The Dissertation Committee for Chunzhi Su
certifies that this is the approved version of the following dissertation:

# Bringing Modular Concurrency Control to the Next Level

Committee:

Lorenzo Alvisi, Supervisor

Emmett Witchel, Co-Supervisor

Christopher J. Rossbach

Johannes Gehrke

# Bringing Modular Concurrency Control to the Next Level

**by**

**Chunzhi Su**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2018

# Acknowledgments

The past six years of my graduate studies have been a unique adventure in my life: there were moments of accomplishments, and there were time of frustration; there was joy, and there was sadness. The best part is that, I was not alone: it is my great honor to have walked through this adventure with great companions.

I would like to express my deep thanks to my advisor, Lorenzo Alvisi, for his guidance throughout my graduate study in these years. He has taught me not only knowledge, but more importantly, how to think, how to conduct research, how to communicate with people, and how to be a good person. I am deeply influenced by his character of always pursuing excellence, his diligence, and his rigor in scientific research. To me, he is not only my mentor, but also my good friend.

I would also like to thank my other committee members: Emmett Witchel, Chris Rossbach, and Johannes Gehrke, for helping me improve my research work in this dissertation with their valuable feedbacks and suggestions.

I would like to thank all my fellow graduate students in the LASR group of UT Austin and in Cornell's Syslab. Especially, I would like to thank Chao Xie and Natacha Crooks. I worked closely with Chao for more than four

years in UT Austin, during which time we have established great collaboration and friendship. Chao has made remarkable contributions to the work in my dissertation, and he also lead the Callas project, from which Tebaldi descends. Being a senior student, Chao gave me good advice on my career growth, which really helped me a lot. Natacha has also made great contribution to the Tebaldi project with her talent and diligent work, and she also helped me improve my writing skills. I also want to thank my other co-workers and friends for their great help. They are: Prince Mahajan, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Cong Ding, and Cody Littley.

My research work benefits a lot from the CloudLab testbed, and I would like to thank their great staff—they have helped me schedule experiment resources and explore hardware and software problems. Thank you!

Finally, I would like to sincerely thank my parents, and my girlfriend, Jingjing, for their love, support, encouragement and accompanying. They are my superheroes! I am really sorry that I wasn't able to spend much time with them during these years, and I am thankful for their understanding.

# Bringing Modular Concurrency Control to the Next Level

Publication No. _____

Chunzhi Su, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Lorenzo Alvisi
Co-Supervisor: Emmett Witchel

Database users face a tension between ease-of-programming and high performance: ACID transactions can greatly simplify the programming effort of database applications by providing four useful properties—atomicity, consistency, isolation, and durability, but enforcing these properties can degrade performance.

This dissertation eases this tension by improving the performance of ACID transactions for scenarios where data contention is the bottleneck. The approach that we take is *federating* concurrency control (CC) mechanisms. It is based on the observation that any single CC mechanism is bound to make trade-offs that cause it to perform well in some cases but poorly in others. A federation opens the opportunity of applying each mechanism only to the set of transactions or workloads where it shines, while maintaining isolation.

In particular, this work builds upon *Modular Concurrency Control (MCC)*, a recent technique that federates CCs by partitioning transactions into groups, and by applying different CC mechanisms in each group.

This dissertation addresses two critical shortcomings in the current embodiment of MCC. First, cross-group data conflicts are handled with a single, unoptimized CC mechanism that can significantly limit performance. Second, configuring MCC is a complex task, which runs counter to MCC's purpose: to improve performance *without sacrificing ease-of-programming*.

To address these problems, this dissertation presents Tebaldi, a new transactional database that brings Modular Concurrency Control to the next level, both figuratively and literally. Tebaldi introduces a new, hierarchical model to MCC that partitions transactions *recursively* to compose CC mechanisms in a multi-level tree. This model increases flexibility in federating CC mechanisms, which is the key to realizing the performance potential of federation. Tebaldi reduces configuration complexity by managing the MCC federation automatically: it can detect performance issues in the current workload in real-time, and automatically adjusts its configuration to improve its performance.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The ACID transaction is a vital primitive for developing many applications over storage systems. By providing four strong semantic guarantees—atomicity, consistency, isolation, and durability—it offers an elegant and powerful abstraction for structuring applications and simplifying reasoning about correctness under failures and concurrency. Performance, however, is not traditionally one of its strong suits. In particular, the concurrency control (CC) mechanisms used to enforce isolation can greatly limit the performance of ACID transactions when there is heavy data contention.

As a result, application developers have been suffering from a tension between two desired properties: *high performance* transaction processing, and *ease-of-programming*.

This dissertation eases this tension by improving the performance of ACID transactions, especially, for scenarios where data contention is the bottleneck. The work it presents builds on *Modular Concurrency Control*, a new approach to *federate* different concurrency control mechanisms in the same database for better performance, that my co-authors and I introduced in prior work [95], refining its design and implementation in ways that make it more

powerful and practical.

This work is among a long history of works to improve the performance of transaction processing, and people have tried to achieve this goal with many different approaches. On the one side of this landscape, people stick with ACID guarantees and explore various optimizations, such as introducing new and potentially more efficient concurrency control mechanisms [32, 50, 59], or leveraging static analysis to create some special optimizations [51, 80, 88, 95, 99]. There are also works that improves the performance of certain types of transactions (such as read-only transactions [40, 84]), or support only a limited transaction model that can perform well [24, 67, 96]. On the other side of this landscape, people try to give up ACID guarantees all together, and embrace the better performance from BASE / NoSQL approach [1, 20, 38, 39, 45, 55, 60, 73]. And some works also explored the middle ground by letting ACID and BASE coexist [94]. Each of these approaches has its own benefits and weakness.

For example, new concurrency control mechanisms, such as optimistic concurrency control [59] and snapshot isolation [29, 35, 50, 72] can perform better than the widely-used two-phase locking technique [30, 33, 53] under certain scenarios. But they may also perform worse in other scenarios. Advanced optimization techniques that leverages static analysis, such as transaction chopping [80, 99] and runtime pipelining [88, 95], can greatly improve the concurrency. But they often come with certain conditions or assumptions on transactions that may not hold in many applications. Works that provide only a limited transaction model, such as mini-transactions [24] or one-shot transac-

tions [67], can often design efficient transaction protocols for these transactions. The drawback, though, is that not all application logics can be implemented easily in these limited models, thus complicating the programming effort.

Frustrated by the performance of ACID, some systems choose to give up ACID properties for a better performance. For example, many NoSQL storage systems [1, 20, 38, 39, 60] choose to give up ACID transactions and embrace the BASE principles [55, 73]. Though their performance numbers are great, programming in the BASE model is challenging and error-prone. Without ACID guarantees, it is hard to reason about the correctness under concurrent execution, and failures can also put the database to an inconsistent state. There are also works that explore the middle ground between the ACID and the BASE approach to combine their benefits. For example, Salt [94] is such a work that my co-authors and I proposed a few years ago. It allows developers to program most transactions with ACID guarantees, and handle a few performance-critical transactions with a BASE transaction model that offers weaker guarantees but better performance. Though programming in Salt is much easier than that in a pure BASE system, it can still be a pain when it comes to those BASE transactions. The loss of ACID guarantees (and ease-of-programming) is often an unaffordable price, especially for applications where data consistency is important [40, 81, 83].

The work in my dissertation takes a different approach: federating concurrency control mechanisms [42, 68, 79, 82, 95].

The rationale for federating concurrency control mechanisms is straight-

forward: any single concurrency control technique is bound to make trade-offs or rely on assumptions that cause it to perform well in some cases but poorly in others. For instance, pessimistic techniques such as two-phase locking [30, 33, 53] do not cause aborts for deadlock-free application even in highly-contended workloads, but may lead to write transactions unnecessarily stalling read transactions; likewise, multi-versioned concurrency control algorithms [29, 31, 32] improve read performance, but may cause additional aborts, and introduce non-serializable behaviors that are difficult to detect [50, 69]. Since concurrent transactions interact in fundamentally different ways across these scenarios, these trade-offs appear unavoidable. A promising approach is instead to *federate* different concurrency control mechanisms within the same database, applying each given mechanism only to the portion of transactions or workloads where it shines, while maintaining the overall correctness of the database.

In practice, however, realizing the performance potential of a federated solution is challenging. Such a solution should be *modular*: it should allow developers to reason about the correctness of any given concurrency control in isolation, without being aware of other coexisting concurrency control mechanisms. The solution should also be *flexible* in determining how to partition the workload and assign them to different concurrency control mechanisms, and be *general*—it should be capable of federating a large set of diverse techniques: optimistic and pessimistic, single-version as well as multi-version.

Prior work has gone some way towards achieving these goals by enabling different concurrency controls to execute on disjoint subsets of either

data [79, 90], transactions [36, 37], or types of data conflicts [31]. But many of these approaches are restricted to specific partitioning of transactions / conflicts, or certain combinations of concurrency control mechanisms, so they are not flexible or general enough [36, 42, 68, 82, 90]. For example, integrated concurrency control [31] only partitions data conflicts into read-write and write-write conflicts, and handles them with different mechanisms. Similarly, multi-version two-phase locking [32, 36] only differentiates read-only and update transactions, and adopts a single combination of concurrency control mechanisms. Local atomicity properties [90] and the work from Sha et al. [79] federate concurrency controls over disjoint data sets, but they place stringent constraints on how they allow data to be partitioned and CCs to be combined.

The starting point of my work is Modular Concurrency Control (MCC), a more general approach to federate CCs that my co-authors and I recently introduced in the Callas database [95]. At a high-level, MCC partitions transactions in groups, giving each group the flexibility to run the concurrency control mechanism that is better suited to regulate concurrency for its transactions. MCC imposes no restriction on the transactions that it works with, or how to partition them. And in principle, it also does not depend on the choice of the in-group concurrency controls: as long as the isolation property holds within each group, MCC guarantees that it will also hold among all transactions.

This dissertation revisits how Callas embodies Modular Concurrency Control, and addresses two of its critical shortcomings.

First, Callas assumes that applications can be partitioned such that the conflicts across partitions are rare or inconsequential to their performance. Consequently, Callas optimizes the concurrency controls within each group, but handles all cross-group conflict using a single, undifferentiated mechanism.

I will show that this assumption is flawed: cross-group conflicts can in fact throttle MCC's performance benefits, as a perfect partitioning of conflicts is, in general, unfeasible. In practice, there is often an inescapable tension between minimizing cross-group conflicts and supporting aggressive CC mechanisms within each group, and Callas' conservative cross-group mechanism can become a performance bottleneck of the entire federation.

Second, the amount of performance benefit that one can gain from Modular Concurrency Control largely depends on its configuration, that is, how to partition transactions and what concurrency control mechanisms to choose. But unfortunately, configuring MCC can be very hard. On the one hand, designing a good MCC configuration often requires thorough exploration of the application and workload to understand its performance characteristics and potential bottlenecks, which can be a daunting task. On the other hand, the MCC technique itself is highly flexible and complicated. There can be exponentially many different ways to configure MCC, and the complicated interaction between concurrency control mechanisms makes it hard to predict the performance of a specific configuration. Also, configuring MCC requires good understanding of different concurrency control techniques, and the MCC framework itself. It is unreasonable to expect most database users to master

such knowledge.

The configuration complexity can be a major challenge in making MCC practical, and, if the job of configuring MCC is left to database users, it may void the key motivation of developing MCC in the first place, namely, improving ACID's performance *without sacrificing its benefit of ease-of-programming*. The paper that describes Callas offers some basic guidelines on how to configure MCC by iteratively optimizing transactions that are the most severe performance bottlenecks, and how this procedure can be automated. But its treatment of these issues is still very elementary. Many details are not justified or simply missing, and reality is often more complicated than what Callas' basic guidelines can handle.

To eliminate these shortcomings, this dissertation presents a new transactional key-value store, Tebaldi [85], bringing Modular Concurrency Control to a new level, both figuratively and literally. To address the first problem, Tebaldi employs a new approach to MCC that is based on a simple, but powerful, insight: the mechanism by which different concurrency controls are federated should *itself* be a federation. Instead of handling cross-group conflicts through a single mechanism, Tebaldi regulates them by applying MCC recursively, adding additional levels to its tree-like structure of federated CC mechanisms. This design increases the flexibility in how conflicts are handled, and is the key to realizing the performance potential of federation. Tebaldi can, for example, combine the benefits of multi-versioning [32, 35] with aggressive single-version techniques such as runtime pipelining [95] at the cross-group

layer.

Realizing this vision in Tebaldi presents two main technical hurdles. First, directly applying CCs hierarchically does not guarantee serializability. We derive a sufficient condition—*consistent ordering*—that CCs must enforce to ensure correctness, and highlight how this property can be achieved in practice. Second, federating different CC mechanisms requires seamlessly managing the different expectations upon which their correctness depends (in terms of protocol, storage, failure recovery, etc.): Tebaldi allows CCs to independently implement the execution logic (including maintaining the necessary metadata) for making ordering decisions, but provides a general framework for composing the CCs execution hierarchically and determining the appropriate version of data to read or write.

To address the second problem, I extended the Tebaldi database with the ability to fully automate how Modular Concurrency Control federates CC mechanisms. The goal is to make the use of Tebaldi as easy as a traditional database: users simply run their applications with real workloads, and the database system automatically optimizes itself by configuring MCC properly. The key technique to achieve this goal is an iterative approach to optimize the MCC federation. In each iteration, the optimization algorithm automatically monitors the database performance, accurately detects the data contention bottleneck, and proposes new MCC configurations to optimize the bottleneck.

There are several technical challenges that are raised by this approach, including how to accurately detect the performance bottlenecks, how to au-

tomatically design new MCC configurations to optimize these bottlenecks, and how to switch the database system between these different configurations. With the careful design to address each of these challenges, our system can retain most of the performance benefits of a manually-configured MCC database, while remove almost all the user interference in programming, performance debugging, and configuration.

In summary, this dissertation makes the following contribution:

- It presents an overview of related work towards the problem of enhancing transaction's performance, and discusses the strength and weakness of different techniques.
- It generalizes the theory of Modular Concurrency Control, a promising technique to achieve high performance ACID, by introducing a new hierarchical model that allows data conflicts to be handled more efficiently, while preserving modularity.
- It identifies a condition for the correct composition of concurrency control techniques in hierarchical MCC, and shows that several existing concurrency controls can be modified to enforce it.
- It presents the design and evaluation of Tebaldi, a transactional key-value store that implements hierarchical MCC.
- It presents a new technique to automatically configure MCC federations in Tebaldi that drastically reduces the complexity of configuring MCC.

Evaluation results show that our technique can retain most of MCC's performance benefit while removing almost all user interferences.

The rest of this dissertation is organized as follows. Chapter 2 provides an overview of ACID transactions, and presents related works in addressing the tension between high performance and strong semantics. Chapter 3 discusses the federated approach to concurrency control. It presents the current design of Modular Concurrency Control, and summarizes its strength and weakness. Chapter 4 discusses in detail how we generalize the theory of Modular Concurrency Control to a hierarchical model in the Tebaldi database. Chapter 5 presents how we automate the procedure of configuring MCC in Tebaldi. Finally, Chapter 6 summarizes this dissertation.

# Chapter 2

# ACID Transactions

This chapter overviews ACID transactions and sets up the background of this dissertation. It articulates why ACID transactions are a powerful primitive for developing applications, and why their performance tends to lag. Finally, it summarizes the strengths and weaknesses of existing efforts towards improving the performance of ACID transactions.

## 2.1 The ACID Properties

A *transaction* consists of a sequence of read and write operations that are carried out *atomically* by the storage system, just as if they were a single operation. As such, transactions allow applications to execute a piece of storage-access logic in an indivisible, all-or-nothing manner. To achieve this, transactions provide four very useful properties: *Atomicity*, *Consistency*, *Isolation*, and *Durability*. Together, they are often called *ACID properties*.

- *Atomicity* states that a transaction always changes database states in an all-or-nothing manner: either all of its changes are applied, or none of them is. In the former case, we say a transaction is *committed*; and in the later case, we say it is *aborted*.

- *Consistency* states that transactions always move database from a consistent state to a consistent state. In general, the definition of *consistent states* is application dependent, but they can often be expressed as a list of predefined invariants / constraints in the database. In such cases, consistency ensures that committed transactions will not violate these predefined constraints. If a transaction does violate them (e.g., because of buggy input / implementation), it will be aborted.

- *Isolation* regulates how concurrent transactions interact with each other when they have conflicting data accesses. Different isolation properties (normally referred to as *isolation levels*) allow different sets of interleavings for concurrent transactions. For example, the *serializable* isolation level [21, 29] states that, despite concurrent execution of transactions, the effect of the concurrent execution is always equivalent to a serial execution of the committed transactions in some order. Techniques to enforce isolation are *concurrency control mechanisms*.

- *Durability* states that committed transactions are durable. That is, once committed, the effects of a transaction on the storage system will not be lost even if the storage system may fail (e.g., crash, power loss) in the future.

Together, the four ACID properties make it easier to both develop applications and reason about their correctness. Atomicity and durability free developers from worrying that failures will leave the database in an inconsistent

state, or cause the loss of committed changes. With proper isolation, developers no longer need to worry about the concurrent execution of transactions, and instead mainly focus on implementing each transaction correctly. In particular, serializable isolation ensures that as long as each transaction, when running alone, transits data from consistent states to consistent states, and the initial state is consistent, the data will always be consistent.

ACID transactions are supported by various storage systems, no matter in commercial SQL databases [7, 8, 10, 13, 15], key-value stores [4, 18, 19], or in the cloud [2, 3, 6].

But ACID transactions do not come at free, and enforcing ACID properties can reduce the performance of transaction processing. My dissertation mainly focuses on the enforcement of the isolation property, and its impact on transactions' semantics and performance. In fact, to pursue better performance, the database community has long been exploring more efficient concurrency control techniques, as well as various isolation levels that are weaker than serializable isolation, and thus allow more concurrent interleavings of operations from different transactions. I will briefly introduce some popular isolation levels and discuss how they are usually enforced in storage systems.

## 2.2 Isolation and Concurrency Control Mechanisms

Concurrent transactions can access overlapping data objects. When two transactions access the same data, and at least one of the data accesses is

a write, a *data contention* (or data conflict[1]) occurs. An isolation property (a.k.a., isolation level) then restricts the possible concurrent interleavings by defining the set of allowed executions for concurrent transactions.

