearizability of and sequential consistency arbitrary data stores [121] and Zellag and Kemme provide algorithms for verifying their serializability [236] and other cycle-based isolation anomalies [235]. As we have discussed, Probabilistically Bounded Staleness provides time- and version-based staleness predictions for eventually consistent data stores [42]. Our focus here is on anomalies as observed by application logic rather than read-write anomalies observed under weak isolation.

Empirical software analysis. Empirical software analysis of open source software is a topic of active interest in the software engineering research community [212]. In the parlance of that community, in this work, we perform a mixed-methods analysis, combining quantitative survey techniques with a confirmatory case study of Rails's susceptibility to validation errors [103]. In our survey, we attempt to minimize sampling bias towards validation-heavy projects by focusing our attention on popular projects, as measured by GitHub stars. Our use of quantitative data followed by supporting qualitative data from documentation and issue tracking—as well as the chronology of methodologies we employed to attain the results presented here—can be considered an instance of the sequential exploration strategy [87]. We specifically use these techniques in service of better understanding use of database concurrency control.

Chapter 8

Conclusions

In this chapter, we conclude this dissertation by reflecting on general design patterns for systems builders, limitations of our approach, and opportunities for future work in coordination avoidance. We conclude with a final discussion on the results contained herein.

8.1 Design Patterns for Coordination Avoidance

During the development of the algorithms and systems described in this dissertation, we encountered a set of recurring patterns that assisted in our design process, both in applying invariant confluence and also deriving coordination-free implementations. Here, we outline four in the hope that they act as helpful rules of thumb and guidelines for future system architects:

Separate progress from visibility. In a coordination-free system, different operations must be able to proceed independently. An inherent side-effect of this behavior is that the progress of one operation may not be visible to other concurrent operations. Thus, a coordination-free system guarantees progress of operations without guaranteeing visibility of their side effects. The side effects can eventually become visible, but coordination-free applications cannot depend on observing them. Put another way, a coordination-free system does not arbitrate whether independent operations should proceed or not—it simply arbitrates when their effects become visible.

Ensure composability of operations. The corollary to the above observation is that a coordination-free application must ensure that its operations are composable. This is the essence of the invariant confluence property, but it bears repeating: in a coordination-free execution, operations will run concurrently, so whether or not their side effects can be rec-

onciled is key to determining whether the execution is safe. We have captured this reconciliation process via an explicit merge operator. In our analyses, merge is simple: typically just set union or a natural extension of abstract data types. However, more complicated merge operations are possible (e.g., to capture behavior such as compensating actions). The guiding question here is: if transactions produce multiple effects independently, do the effects make sense when combined?

Control visibility via multi-versioning. In our coordination-free implementations and system designs, we have relied heavily on the use of multi-versioning. In some cases, this appears a necessity: to avoid revealing intermediate data while allowing updates to existing data, multiple versions are required (e.g., as in RAMP). The subtlety in each implementation is due to two related factors. First, when are new writes revealed? For example, Read Atomic isolation and causal consistency each provide a different answer that affects the algorithm design. Second, how is dependency information encoded in the database? Again, as an example, RAMP and causally consistent algorithms take various different approaches. For RAMP, we have provided three options that offer a trade-off between efficiency of reads and compactness of metadata. For causal consistency, we have another set of options, from vector clocks to dependency trees (Section 5.6.3). Nevertheless, the core concepts are the same, and they are repeated in many of our algorithmic and systems contributions, from Read Committed isolation to our invariant confluent implementation of TPC-C transactions.

Limit the scope of coordination (when required). Our primary focus in this thesis is determining when coordination-free implementations of common semantics are achievable. As we have seen, sometimes coordination is required. As a simple design axiom that is intuitive but nevertheless useful, when coordination is unavoidable, it is desirable to limit its scope, both in time and space. That is, first, it is best to coordinate over as few operations as possible (e.g., in TPC-C, instead of coordinating for all operations, we only coordinate for those that require it, allowing the rest to execute without coordination). Second, coordination within a single node is much cheaper than coordinating across nodes, so minimizing the distribution of coordinating processes is also beneficial. We expand on this final theme in the remainder of this chapter.

8.2 Limitations

While we believe that the techniques in this thesis are useful, they have several limitations; in this section, we outline four of them. We discuss avenues for addressing them in Section 8.3.

Advance knowledge. In our invariant confluence analysis, we have effectively assumed that all program text and constraints are known in advance. Of course, by restricting program operations and constraints to only those constructs that we have shown to be invariant confluent, we can construct safe "languages" from which one can dynamically specify programs. However, without further consideration of the admissible logics and complexity classes contained within the space of invariant confluent semantics, it is difficult to precisely characterize the utility of this alternative.

Complete specifications. A related concern is that invariant confluence requires a complete specification of correctness for a given task or program. If a particular invariant does not appear in a specification, invariant confluence assumes that the invariant does not matter for correctness. This leaves a considerable burden on the programmer. We have attempted to mitigate this burden by considering existing, widely-used semantics in this dissertation.