The database community has proposed many different isolation levels, each with different semantic guarantees and performance implications. Perhaps surprisingly, precise definitions for these isolation levels have, for a long time, proved elusive.

### 2.2.1 ANSI Isolation Definitions

The ANSI SQL-92 document [21] introduces four different isolation levels: *Serializable*, *Repeatable Read*, *Read Committed*, and *Read Uncommitted*. These isolation levels are defined in terms of whether or not they three types of *anomalies*: *Dirty Read*, *Non-repeatable Read* and *Phantom*. The anomalies are described in English, as follows [21, 29]:

- *Dirty Read*: transaction $T_1$ modifies a data item. Another transaction $T_2$ then reads that data item before $T_1$ performs a commit or abort. If $T_1$ then performs an abort, $T_2$ has read a data item that was never committed and so never really existed.

- *Non-repeatable Read*: Transaction $T_1$ reads a data item. Another transaction $T_2$ then modifies or deletes that data item and commits. If $T_1$

[1] In this dissertation, I treat *data contention* and *data conflict* as synonyms.

then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.

- *Phantom*: Transaction $T_1$ reads a set of data items satisfying some search condition. Transaction $T_2$ then creates data items that satisfy $T_1$s search condition and commits. If $T_1$ then repeats its read with the same search condition, it gets a set of data items different from the first read.

Table 2.1 shows the anomalies proscribed by each isolation level. The ANSI prefix is intended to distinguish these definitions from alternative formulations that will be discussed later in this chapter.

| Isolation Level | Dirty Read | Non-repeatable Read | Phantom |
|---|---|---|---|
| ANSI Serializable | ✗ | ✗ | ✗ |
| ANSI Repeatable Read | ✗ | ✗ | ✓ |
| ANSI Read Committed | ✗ | ✓ | ✓ |
| ANSI Read Uncommitted | ✓ | ✓ | ✓ |

Table 2.1: Isolation level definitions from ANSI SQL-92 document.

Intuitively, anomalies correspond to concurrent interleavings that cannot occur in serializable executions. By defining isolation levels using anomalies, the ANSI document aims to define isolation levels independently of how they are implemented by specific concurrency control mechanisms. In particular, disallowing all three anomalies, executions that are ANSI serializable should be equivalent to an execution where transactions are executed sequentially.

**Critique and Refinement to ANSI Definitions** In their pointed critique, Berenson et al. submit that any implementation independence gained through the ANSI definition has been acquired at the cost of infusing them with ambiguities [29]. They show that the three anomalies can be interpreted in at least in two different ways: a *strict interpretation* that leads to broad isolation level definitions, and a *broad interpretation* that leads to strict isolation level definitions. In the strict interpretation, anomalies capture executions that are *actually* not serializable. For example, Dirty Read under this interpretation refers to a committed transaction reading from an aborted transaction:

$$T_1.\text{write}(x), ..., T_2.\text{read}(x), ..., (T_1.\text{abort and } T_2.\text{commit in any order})$$

In the broad interpretation, instead, anomalies also include executions that *may* lead to non-serializable results in the future. For example, Dirty Read will now refer to any transaction $T_1$ reading a value from an unfinished transaction $T_2$—it does not require $T_1$ to actually abort, or $T_2$ to actually commit:

$$T_1.\text{write}(x), ..., T_2.\text{read}(x), ..., (T_1.\text{commit or abort})$$

To distinguish these two interpretations, Berenson et al. called the broad version of anomalies *phenomena*.

Berenson et al. find that just preventing the three anomalies in the strict interpretation is not enough to ensure serializable. They give the following example, where $T_1$ transfers 40 dollars from $x$ to $y$, while $T_2$ checks the total balance of $x$ and $y$. The execution does not trigger any anomalies in the strict

interpretation, but it is not serializable, since $T_2$ reads a wrong total balance:

$$T_1.\text{read}(x = 50), T_1.\text{write}(x = 10), T_2.\text{read}(x = 10), T_2.\text{read}(y = 50),$$

$$T_2.\text{commit}, T_1.\text{read}(y = 50), T_1.\text{write}(y = 90), T_1.\text{commit}$$

This execution, however, will be captured by the loose interpretation of Dirty Read, and therefore disallowed.

Moreover, they found that even disallowing all three *phenomena* will not guarantee serializable execution (in the common sense): running a set of write-only transactions is not regulated by the three phenomena at all.

To address any ambiguity and inaccuracy in the original ANSI isolation definitions, Bernenson et al. propose new and formal definitions of the four isolation levels. These definitions take the loose interpretation of anomalies (i.e., they focus on phenomena), and they include a fourth phenomenon, *Dirty Write*, that all four isolation levels should prevent.

Formally, let an *execution history* be a linear order of read, write, commit, and abort operations that represents the concurrent execution of a set of transactions. Assume $T_1$ and $T_2$ are two transactions, $x$ is a data object, and $P$ is a predicate used in search. The four phenomena are defined by the following structures on execution histories:

- Dirty Write: $T_1$.write($x$), ..., $T_2$.write($x$), ..., $T_1$.commit or abort.
- Dirty Read: $T_1$.write($x$), ..., $T_2$.read($x$), ..., $T_1$.commit or abort.

- Non-repeatable Read: $T_1$.read($x$), ..., $T_2$.write($x$), ..., $T_1$.commit or abort.
- Phantom: $T_1$.read($P$), ..., $T_2$.write($x \in P$), ..., $T_1$.commit or abort.

The four isolation levels are then defined by disallowing different sets of phenomena, as shown in Table 2.2.

| Isolation Level | Dirty Write | Dirty Read | Non-repeatable Read | Phantom |
|---|---|---|---|---|
| Serializable | ✗ | ✗ | ✗ | ✗ |
| Repeatabler Read | ✗ | ✗ | ✗ | ✓ |
| Read Committed | ✗ | ✗ | ✓ | ✓ |
| Read Uncommitted | ✗ | ✓ | ✓ | ✓ |

Table 2.2: Refined definitions of ANSI isolation levels.

With the new definitions, the four isolation levels can indeed rule out undesired concurrent interleavings, and any serializable history under this definition is equivalent to a history where transactions are executed in some sequential order.

These definitions, however, also bring new problems. First, they are unnecessarily strict, ruling out many executions that are actually serializable. For example, preventing dirty reads in this definition generally disallows any data flow from an ongoing transaction to another, even if the writing transaction eventually commits, or if both transactions abort. Second, they are no longer implementation-independent. Indeed, Berenson et al. are the first to point out that they amount to a disguised re-statement of *two-phase locking* [30, 33, 53], a pessimistic concurrency control mechanism that *implements* these isolation levels. Consequently, these definitions may be incompatible with concurrency

control mechanisms based on optimistic or multi-versioned schemes, such as serializable snapshot isolation (discussed below) that can often achieve better performance than two-phase locking.

### 2.2.2 Snapshot Isolation

Snapshot isolation is also defined by Berenson et al. [29]. It is a *multi-versioned* concurrency control mechanism that keeps multiple versions of the same object in the database. In snapshot isolation, a transaction effectively read data from a database *snapshot* taken at the time when the transaction starts, and its updates create a new database snapshot at the time when the transaction commits.

Each transaction is assigned two unique timestamps, a *start timestamp* at its start time, and a *commit timestamp* at its commit time. The database keeps all committed versions of a data object, with each version identified by the commit timestamp of the writing transaction. When a transaction $T_1$ reads a data object, it returns the latest (committed) version whose timestamp is smaller than $T_1$' start timestamp. $T_1$'s writes are buffered until it commits. At commit time, $T_1$ checks for concurrent writes on data objects in its write set, i.e., whether a transaction $T_2$ updated one or more of objects modified by $T_1$, and committed between $T_1$'s start and commit timestamp. If so, $T_1$ has to abort; otherwise, $T_1$ commits.

Snapshot isolation differs from the four ANSI isolation levels. It is weaker than serializable, in that it allows a non-serializable anomaly called

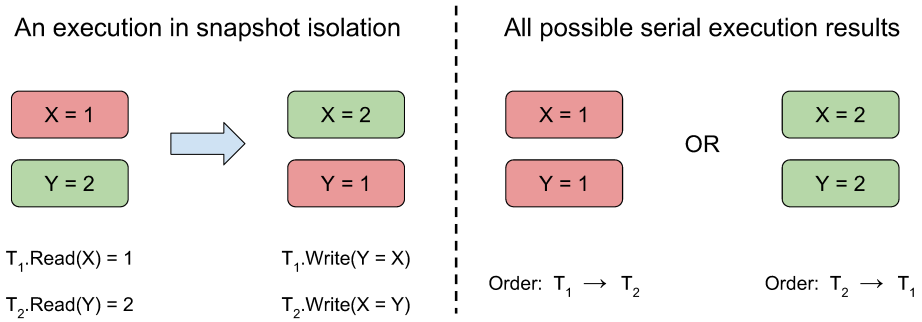



Figure 2.1: The write skew anomaly in snapshot isolation.

*write skew*. Figure 2.1 shows an example: $T_1$ writes the value of object $x$ into object $y$, while $T_2$ writes the value of $y$ into $x$. In snapshot isolation, both transactions can read the original value of the two objects from an initial snapshot, and update $x$ and $y$ separately. The end result is that the concurrent execution of $T_1$ and $T_2$ swaps the value of $x$ and $y$, an outcome that is impossible in any serial history. However, snapshot isolation is stronger than read committed, in that the set of non-serializable histories that are allowed by snapshot isolation is a strict subset of that allowed by read committed. Finally, snapshot isolation is not comparable with repeatable read, as either allows some non-serializable histories that are disallowed by the other.

Fekete et al. [50] and Cahill et al. [35] explored how to achieve serializable isolation level with the snapshot isolation technique. The result is a new concurrency control mechanism called *serializable snapshot isolation (SSI)*. The key idea is to detect and prevent a dangerous structure in called the *pivot* that all non-serializable executions in snapshot isolation exhibit. A pivot in-

volves three transactions, $T_1$, $T_2$, and $T_3$, where $T_2$ is *concurrent* with both $T_1$ and $T_3$ (that is, the interval between $T_2$'s start and commit timestamp overlaps with that interval of both $T_1$ and $T_3$). Furthermore, there is a pair of conflicting read and write operations between $T_1$ and $T_2$, where $T_1$'s read misses $T_2$'s write, and a pair of conflicting read and write operations between $T_2$ and $T_3$, where $T_2$'s read misses $T_3$'s write. Once detected, the pivot structure is removed by aborting one of these transactions. SSI may cause unnecessary aborts though, since not all pivot structures will eventually cause a non-serializable history.

### 2.2.3 A Graph-based Isolation Definition

To address problems in Bernenson et al.'s isolation theory (Section 2.2.1), Atul Adya et al. proposed a new and elegant isolation definition that is based on graphs [22, 23]. Their key observation is that though many consistency conditions involve multiple data objects, the phenomena proscribed by Berenson et al. are expressed in terms of accesses to a single object. To be sufficiently expressive to nonetheless capture multi-object conditions, Berenson's formulations end up being overly restrictive. Adya's new isolation theory avoids this problem by directly expressing multi-object restrictions (with the help of a graph), so it can more accurately capture the fundamental differences between the allowed and disallowed interleavings in each isolation level.

Adya's theory also aims to be implementation-independent, and be compatible with both single-version and multi-versioned data models. Thus, it adopts a database model where each write operation creates a new *version*

of a data objects, so read operations can read different versions; committed versions of each data object are totally ordered. The definition of an *execution history* is also generalized to be a *partial* order of read, write, commit, and abort operations[2].

At the core of this new isolation theory is a data structure called *Direct Serialization Graph (DSG)*. It is a directed graph derived from a given execution history. Different isolation levels, then, are characterized by disallowing specific cycles in the graph.

Each node in a DSG is a *committed* transaction in the underlying execution history, and each edge represents a *direct dependency*, i.e., a logical *happen-before* relationship on conflicting data accesses between the two transactions. There are three types of direct dependencies between two transactions, creating three different types of edges:

- There is a *direct write-read dependency* from $T_1$ to $T_2$ if $T_1$ installs a version on some object $x$ that was read by $T_2$. In a DSG, this is represented by an edge $T_1 \xrightarrow{wr} T_2$.

- There is a *direct write-write dependency* from $T_1$ to $T_2$ if $T_1$ installs a version on some object $x$, and $T_2$ installs the next version on $x$ with respect to the version order. In a DSG, this is represented by an edge $T_1 \xrightarrow{ww} T_2$.

[2]The single-version database model is a special case of this new model, so it can still support single-versioned databases.

- There is a *direct read-write dependency*, or *direct anti-dependency* from $T_1$ to $T_2$ if $T_1$ reads a version on some object $x$, and $T_2$ installs the next version on $x$ with respect to the version order. In a DSG, this is represented by an edge $T_1 \xrightarrow{rw} T_2$.

We also say $T_2$ *depends on* $T_1$, or $T_1$ *happens before* $T_2$, if there is a dependency from $T_1$ to $T_2$.

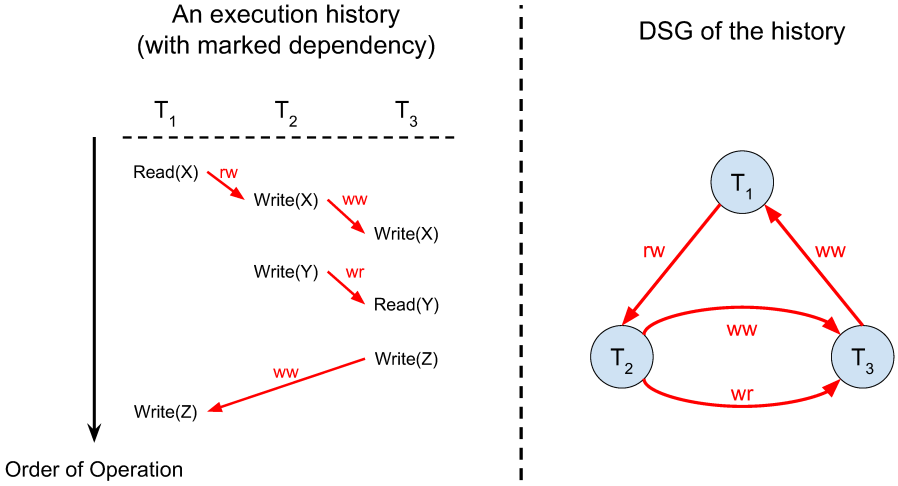



Figure 2.2: An example of execution history and its DSG.

Figure 2.2 gives an example of execution history and its corresponding DSG. For simplicity, the example uses a single-version, totally-ordered history, and it omits the commit operations.

Note that, Adya's theory discusses dependency relationships not only for operations that access a single data object (which form *item-level* dependencies), but also for operations that affect all data objects that match a

given predicate (which form *predicate-level* dependencies). But for the purpose of this dissertation, I will only discuss item-level dependencies, since the database system that we propose in this dissertation does not involve predicate operations, such as searches.

Different ANSI isolation levels can now be characterized by proscribing two anomalies defined directly on execution histories, and certain cycles in DSGs associated with their execution histories, as follows.

First, let $T_1$, $T_2$ be two transactions, $x_i$, $x_j$ be versions of data object $x$. All ANSI isolation levels, except for read uncommitted, proscribe the following two anomalies:

- *Aborted Read*: A committed transaction reads a version that was installed by an aborted transaction. Formally,

  $$T_1.\text{write}(x_i), ..., T_2.\text{read}(x_i), ..., (T_1.\text{abort and } T_2.\text{commit in any order})$$

- *Intermediate Read*: A committed transaction reads a version other than the last version of an object installed by another transaction. Formally,

  $$T_1.\text{write}(x_i), ..., T_2.\text{read}(x_i), ..., T_1.\text{write}(x_j), ..., T_2.\text{commit}$$

Second, each ANSI isolation level proscribes a corresponding type of cycles in DSG:

- The read uncommitted isolation level disallows any execution history whose DSG contains a cycle that only consists of ww-dependency edges.

- The read committed isolation level disallows any execution history whose DSG contains a cycle that only consists of ww- and wr-dependency edges.
- The repeatable read isolation level disallows any execution history whose DSG contains a cycle that only consists of ww-, wr-, and item-level rw-dependency edges.
- The serializable isolation level disallows any execution history whose DSG contains a cycle of any kind.

Since we only discuss item-level dependencies here, repeatable read and serializable are essentially the same.

This isolation theory can also describe snapshot isolation. To do so, it extends the DSG with a new type of dependency edge that captures temporal dependencies: it connects two transactions if the first commits before the second starts. Further details can be found in Adya's paper [23].

This graph-based definition of ANSI isolation levels captures the *essence* of isolation properties: fundamentally, isolation is about properly ordering conflicting *operations*, so that they can eventually induce certain ordering of *transactions*. Unlike the phenomena-based definitions, Adya's definition is independent of the implementation of certain concurrency control mechanisms, and it is not overly strict. For example, it allows concurrent transactions to write the same set of data and commit: as long as the order of their writes is consistent, they do not create any cycle in the DSG. It also allows concurrent transactions to expose uncommitted data, as long as the no-cycle requirements are

met, and both aborted reads and intermediate reads are avoided, if necessary, by aborting the reader. For these reasons, the rest of this dissertation adopts Adya's isolation definitions.

### 2.2.4 Enforcing Isolation with Concurrency Control

Storage systems use *concurrency control mechanisms* to enforce the desired isolation properties. A concurrency control mechanism must always be *correct*: all execution histories that it generates must be a subset of the allowed histories of the targeted isolation level. It does not need to be tight though: indeed, many commonly-used concurrency control mechanisms disallow some histories that are allowed by the targeted isolation level.

Different Concurrency control mechanisms can take very different approaches to enforce isolation. Pessimistic mechanisms use locks to prevent undesired interleavings from happening in the first place [53]. Optimistic mechanisms operate under the assumption that concurrent transactions do not violate isolation properties in most cases, and check for isolation violation just before a transaction commits [59]. Single-versioned concurrency control mechanisms keep only the most recent committed value of each data object. Multi-versioned mechanisms keep, in addition, some earlier values of each data object, which can be returned by some read operations [32, 35].

## 2.3 High Performance and ACID Guarantees

Concurrency control mechanisms used to enforce isolation can greatly limit the performance of ACID transactions when there is heavy data contention.

Consider, for example, the two-phase locking concurrency control mechanism [30, 33, 53], When a transaction accesses a data object, it acquires a lock on that object, and does not release it until the transaction finishes (except for read operations in weaker isolation levels). As a consequence, transactions can prevent other transactions from accessing locked data objects for long periods of time, which can become a major bottleneck if the workload contains heavy data contentions.

Performance often worsens when the storage system is distributed over a cluster of machines, as costly network round-trips further increase the duration of data conflicts. Distributed commit protocols, such as two-phase commit [65], also delays the time when a transaction commits, so a transaction must hold locks for a longer time.

This dissertation aims to improve the performance of ACID transactions under heavy data contentions without compromising their semantic guarantees. To put my work in context, I discuss below earlier efforts at resolving the tension between performance and ACID guarantees.

### 2.3.1 Optimizing Transaction Protocols

The first category of work insists on providing strong ACID guarantees, (e.g., serializable isolation level), and focuses on improving the transaction protocols, including the concurrency control mechanisms.

**Proposing New Concurrency Control Mechanisms** As we have seen in Section 2.2.4, the database community has introduced various concurrency control mechanisms over the years. Techniques like optimistic concurrency control (OCC) [59], serializable snapshot isolation (SSI) [35] and multi-versioned timestamp ordering (TSO) [32, 76] can perform better than two-phase locking under certain workload scenarios. But in general, there are always trade-offs: these mechanisms can suffer from new problems in other scenarios. SSI, for example, handles write-read contentions more efficient than 2PL, but it may experience high abort rates when write-write conflicts are frequent.

**Leveraging Static Analysis** Some concurrency control techniques perform static analysis on transaction codes, or even maneuver them, to figure out more efficient ways to run these transactions. The effectiveness of these techniques can depend highly on whether the transaction codes exhibit certain structures or properties that these techniques leverage: when the condition is in favor, these techniques can be very efficient; otherwise, they may simply not work.

One example is transaction chopping [80, 99]. The basic idea of this technique is to optimize 2PL by reducing locking period and allowing transac-

tions to expose their intermediate states. To do so, it chops large transactions that contain many operations into several smaller *sub-transactions*. The chopping ensures that any serial history of the sub-transactions will be equivalent to a serial history of the original transactions, so 2PL only needs to hold locks during each sub-transaction, rather than the entire transaction. By reducing the locking period, transaction chopping can enhance the performance when data contention is the bottleneck.

But there are several limitations in this approach. First, it requires all transaction codes to be known in advance, since it needs to perform a static analysis to chop transactions. Second, to prevent aborted-read, a transaction must commit once its first sub-transaction finishes. So user-issued aborts can only be in the first sub-transaction. Third, the chopping must meet a global condition called *no SC-cycle* [80]. Specifically, the static analysis constructs an *SC-graph* where each node is a sub-transaction. Sub-transactions from the same transaction are chained with an *S-edge*, and sub-transactions from different transactions that *may* conflict are connected with a *C-edge*. The chopping must ensure that the graph does not contain cycles with both S- and C-edges. If an SC-cycle exists, sub-transactions need to be merged to remove the cycle, but doing so will reduce the performance benefit. In reality, complicated applications may have many conflicting operations that cause many SC-cycles, which can make transaction chopping inefficient.

Like transaction chopping, sagas [51] improves the performance of transactions by breaking long-lived transactions into sub-transactions. The chal-

lenge of using sagas, however, is that it does not retain isolation by-default: users have to manually figure out how to chop transactions so that the interleaving of sub-transactions does not violate application semantics. In case a transaction fails in the middle, sagas relies on user-defined *compensating transactions* to amend partial executions. Sagas ensures that either all sub-transactions are executed, or compensating transactions are called.

Runtime pipelining (RP) [88, 95] is another technique to optimize 2PL, introduced in the Callas project by my co-authors and me. RP also allows transactions to expose intermediate states, but comparing to transaction chopping, RP uses more sophisticated static analysis and runtime techniques to allow finer chopping that contains SC-cycles. RP is based on an observation that if transactions access data objects in the same global order, concurrent transactions can run in a *pipelined* manner: once a transaction finishes accessing the $i$-th data object, the next transaction can start accessing that object. RP's static analysis detects such global order, or reorder operations to form such order. It then chops transactions into *steps* according to the global data access order. At runtime, transactions can release locks in a step once the step completes. If $T_2$ is ordered after $T_1$ in a step, for future steps, $T_2$ can only start a step after $T_1$ finishes that step.

Runtime pipelining has its own limitations: it is conditioned on a global data access order. If such order does not exist (which can be common in complicated applications), some data objects need to be logically combined into a single step in the pipeline. This combination does not require the involved

data objects to share the same lock, but it still results in coarser chopping and less performance benefit.

ROCOCO [66] introduced another optimization technique that relaxes the no SC-cycle requirement in transaction chopping. Transactions in ROCOCO consists of several *atomic pieces*. At runtime, ROCOCO tracks dependencies between concurrent transactions. At commit time, the dependency information are interchanged among the participant database servers, and any isolation violation is detected and resolved by *reordering* the conflicting pieces deterministically across all servers.

ROCOCO's static analysis still needs to construct an SC-graph, but it distinguishes C-edges into *immediate* and *deferrable* ones, according to whether the outputs of the involved transaction pieces are immediately used in the next piece of the transaction. Only SC-cycles whose C-edges are all immediate are prohibited in the static analysis.

**Optimizing Certain Types of Transactions** Many storage systems choose to optimize certain types of transactions that are common or important in many application workloads.

Many systems come with optimizations for read-only transactions, such as Spanner [40], F1 [81], and Carousel [96]. Spanner, for example, uses multi-versioning to support lock-free read-only transactions. It further reduces the chance of blocking by carefully choosing read timestamps, and using the TrueTime API to manage clock synchronization.

Multi-version two-phase locking (MV2PL) [32, 36, 37, 47, 89] uses multi-versioning to handle read-write conflicts between read-only and update transactions. Read transactions are assigned read timestamps at their start time, and they read the latest committed version of data that is smaller than the read timestamp—no locks are needed for read-only transactions. Update transactions acquire commit timestamps at their commit time, and they store their write values with that timestamp. This way, read-only and update transactions never block (or abort) each other. Among update transactions, data conflicts are regulated with a standard 2PL, so operations in update transactions, including reads, need to acquire locks.

Granola [42] optimizes *one-round independent* transactions, where each involved database partition can execute the transaction without communicating with others, and achieve the same commit / abort decisions at end. Granola employs an efficient timestamp-based mechanism for these transactions to avoid the cost of locking and two-phase commit. Likewise, H-Store [57, 84] handles *single-sited* transactions (whose operations access the same partition) and *one-shot* transactions (whose operations do not depend on each other) with optimized mechanisms. For example, H-Store can avoid the cost of unnecessary network communication, concurrency control, and undo logs for single-sited transactions.

**Supporting a Limited Transaction Model** Another common approach to improve the performance of transaction processing is to only support a

limited transaction model, so that storage systems can leverage additional properties that may not hold in a more general transaction model to optimize their performance.

Janus [67] limits its transaction model to *one-shot* transactions, which we have mentioned in H-Store [57, 84]. Janus allows one-shot transactions to have data or control-flow dependencies among operations in the same partition (which is slightly different from H-Store), but disallows operations from different partitions to affect each other. This property can help storage systems reduce the number of network round-trips during the transaction execution and the commit protocol.

Sinfonia [24] restricts its transaction model to *minitransactions*. They are lightweight transactions that consist of three data access sets: a *compare* set, a *read* set, and a *write* set. All the object addresses, the values to write, and the values to compare with, must be known at the beginning of the transaction. A minitransaction checks if the data objects in the compare set match those given values. If the check passes, data in the read set are retrieved, and data in the write set are modified. Otherwise, the transaction aborts. This transaction model allows Sinfonia to provide efficient and consistent data accesses, but it also limits its use in many applications.

Carousel [96] introduced two-round fixed-set interactive (2FI) transactions. A 2FI transaction consists of a read round followed by a write round. It allows write values to depend on read results, even if they are from different partitions. However, the *address* of all data accesses must be known at the

beginning of the transaction, and cannot depend on prior operations. These features make 2FI transactions more expressive than one-shot transactions. But the lack of address-dependency still makes it hard to implement many common data structures and logics, such as secondary indices.

Calvin [86] eliminates the cost of running distributed commit protocols by determining the execution schedule for concurrent transactions in advance (*before* they acquire locks and start to execute). But like Carousel, Calvin also requires knowledge about transactions' read and write sets at their beginning, so it cannot natively support transactions whose read and write sets depend on prior read results.

A common walk-around to the address-dependency problem is to modify the original transaction and introduce a *reconnaissance* transaction [86, 96], which reads necessary data to generate the read and write sets. However, this requires the actual transaction to re-read these reconnaissance queries, and abort if any result changes. This may cause a lot of aborts when data contentions are heavy, offsetting their performance benefits.

**Optimizations for Geo-distributed Transactions** The work presented in this dissertation focuses on distributed databases that are deployed within a single datacenter. There are other database systems [40, 58, 67, 96, 98] that work in a geo-distributed setting, spanning multiple datacenters. These are very different scenarios: sending and receiving a message in a single datacenter usually takes tens to hundreds of microseconds (though this is still longer

than the computational time of transaction protocols); but doing so across datacenters can take *much* longer time (100s of milliseconds). Therefore, a key focus in geo-distributed systems is to reduce the number of sequential wide-area (i.e., cross-datacenter) network round-trips.

One technique to achieve this goal is to use a limited transaction model, as running general-purpose, interactive transactions can be costly in a geo-distributed setting. Indeed, some techniques that we have discussed, such as one-shot (Janus [67]) and 2FI (Carousel [96]) transactions, are used in geo-distributed systems.

Another common technique to reduce the number of wide-area round-trips is to co-design the transaction and replication protocol, rather than layering one on top of the other. For example, TAPIR [98] co-designs the two protocols so they can provide strongly consistent transactions with an inconsistent replication protocol, and commit most transactions in a single wide-area round-trip. Similar techniques are also introduced in MDCC [58], Janus [67], and Carousel [96].

### 2.3.2 Weakening Isolation

Another way to improve performance is to give up some ACID guarantees by adopting an isolation level that is weaker than serializable. Weakening the isolation property can bring more interleavings, increasing the overall performance when data contention is the bottleneck. Applications can still enjoy a well-defined (but weaker than serializable) isolation property, together with

the other three ACID properties. In fact, some isolation levels, such as snapshot isolation and repeatable read, are close to serializable, and some applications can work correctly with them.

Indeed, many commercial database systems choose a weak isolation level as their default settings. For example, MySQL database with InnoDB backend uses repeatable read as its default isolation level [12]. MySQL database with NDBCluster backend uses read committed as default—it is actually the only isolation level supported by NDBCluster [11]. Microsoft's SQL Server [9], Oracle Database 18c [14], and PostgreSQL [17] also uses read committed as their default settings. Their choices are not surprising at all: strong isolation levels, such as serializable, when implemented with lock-based concurrency control mechanisms, can cause severe blocking between read and write operations, which is often undesirable. This is yet another evidence of the tension between high performance and strong semantic guarantees.

However, weakening isolation levels cannot address all data contention bottlenecks. It is most effective to reduce the cost of read-write conflicts. Write-write ones, though, are still regulated by most isolation levels, and therefore cannot benefit from this approach. In fact, it may not even address all problems in read-write conflicts. Two-phase locking, for example, still acquires short-term read locks under read committed isolation level, so write operations may still block read operations.

### 2.3.3 Renouncing ACID Guarantees

In pursuit of better performance, some storage systems choose to give up ACID properties all together. For example, many NoSQL storage systems [1, 20, 38, 39, 45, 60] choose to either provide very limited transaction support on single data object, or give up ACID transactions and embrace the BASE principles [55, 73], which stands for only providing basically available, soft state and eventual consistency guarantees.

By weakening or removing the transactional guarantees, these systems can perform very well, but programming applications with these systems can be very challenging. Without ACID properties, it is very hard to reason about the correctness of applications under concurrent execution—it is now the application developer's job to ensure such correctness. Moreover, failures of application or database can also put data to an inconsistent state. The lose of ease-of-programming is often an unaffordable price, especially for applications where data consistency is important. In fact, as Shute et al. mentioned in their paper of F1 database [81], "Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains".

Some systems, like ElasTras [43] and G-Store [44] try to mitigate the programming complexity by providing transactions within a single partition or key group. But the lack of cross-partition transactions can still limit their usages.

Salt [94] tries to combine the benefit of the ACID and the BASE approach. It allows developers to program a few performance-critical transactions in a BASE transaction model that offers weaker guarantees but better performance, and to program the rest of transactions with ACID guarantees. As such, programming in Salt is much easier than programming in a pure BASE system. However, it can still be a pain when it comes to those BASE transactions.

# Chapter 3

# Federating Concurrency Control Mechanisms

Instead of proposing new and optimized concurrency control mechanisms, my work takes a different approach to improve the performance of ACID transactions: *federation* [42, 68, 79, 82, 95].

The idea is to *compose multiple concurrency control mechanisms within the same database*. Each of these mechanisms only regulates the concurrent execution of a fraction of the workload (e.g., a subset of transactions, data conflicts, or data objects), while together, they ensure correct isolation over the entire workload.

In this chapter, I will first explain why federating concurrency controls is a promising approach to improve the performance of ACID transactions, and provide an overview of some existing work that takes this approach. I will then focus on *Modular Concurrency Control (MCC)*, a more recent work in this vein that my co-authors and I introduced in the Callas database [95]. Though Callas is not part of my dissertation, it serves as the basis of my work. I will discuss the limitations of Callas' implementation of MCC, and give a brief overview of how they will be addressed in the rest of this dissertation.

## 3.1 The Benefits of Federation

The rationale for federating concurrency control mechanisms is based on the observation that any single concurrency control technique is bound to rely on assumptions that cause it to perform extremely well in some cases but poorly in others. For example, multi-versioned concurrency control mechanisms, such as snapshot isolation, can improve read performance, since reads and writes do not block each other. But they can also cause aborts on write-write conflicts, and introduce non-serializable behaviors, like write-skews, that are difficult to detect [50, 69]. Meanwhile, pessimistic techniques such as two-phase locking [33] do not cause aborts even in highly-contended workloads (as long as the application is deadlock-free). But they may lead to write transactions unnecessarily stalling read transactions. If a database uses only a single concurrency control mechanism, these trade-offs appear unavoidable, since even transactions within the same workload can interact with each other in fundamentally different ways. *Federating* different concurrency control mechanisms within the same database opens the opportunity of applying each given concurrency control mechanism only to the part of transactions or workloads where it shines, while maintaining the overall isolation property. For example, it is possible to get better performance by combining the multi-versioned snapshot isolation technique with two-phase locking, using locking to handle write-write conflicts, and using multi-versioning to handle read-write ones [32, 36, 37, 47, 89].

The drive towards federating concurrency controls also stems from a

tension between a concurrency control's generality and its ability to aggressively handle conflicts. More general concurrency control mechanisms like two-phase locking and optimistic concurrency control make few assumptions about the application or the workload, but they are often overly pessimistic in dealing with conflicts. More specialized optimizations, like transaction chopping and runtime pipelining, however, rely on properties that are unlikely to hold *globally* for an entire application workload: full knowledge of the read and write set [49, 86], lack of SC cycles [80], the ability to statically determine a total order of tables [95], or access locality [56, 61]. By federating these mechanisms, the scope of these optimizations can be restricted only to the portions of the application for which their assumptions hold, allowing for higher performance without sacrificing generality.

Consider, for example, runtime pipelining (RP) [95] and deterministic concurrency control (DCC) [49, 86]. As we have seen in the previous chapter (Section 2.3.1), runtime pipelining efficiently pipelines transactions by allowing operations to observe the result of uncommitted writes, but it requires transactions to have a consistent global data access order. Therefore, runtime pipelining is most effective when transactions generate few circular dependencies when accessing tables. As the number of transactions grows, however, such dependencies are increasingly likely. Runtime pipelining is, for instance, of limited use when applied to the full TPC-C (Figure 3.1), as there exists a circular dependency in the `new order` and `stock level` transactions between the `stock`, `order line`, and (the preferred execution order of) `district` tables,

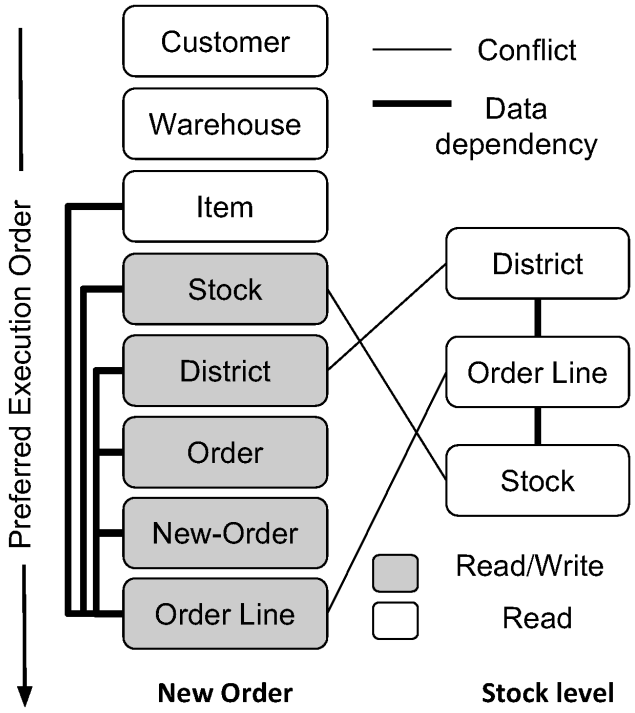



Figure 3.1: TPC-C `new order`/`stock level` transactions.

which prevents RP from efficiently pipelining these operations. If its scope were instead restricted to regulating multiple concurrent instances of `new order`, RP could choose a finer grained pipeline, improving performance [95].

Similarly, *deterministic concurrency control* (DCC) [49, 86] shines when the complete data access set of transactions is known at start time, as then DCC can pre-order transactions according to their data-access, removing the overhead of runtime conflict detection. Otherwise, DCC's benefit is limited by its need for issuing pre-transaction reconnaissance reads to construct the data access set, and the transaction may have to abort if these read results are changed in between.