Manual process. Our application of invariant confluence analysis and development of coordination-free algorithms is primarily manual, relying on proofs, rudimentary static analysis, and a set of design principles as outlined above. We have not found this process to be exceedingly onerous in practice, but, nevertheless, for others to rigorously apply these ideas may require considerable familiarity with the details in this work.

Replicated and partitioned model.

Mixed execution. Our primary goal has been to determine whether coordination-free execution is possible and, when it is, how to realize coordination-free implementations of this task. This leaves a question of how to combine coordination-free and coordinated semantics within a given application. A trivial answer is to always coordinate in the event that invariant confluence does not hold. However, in our experience, this empirically wastes considerable opportunity for more efficient execution. Rather, fully realizing coordination avoidance requires embracing mixed-mode execution within system runtimes.

8.3 Future Work

We see several promising directions for future work on coordination avoidance, which we discuss here. A subset of the directions here address the limitations of our existing work that we have discussed in Section 8.2, while others represent new questions arising from this work.

8.3.1 Automating Coordination Avoidance

In this dissertation, we have developed a foundation for determining when coordination-free execution is possible; we have applied the invariant confluence principle to a range of semantics found in applications today. However, this process is primarily manual and requires human intervention at several stages of the process, from proving or disproving the invariant confluence of a set of semantics to deriving a coordination-free algorithm for implementing an invariant confluent set of semantics. This raises a natural question: which aspects of this process can be automated? We see several avenues for progress:

Automatically proving invariant confluence. Given a set of operations, an invariant, and a merge function, it would be desirable to automatically prove whether invariant confluence holds for the combination. For arbitrary logic, this is undecidable via trivial application of Rice's Theorem. However, given the brevity of most of our proofs in this work, we believe that, for practical programs, this is feasible. The key to achieving this possiblity is to determine a sufficiently restricted set (or language) of operations and constraints that can be efficiently checked.

Synthesizing coordination-free algorithms. As we have discussed, many of our algorithms fit a common pattern—to what extent can we automatically synthesize coordination-free algorithms from their specification (similar to synthesis of concurrent data structures [131])? Could we build a synthesizer that could automatically output the RAMP protocols described in this dissertation, using only the RA isolation specification? Could we do the same for causal consistency? We believe the answer is yes to all: the base implementation of each is relatively similar (a multi-versioned database), and each implementation is effectively parametrized by a delivery policy and metadata. Thus, a synthesizer could efficiently search the space of delivery policies and metadata to find an appropriate match for a given set of semantics. For the semantics we have described here, the search space is relatively constrained.

Mining constraints. The task of specifying a complete declarative set of invariants for arbitrary applications is a considerable burden on developers. Is it possible to automate this process? One promising possibility is to use recent program analysis techniques such as the Homeostasis Protocol [198] to first generate a conservative set of invariants from application code (e.g., using serializability as a correctness specification). Subsequently, given this conservative specification, we can engage the programmer to increase its precision. Specifically, given a cost model for each invariant (e.g., determined by applying invariant confluence analysis and examining the distribution of data and which invariants require single-node versus multi-partition coordination to enforce), we can present the user with a list of "expensive" invariant-operation pairs ranked by cost. For each operation, we can

present a set of cheaper alternatives to choose from that relax the specification. For example, we might explain that, while assigning IDs sequentially is expensive, using a nonce unique ID generator is considerably less expensive. Thus, even if program text indicates the need for coordination, involving the programmer in the program rewriting phase may allow more flexibility. By using cost models to guide the optimization, we can address the most expensive components of the application first.

Bespoke coordination plans. Given a set of operations and constraints, how should a system actually proceed to enforce them? Our current approach is relatively simple: each constraint and operation pair has an associated implementation (e.g., foreign key insertions use RAMP transactions). However, for more complex constraints and operations, this approach can quickly become onerous and even untenable. We see an opportunity for both more principled study of distributed invariant enforcement (in the spirit of active database systems, especially in a multi-partition, geo-replicated environment) as well as, in effect, "query planning" for coordination. If coordination is required, what mechanisms should be chosen? On what partitions should they be deployed? As an example, lock-based coordination is typically partitioned by item (e.g., the lock for item x belongs on the server for x). However, if we consider higher-order data types like publish-subscribe queues, we have a number of options, including coordinating at the publisher(s), coordinating at the subscriber(s), or neither. The Blazes system [18] chooses between totally ordered, partially ordered, and unordered delivery in a stream processing engine in order to guarantee output determinism; similar choices exist for invariant enforcement. Given that there are rarely unilateral "best" strategies in concurrency control, runtime adaptivity may confer serious advantages.

Maintaining constraints and operations. Applications change over time: new constraints and operations will be added to applications. How should we incrementally maintain coordination plans, and how can we incrementally check (and maintain) invariant confluence? New constraints must be compatible with existing constraints, otherwise an unsatisfiable set of constraints will result in operation unavailability and database "inconsistency." One simple strategy is to treat code modifications as a coordinated operation, much as schema changes are rolled out across clusters today.