## 3.2 Prior Work on Federating Concurrency Controls

The federated approach to concurrency control has a great performance potential, but in practice, realizing this potential is challenging. A good solution should be *modular* in correctness: it should come with a well-defined correctness condition for participating concurrency control mechanisms, so that the entire federation is correct as long as each individual mechanism meets the condition. Also, each mechanism should be able to reason about such correctness condition in isolation, without being aware of the implementation of other coexisting mechanisms. Meanwhile, for better performance, the federated solution should also be *flexible* with how to partition the workload and assign them to different concurrency control mechanisms, and be *general* enough to federate a large set of diverse techniques: optimistic and pessimistic, single-version as well as multi-version.

Prior work has gone some way towards achieving these goals, enabling different concurrency controls to execute on disjoint subsets of data [79, 90], transactions [36, 37], types of data conflicts [31], and so on. This section will introduce these related work, and discusses their strength and weakness. To give a clear picture, I organize these work into several categories according to how they compose concurrency control mechanisms (e.g., apply different mechanisms among different transactions / conflicts / data, etc.).

**Partitioning by Transactions or Conflicts** Many systems seek to improve performance by tailoring concurrency controls to specific transactions

or conflicts. Integrated concurrency control [31], for instance, uses different concurrency control mechanisms to handle write-write and write-read conflicts. Likewise, multi-version two-phase locking (MV2PL) [32, 36, 37, 47, 89], which we have mentioned in the Section 2.3.1, partitions transactions into read-only and update transactions; the latter are regulated using 2PL, while read-only transactions are allowed to read, without blocking, from a consistent snapshot by using a multiversioned protocol. Section 2.3.1) also discusses H-Store [84], which optimizes transactions that can execute on a single partition by removing unnecessary network communication, and Granola [42], which optimizes independent distributed transactions, i.e., transactions where each site can reach the same decision even without communication, by eliminating the two-phase commit protocol. These techniques can be seen as federations, since they optimize certain types of transactions with a different protocol.

But these federation techniques are not flexible enough, since they simply partition transactions or conflicts in a fixed manner, and apply one concurrency control mechanism in each of these fixed partitions.

**Partitioning by Data** Other systems instead partition the database into multiple disjoint components, allowing each component to execute its own concurrency control while preserving consistency globally. Federated databases [34, 74, 75, 78] for instance, support serializable *global transactions* that touch multiple local databases running different concurrency control mechanisms, and some systems [62] can even compose federated databases into a hierarchical

structure. Unlike the work described in this dissertation, federated databases are motivated primarily by functionality requirements (i.e., the ability to execute transactions across different databases) rather than performance: indeed, they can perform *worse* than local DBs.

Some works in this category partition data so that global serializable execution is directly guaranteed if transactions are per-partition serializable, removing the need for cross-partition concurrency controls. For instance, the work on local atomicity properties [90] explores the required properties of per-object concurrency control needed to guarantee database-level serializability. Likewise, Sha et al.[79] partitions the database into atomic datasets, each with its own concurrency control, such that no consistency invariant spans multiple datasets. Database-level consistency then directly follows from per-dataset serializability. These approaches suffer from two main limitations: first, they place stringent constraints on how they allow data to be partitioned and concurrency controls to be combined. Second, they require all conflicts on the same data partition to be handled by the same concurrency control.

**Partitioning by Time** Certain systems choose to partition the execution of transaction protocol into distinct phases during which different concurrency control mechanisms can execute. For example, Granola [42] uses time partitioning to assign different protocols to different types of transactions. Specifically, Granola switches between a *timestamp mode* and a *locking mode*. In timestamp mode, it runs single-sited and *independent* transactions (i.e., transactions

whose commit / abort decision is deterministic) using an efficient lock-less timestamp-based protocol. Transactions that require coordination are instead constrained to execute in the less efficient *locking mode* that relies on traditional two-phase locking. Likewise, Doppel [68] is a multi-core main-memory database that distinguishes between joined, split and reconciliation phases. Transactions in the joined phase use traditional optimistic concurrency control. Transactions in the split phase, assuming their operations commute, are guaranteed never to conflict: instead, they modify split per-core state that is subsequently merged in the reconciliation phase.

**Hierarchical decomposition** Multi-level serializability [28, 77, 91–93] observes that transactions can be hierarchically decomposed in a tree such that each level captures operations at a different level of abstraction: operations that appear to be atomic at a given level may be in fact be implemented by a collection of operations at the preceding level, with different concurrency controls used at different levels. A sufficient condition to ensure serializability in this context is level-by-level serializability [27]: assuming conflicting operations at level $i+1$ generate at least one conflicting operation at level $i$, serializability is guaranteed if the serialization graph between levels $i$ and $i+1$ is acyclic. Intuitively, this means that if a level orders conflicting operations, the corresponding operations at higher level should be ordered consistently. In the context of multi-level serializability, individual concurrency controls regulate the interactions of *all* transactions at a given level. In contrast, in our work,

as I will show later, concurrency controls are responsible only for a subset of transactions.

## 3.3 Modular Concurrency Control

The starting point of my work is *Modular Concurrency Control* (MCC), a technique introduced by my co-authors and me in the Callas database [95]. Unlike much of the work surveyed earlier in this chapter, MCC is not limited to specific partitioning of transactions / conflicts, or specific concurrency control combinations [36, 42, 68, 82, 90], but aims to provide a general approach to federating concurrency controls. In this section, I will give a brief overview of MCC, and introduce its design and implementation in the Callas database.

Callas' Modular Concurrency Control implementation partitions transactions in *groups*, allowing each group to run its own private concurrency control mechanism. Each mechanism is charged with regulating concurrency only for the transactions within its group. As such, one can improve concurrency by choosing the best-suited existing mechanism for each group, or even by designing new and more aggressive mechanisms. Besides these *in-group* concurrency control mechanisms, Callas's MCC provides a special *cross-group mechanism* to handle data conflicts that happen between transactions from different groups to ensure isolation of all transactions.

Modular Concurrency Control emphasizes its *generality* and *modularity*. It imposes no restriction on the transactions that it handles, or on how to partition them. In principle, the MCC approach does not depend on the

choice of the in-group concurrency controls, and how they are implemented: no matter locks or OCC, single-versioned or multi-versioned. As long as the isolation property holds within each group, MCC guarantees that it will also hold among all transactions. In practice, the implementation in Callas only explored an instance of the MCC design that works with single-version concurrency controls. Indeed, realizing MCC's full generality is a key contribution of this dissertation.

I will next quickly discuss the design and implementation of the in-group and cross-group layer in the Callas database, and finally, describe how Callas groups transactions.

### 3.3.1 In-group Layer

This is the layer that gives Callas its performance benefits over monolithic concurrency controls.

Concurrency control mechanisms in this layer have two goals. First, they must ensure in-group correctness, namely, that transactions in each group are isolated. Second, they are dedicated to optimize the performance of transactions in their groups.

Following the isolation definition from Adya's graph-based model [23], Callas formally defines *in-group correctness* as following [95]:

**In-group Correctness** *Concurrency controls in a group G must prevent Aborted Reads and Intermediate Reads between transactions from G, and pre-*

*vent Circularity if all transactions in the cycle are in G.*

To optimize in-group performance, Callas explored both existing (transaction chopping [80, 99]) and new (runtime pipelining [95]) techniques as in-group mechanisms. As I have mentioned in Section 2.3, both techniques improve concurrency by splitting transactions into multiple pieces, which reduces transactions' locking periods and exposes their intermediate states. Splitting, however, can only occur if certain conditions are met. In particular, transaction chopping requires the absence of SC-cycles, while runtime pipelining requires transactions to access data objects in the same order. These requirements are hard to meet when they must apply to all transactions in a workload, which often reduces the effectiveness of these techniques. Modular Concurrency Control limits the scope of these requirements only to the transactions within each group, greatly improving their applicability.

### 3.3.2 Cross-group Layer

The goal of this layer is to regulate data conflicts across different groups, so that all transactions are isolated properly. To achieve this, the cross-group layer needs to ensure the following *cross-group correctness* property [95]:

**Cross-group Correctness** *Aborted Reads and Intermediate Reads must be prevented between transactions from different groups. Circularity must also be prevented if at least two transactions in the cycle come from different groups.*

Callas' cross-group layer employs a single concurrency control mecha-

nism based on two-phase locking. The key technique used by this mechanism is a new type of locks called *nexus locks* [95]. As in two-phase locking, before any transaction can access any data object, it must first acquire a nexus lock on that data object, and these locks are not released until the transaction commits. But unlike the traditional locks in 2PL, the behavior of nexus locks depends on which group the transactions that are accessing the data object belong to:

- If two transactions from different groups are trying to acquire the same nexus lock, one of them must wait, unless both of them are reading that data object.
- If transactions are from the same group, they can acquire the same nexus lock simultaneously.

Intuitively, a nexus lock only regulates conflicting data accesses from different groups, and it always allows transactions from the same group to access the data object concurrently—it is up to their in-group mechanism to handle such data conflicts.

However, these rules alone are not enough to ensure cross-group correctness. As it regulates conflicts between transactions in different groups, the cross-group mechanism may develop, by transitivity, dependencies between two transactions of the same group. Circularity can happen if the order implied by these transitive dependencies conflicts with the order assigned to these transactions by their in-group mechanism. It is possible to prove [95]

that correctness can be ensured by adding the following requirement to the management of nexus locks:

**Nexus Lock Release Order** *If transaction $T_1$ and $T_2$ are from the same group, and $T_2$ depends on $T_1$ within the group, then $T_2$ cannot release its nexus locks until $T_1$ does.*

### 3.3.3 How to Group Transactions

Modular Concurrency Control does not constrain how to group transactions, or what concurrency controls to use in each group—these are simply considered as configuration choices. In practice, however, configuring MCC can be complicated, since there are exponentially many different ways to partition transactions with respect to the number of transactions.

Callas offered some basic heuristics for configuring MCC. First, it recommended to place in separate groups, with specialized concurrency control mechanisms, only the few transactions that are performance critical—all other transactions can be kept in a single group, regulated by a simple 2PL mechanism.

Second, it proposed an iterative algorithm to group transactions. In each iteration, the algorithm identifies transactions that are suffering from heavy data contention, and tries to improve performance by moving them to different groups, or by creating new groups for them. To detect heavily contended transactions, the algorithm increases the workload's request rate,

and looks for transactions whose latencies increase much faster than other transactions.

These guidelines are good starting points for automatic configuration. But as we will see later in Section 3.4.2, they are still very elementary, and need to be improved.

## 3.4 Problems in the Current MCC Design

Callas' design and implementation of MCC are not flawless. We identify two problems. First, Callas' inflexible and conservative cross-group mechanism may limit its performance. Second, Callas does not fully address MCC's configuration complexity, which could become a major challenge in making this technique practical. This section discusses them in detail.

### 3.4.1 Limitation of the Inflexible Cross-group Mechanism

Callas assumes that an application's transactions can be cleanly partitioned into groups such that conflicts across groups are rare or inconsequential to performance. Consequently, its prescription to coordinate different groups is to complement those efficient in-group concurrency control mechanisms with a single, catch-all mechanism based on two-phase locking.

The findings in my work, however, tell a different story. We find that combining aggressive in-group optimizations with an inflexible, conservative cross-group mechanism exposes Callas' embodiment of MCC to a dilemma. On the one hand, in-group mechanisms, to be effective, must handle a very specific,

| Same group | Separate - Deadlock | Separate - No Deadlock | Separate - No Conflict |
|---|---|---|---|
| $3{,}207 \pm 1$ | $158 \pm 9$ | $3{,}598 \pm 14$ | $23{,}834 \pm 5$ |

Table 3.1: Impact of grouping on throughput (txn/sec).

and hence narrow, subset of conflicts. On the other, pushing the remaining conflicts to the cross-group layer can cripple Callas' conservative concurrency control mechanism, and in turn the performance of the whole system.

The example from TPC-C that I introduced earlier in this chapter (Section 3.1 and Figure 3.1) highlights this dilemma. Results are shown in Table 3.1. The first column shows the throughput of running `stock level` and `new order` in the same group (using runtime pipelining as the in-group mechanism). As we discussed, this arrangement creates circular dependencies that void much of the potential benefit of runtime pipelining. Perhaps surprisingly, placing these transactions in separate groups (using 2PL as cross-group concurrency control mechanism) yields no benefit: throughput actually *drops* by an order of magnitude, as runtime pipelining's preferred ordering of read-write accesses (Section 3.1) creates deadlocks at the cross-group level.

Removing these deadlocks by reordering `new order`'s access to the `district` and `stock` table (third column) improves performance somewhat, but the result is still only marginally better than placing the transactions in the same group. To offer a sense for the role that the cross-group mechanism plays in determining these results, the last row shows the throughput of a best-case scenario for partitioning. `Stock level` and `new order` are placed in separate groups, and artificially restricted to accessing different warehouses as

a way to eliminate all cross-group read-write conflicts: performance soars by almost an order of magnitude over the value reported in the third column.

These results suggest three observations. First, *cross-group conflicts matter*. The performance bottleneck in our example is the 2PL concurrency control mechanism at the cross-group layer, which cannot efficiently handle the read-write conflicts on the `district` table between the two transactions. Second, *the cross-group mechanism matters*. These read-write conflicts would have been better handled using a multi-versioned concurrency control mechanism. Third, *no single cross-group mechanism can effectively address all conflicts*. This same multi-versioned concurrency control would fare poorly under write-write cross-group conflicts.

### 3.4.2 Revisiting MCC's Configuration Complexity

MCC's performance edge comes from the discrimination of different data contention in a workload, and, as we have seen, this is achieved by properly configuring the grouping of transactions and in-group mechanisms. Consequently, MCC's performance largely depends on its configuration. A good configuration can improve the performance by properly partitioning transactions to separate major contention bottlenecks, and applying the right concurrency control mechanism to handle each of them. A bad one, however, may miss such opportunities, or even harm the performance, e.g., by introducing additional computational costs or deadlocks.

But configuring MCC is not a trivial task, and if this task is left to

database users, it can become a major burden.

First, to find a good configuration, one must thoroughly explore the application and its workload to understand its performance characteristics and potential bottlenecks. As we know, reasoning about performance requires advanced skills and substantial efforts from application developers; this is especially true for database workloads that involve extensive concurrent execution.

Second, the MCC has itself many "turning knobs", so configuring it can be very complicated. On the one hand, there are exponentially many different ways to partition transactions and assign concurrency control mechanisms to each group. On the other hand, the performance of a configuration is often hard to predict. As we have seen from the example in Section 3.4.1, there are many factors that can affect MCC's performance: they include not only how well each concurrency control can handle its data conflicts, but also the interaction between different mechanisms, which may end up producing deadlocks.

Third, configuring MCC manually also requires users to be familiar with both the MCC technique and the specific concurrency control mechanisms that the database supports. As concurrency control techniques are buried deep in the implementation of a database, rather than part of its API, it is unreasonable to ask users to master such domain-specific knowledge.

Indeed, if left unsolved, the complexity of configuring MCC may void a key motivation for introducing this technique in the first place, namely, to improve transactions' performance *without sacrificing ease of programming.*

Callas offered some basic guidelines to group transactions (Section 3.3.3), but they are very elementary, and suffer from several problems. For example, we find that the profiling technique proposed in Callas cannot reliably detect the performance bottleneck (see Section 5.3.1). Also, these guidelines assume the ability to adjust workload parameters, which may not hold if the configuration algorithm is integrated into the database system to transparently manage its configuration *on-the-fly*. In reality, configuring MCC is still largely an ad-hoc procedure that involves a lot of human efforts, and this is especially true after Tebaldi introduces hierarchical MCC, whose multi-layer structure to federating concurrency control further complicates the configuration.

## 3.5 From Callas to Tebaldi

To address problems in the current MCC design, we present Tebaldi [85], a distributed key-value store that takes significant steps towards harnessing the performance opportunities offered by federating different concurrency control mechanisms. Comparing to Callas, Tebaldi makes two major improvements. First, it proposes a new, hierarchical approach to MCC. Second, it is capable to automatically manage its configuration of MCC.

**Hierarchical MCC** Tebaldi employs *hierarchical MCC* to address the problem that Callas' inflexible, conservative cross-group mechanism can limit the performance of the entire federation. The design of hierarchical MCC starts with the simple premise that the key to performance is, once again, a federation

of concurrency control mechanisms, this time deployed to resolve *cross-group* conflicts. The vision of this approach is to further increase the *flexibility* in how data conflicts are handled in the federation, which, in turn, improves performance.

To realize this vision, we need to clear several technical hurdles. First, we need to determine the inner structure of the new federations that Tebaldi enables: our goal is to ensure that these new degrees of freedom do not come at the expense of modularity. Second, we need to identify the conditions that ensure the correctness of Tebaldi's more general federations. Finally, we need to develop the system-level support needed to bring this vision to fruition.

In Chapter 4, I will address these challenges, and discuss the design, implementation and evaluation of hierarchical MCC in Tebaldi.

**Automatic Configuration** To mitigate the challenge in configuring hierarchical MCC, we equip Tebaldi with the ability to manage its own configuration. The key technique is an automatic configuration algorithm that can diagnose performance issues on-the-fly, and reconfigure MCC to improve the performance.

Our algorithm takes an iterative approach that is similar to the initial proposal in Callas, but addresses many challenges that were not solved in Callas. It adopts a new profiling technique that can detect and describe performance bottlenecks more reliably and more accurately, and we propose new strategies to adjust the federation that can better fit Tebaldi's hierarchical

approach to MCC. Our algorithm is fully integrated into the Tebaldi database, runs in real time, and requires minimal, if any, user-involvement.

In Chapter 5, I will discuss the detail of automatic configuration in Tebaldi, and evaluate its performance.

# Chapter 4

# Tebaldi's Hierarchical Approach to MCC

In this chapter, I will present *hierarchical MCC*, Tebaldi's new approach to federating concurrency controls for better performance. I will first introduce the theoretical foundations of hierarchical MCC, and then go through its design and implementation in Tebaldi, highlighting its benefits and limitations. Finally, I quantify the benefits of this new approach.

## 4.1 Overview of Hierarchical MCC

To harness the full power of the federated approach, Tebaldi seeks to maximize its flexibility in federating concurrency controls while preserving modularity. The benefits of *flexibility* are clear: finer control in determining the mapping between sets of conflicts and concurrency control (CC) mechanisms enables greater concurrency and higher performance. Unhinged flexibility can, however, come at the cost of *modularity*, defined as the ability of individual CCs to order conflicts independently while guaranteeing isolation.

This chapter, as well as part of Chapter 3, is based on the paper *Bringing Modular Concurrency Control to the Next Level* by Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi and Chao Xie, which was published in *SIGMOD 2017*. I led this research project, and made major contributions to the design, implementation, and evaluation of the Tebaldi database.

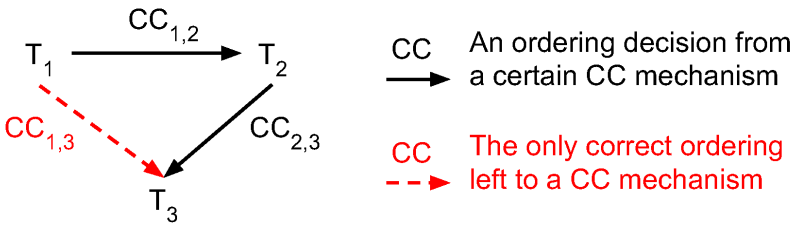



Figure 4.1: Unhinged flexibility in a federation may harm its modularity.

Consider, for instance, a set of three transactions, $T_1$, $T_2$, and $T_3$, as shown in Figure 4.1, and assume that, to maximize flexibility, conflicts between each pair of transactions are governed by a separate CC mechanism. Suppose $CC_{1,2}$ orders $T_1$ before $T_2$, and that $CC_{2,3}$ orders $T_2$ before $T_3$. $CC_{1,3}$ is then left with only one correct choice, i.e., ordering $T_1$ before $T_3$, and it needs to become aware of that (as otherwise a circle will be created in the Direct Serialization Graph). In general, a CC mechanism may have to learn the ordering decisions of all other CCs to guarantee correctness.

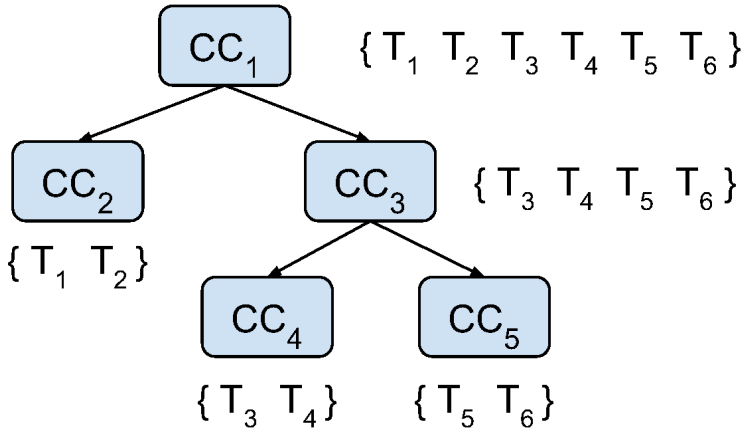



Figure 4.2: Tebaldi's hierarchical approach to Modular Concurrency Control.

Tebaldi balances these competing concerns by applying the theory of Modular Concurrency Control [95] recursively, organizing CC mechanisms in a multi-level tree, as shown in Figure 4.2. Each node in the tree is a concurrency control mechanism, and it is associated with a set of transactions $\mathcal{T}$. The node is then responsible for regulating data conflicts among that set of transactions. Initially, the root node is assigned with all transactions; in turn, a non-leaf node can choose to *delegate* some of its responsibility by assigning disjoint subsets of $\mathcal{T}$ to children nodes better suited to handle their conflicts, while only retaining responsibility for regulating conflicts across children.

We call this new approach to federating concurrency controls *Hierarchical Modular Concurrency Control*, or *hierarchical MCC / HMCC* for short. The hierarchical refinement of MCC yields two key benefits. On the one hand, it enables greater flexibility: applying MCC recursively largely removes the concern that using aggressive in-group mechanisms may unnecessarily push conflicts to the conservative cross-group CC. Instead, these "cross-group" conflicts are themselves further partitioned and mapped to various efficient CCs—and so on recursively, until one reaches the root of the tree. On the other hand, Tebaldi's multi-level tree preserves a high-degree of modularity by retaining a key feature of MCC: the mapping from sets of data conflicts to CCs is derived by subdividing a given set of transactions into *mutually disjoint* subsets. This makes it impossible for sibling nodes on Tebaldi's tree to make conflicting ordering decisions, as they regulate disjoint portions of the Direct Serialization Graph. Instead, CCs need only communicate their ordering choices to (and, in

turn, have their ordering choices constrained by) the CC mechanism of their parent node. As we will see in Section 4.2, this structural property is instrumental to guaranteeing that no two concurrency controls can make conflicting decisions on how to order a pair of transactions.

## 4.2 Ensuring Correctness

Tebaldi builds upon Adya's [22] general theory for expressing isolation. As we saw in Chapter 2, this theory associates with every execution a *direct serialization graph* whose nodes consist of committed transactions and whose edges mark the dependencies (write-read, write-write, or read-write) that exist between them. An execution satisfies a given isolation level if it disallows three properties: *aborted reads*, *intermediate reads*, and *circularity*.

In the spirit of modularity, the MCC implementation in Callas [95] articulates these global requirements separately for in-group and cross-group mechanisms. In-group mechanisms must prevent circularity, aborted reads, and intermediate reads that solely involve the subset of transactions that they are responsible for. Cross-group CCs must prevent cycles, as well as aborted and intermediate reads, involving transactions from different groups.

Tebaldi blurs the distinction between cross-group and in-group mechanisms: every concurrency control in the CC tree acts as an in-group mechanism in the eyes of its parent, and as a cross-group mechanism in the eyes of its children. Correctness can then simply be defined as follows:

**Definition 4.2.1.** *A CC tree is correct if every concurrency control in the tree prevents aborted reads, intermediate reads, and circularity for the committed transactions in its group.*

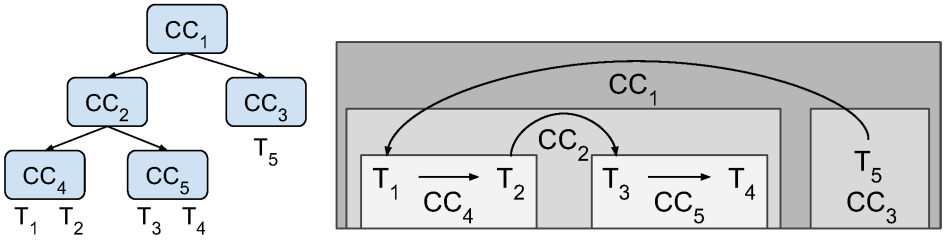
As it is, this definition says little about *how* CCs can achieve the correct isolation in a modular yet flexible fashion. In Callas, this is achieved by introducing a single, 2PL-based *instance* of cross-group mechanism. Tebaldi is more sophisticated: its goal is to establish a general model of federation that allows a variety of CC mechanisms to regulate conflicts across groups (i.e., to serve as an internal node of the HMCC tree). We then need a *general condition* on the composability of CC mechanisms, so that any combination of CC mechanisms that meet the condition will isolate transactions correctly.

### 4.2.1 Consistent Ordering

The correctness condition that we propose is based on the key concept in hierarchical MCC—delegation: a parent concurrency control delegates conflicts that its child is better suited to handle. So the parent's only responsibility is then to ensure that subsequent ordering decisions will be *consistent* with those of its children. Formally:

**Consistent Ordering** *For committed transactions $T_1$ and $T_2$, if a concurrency control node in the tree creates a path from $T_1$ to $T_2$ in the DSG, then its parent CC node must never create a path from $T_2$ to $T_1$ among the larger set of transactions that it manages.*

In effect, each subtree in the CC tree defines a partial ordering on the subset of transactions that it manages. Outer concurrency controls merge different subtrees, extending the partial order and ensuring that the resulting transaction ordering does not violate circularity.




(a) A 3-layer CC tree. (b) Each CC's Ordering decisions.

Figure 4.3: Ordering responsibilities of each CC mechanism in the tree.

We illustrate this process in Figure 4.3. $CC_4$ orders transactions $T_1$ and $T_2$ and $CC_5$ orders transactions $T_3$ and $T_4$. $CC_2$'s only responsibility is to ensure that $T_1/T_2$ and $T_3/T_4$ are ordered consistently—in this case, by ordering $T_3$ after $T_2$. Similarly, $CC_1$ orders $T_5$ before $T_1$.

Most off-the-shelf concurrency control mechanisms, however, are not quite as diligent as $CC_1$ and $CC_2$. In general, the parent CC can constrain the ordering decisions between transactions assigned to one of its children both directly (e.g., by timestamping transactions at their start time) and indirectly (by making cross-group ordering decisions that limit, in the service of correctness, a child CC's discretion in deciding how to order its own transactions). Ignoring these effects can lead to violations of consistent ordering.

For example, Consider a non-leaf node, $CC_n$, in our tree, which runs

2PL. Assume $CC_n$ has two children CCs: $CC_1$, in charge of ordering transactions $T_1$ and $T_2$, and $CC_2$, in charge of $T_3$. Suppose $CC_1$ serializes $T_1$ before $T_2$, and that $T_2$ commits first, releasing all its 2PL locks at $CC_n$. Nothing now prevents $CC_n$ from letting $T_3$ read a data object written by $T_2$, thus forming a cross-group dependency $T_2 \rightarrow T_3$. Similarly, nothing forbids $T_3$ from writing some data object $x$ and committing (thus releasing all locks at $CC_n$). Suppose $T_1$ now reads $x$, causing a write-read cross-group conflict at $CC_n$ between $T_3$ and $T_1$. If $CC_n$ orders $T_1$ after $T_3$, its two ordering decisions ($T_2 \rightarrow T_3$ and $T_3 \rightarrow T_1$) create a path from $T_2$ to $T_1$, violating consistent ordering.

### 4.2.2 Preserving Consistent Ordering in CC Mechanisms

We identify three general strategies by which nodes at adjacent levels of the CC tree can coordinate their ordering decisions and preserve consistent ordering. When a parent CC is faced with an ordering decision, it can either straightforwardly *adopt* the decision of one of its children CCs, take actions that *constrain* the future ordering decisions of its children, or *procrastinate*, leaving more time for its children to propose an ordering. The choice among these strategies largely depends on the timing of the decision and on the specifics of the parent's CC mechanism; indeed, mechanisms that take multiple steps to reach their final ordering decision may use a combination of these strategies.

When the parent CC's ordering decision comes *after* a child CC has decided how to order the transactions in its group, the simplest strategy to

ensure consistent ordering is to fully embrace Tebaldi's emphasis on delegation and adopt the child's ordering decisions. This strategy proves particularly useful when the parent CC orders transactions at `commit` time, since by then it is likely to know its children's ordering decisions. Consider again the above example where $CC_n$, the parent CC, runs 2PL. $CC_n$, once it learns of its children's ordering decisions (e.g., $T_1 \rightarrow T_2$), must simply respect that ordering when committing transactions (so that the locks held by $T_2$ are only released after $T_1$ commits). This is exactly the *Nexus Lock Release Order* [95] that we adopted in Callas (See Section 3.3.2).

When instead the parent CC's ordering decision comes *before* the child has had time to decide, two strategies remain: constraining the child's decisions, or procrastinating until the child decides (and then adopting its decisions).

The constraints imposed by the parent CC can vary from subtle to overbearing. At the subtle end of the spectrum, how a parent resolves a cross-group conflict (for instance, by blocking conflicting transactions across groups in 2PL and runtime pipelining) can often indirectly limit how the children CCs can serialize these transactions. At the other end of the spectrum, a parent CC could simply dictate the order of transactions to its children (for example, Timestamp Ordering [32] assigns each transaction a unique timestamp at begin time). This degree of micromanagement, of course, would run counter to Tebaldi's design, which leverages delegation as the key to greater performance. Nonetheless, such CCs can serve effectively as inner nodes in Tebaldi's

hierarchy by selectively *procrastinating* ordering decisions until their children make theirs. For example, rather that labeling each transaction instance with a unique timestamp, the parent CC could assign the *same* timestamp to a batch of transactions from the same group. By waiving its chance to order these transactions, the parent would in effect delegate their ordering to the child CC responsible for that group, while still constraining the child by preventing it from ordering the transactions in batch $i + 1$ before those in batch $i$.

In Section 4.4, we will discuss in detail how Tebaldi uses adoption, constraining, and procrastination to integrate several widely used CCs in its hierarchical architecture.

## 4.3 Tebaldi's Design

Having sketched out the correctness requirements of hierarchical MCC, we describe next how Tebaldi, our new transactional key-value store, enforces these requirements. Similarly to several distributed, disk-based, commercial systems [10, 26, 40], Tebaldi separates its concurrency control logic from its storage management and keeps metadata associated with CC protocols (like timestamps and version lists in snapshot isolation, and locks in 2PL) as transient state in the concurrency control module.

The *concurrency control module* coordinates how the diverse CC protocols in Tebaldi's hierarchy collectively determine the order of transactions. Tebaldi's framework for CC coordination leverages the observation that, despite their diversity, the steps that most CC protocols take in determining the

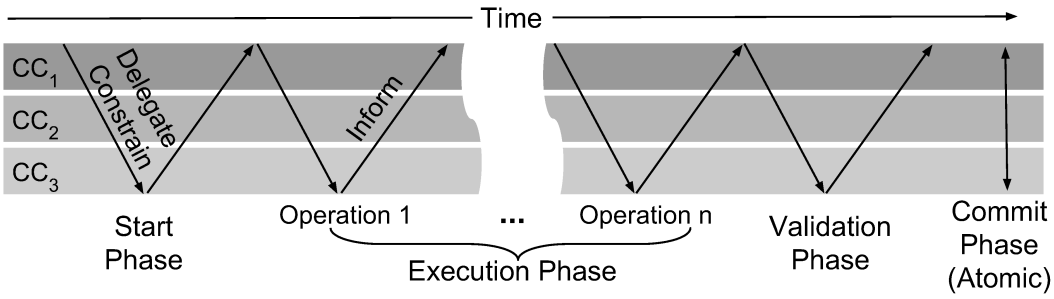



Figure 4.4: An overview of Tebaldi's transaction execution protocol.

ordering of a transaction $T$ can be grouped into four distinct phases: a *start phase*, an *execution phase*, a *validation phase*, and a *commit phase*. Tebaldi executes each phase in two passes, as shown in Figure 4.4. The first pass, top-down, gives parent nodes the opportunity to constrain how their children's ordering decisions affect the ordering of $T$; the second pass, bottom-up, lets children (optionally) inform their parent of $T$'s current *dependency set*, i.e., the list of transactions in its group on which $T$ directly depends.

This structure gives Tebaldi its generality: Tebaldi can support a maximum of CC combinations by giving every concurrency control, in each phase, the opportunity to constrain or delegate to its child, while the child can in turn inform its parent as soon as dependencies become known. This generality does not come at the cost of modularity: the implementation of each concurrency control remains independent from that of its parents (or children) as they communicate only via well-defined communication channels. Further, Tebaldi is extensible: it provides a blueprint for adding a new CC to an existing CC hierarchy tree: all that is required is to identify and integrate the new CC's

four phases in the tree.

The *storage module* stores and retrieves the appropriate version of a data object according to the CCs' ordering decisions. To support both single version and multiversion concurrency control protocols, this module is implemented as a multiversion storage that keeps all the committed and uncommitted writes on each object.

Naturally, there might be exceptions: some specialized concurrency controls may not fit well in Tebaldi's four-phase model or may expect a specific storage layout. Likewise, there may be inner CCs whose ordering decisions are not known at the time when parent CCs need them (or even at commit time). We discuss the limitations of our approach in Section 4.3.2.

### 4.3.1 Execution Protocol

Executing a transaction $T$ in Tebaldi requires coordinating the actions of all the CCs handling $T$. These CCs form a path $\pi$ in the CC tree (starting at the root and ending in $CC_n$) as each CC delegates some of $T$'s conflicts to a child. Tebaldi executes $T$ as follows:

**Start phase** In the top-down pass, each CC on $\pi$ allocates the specific metadata that it requires. A CC may, for example, initialize data-structures that either uniquely identify the transaction or order it relative to concurrently running transactions (e.g., start timestamps in serializable snapshot isolation and timestamp ordering, or transaction id in lock-based protocols). More complex

protocols, like Calvin [86], may also use the start phase to batch transactions, pre-ordering transactions within a batch. At the end of the phase, the bottom-up pass, starting from $CC_n$, lets children inform their parent of $T$'s new dependency set.

**Execution phase** In this phase, each CC along $\pi$ runs its execution phase for each read and write operation in $T$. In doing so, CCs refine $T$'s position in the overall transaction schedule. For example, choosing to read from a version created by a transaction $T_i$ orders $T$ after $T_i$, while inserting a new object version before $T_j$ orders $T$ before $T_j$.

In the top-down pass, each CC executes its own CC-specific logic and appropriately constrains the ordering decisions of its children by blocking (or aborting) operations. Lock-based systems, for instance, delay operations until conflicting locks have been released, before placing a lock on the chosen object and executing their children's execution phase. The process is similar for multiversioned systems: though these techniques do not require blocking, they may decide to abort $T$ on write-write conflicts.

The bottom-up pass has two components. First, as in the start phase, a child can forward $T$'s dependency set to its parent. Second, the CCs on $\pi$ collaboratively identify the appropriate version to return on a read operation. Specifically, a child CC proposes on a read operation a "candidate" version to return. Its ancestors can then amend the child CC's proposal based on transactions that may have written to that same object in sibling groups.

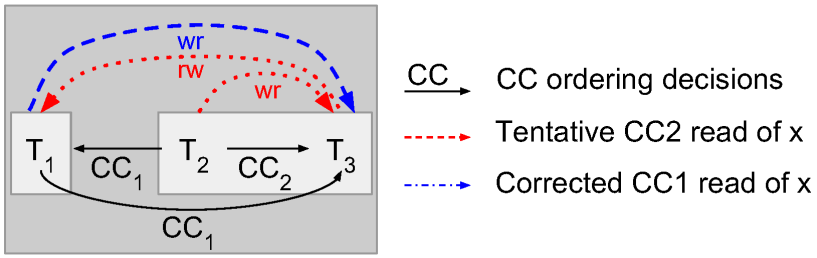



Figure 4.5: Read logic in the bottom-up pass of execution phase.