8.3.2 Comprehending Weak Isolation

As discussed in Section 4.1, few relational database engines today actually provide serializability by default or even as an option at all. This was a somewhat surprising result to us, as much of the power and beauty of the transaction concept is predicated on serializable isolation. As we have in Section 6.5, these weak isolation guarantees can corrupt applica-

tion integrity if not correctly applied. This raises a set troubling questions for the database community. Primarily, how is it that weak isolation is so prevalent and yet transaction processing is successful in practice?

One possibility for this behavior is that, in practice, there is little concurrency. For many high value applications such as point of sale systems, volumes on the order of hundreds of transaction per second are impressive yet are easily handled by a single server runtime. Without high concurrency, there are few conflicts. However, if this is the case, increasing query and data volumes may lead to an increased incidence of consistency errors due to weak isolation.

Another possibility is that programmers are simply compensating for weak application in the application. This is possible via use of language constructs like `SELECT FOR UPDATE`, which acquire exclusive access to a record, compensating actions, and a range of alternative concurrency control strategies. Determining whether this is the case will require additional investigation of programmer behavior.

In either of these cases, we see a number of opportunities for improving programmer experiences, including:

Automated isolation analyses. In line with the automated invariant confluence analyses above, one could check a given application to determine its susceptibility to inconsistencies arising from weak isolation anomalies. This has been an active area of research for weak memory models in multi-processor systems, and we see an analogous opportunity here.

Debugging weak isolation. When inconsistencies in data occur, from where did they come? Leveraging provenance-style analyses to determine the origins of inconsistency may assist programmers in understanding why errors occurred and how to avoid them in the future.

Automated isolation repair. Given an analysis as above, can we synthesize the use of constructs like `SELECT FOR UPDATE` to correct for anomalies? Once an inconsistency occurs, can a system automatically generate a compensating action that will repair it without compromising correctness?

8.3.3 Emerging Application Patterns

Given the ascendancy of open source, there is unprecedented opportunity to empirically and quantitatively study how our systems are and are not serving the needs of application programmers. Lightweight program analysis has never been easier, and the corpus of readily-accessible code—especially in an academic context—has never been larger.

The applications we study here are undoubtedly dwarfed by many other commercial and

enterprise-grade codebases in terms of size, quality, and complexity. However, compared to alternatives such as TPC-C, which today is almost 23 years old and is still the preferred standard for transaction processing evaluation, open source corpuses are arguably better proxies for modern applications. Recent efforts like the OLTPBenchmark suite [98] are promising but are nevertheless (and perhaps necessarily) not a substitute for real applications. The opportunity to perform both quantitative surveys across a large set of applications as well as longitudinal studies over the history of each application repository (and the behavior of a given programmer over time and across repositories) is particularly compelling. While these studies are inherently imprecise (due to limitations of the corpuses), the resulting quantitative trends are invaluable.

Thus, in this era of "Big Data" analytics, we see great promise in turning these analyses inwards, towards an empirical understanding of the usage of data management systems today, in service of better problem selection and a more quantitatively informed community dialogue. Our previous discussion in Section 6.7 hints at what is possible when we re-evaluate abstractions such as the transaction concept in light of developer trends.

8.3.4 Statistical Coordination Avoidance

In this work, we have largely concerned ourselves with the problem of deterministically maintaining safety guarantees. These guarantees are true guarantees in that they will hold in all scenarios: this is useful for application programmers as they do not have to reason about exceptional behavior under which safety guarantees do not hold. However, many emerging analytics tasks may not require such rigid guarantees. Instead, we might consider more relaxed safety guarantees as in PBS, wherein safety is guaranteed only in expectation or in a probabilistic sense. Numerical consistency guarantees and their enforcement have a long history in the literature [148, 185, 232, 237] but have seen limited adoption for want of a practical use case; we view these statistical analytics tasks as a new killer application.

Adapting the ideas we have discussed here to this statistical context provides a number of opportunities. A modified invariant confluence analysis might account for numerical robustness by incorporating bounded drift between merge invocations or incorporating network delay in analysis. In turn, coordination avoiding algorithms and variants of classic techniques from as escrow transactions [186] and approximate replication [185] for rebalancing could be used to increase the efficiency of statistical analysis tasks.

As an example, in the Alternating Direction Method of Multipliers (ADMM) [61], a number of processes coordinate to perform a distributed convex optimization routine. A core component of ADMM is a mathematical "consensus term" that prohibits individual processes from diverging from the global solution. We can treat this term as, in effect, a

quadratic penalty that is analogous to the numerical inequalities enforced by the escrow transaction method. In ADMM, all processes typically communicate to update the consensus term. However, in the escrow transaction method, processes can communicate pair-wise to rebalance slack in the allocation of divergence to individual processes. Processes might use a heuristic to borrow slack from the least-loaded process, thus reducing overall communication. Could escrow's pair-wise rebalancing be employed in ADMM in order to reduce communication costs without affecting convergence? We believe so.

Our initial experiences in this space have been promising. First, the Velox [86] system provides scalable model predictions as a service by treating models as statistically robust materialized views. In contrast with traditional materialized views, Velox defers the maintenance of models as new training data arrives and instead prioritizes model maintenance according to robustness. Second, our experiences integrating asynchronous model training into distributed dataflow engines [122] indicates that the degree of coordination required for efficient convergence of convex optimization tasks is closely linked to model skew. Both of these projects concretely highlight the potential of statistically-motivated coordination avoidance.

In general, we see great promise in systematically exploiting numerical robustness of statistical analytics routines. If a procedure is robust to numerical inaccuracy, an execution framework can systematically introduce inaccuracy to improve execution efficiency: for example, by introducing asynchrony into processing, by batching messages, or by operating over stale data. This is not a new idea by itself, but we see a wealth of unexplored connections in the domain of data serving and transaction processing that, in our initial investigations, have borne considerable fruit.

8.4 Closing Thoughts

ACID transactions and associated strong isolation and consistency levels dominated the field of database concurrency control for decades, due in large part to their ease of use and ability to automatically guarantee application correctness criteria. However, these powerful abstractions come with a hefty cost: concurrent transactions must coordinate in order to prevent read/write conflicts that could compromise equivalence to a serial execution. At large scale and, increasingly, in geo-replicated system deployments, the coordination costs necessarily associated with these implementations produce significant overheads in the form of penalties to throughput, latency, and availability. Today, these overheads necessitate a re-evaluation of concurrency control best practices.

In this dissertation, we developed a formal framework, called invariant confluence, in

which application invariants are used as a basis for determining if and when coordination is strictly necessary to maintain correctness. With this framework, we demonstrated that, in fact, many—but not all—common database semantics and integrity constraints are actually achievable without coordination. By applying these results to a range of actual workloads, we demonstrated multiple opportunities to avoid coordination in many cases that traditional serializable mechanisms would otherwise coordinate. The order-of-magnitude performance improvements we demonstrated via novel coordination-avoiding concurrency control implementations provide compelling evidence that coordination avoidance is a promising approach to meaningfully scaling future data management systems.

As a final note, today, the database and distributed computing communities are somewhat separate. While these communities have considerable shared interests in replicated data, their terminology and transfer of ideas are limited beyond the basics. This is unfortunate: the distributed computing literature has much to offer students of distributed databases. As we have highlighted here, the distributed computing community's emphasis on formal specification enables very useful analyses, without which we might spend years looking for algorithms that do not exist. Moreover, the distributed computing community's emphasis on modular abstractions, including atomic commitment, consensus objects, and broadcast primitives will be familiar to systems builders. Conversely, while the distributed computing readership of this thesis may be more limited, distributed databases also have much to offer scholars of distributed computing, including but not limited to new transaction models and safety properties. Perhaps most unfortunate is the reality that an increasing number of systems designers and practitioners today are exposed to the complexities of distribution, and there is little principled *and* practical guidance as to when to employ each of these mechanisms. Accordingly, this thesis is an attempt to merge these worlds and use principled analyses of network behavior to improve the practical development of distributed database systems. By making judicious use of coordination, we can build systems that guarantee application safety while maximizing their scalability.

Bibliography

[1] How a quiet developer built Goodreads.com into book community of 2.6+ million members – with Otis Chandler, November 2009. http://mixergy.com/interviews/goodreads-otis-chandler/.

[2] Java EE 7 API: Package javax.persistence, 2013. http://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html.

[3] CakePHP, 2014. http://cakephp.org/ and http://book.cakephp.org/2.0/en/index.html.

[4] Django: The Web framework for perfectionists with deadlines, 2014. https://www.djangoproject.com/ and https://github.com/django/django.

[5] Laravel: The PHP Framework for Web Artisans, 2014. http://laravel.com/ and https://github.com/laravel/laravel.

[6] RailsGuide: Active Record Validations, 2014. http://guides.rubyonrails.org/active_record_validations.html.

[7] SchemaPlus, 2015. https://github.com/SchemaPlus/schema_plus.

[8] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[9] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[10] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy. Relative serializability (extended abstract): An approach for relaxing the atomicity of transactions. In *PODS*, 1994.

[11] D. Agrawal and V. Krishnaswamy. Using multiversion data for non-interfering execution of write-only transactions. In *SIGMOD*, 1991.

[12] Divyakant Agrawal et al. Consistency and orderability: semantics-based correctness criteria for databases. *ACM TODS*, 18(3):460–486, September 1993.

[13] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and P.W. Hutto. Causal memory: Definitions, implementation and programming. *Dist. Comp.*, 9(1), 1995.

[14] Alexander Aiken, Jennifer Widom, and Joseph M Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *SIGMOD*, 1992.

[15] Ross Allen. Airbnb Engineering Blog: "Upgrading Airbnb from Rails 2.3 to Rails 3.0", October 2012. http://nerds.airbnb.com/upgrading-airbnb-from-rails-23-to-rails-30/.

[16] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web services*. Springer, 2004.

[17] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[18] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis for distributed programs. In *ICDE*, 2014.

[19] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, 2011.

[20] Peter Alvaro et al. Consistency without borders. In *ACM SoCC*, 2013.

[21] Tom J. Ameloot, Frank Neven, and Jan Van Den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15:1–15:38, May 2013.

[22] ISO/IEC 9075-2:2011 *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*, year=2011.

[23] Austin Appleby. Murmurhash 2.0, 2008. http://murmurhash.googlepages.com/.