To illustrate, consider the CC tree in Figure 4.5. $T_1$ is in one group, while $T_2$ and $T_3$ are in another (controlled by $CC_2$); the interaction between the two groups is regulated by the cross-group concurrency control $CC_1$. Suppose transactions $T_1$ and $T_2$ both write object $x$ (producing, respectively $x_1$ and $x_2$) and that $CC_1$ chooses to order $T_1$ after $T_2$ but before $T_3$ (solid edges). Now consider a read of $x$ by $T_3$: as $T_1$ and $T_3$ are in different groups, $CC_2$ is unaware that $T_1$ wrote $x_1$. It thus proposes $T_2$'s write $x_2$ as a candidate read value (red dashed edges). Returning this value would create a cycle in the final transaction schedule that consists of an anti-dependency edge from $T_3$ to $T_1$ (as $T_3$ misses $T_1$'s write) and an ordering edge from $T_1$ to $T_3$. $CC_1$ thus "corrects" $CC_2$ and instead returns $x_1$, removing the cycle (blue dotted edges). Importantly, $CC_2$ never becomes aware of the existence of $x_1$. Restricting $CC_2$ to this partial view is necessary to preserve Tebaldi's modularity, which hinges on concurrency controls making ordering decisions solely for transactions in their group.

**Validation phase** This phase makes the ultimate decision [33] on whether $T$ can commit and on its position in the final transaction schedule. In the top-down pass, each CC along $\pi$ determines whether $T$ is committable and, if desired, constrains its children's ordering decisions by delaying their validation phase. In the bottom-up pass, starting from the last CC on $\pi$, each CC forwards to its parent either $T$'s dependency set or its decision to abort $T$. The parent CC can in turn use that information to determine whether it can commit $T$ in an order consistent with its child's decision.

The process of deciding whether $T$ can commit varies widely across CCs. Validation is trivial in lock-based protocols, as having acquired all locks is sufficient to ensure commit. Optimistic protocols must instead verify whether the objects read by $T$ are still current, or otherwise abort $T$. Likewise, the ease with which the full set of dependent transactions can be reported also varies. In most single-version systems, $T$ always knows its dependency set by the end of the validation phase. In contrast, that information is not available in multi-versioned CC mechanisms like SSI [50] until all transactions that were concurrent with $T$ have committed.

**Commit phase** Tebaldi guarantees that $T$ is committed atomically across all CCs by ensuring that the chained commit phases execute uninterrupted, starting from the leaf CC on $\pi$.

### 4.3.2 Limitations

Tebaldi's framework strives to be general, modular, and extensible. These benefits, however, come at the cost of some efficiency, as, unlike systems designed to work solely with a fixed subset of CCs, Tebaldi cannot co-design components. Single-versioned concurrency controls, for instance, do not need to keep version histories, whereas Tebaldi's storage module must store them to support multi-versioned CCs. Similarly, while many specialized systems can benefit from co-locating concurrency controls' metadata (such as locks and timestamps) with the actual data, Tebaldi's generality requires their separation. Finally, some CCs, such as 2PL, do not need a validation phase.

Moreover, Tebaldi's ability to incorporate a given CC is based on the assumption that the CC can be expressed using its protocol's four phases and modified to guarantee consistent ordering (Section 4.2). These assumptions, though valid for common concurrency controls, may not hold universally; and, even when they do, may reduce efficiency. For example, batching increases the probability of write-write conflicts in snapshot isolation (Section 4.4, Section4.6.4) and may reduce the scheduling flexibility of time-travelling concurrency controls like TicToc [97]. Also, some CC mechanisms require their children to report dependency information at certain phases (e.g., at execution / validation phase) to enforce consistent ordering, but not all children CCs can meet these requirements. In such cases, the two CC mechanisms will not be able to work together as parent and child.

## 4.4 Use Cases

This section sketches the different concurrency control protocols currently supported by Tebaldi. This initial selection achieves a dual purpose. First, it illustrates how one can guarantee consistent ordering for real, well-known concurrency controls and how these CCs can be implemented in Tebaldi. Second, it speaks to the generality of our approach: Tebaldi supports two lock-based, single-versioned protocols (traditional two-phase locking [30, 48] and the recently proposed runtime pipelining [95]), and two multiversioned protocols (serializable snapshot isolation [35, 50, 72], and multiversioned timestamp ordering [76]). We describe each mechanism in turn.

### 4.4.1 Two-Phase Locking (2PL)

Our implementation directly follows the original algorithm [30, 48]: transactions acquire shared read locks when executing a read operation and exclusive write locks when executing write operations. Every transaction holds these locks until commit, so as to guarantee serializability. Any deadlocks are handled by timing out transactions.

Implementing 2PL as a non-leaf CC requires only two small changes to the algorithm, which we have described in the Callas paper [95]. First, 2PL delegates in-group concurrency control to its children CCs by marking all locks acquired by transactions from the same group as non-conflicting. Second, 2PL ensures consistent ordering by delaying a transaction's commit until all its in-group dependencies have also committed.

2PL takes no action in the start phase. All necessary locks are acquired in the top-down pass of the execution phase. The bottom-up pass of the execution phase decides the appropriate read version to return: 2PL accepts the child's proposal if it is an uncommitted value from its group or else returns the latest committed value. The validation phase then gathers the committing transaction's dependency set from the child CC and delays commit until all transactions in that set have committed. Finally, it releases the locks in the commit phase.

### 4.4.2 Runtime Pipelining (RP)

As we have introduced in Chapter 2, Runtime pipelining [95] handles data conflicts more efficiently than 2PL through a combination of static analysis and runtime constraints. Specifically, RP first statically constructs a directed graph of tables, with edges representing transactional data / control-flow dependencies, and topologically sorts each strongly connected set of tables. Transactions are correspondingly reordered and split into steps, with step $i$ accessing tables in set $i$. A runtime pipeline ensures isolation: once $T_2$ becomes dependent on $T_1$, $T_2$ can execute step $i$ only once either $T_1$ has terminated, or $T_1$ is executing a step larger than $i$. Operations within a step are isolated using 2PL.

RP's start phase initializes the step counter and dependency set. In the execution phase, RP delegates concurrency control within each group to the child CC by allowing transactions from the same group to execute the

same step concurrently. Upon starting a new step $i$, RP first "commits" the previous step by releasing the step-level lock (after in-group dependencies have also step-committed). It then waits both for all cross-group dependencies to *finish* executing step $i$, and for all in-group dependencies to *start* executing step $i$, before acquiring the step-level lock. The bottom-up pass of the execution phase gathers in-group dependency reports from children CC. It also decides the appropriate read version to return: RP accepts the child's proposal if it is a write from its group that has not step-committed. Otherwise, it returns the latest step-committed value. The validation phase delays a transaction's commit until transactions in its dependency set have committed.

### 4.4.3 Serializable Snapshot Isolation (SSI)

Tebaldi supports a distributed implementation of serializable snapshot isolation [35, 50, 72], a multiversioned protocol that rarely blocks readers. As we have mentioned in Chapter 2, SSI orders transactions using start/commit timestamps: transactions read from a snapshot at the start timestamp, while writes become visible at the commit timestamp. SSI ensures serializability by detecting (and preventing) "pivot" transactions that have both incoming and outgoing anti-dependency edges.

Enforcing consistent ordering in SSI requires care. Firstly, unlike 2PL and RP, SSI partially decides transaction ordering through start timestamp assignment. Consider for example two transactions $T_1$ and $T_2$ from the same group, with $T_1$ having a smaller start timestamp. Consistent ordering can

be violated if the child CC orders $T_2$ before $T_1$, as $T_2$ may observe a write from another group that $T_1$ cannot see. To address this, Tebaldi uses *batching*. Instances of transactions from the same group are placed in a batch and assigned the *same* start timestamp, delaying their relative ordering until commit. A child CC is then free to order batched transactions without violating consistent ordering. Though transactions in a batch share a start timestamp, they can commit individually with different commit timestamps (once all their in-group dependencies have already committed). Introducing grouping and batching means that SSI must detect and prevent *pivot batches*, with both incoming and outgoing anti-dependencies.

The start phase assigns the batch's start timestamp to the transaction (determined by a centralized timestamp server). During the execution phase, SSI tracks pivot batches by asynchronously querying a *group manager* that keeps track of batches' anti-dependencies. Cross-group write-conflicting transactions are aborted. In the bottom-up pass of the execution phase, SSI decides on the appropriate read version: SSI accepts the child's proposal if it is a value from its own batch, and otherwise returns the latest committed version whose commit timestamp is smaller than the transaction's start timestamp. Finally, the validation phase waits for the asynchronous pivot-check replies, and for dependent transactions to commit, before acquiring the final commit timestamp and reporting the transaction's dependency set to the parent CC. Doing so may require additional waiting: the full set of anti-dependencies is not known until all transactions with a smaller start timestamp finish executing.

As we mentioned in the previous section, batching may reduce efficiency of SSI, since it increases the possibility of write-write conflicts (and write-skews). Luckily, the SSI protocol can be further optimized under certain assumptions. For instance, if SSI is used as the CC at the root of the CC tree to separate read-only transactions from update transactions (as is often the case), the protocol can be optimized as follows. First, SSI does not need to wait for concurrent transactions to finish executing, as root CC does not need to report a transaction's dependency set. Second, in the presence of a single update child group (which can be further partitioned), batching is no longer necessary. Indeed, as transactions in the update group will never observe values from the read-only group, it is not necessary to assign them start timestamps. Consistent ordering can be achieved simply by committing transactions in the update group according to their in-group order. Finally, checking for pivot batches is not necessary, as a pivot batch must involve at least two update groups.

### 4.4.4 Multiversioned Timestamp Ordering (TSO)

Multiversioned timestamp ordering [32, 76] minimizes snapshot isolation's high abort rates under heavy write-write conflicts. TSO decides the serialization order by assigning a timestamp to every transaction at their start time. A writer creates a new object version marked with its timestamp, unless a reader with a larger timestamp has read the prior version (i.e., has missed this write), in which case the writer is aborted. A read returns the latest ver-

sion with a timestamp smaller than the reader's. To prevent aborted reads, a transaction logs the write-read dependencies, and only commits after all these dependencies have committed. Tebaldi, inspired by Faleiro et al. [49] implements an optimization: promises. Transactions can optionally specify at start time object keys that they will write during their execution. Tebaldi then delays any transactions that attempt to read those values until the corresponding write occurs (instead of eventually having to abort the write transaction).

To enforce consistent ordering in TSO, Tebaldi can once again use batching. The start phase creates a batch of transactions for each child group and assigns the same timestamp to all transactions in the same batch. As in SSI, batching delays TSO's ordering decisions for a batch until commit time, giving children CCs complete freedom in how they order transactions.

In the execution phase, the write logic remains identical. For reads, TSO accepts the child's proposal if it is a write from its own batch. Otherwise, it returns the latest version of that object with a smaller timestamp. In the validation phase, TSO uses the in-group dependency reports to decide on the order of transactions within a batch, and commits a transaction only after all its in-group dependencies have committed. As TSO exposes uncommitted values across groups (unlike SSI), the protocol must additionally verify that later batches read the latest write from previous batches: consistent ordering can be violated if the final order of writes differs from their execution order. Suppose, for instance, that two transactions $T_1$ and $T_2$ in the same batch $B$ write the same object. If $T_1$ *executes* the write before $T_2$, a reader from another group,

ordered after $B$, will read $T_2$ instead of $T_1$, even if the child CC eventually orders $T_1$ after $T_2$. To prevent this, TSO delays committing a transaction $T$ until all batches with lower timestamps have committed. It aborts $T$ if there exists a later object version that has the same timestamp as the version observed by $T$. Finally, TSO can conservatively report a transaction's dependency set to its parent to include all transactions with a smaller timestamp.

As was the case in SSI, batching in TSO can degrade its performance. In fact, TSO is most efficient when serving as a leaf node in hierarchical MCC. In this case, it does not need to adopt batching to enforce consistent ordering.

## 4.5 Implementation

The current prototype of Tebaldi provides support for tables, variable sized columns, and read-modify-write operations. It also supports durability. It does not however currently support range operations or replication.

### 4.5.1 Cluster Architecture

A Tebaldi cluster consists of two major types of nodes. The *transaction coordinators* (TC) manage transactions' states, while *data servers* (DS) hold partitions of the data and handle data access requests from TCs. Tebaldi can be scaled up easily by adding more TC and DS nodes to the cluster.

The implementation of the four phases discussed in Section 4.3.1 is split between TC and DS nodes and follows a common pattern: for each phase, the TC issues request(s) to the appropriate DS and waits for the reply. In

certain phases, some CCs may omit contacting the DS, while others may require additional communication. For example, in runtime pipelining the TC for transaction $T$ contacts the TCs for the transactions in $T$'s dependency set to determine when it is safe for $T$ to begin executing a new step. The DS, meanwhile, maintains a lock table and manages timeouts.

### 4.5.2 Optimizations

**Phase optimizations** We previously introduced each of the four protocol phases as two-pass procedures: a top-down pass where the parent CC constrains the execution of its children, followed by a bottom-up pass where the child CC informs the parent of its ordering decisions. In our experience however, few CCs leverage the bottom-up pass in the start phase or the top-down pass in the validation phase. Our current implementation removes them for efficiency.

**Latency reduction** Sequentially executing the respective phases of every CC in a CC tree of height $h$ could result in up to $O(h)$ network round-trips, as each CC's logic may involve communication between the TC and DS. To sidestep this issue, Tebaldi, for each phase, first executes the TC component of every CC, batching communication to the DS. The DS in turn executes the DS part of every CC, and batches replies to the TC. This reduces the framework's latency to a single round-trip per phase, in line with prior non-hierarchical approaches.

### 4.5.3 Garbage Collection

Tebaldi implements a garbage collection service that prunes stale versions from multiversioned storage. Logically, a write can be GCed when all CCs agree that it will never be read again. For efficiency, Tebaldi processes records in batch within a *GC epoch*: Tebaldi assigns a GC epoch id to every transaction, periodically incrementing the epoch. When all transactions in an epoch finish, Tebaldi asks all CCs to confirm that they will never order ongoing or future transactions before a transaction in this epoch. Once all CCs have confirmed, all stale writes in the epoch can be GCed.

### 4.5.4 Supporting Durability

We implemented a durability module for Tebaldi. It is based on the widely-used write-ahead logging and two-phase commit technique [52, 64, 65].

During the execution phase, data servers create *operation logs* to store information of write operations. At the precommit phase, each participating data server generates a *precommit log* when all the involved CCs pass the precommit on that server. This log keeps the number of participating data servers in the transaction and the ordering information for writes (to reconstruct latest version on each data object in recovery). This log is saved to persistent storage before the data server notifies the transaction coordinator. A transaction is ready to commit (and is guaranteed to committed) once all precommit logs have been made persistent on all involved data servers.

Tebaldi does not implement its own persistent storage; instead, it *out-*

*sources* this responsibility to an underlying storage system. Besides ensuring persistency, the only requirement on this storage system is to provide a key-value interface. In this way, Tebaldi can transform a single-machine, non-transactional key-value store into one that is distributed and offers transactional support[1]. Tebaldi currently uses Redis [18] and RocksDB [19] as its underlying storage; new ones can be added easily.

**Recovery Protocol** Recovery in Tebaldi is a three-step procedure. The first step retrieves logs from the persistent storage on each data server. The second step then reconstructs the database state: data servers coordinate with each other (via transaction coordinators) to discard any transaction that has fewer precommit logs than the number of participating data servers. The remaining transactions are committed. Tebaldi reconstructs the database state by keeping the latest committed version on each data object. Finally, the third step reconstructs concurrency control's internal states, such as data indices, version maps, and lock tables. Here, Tebaldi only needs to reconstruct the root CC's state. Logically, this is equivalent to having a recovery transaction that writes all the recovered data objects to an empty database, with the transaction coming from a virtual child node beneath the root, so that only the root CC knows this transaction. After recovery, the read logic in the execution phase will automatically fix any incorrect read result.

[1]The underlying storage has all the data in Tebaldi database, but in the form of transaction logs, rather than actual key-value pairs.

**Asynchronous flushing** Flushing transactions synchronously can severely reduce Tebaldi's performance. To mitigate this problem, Tebaldi provides an asynchronous flushing protocol. It separates *commit notification* from *durable notification*: a committed transaction may still get lost if Tebaldi fails soon after, until a durable notification is issued at a later time. This allows Tebaldi to batch and flush logs asynchronously in background. Most importantly, to CC mechanisms, a committed but not-yet-durable transaction is no different from a committed and durable one, so durability won't affect concurrency (e.g., 2PL can release all locks at commit time). Applications can choose to wait for either of the notifications, or both of them.

The key challenge is to ensure that Tebaldi can recover to a consistent state: as committed transactions may get lost, a recovered transaction may read from a committed transaction that was lost in the failure. This has similar effects as an aborted read, rendering the database in an inconsistent state. Our solution to this problem is to use a global checkpoint (GCP) protocol. Transaction logs are flushed in batches called *GCP epochs*. Each data server holds the current GCP epoch id, and assigns this id to transactions' precommit logs. The transaction coordinator then calculates a transaction's *global epoch id* to be the largest epoch id from participating data servers (data servers may hold different epoch ids during an epoch change). At commit time, the transaction coordinator notifies data servers with the transaction's global epoch id. If this id is larger than a data server's current epoch id, the data server updates its current epoch id before executing any CC's commit phase.

This protocol ensures that if transaction $T_2$ reads data from transaction $T_1$, $T_2$ will have a global epoch id larger than that of $T_1$. During recovery, Tebaldi discards transactions whose global epoch id is larger than the latest persistent epoch id.

## 4.6 Evaluation

Tebaldi seeks to unlock the full potential of federating concurrency controls by applying MCC hierarchically. To quantify its benefits and limitations, we ask the following questions:

- How does Tebaldi perform compared to monolithic concurrency controls and two-layered MCC systems? (Section 4.6.1 and Section 4.6.2)
- Is Tebaldi's framework conducive to adapting CC trees to changes in the workload? (Section 4.6.3)
- How do Tebaldi's different features contribute to increasing concurrency? (Section 4.6.4)
- What is the overhead of running multiple CCs? (Section 4.6.5)

**Experimental setup** We configure Tebaldi to run on a CloudLab [5] cluster of Dell PowerEdge C8220 machines (20 cores, 256GB memory) connected via 10Gb Ethernet. The ping time between machines ranges from 0.08ms to 0.16ms. The cluster contains 20 machines for the TPC-C and SEATS experiments, and ten machines for microbenchmarks; each machine runs ten data

server nodes and ten transaction coordinators (with one additional machine for timestamp assignment and batch management under SSI). These experiments were carried out before the durability feature was designed and added to Tebaldi, so there is no durability in these experiments. We add an experiment in the end of this section to measure the overhead of durability: with asynchronous flushing, durability only leads to about 5% performance overhead.