[24] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[25] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the Facebook social graph. In *SIGMOD*, 2013.

[26] Rony Attar, Philip A. Bernstein, and Nathan Goodman. Site initialization, recovery, and backup in a distributed database system. *IEEE TSE*, 10(6):645–650, November 1984.

[27] Hagit Attiya, Rachid Guerraoui, Danny Hendler, et al. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, 2011.

[28] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[29] AWS. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://tinyurl.com/6ab6el6, April 2011.

[30] James Aylett. django-database-constraints, 2013. https://github.com/jaylett/django-database-constraints.

[31] Shivnath Babu and Herodotos Herodotou. Massively parallel databases and mapreduce systems. *Foundations and Trends in Databases*, 5(1):1–104, 2013.

[32] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and limitations. In *VLDB*, 2014.

[33] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An empirical investigation of modern application integrity. In *SIGMOD*, 2015.

[34] Peter Bailis, Alan Fekete, Michael J. Franklin, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Coordination avoidance in database systems. In *VLDB*, 2015.

[35] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *ACM SoCC*, 2012.

[36] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, not CAP: Introducing Highly Available Transactions. In *HotOS*, 2013.

[37] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.

[38] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), 2013.

[39] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *SIGMOD*, 2013.

[40] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 12(7), July 2014. Also appears in *Communications of the ACM* 57(9):48-55, September 2014.

[41] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 12(7):20, 2014.

[42] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically Bounded Staleness for practical partial quorums. In *VLDB*, 2012.

[43] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. PBS at work: Advancing data management with consistency metrics. In *SIGMOD*, 2013. Demo.

[44] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Quantifying Eventual Consistency with PBS. *The VLDB Journal*, 23(2):279–302, 2014. "Best of VLDB 2012" Special Issue.

[45] J. Baker, C. Bond, J.C. Corbett, JJ Furman, A. Khorlin, J. Larson, J.M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[46] Balderdash. Sails.js: Realtime MVC Framework for Node.js, 2014. https://github.com/balderdashy/sails.

[47] Balderdash. Waterline: An adapter-based ORM for Node.js with support for mysql, mongo, postgres, redis, [sic] and more, 2014. https://github.com/balderdashy/waterline.

[48] Bean Validation Expert Group. Jsr-000303 bean validation 1.0 final release specification, 2009. http://download.oracle.com/otndocs/jcp/bean_validation-1.0-fr-oth-JSpec/.

[49] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[50] David Bermbach and Stefan Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *MW4SOC*, 2011.

[51] Emmanuel Bernard. Java Specification Request 349: Bean Validation 1.1, 2013. https://jcp.org/en/jsr/detail?id=349.

[52] Arthur J Bernstein and Philip M Lewis. Transaction decomposition using transaction semantics. *Distributed and Parallel Databases*, 4(1):25–47, 1996.

[53] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[54] Phil Bernstein and Sudiptdo Das. Rethinking eventual consistency. In *SIGMOD*, 2013.

[55] Philip A. Bernstein, David W. Shipman, and James B. Rothnie, Jr. Concurrency control in a system for distributed databases (SDD-1). *ACM TODS*, 5(1):18–51, March 1980.

[56] Phillip A Bernstein, Alon Y Halevy, and Rachel A Pottinger. A vision for management of complex models. *ACM Sigmod Record*, 29(4):55–63, 2000.

[57] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, June 2009.

[58] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

[59] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[60] Ivan Bocić and Tevfik Bultan. Inductive verification of data model invariants for web applications. In *ICSE*, 2014.

[61] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[62] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, 2008.

[63] Eric Brewer. Towards robust distributed systems. 2000. Keynote at PODC.

[64] Eric Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[65] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, et al. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[66] Jordan Brough. #645: Alternative to validates_uniqueness_of using db constraints, 2011. rails/rails at https://github.com/rails/rails/issues/645.

[67] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *PDP*, 2004.

[68] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[69] Phil Calcado. Building products at SoundCloud – Part I: Dealing with the monolith, June 2014. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith.

[70] Michael J Carey and David J DeWitt. Of objects and databases: A decade of turmoil. In *VLDB*, 1996.

[71] Michael J Carey et al. Shoring up persistent applications. In *SIGMOD*, 1994.

[72] Andrew Carter. Hulu Tech Blog: "At a glance: Hulu hits Rails Conf 2012", May 2012. http://tech.hulu.com/blog/2012/05/14/347/.

[73] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE TSE*, 11(2):205–212, 1985.

[74] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[75] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.

[76] Avik Chaudhuri and Jeffrey S Foster. Symbolic security analysis of Ruby-on-Rails web applications. In *CCS*, 2010.

[77] Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[78] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C Myers. Automatic partitioning of database applications. In *VLDB*, 2012.

[79] Rada Chirkova and Jun Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[80] Austin T. Clements et al. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP*, 2013.

[81] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. Logic and lattices for distributed programming. In *ACM SoCC*, 2012.

[82] Blaine Cook. Scaling Twitter, SDForum Silicon Valley Ruby Conference, 2007. http://www.slideshare.net/Blaine/scaling-twitter.