**Benchmarks** We evaluate the performance of our system using several microbenchmarks, TPC-C [41], and SEATS [46]. TPC-C models the business logic of a wholesale supplier and is the de-facto standard for OLTP workloads, while SEATS simulates an airplane ticket selling service.

We adapt the TPC-C benchmark to our transactional key-value store interface: we remove the scan over a customer's last name in the `payment` and `order status` transactions. We additionally use a separate table as a secondary index on the order table to locate a customer's latest order in the `order status` transaction. Our current implementation does not contain the 1% of aborted `new order` transactions. In line with prior work that focuses on contention-heavy workloads [66, 94, 95, 99], we run TPC-C test clients in a closed-loop and populate ten warehouses.

We also adapt the SEATS benchmark: we keep its application logic but reduce the number of available flights to demonstrate the benefits of hierarchical grouping under high-contention, and significantly increase the number of seats in each "flight" to run the benchmark for sufficiently long. The configu-

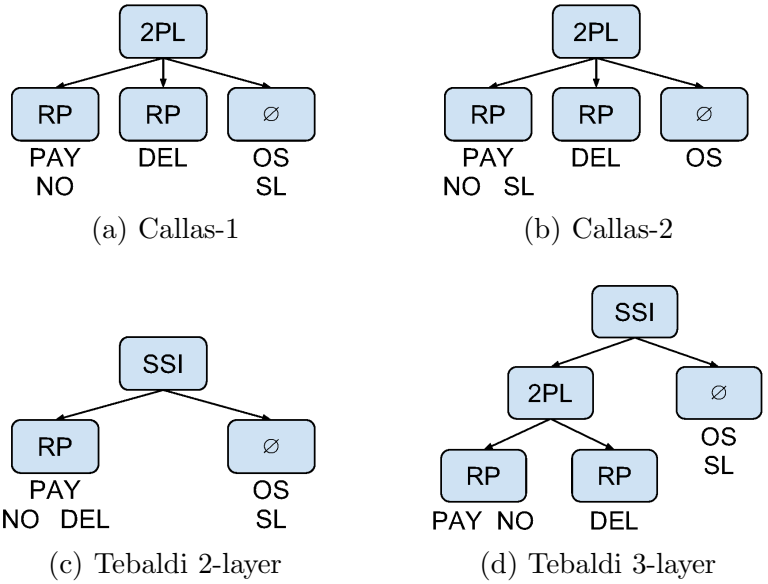
ration we ultimately adopt, though unrealistic for airlines, may model seating assignments for a small number of sporting events, each with a large number of seats. Specifically, we remove the scan over the customer name (in `delete reservation` and `update customer`) and use separate tables as secondary indices on the `reservation` table to locate the reservation id based on the flight id and seat / customer id. Further, we reduce the number of available flights to 50, increase the number of seats available per flight to 30,000, and reduce the number of seats accessed in `find open seats` to 30.

Optimal grouping for both benchmarks is obtained manually in this chapter: we recursively identify highly contended transactions with human effort, and use our domain-specific experience to pair them with concurrency controls well-suited to the transactions' inherent structure. We give specific details for each benchmark in Section 4.6.1 and Section 4.6.2.

### 4.6.1 Tebaldi's performance on TPC-C

**Baselines** We compare Tebaldi against two monolithic concurrency controls (2PL and SSI) and the federated system Callas [95]. These systems are implemented within the Tebaldi framework, and hence make use of the same network and storage stack. In SSI, we allow aborted transactions to backoff for 5 milliseconds before retrying to reduce the resource consumption.

**Grouping** To configure the Callas system, we start with the grouping strategy proposed in the Callas paper. This initial grouping (Callas-1, Figure 4.6a)




(a) Callas-1 (b) Callas-2

(c) Tebaldi 2-layer (d) Tebaldi 3-layer

Figure 4.6: CC trees used in TPC-C. Leaf nodes are labeled with transactions: `payment` (PAY), `new order` (NO), `delivery` (DEL), `order status` (OS), and `stock level` (SL).

partitions transactions into three groups. The first group contains `new order` and `payment`, whose conflicts can be aggressively optimized by runtime pipelining. The second group uses another instance of runtime pipelining for the `delivery` transaction. Finally, the two read-only transactions are in the third group. This grouping, when running under serializability (Callas originally runs under read-committed), introduces a large number of cross-group read-write conflicts between the `stock level` and `new order` / `payment` transactions. Because of the fixed structure of Callas, these conflicts must be handled by 2PL. To mitigate this problem, we modify Callas-1 by moving `stock level` in the first group, where it can be pipelined with `new order` (Callas-2, Figure 4.6b).

Thanks to the flexibility of hierarchical MCC, Tebaldi supports a wider variety of grouping strategies than its cousin Callas. We propose two such groupings, in Figure 4.6c and Figure 4.6d respectively. The first grouping strategy (Tebaldi 2-layer) leverages Tebaldi's ability to support cross-group protocols other than 2PL by selecting a multiversion cross-group protocol, SSI. It then partitions transactions into two groups: a read-only group containing transactions `order status` and `stock level` that requires no in-group concurrency control; and an update transaction group that uses runtime pipelining to optimize the `new order`, `payment`, and `delivery` transactions.

The second grouping strategy (Tebaldi 3-layer) instead leverages Tebaldi's hierarchical model by creating a concurrency control tree of depth three. This approach partitions transactions into the same leaf-level groups as the original Callas grouping (Callas-1). However, it then uses two distinct cross-group mechanisms to handle the remaining conflicts: 2PL for conflicts between `new order / payment` group and the `delivery` group, and SSI for conflicts between the read-only group and all other groups.

**Results** Figure 4.7 compares the performance of Tebaldi's grouping strategies against those of Callas, and against two monolithic concurrency control protocols: two-phase locking (2PL) and serializable snapshot isolation (SSI).

Consider first the performance of the two monolithic concurrency controls: the peak throughput of SSI is 7× higher than that of 2PL, because of the high read-write conflict ratio between `new order` and `payment` (the trans-

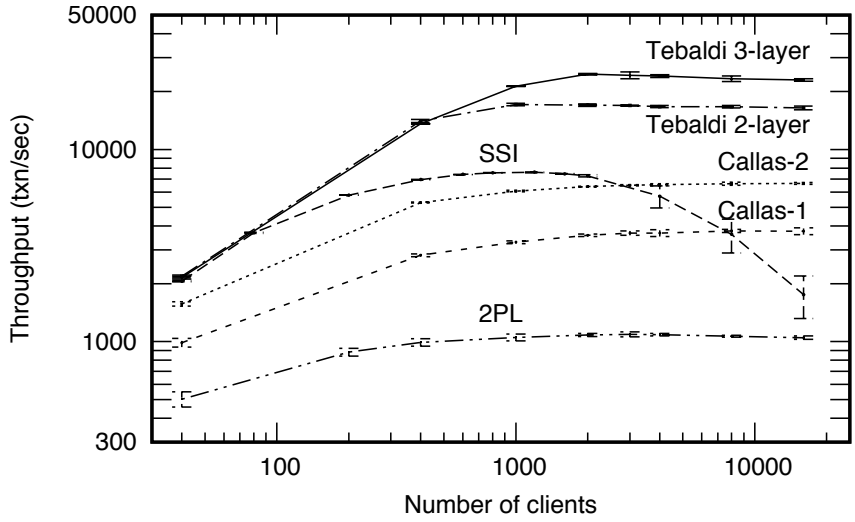



Figure 4.7: Performance of TPC-C benchmark.

actions in fact read and write different columns, but Tebaldi, like most other systems, takes row-level locks). As contention increases (by increasing the number of clients), the performance of SSI drops steeply as the high write-write conflict rate causes SSI to repeatedly abort transactions.

When contention is high, the performance of SSI and 2PL is lower than that of Callas and Tebaldi. Callas's initial grouping strategy (Callas-1) is bottlenecked by the heavy read-write conflicts between `stock level` and `new order` / `payment`. Revising this partitioning (Callas-2) yields a 77% throughput increase, but decreases the efficiency of RP (by moving `stock level` and `new order` to the same group): `new order`'s writes to `order`, `new order` and `order line` tables, which could never conflict in the previous grouping (as order ids are unique), can now read-write conflict with `stock level`'s reads (creating additional synchronization in RP). Additionally, combining

`stock level` and `new order` creates circular table dependencies, resulting in a coarser-grained pipeline. There is once again a tension between the potential inefficiency of a monolithic cross-group mechanism, and the in-group mechanism's desire to handle few transactions.

To side-step this tension, Tebaldi has two options: select a more suitable cross-group mechanism (Tebaldi 2-layer), or create a deeper grouping hierarchy (Tebaldi 3-layer). The former yields a 2.6× improvement over the best grouping strategy in Callas: SSI, as a cross-group mechanism, can efficiently handle the read-write conflict between `stock level` and `new order`, while the three update transactions can be pipelined fairly efficiently. This pipelining remains suboptimal however, as the potential conflict between `new order` and `delivery` voids `new order`'s unique access to tables, creating additional synchronization in `new order`'s execution. The latter grouping strategy (Tebaldi 3-layer) addresses this issue: the small and carefully selected scope of each group gives every in-group concurrency control the opportunity to perform well. The rare conflicts between the `new order` and `delivery` transactions are regulated by 2PL, while the common read-write conflicts are resolved using SSI. This careful tailoring of cross-group CCs to cross-group conflicts allows the three leaf groups to remain small. This narrow scope results in optimal pipelines for the two groups running RP, while the read-only group can operate without any concurrency control, leading to a further 44% improvement.

### 4.6.2 Tebaldi's performance on SEATS

**Grouping** We compare three grouping strategies. The baseline is a monolithic 2PL system. To optimize the read-write conflicts, our second grouping uses SSI to separate the read-only transactions (`find flights` and `find open seats`) from the update transactions, and uses 2PL to regulate the remaining update transactions. The third grouping further optimizes the conflicts among the update transactions. Unlike TPC-C however, these highly contended read-write transactions (`create`, `update` and `delete` of reservation) cannot be efficiently pipelined using RP because of the circular dependency between tables (`flight`, `customer` and `reservation`). Nonetheless, TSO can still pipeline these transactions by preordering them at runtime using timestamps. In doing so, however, it creates many spurious dependencies between non-conflicting transactions. To alleviate this concern, we observe that transactions that access different flights rarely conflict. We leverage Tebaldi's flexible grouping to create not one, but multiple TSO instances, one for each flight, and assign transactions to their group at start time according to their input, using 2PL as cross-group mechanism. This approach efficiently pipelines the (likely) conflicts for transactions that access the same flight but does not unnecessarily order those that don't (in the rare cases when conflicts between transactions accessing different flights arise, they are still handled by 2PL).

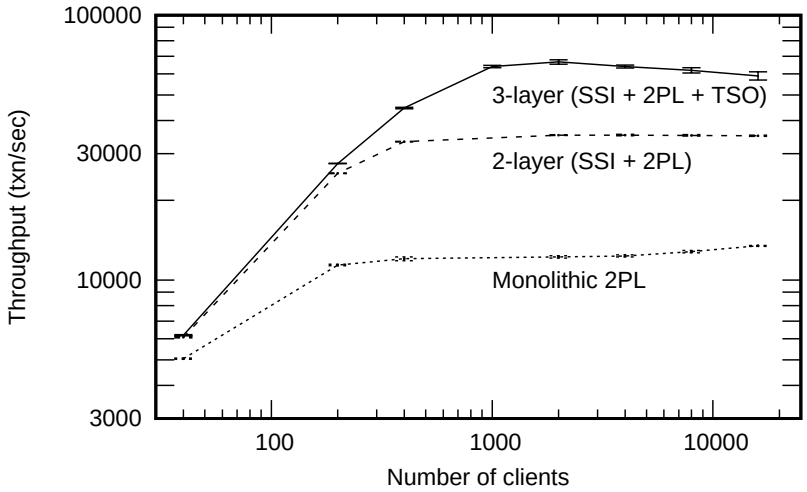



Figure 4.8: Performance of SEATS benchmark.

**Results** Figure 4.8 compares the performance of Tebaldi's three-layer hierarchy against those of two-phase locking (2PL) [2] and the two-layer hierarchy (SSI+2PL). We find that, unsurprisingly, the peak throughput of the two-layer setting is 2.6× higher than that of 2PL as it minimizes the effect of the read-write conflicts introduced by the long-running read-only transactions `find flights` and `find open seats`. As contention increases, however, `new reservation` transactions spend a prohibitive time waiting to acquire exclusive locks, hampering performance. Pipelining transactions that access the same flights using TSO yields a further about 2× speedup: TSO can pipeline operations and expose uncommitted writes to subsequent transactions without

[2]There was an error in the Tebaldi paper when we calculated the performance of the 2PL baseline. The reported 2PL performance was about 33% higher than the actual value, and we *under-estimated* MCC's benefit. I fixed this error here by re-calculating the performance of 2PL using the *original* experiment logs. This error only affected the 2PL baseline.

```
1  // input: w_id, d_id
2  begin transaction
3      o_id = district[w_id, d_id].next_order_id;
4      ol_num = order[w_id, d_id, o_id].ol_num;
5      for (i = 0; i < ol_num; ++i) {
6          item_id = order_line[w_id, d_id, o_id, i].ol_i_id;
7          item_stats[item_id]++;
8      }
9  commit
```

Figure 4.9: Pseudocode of `hot item`.

delays. Tebaldi's flexibility enables a "hybrid" type of grouping: first by transaction type, and then by transaction instances (according to their inputs), once one knows whether two transaction instances will actually conflict. Thus, Tebaldi gets the best of both worlds: the efficiency of 2PL when transactions rarely conflict, and the localized performance gains of TSO's pipeline when transactions do.

### 4.6.3 Extensibility

For moderate changes in the workload, Tebaldi's modular and flexible design makes it possible to handle the new conflict patterns by adding new concurrency controls to the existing CC trees. To showcase this benefit, we add a new transaction `hot item` to TPC-C and sketch how the prior Tebaldi 3-layer hierarchy can be refined to account for the additional conflicts.

This new transaction (shown in Figure 4.9) computes popular or "hot" items by randomly sampling recent orders in the database, and aggregating the per-item sale count over all warehouses. We set the new TPC-C workload distribution to be the following: 41.8% of transactions are `new order` transactions, 41.8% are `payment`, while the remaining transactions all run 4.1% of

the time.

The `hot item` and `new order` transactions read-write conflict heavily: neither of the current cross-group 2PL or SSI are thus good choices for regulating their behavior (the batching in SSI will periodically promote write-write conflicts between `new order` instances to cross-group conflicts, causing aborts). Instead, we have two solutions: we can keep the same three-layer hierarchy, placing the new transaction in the same group as `new order` and `payment` at the cost of a less efficient pipeline. Alternatively, we can leverage Tebaldi's flexibility to place the transaction in a separate group and use RP as the cross-group mechanism to regulate conflicts with the `new order/payment` group.

The experiment shows that the three-layer approach has a throughput of $16{,}417 \pm 192$ txn/sec, while the four-layer approach gives $23{,}232 \pm 111$ txn/sec. Placing `new order` and `hot item` in the same group reduces the pipeline's efficiency as `new order`'s accesses to tables are no longer guaranteed to be non-conflicting. In the four-layer solution, we side-step this issue by placing `hot item` and `new order` in their own group, yielding a 42% throughput increase.

### 4.6.4 Impact of flexibility

We next investigate in more detail how Tebaldi's higher flexibility enhances concurrency. Tebaldi increases flexibility over prior federated systems in two ways: by supporting multiple cross-group CCs, and by enabling finer

partitioning of conflicts.

**Support for different cross-group protocols** We first quantify the potential gains associated with Tebaldi's support for different cross-group concurrency controls. To do so, we compare the performance of different cross-group CCs for different conflict patterns. We use a two-layer hierarchy with two groups; we fix the in-group CCs and set the cross-group CC to be either 2PL, SSI, or RP. In the first three workloads, each group contains an update transaction consisting of seven write operations. The first operation writes to a shared table consisting of $n$ rows, so the conflict rate for both in-group and cross-group is $1/n$. The second operation writes to a group-local table of ten rows, adding only in-group conflicts. The remaining operations within the group conflict with low probability (1/10,000). Both groups use RP to handle in-group conflicts. By tuning $n$, we vary the cross-group conflict ratio in each workload (for benchmarks *ww-1*, *ww-5*, and *ww-10*, respectively 1%, 5%, and 10% of write-write conflicts). In the three remaining workloads, we replace one of the write-only groups with a read-only group. As read-only transactions never conflict with each other, we use an empty in-group concurrency control protocol. As above, we vary the cross-group conflict rate, this time of read-write conflicts, in each benchmark (for benchmarks *rw-1*, *rw-5*, and *rw-10*, respectively 1%, 5%, and 10%).

Figure 4.10 summarizes the throughput for each workload. We report the results in transactions per second as the average of three runs. We find that

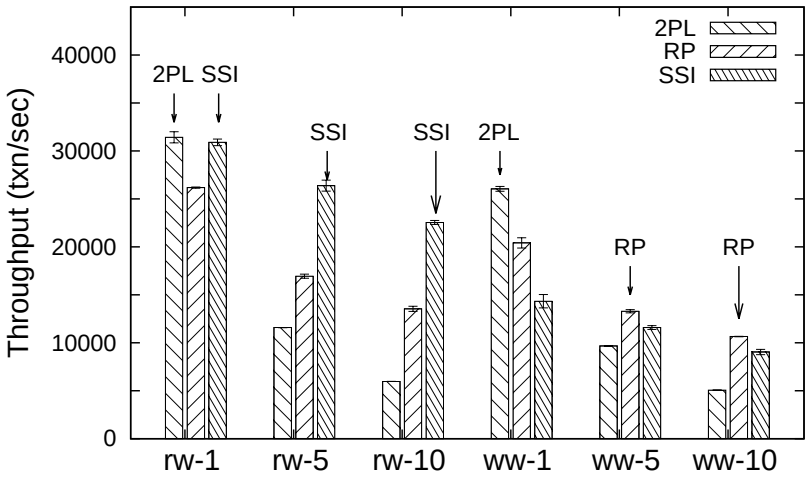



Figure 4.10: Cross-group CCs' performance.

no single cross-group protocol outperforms the others in all cases, underscoring the practical importance of selecting the cross-group mechanism most suitable for a given workload. Specifically, we find that, SSI, unsurprisingly, performs best when handling read-write conflicts across groups, as readers and writers never block each other. In contrast, SSI performs worse in the presence of write-write conflicts because of repeated aborts. Its poor performance is exacerbated by the need for batching, as write-write conflicts cannot be resolved until the next batch change (in *ww-1* transactions already retry on average more than 2.5 times). Aborts in this benchmark are relatively cheap (they mostly happen on the first operation). Costlier aborts would likely cause SSI's performance to drop. Runtime pipelining, in contrast, performs best in scenarios with medium to high amounts of write-write contention (*ww-10*,*ww-5*). When write-write conflicts are rare, however, the overhead of maintaining the pipeline outweighs

its benefit: the more conservative but simple 2PL performs best (*ww-1*).