[83] B.F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[84] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM SoCC*, 2010.

[85] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[86] Dan Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. In *CIDR*, 2015.

[87] John W Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage, 2013.

[88] Kyle Crum. #3238: Activerecord::staleobjecterror in checkout, 2013. spree/spree at https://github.com/spree/spree/issues/3238.

[89] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, 2010.

[90] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *ACM SoCC*, 2010.

[91] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *ICDE*, 2004.

[92] S.B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM CSUR*, 17(3):341–370, 1985.

[93] Jeff Dean. Designs, lessons and advice from building large distributed systems. Keynote at LADIS 2009.

[94] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[95] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[96] Alan Demers et al. Epidemic algorithms for replicated database maintenance. In *PODC*, 1987.

[97] Peter Deutsch. The eight fallacies of distributed computing. http://tinyurl.com/c6vvtzg, 1994.

[98] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. In *VLDB*, 2014.

[99] Romain Dillet. Update: Amazon Web Services down in North Virginia — Reddit, Pinterest, Airbnb, Foursquare, Minecraft and others affected. TechCrunch http://tinyurl.com/9r43dwt, October 2012.

[100] GregoryJ. Duck, PeterJ. Stuckey, and Martin Sulzmann. Observable confluence for constraint handling rules. In *ICLP*, 2007.

[101] John Duff. How Shopify scales Rails, Big Ruby 2013, April 2013. http://www.slideshare.net/jduff/how-shopify-scales-rails-20443485.

[102] Edd Dumbill. O'Reilly: "Ruby on Rails: An interview with David Heinemeier Hansson", August 2005. http://www.oreillynet.com/pub/a/network/2005/08/30/ruby-rails-david-heinemeier-hansson.html.

[103] Steve Easterbrook et al. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[104] Max Ehsan. Input validation with Laravel, 2014. http://laravelbook.com/laravel-input-validation/.

[105] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[106] Jose Faleiro, Alexander Thomson, and Daniel J Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.

[107] Abdel Aziz Farrag and M Tamer Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM TODS*, 14(4):503–525, 1989.

[108] Alan Fekete, Shirley N Goldrei, and Jorge Pérez Asenjo. Quantifying isolation anomalies. In *VLDB*, 2009.

[109] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *PODC*, 1996.

[110] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, June 2005.

[111] Hardy Ferentschik. Accessing the Hibernate Session within a ConstraintValidator, May 2010. https://developer.jboss.org/wiki/AccessingtheHibernateSessionwithinaConstraintValidator.

[112] Hardy Ferentschik and Gunnar Morling. Hibernate validator JSR 349 reference implementation 5.1.3.final, 2014. https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html/.

[113] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[114] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[115] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS*, 8(2):186–213, June 1983.

[116] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD*, 1987.

[117] David Geer. Will software developers ride Ruby on Rails to success? *Computer*, 39(2):18–20, 2006.

[118] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[119] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, 2011.

[120] Parke Godfrey et al. *Logics for databases and information systems*, chapter Integrity constraints: Semantics and applications, pages 265–306. Springer, 1998.

[121] Wojciech Golab, Xiaozhou Li, and Mehul A Shah. Analyzing consistency properties for fun and profit. In *PODC*, 2011.

[122] Joseph E. Gonzalez, Peter Bailis, Michael J. Franklin, Joseph M. Hellerstein, Michael I. Jordan, and Ion Stoica. Asynchronous complex analytics in a distributed dataflow architecture.

[123] Jim Gray. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[124] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM TODS*, 31(1):133–160, March 2006.

[125] J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1976.

[126] Paul WPJ Grefen and Peter MG Apers. Integrity control in relational database systems–an overview. *Data & Knowledge Engineering*, 10(2):187–223, 1993.

[127] Ashish Gupta and Jennifer Widom. Local verification of global integrity constraints in distributed databases. In *SIGMOD*, 1993.

[128] James Hamilton. Stonebraker on CAP Theorem and Databases. http://tinyurl.com/d3gtfq9, April 2010.

[129] David Heinemeier Hansson. active_record/transactions.rb, 2004. rails/rails githash db045db at https://github.com/rails/rails/blob/db045dbb.

[130] David Heinemeier Hansson. Choose a single layer of cleverness, September 2005. http://david.heinemeierhansson.com/arc/2005_09.html.

[131] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[132] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, 2007.

[133] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, 2007.

[134] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. 2008.

[135] Hibernate Team and JBoss Visual Design Team. Hibernate reference documentation 4.3.7.final, 2014. http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/.

[136] Matthew Higgins. Foreigner. https://github.com/matthuhiggins/foreigner.

[137] Timothy Hinrichs et al. Caveat: Facilitating interactive and secure client-side validators for ruby on rails applications. In *SECURWARE*, 2013.

[138] Sean Hull. 20 obstacles to scalability. *Communications of the ACM*, 56(9):54–59, 2013.

[139] Nam Huyn. Maintaining global integrity constraints in distributed databases. *Constraints*, 2(3/4):377–399, January 1998.

[140] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Eliminating unscalable communication in transaction processing. *The VLDB Journal*, pages 1–23, 2013.

[141] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.

[142] Robert Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB*, 2008.

[143] Bettina Kemme. *Database replication for clusters of workstations*. PhD thesis, EPFL, 2000.

[144] Jan Willem Klop. *Term rewriting systems*. Stichting Mathematisch Centrum Amsterdam, 1990.

[145] Henry K Korth and Gregory Speegle. Formal model of correctness without serializabilty. In *SIGMOD*, 1988.

[146] Michael Koziarski. Warn users about the race condition in validates_uniqueness_of. [koz], 2007. rails/rails githash c01c28c at https://github.com/rails/rails/commit/c01c28c.

[147] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[148] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An adaptive quality of service aware middleware for replicated services. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1112–1125, 2003.

[149] Hsing-Tsung Kung and Christos H Papadimitriou. An optimality theory of concurrency control for databases. In *SIGMOD*, 1979.

[150] Craig Labovitz, Abha Ahuja, and Farnam Jahanian. Experimental study of internet stability and backbone failures. In *FTCS*, 1999.

[151] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4):360–391, November 1992.

[152] Hongli Lai. Document concurrency issues in validates_uniqueness_of., 2008. rails/rails githash adacd94 at https://github.com/rails/rails/commit/adacd94.

[153] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. In *LADIS*, pages 35–40, 2008. Project site: http://cassandra.apache.org (2012).

[154] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, 1991.

[155] Leslie Lamport. Towards a theory of correctness for multi-user database systems. Technical report, CCA, 1976. Described in [12, 202].

[156] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions Software Engineering*, 3(2):125–143, March 1977.

[157] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[158] Jon Leighton. Support for specifying transaction isolation level, 2012. rails/rails githash 392eeec at https://github.com/rails/rails/commit/392eeec.

[159] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[160] Cheng Li, Joao Leitao, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, et al. Automating the choice of consistency levels in replicated systems. In *USENIX ATC*, 2014.

[161] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, et al. Snapshot isolation and integrity constraints in replicated databases. *ACM TODS*, 34(2), July 2009.

[162] Greg Linden. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. 9 November 2006.

[163] Greg Linden. Make data useful. https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-29.ppt, 2006.

[164] R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, 1988.

[165] V Liu, D Halperin, A Krishnamurthy, and T Anderson. F10: Fault tolerant engineered networks. In *NSDI*, 2013.

[166] Francois Llirbat, Eric Simon, Dimitri Tombroff, et al. Using versions in update transactions: Application to integrity checking. In *VLDB*, 1997.

[167] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[168] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[169] Shiyong Lu, Arthur Bernstein, and Philip Lewis. Correct execution of transactions at different isolation levels. *IEEE TKDE*, 16(9), 2004.

[170] Nancy A Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions: In Concurrent and Distributed Systems*. Morgan Kaufmann Publishers Inc., 1993.

[171] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, CS Department, UT Austin, May 2011.

[172] Dahlia Malkhi, Michael Reiter, Avishai Wool, and Rebecca Wright. Probabilistic quorum systems. *Information and Communication*, 170:184–206, 2001.

[173] Ankit Malpani et al. Reverse engineering models from databases to bootstrap application development. In *ICDE*, 2010.

[174] Adam Marcus. The nosql ecosystem. In *HPTS*, 2011. http://www.slideshare.net/AdamMarcus/nosql-ecosystem.

[175] Athina Markopoulou et al. Characterization of failures in an operational IP backbone network. *IEEE/ACM TON*, 16(4), 2008.

[176] Declan McCullagh. How Pakistan knocked YouTube offline (and how to make sure it never happens again). CNET, http://tinyurl.com/c4pffd, February 2008.

[177] Sean McCullough. Groupon Engineering Blog: "Geekon: I-Tier", October 2013. https://engineering.groupon.com/2013/node-js/geekon-i-tier/.

[178] Robert McMillan. Research experiment disrupts internet, for some. Computerworld, http://tinyurl.com/23sqpek, August 2010.

[179] C Mohan. History repeats itself: Sensible and NonsenSQL aspects of the NoSQL hoopla. In *EDBT*, 2013.

[180] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[181] P. P. S. Narayan. Sherpa update. YDN Blog, http://tinyurl.com/c3ljuce, June 2010.

[182] Stephen Nguyen. A mind boggling 9 billion queries per second at facebook @rocksdb, December 2011. Tweet by @Stephenitis, re-tweeted by @RocksDB.

[183] Jaideep Nijjar and Tevfik Bultan. Bounded verification of ruby on rails data models. In *ACM ISSTA*, 2011.

[184] Charles Nutter. Q/a: What thread-safe Rails means, August 2008. http://blog.headius.com/2008/08/qa-what-thread-safe-rails-means.html.

[185] Christopher Olston. *Approximate Replication*. PhD thesis, Stanford University, 2003.

[186] Patrick E O'Neil. The escrow transactional method. *ACM TODS*, 11(4):405–430, 1986.

[187] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, 2012.

[188] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[189] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, 1997.

[190] Shirish Hemant Phatak and BR Badrinath. Multiversion reconciliation for mobile databases. In *ICDE*, 1999.

[191] Tom Preston-Werner. How we made GitHub fast, October 2009. https://github.com/blog/530-how-we-made-github-fast.

[192] Lin Qiao et al. On brewing fresh Espresso: LinkedIn's distributed data serving platform. In *SIGMOD*, 2013.

[193] Rajiv Ranjan. Streaming big data processing in datacenter clouds. *Cloud Computing, IEEE*, 1(1):78–83, 2014.

[194] Rajeev Rastogi, Sharad Mehrotra, Yuri Breitbart, Henry F. Korth, and Avi Silberschatz. On correctness of non-serializable executions. In *PODS*, 1993.

[195] Kun Ren, Alexander Thomson, and Daniel J. Abadi. Lightweight locking for main memory database systems. VLDB, 2013.

[196] John Rizzo. Twitch: The official blog "Technically Speaking – Group Chat and General Chat Engineering", April 2014. http://blog.twitch.tv/2014/04/technically-speaking-group-chat-and-general-chat-engineering/.

[197] Dave Roberts. #13234: Rails concurrency bug on save, 2013. rails/rails at https://github.com/rails/rails/issues/13234.

[198] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.

[199] Sam Ruby, Dave Thomas, David Heinemeier Hansson, et al. *Agile web development with Rails 4*. The Pragmatic Bookshelf, Dallas, Texas, 2013.

[200] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), March 2005.

[201] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS*, 2010.

[202] Fred B Schneider. *On concurrent programming*. Springer, 1997.

[203] E. Schurman and J. Brutlag. Performance related changes and their user impact. Presented at Velocity Web Performance and Operations Conference, June 2009.

[204] Laurie Segall. Internet routing glitch kicks millions offline. CNNMoney, http://tinyurl.com/cmqqac3, November 2011.

[205] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. INRIA TR 7506, 2011.

[206] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: algorithms and performance studies. *ACM TODS*, 20(3):325–363, September 1995.

[207] Jeff Shute et al. F1: A distributed SQL database that scales. In *VLDB*, 2013.

[208] Neeraj Singh. update_attributes and update_attributes! are now wrapped in a transaction, 2010. rails/rails githash f4fbc2c at https://github.com/rails/rails/commit/f4fbc2c.

[209] Nazari Skrupsky et al. Waves: Automatic synthesis of client-side validation code for web applications. In *IEEE CyberSecurity*, 2012.

[210] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400, 2011.

[211] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[212] Klaas-Jan Stol et al. The use of empirical methods in open source software research: Facts, trends and future directions. In *FLOSS*, 2009.

[213] Code Stoltman. initial stab at creating has_many relationships, 2013. balderdashy/waterline githash b05fb1c at https://github.com/balderdashy/waterline/commit/b05fb1c. As of November 2014, this code has been moved but is still non-transactional and the comment remains unchanged.

[214] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9, 1986.

[215] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer, 2011.

[216] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, et al. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[217] A. Thomson, T. Diamond, S.C. Weng, K. Ren, P. Shao, and D.J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[218] Tim O'Reilly. What is web 2.0: Design patterns and business models for the next generation of software. *Communications and Strategies*, 65(1):17–37, 2007.

[219] TPC Council. TPC Benchmark C revision 5.11, 2010.

[220] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM TODS*, 7(3):323–342, 1982.

[221] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[222] Daniel Turner, Kirill Levchenko, Jeffrey C Mogul, Stefan Savage, and Alex C Snoeren. On failure in managed enterprise networks. HP Labs HPL-2012-101, 2012.

[223] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. California fault lines: understanding the causes and impact of network failures. In *SIGCOMM*, 2011.

[224] Werner Vogels. Eventually consistent. *CACM*, 52(1):40–44, January 2009.

[225] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, 2011.

[226] William Weihl. *Specification and implementation of atomic data types*. PhD thesis, Massachusetts Institute of Technology, 1984.

[227] Kevin Weil. Rainbird: Real-time analytics at Twitter. Strata 2011 http://slidesha.re/hjMOui.

[228] Jennifer Widom and Stefano Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[229] Alex Williams. Techcrunch: Zendesk launches a help center that combines self-service with design themes reminiscent of Tumblr, August 2013. http://on.tcrn.ch/l/XMdz.

[230] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: avoiding long tails in the cloud. In *NSDI*, 2013.

[231] Fan Yang et al. Hilda: A high-level language for data-drivenweb applications. In *ICDE*, 2006.

[232] 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–âĂŞ282, 2002.

[233] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Middleware*, 2015.

[234] Stanley B. Zdonik. Object-oriented type evolution. In *DBPL*, pages 277–288, 1987.

[235] Kamal Zellag and Bettina Kemme. Real-time quantification and classification of consistency anomalies in multi-tier architectures. In *ICDE*, 2011.

[236] Kamal Zellag and Bettina Kemme. How consistent is your cloud application? In *ACM SoCC*, 2012.

[237] Chi Zhang and Zheng Zhang. Trading replication consistency for performance and availability: an adaptive approach. In *ICDCS*, 2003.

[238] Jingren Zhou et al. Lazy maintenance of materialized views. In *VLDB*, 2007.