**Hierarchical application of MCC** The previous microbenchmark was restricted to two-layer grouping strategies with a single cross-group mechanism per configuration. We next quantify the benefits of using deeper hierarchies in which we can combine multiple cross-group protocols. To do so, we focus on a scenario in which no single cross-group mechanism can efficiently handle the pairwise interactions of all transaction groups. This represents a best-case scenario for Tebaldi; we quantify potential overheads associated with deeper hierarchies in Section 4.6.5.

The microbenchmark consists of one read-only transaction, $T_1$ and two update transactions, $T_2$, and $T_3$. $T_1$ read-write conflicts heavily with $T_2$ and $T_3$, while $T_2$ and $T_3$ cannot be efficiently handled within a group. Table $A$ suffers from heavy contention as it contains only ten rows, while all other tables ($B$ to $E$) contain 10,000 rows and rarely contend. Transaction $T_1$ reads a single row in $A$, and ten rows from the remaining tables. Transaction $T_2$ first writes a row in $A$, and subsequently writes a random key from every table $B$ to $E$. Transaction $T_3$ does not access table $A$. Instead, it reads a random key from tables $B$ to $E$, and subsequently writes back to $B$. Considering the previous in-group mechanism, runtime pipelining: RP can handle $T_2$ efficiently, but not $T_2$ and $T_3$ (or $T_1$).

Constructing a three-layer CC tree in Tebaldi side-steps this issue. The read-write conflict between $T_1$ and $T_2/T_3$ can be handled efficiently through

selecting SSI as root CC, placing $T_1$ and $T_2/T_3$ in separate groups. Next, as $T_2$ and $T_3$ rarely conflict with each other, they can be placed in separate groups with 2PL as cross-group CC. Finally, conflicts between different instances of $T_2$ can be efficiently pipelined with RP. As contention is low for $T_3$, we simply use 2PL as its in-group mechanism.

We compare our solution to the four most promising two-layer hierarchies. The first two hierarchies use SSI as cross-group mechanism to optimize the read-write conflict between $T_1$ and $T_2$, but differ in how they handle $T_2$ and $T_3$: the first grouping strategy (Two-layer 1) puts $T_2$ and $T_3$ in separate groups. It allows $T_2$ to be efficiently pipelined, but requires SSI to run with batching enabled, which can periodically promote in-group conflicts to cross-group conflicts. The second baseline (Two-layer 2) places $T_2$ and $T_3$ in the same group. This gives SSI better performance at the cost of a less efficient in-group pipeline. The third grouping strategy (Two-layer 3) has $T_1$ and $T_2$ in one group running RP, and $T_3$ in another group running 2PL. 2PL is used across these two groups. This avoids the issues associated with having SSI across groups, yet still optimizes the conflict between $T_1$ and $T_2$ at the cost of a less efficient pipeline for $T_2$. The last baseline (Two-layer 4) runs all three transactions in separate groups (2PL cross-group). It prioritizes $T_2$ by using the optimal pipeline. None of these four solutions is perfect: while $T_1$, $T_2$ and $T_3$ would all benefit from being in a single group, no single concurrency control is well-suited to handle conflicts between $T_1/T_2$ and $T_2/T_3$.

Figure 4.11 confirms our intuition. The peak throughput achieved by

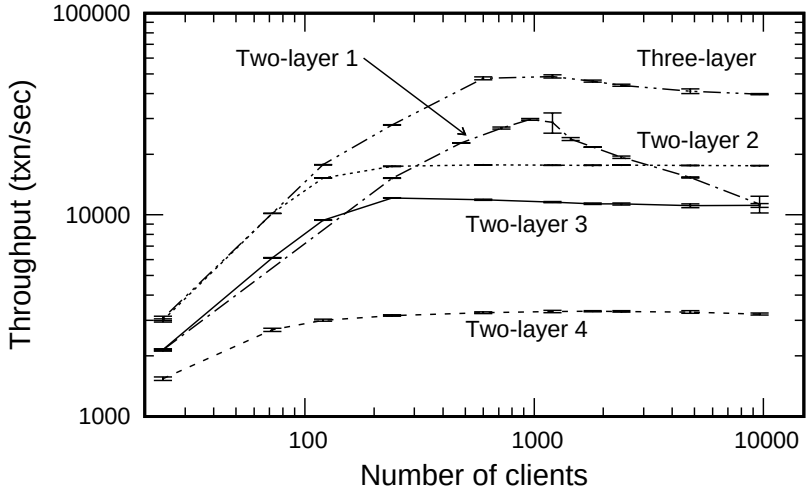



Figure 4.11: Two-layer vs. three-layer.

the three-layer hierarchy is 63% higher than the best performing two-layered grouping strategy. The fourth (Two-layer 4) grouping strategy performs worst as 2PL cannot efficiently handle the frequent read-write conflicts between $T_1$ and $T_2$. The first baseline (Two-layer 1) performs best under moderate number of concurrent clients but suffers from a high abort rate due to SSI's sensitivity to the write-write conflicts: its performance drops as the number of clients increases. Finally, the performance of both the second and third grouping options (Two-layer 2 and 3) is hampered by a sub-optimal runtime pipeline.

### 4.6.5 Overhead of Additional Layers

Tebaldi attempts to improve the performance of applications that are bottlenecked on data conflicts. It does so by enhancing the concurrency of these applications, at the cost of more complex control logic: each transaction needs

| Setting | Latency (ms) | Throughput (K txn/sec) |
|---|---|---|
| stand-alone RP | 2.969 ± 0.004 | 490.0 ± 1.7 |
| 2PL - RP | 3.068 ± 0.004 | 386.1 ± 0.4 |
| SSI - RP | 3.259 ± 0.006 | 368.4 ± 1.7 |
| RP - RP | 4.047 ± 0.004 | 291.9 ± 0.8 |

Table 4.1: Latency and resource cost of adding additional layers.

to percolate through every concurrency control in its path on the CC tree. This additional complexity can negatively impact the application in two ways: it can increase the latency of transactions, and can increase the application's resource consumption. We quantify these potential drawbacks in this section. To do so, we run a microbenchmark with grouping strategies that do not yield any additional concurrency, and measure the resulting cost increase. The benchmark consists of a single transaction type with seven write operations, and ensures concurrent transactions never conflict with each other. We use a stand-alone runtime pipelining protocol as the baseline, and add either 2PL, RP, or SSI cross-group layers to the hierarchy.

**Latency overhead** To measure the impact of the hierarchy's depth on latency, we run the benchmark with a small number of clients (20) to ensure that the resource (CPU / network) consumption is low.

Our results are shown in the second column of Table 4.1 and denote the transaction's average latency over ten 60 seconds runs. We find that the relative latency increase of adding an additional layer in the hierarchy depends heavily on the cross-group concurrency control being added. Adding a 2PL

cross-group layer yields a small 3.3% increase in latency. This increase is exclusively due to computational overhead: the number of round-trips in Tebaldi is independent of the hierarchy depth (Section 4.5). Any necessary additional round-trip is thus a property of the concurrency control itself: 2PL requires no additional round-trips, while SSI requires an additional round-trip to contact the timestamp server, and RP requires an additional round-trip per operation. These additional network trips are reflected in our result: adding an SSI cross-group layer increases latency by 9.8%, while adding an RP cross-group layer increases latency by 36.3%.

**Computation resources overhead** Under high system load, the computational overhead of adding CCs could become prohibitive, bottlenecking the system on CPU or network resources. To quantify this overhead, we increase the workload to measure the peak throughput of the microbenchmark, ensuring that the CPU is the bottleneck each time.

Our results are summarized in the third column of Table 4.1. Adding a 2PL layer over an RP in-group mechanism leads to a 21% decrease in throughput while adding an SSI layer leads to 25% drop. The overhead is relatively small, as 2PL and SSI remain fairly light-weighted when compared to RP. The overhead of adding an RP layer is more significant: throughput drops by 40%, as RP is fairly complex. Note that, even when adding an RP layer over the RP in-group mechanism, the throughput does not simply halve, as many components of the framework are independent of the hierarchy depth.

### 4.6.6 Overhead of Durability Protocol

To understand the overhead of the durability feature, we rerun the TPC-C benchmark with our latest Tebaldi codebase that supports durability. We use RocksDB [19] as the underlying persistent storage when durability is ON, and employ the asynchronous flushing protocol (Section 4.5.4). When committing transactions, testing clients wait for commit notifications (rather than durability notifications). This reduces the latency, and allows us to reach peak throughput with much fewer concurrent clients. The drawback, though, is that committed transactions in the last GCP epoch may get lost in case of failure. We configure the length of a GCP epoch to be one second, so a transaction will become durable soon after it commits (one second plus time to flush data to disks) as long as Tebaldi does not fail in that period.

| Setting | Throughput (txn/sec) |
|---|---|
| Durability ON | $22{,}390 \pm 60$ |
| Durability OFF | $23{,}415 \pm 81$ |

Table 4.2: Overhead of durability protocol on TPC-C benchmark.

We run TPC-C benchmark with the 3-layer configuration shown in Figure 4.6d, and measure the peak throughput with durability turned on and off. The performance numbers are shown in Table 4.2. As we can see, the durability feature only costs about 5% of performance. The asynchronous flushing protocol is the key reason why we can achieve such low cost: as it decouples concurrency control from durability, CC mechanisms can commit a transaction and release its resources (such as locks) before it actually becomes durable.

# Chapter 5

# Automatic Configuration

The performance benefits of MCC largely depend on the configuration of the concurrency control federation, but as I have discussed in Chapter 3, configuring MCC is a non-trivial task. This chapter focuses on this problem, with the goal of a MCC-based database as easy to use as a traditional one: users should be able to enjoy the performance benefits of MCC without paying much additional effort in programming applications or configuring the database.

To achieve this goal, we equip Tebaldi with the ability to manage its MCC configuration in a fully automatic manner. The key technique is a configuration algorithm that allows Tebaldi to monitor the workload, identify data contention bottlenecks, and adjust automatically to improve the *throughput* of the current workload.

The chapter starts with a discussion of the overall design of the algorithm, and proceeds to detail each of its components, and finally, presents an evaluation of how Tebaldi performs under automatic configuration.

## 5.1 Towards Automatic Configuration

When configuring MCC, the ideal outcome is one that, for any given workload, can produce the configuration that produces the highest throughput. Two challenges make this goal hard to achieve efficiently. The first is the size of the search space: even for Callas' two-layer MCC, and even if we only partition transactions by *type* (i.e., by the static transaction code), the number of different configurations can grow exponentially with the number of transaction types. Adopting hierarchical MCC, or partitioning transactions by instances (like in the SEATS benchmark in Section 4.6.2) can further complicate the problem: even a moderate number of transaction types can result in a huge search space.

The second challenge is that predicting the performance of a given configuration is hard. Many factors can affect MCC's performance, including the workload characteristics, how MCC partitions data conflicts, how effectively the CC mechanisms handle their data conflicts, and how they interact with one another. These factors can be very specific to CC mechanisms, and are highly sensitive to any small changes in the MCC configuration or the workload. For example, the change in data access patterns caused by adding one transaction to a runtime pipelining group can fundamentally alter its performance (see Section 4.6.3 for an example). Similarly, reordering operations in transactions may introduce deadlocks in a 2PL group, spoiling its performance. Essentially, the only reliable way to know how a given MCC configuration will perform for a given workload is to run it, but doing so for a large set of configurations

becomes quickly prohibitive.

These considerations lead us to adopt for this dissertation a more modest goal: we aim to identify solutions that, though not optimal, can be calculated efficiently, while still being likely to increase concurrency and yield substantial performance benefits. As such, we make two simplifications. First, unless otherwise stated (Section 5.4.2), our search only aims at configurations that partition transactions by types. This assumption allows us to treat transactions of the same type together, avoiding the complexity of a per-instance analysis. While this simplification may lead us to miss some potential optimizations, in most cases, partition-by-type is very effective: transactions of the same type often share a similar data conflict pattern, so they can be optimized in the same way. Second, rather than seeking a solution that addresses at the same time all performance-limiting data conflicts, to achieve a good balance between complexity and performance, we prioritize the most severe performance bottlenecks, and only resolve one bottleneck at a time.

In the spirit of the above discussion, we take an *iterative* approach to optimize MCC's configuration. Each iteration of our algorithm (shown in Figure 5.1) identifies the most severe data contention bottleneck in the MCC configuration from the previous iteration, and proposes new configurations to optimize it. It then evaluates each of the candidates, and picks the best performing one as the new configuration from which to start the next iteration.

This approach follows common performance debugging practice, and shares the same rationale. It is reasonable to prioritize different contention

```
1  // Start with an initial configuration
2  // mcc_config is the active configuration
3  mcc_config := initial_config;

5  // EndOfOptimization() defines the termination condition
6  while (!EndOfOptimization()) {
7      // Analyze current performance
8      bottleneck := FindFirstBottleneck();

10     // Propose optimization candidates
11     candidates := ProposeOptimization(mcc_config, bottleneck);

13     // Test each candidate and make a decision
14     best_config := mcc_config;
15     best_performance := CurrentThroughput();
16     foreach (new_config in candidates) {
17         mcc_config := new_config;
18         performance := TestThroughput();
19         if (performance > best_performance) {
20             best_config := mcc_config;
21             best_performance := performance;
22         }
23     }
24     mcc_config := best_config;
25 }
```

Figure 5.1: Pseudocode of our configuration algorithm.

bottlenecks by their severity, since the end-to-end performance of a system is often largely determined by its most severe bottleneck. In addition, the most severe performance bottleneck is typically easier to isolate than secondary ones. By always focusing on the most severe bottleneck, we are more likely to work on a real performance problem. Meanwhile, by tackling only one bottleneck at a time, we can reduce the search space significantly.

This iterative approach is similar in structure to the one adopted by Callas, but besides addressing the novel challenges introduced by hierarchical MCC, Tebaldi's algorithm revisits, and improves upon, many of the major components of Callas' algorithm. In particular, it includes four significant innovations:

- A new profiling algorithm that can more reliably and accurately detect and describe performance bottlenecks.
- New strategies to adjust MCC's configuration, designed to leverage the benefit of hierarchical MCC.
- A new framework to deal with the additional preprocessing steps required by specific CC mechanisms, such as the static analysis step used by runtime pipelining, that enables Tebaldi to apply them automatically.
- New online reconfiguration protocols that allow Tebaldi to change the configuration of hierarchical MCC efficiently at runtime.

The algorithm is fully integrated in the Tebaldi database, and can automatically manage Tebaldi's configuration in real time with minimal, if any, user involvement.

The rest of this chapter details each component in our algorithm. Section 5.2 discusses how the algorithm is initialized and the condition under which it terminates. Next, we discuss the three stages of each iteration of the algorithm. Section 5.3 describes the *analysis stage*, whose purpose is to monitor the performance of the database and determine the most significant remaining performance bottleneck; Section 5.4 discusses the *optimization stage*, charged with proposing new MCC configurations and dealing with CC-specific preprocessing; and Section 5.5 presents the online reconfiguration protocols that we use in the *testing stage*, which allows Tebaldi to efficiently switch among

candidate configurations, measure their performance, and decide on the best configuration.

## 5.2 Initial Configuration and Termination Condition

**Initial Configuration** The main requirement for the algorithm's initial configuration is that it should be general: it should work with most transactions and workloads. One option is to start with all transactions in a single group running two-phase locking, since this is the standard mechanism used by many database systems [8, 10, 70].

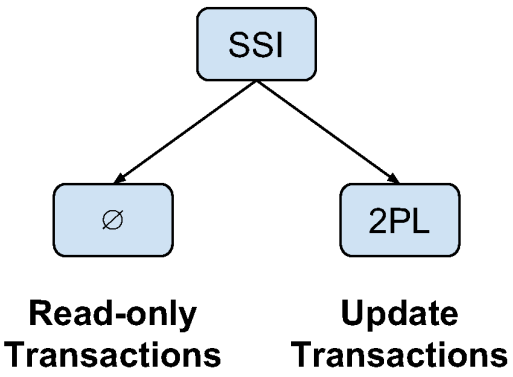



Figure 5.2: The initial configuration used in our algorithm.

Tebaldi, however, takes advantage of MCC to instantiate a more sophisticated initial configuration that already optimizes some common types of workloads. Inspired by various database systems that adopt optimizations for read transactions [40, 42, 81, 96], Tebaldi's initial configuration (shown in Figure 5.2) places serializable snapshot isolation (SSI) at the root node, with two

children: a node with no CC mechanism grouping all read-only transactions, and a node running two-phase locking grouping all update transactions. As I have mentioned in Section 4.4.3, running SSI in this setting is effectively equivalent to running multi-version two-phase locking (MV2PL) [32, 36, 37, 47, 89], which ensures that read-only and update transactions do not interfere with each other. Future optimizations proposed by the automatic configuration algorithm may further partition transactions in the 2PL group, and handle their data conflicts using more efficient CC mechanisms.

**Terminating Condition** Tebaldi currently uses a very simple terminating condition to decide when to stop the iterative optimization process. It stops when an iteration fails to find a data contention bottleneck, or when all the proposed new configurations perform worse than the current configuration.

## 5.3 Analysis Stage

The analysis stage monitors the database's performance and identifies the most prominent source of data contention under MCC's current configuration, which then becomes the target to optimize in the current round of iteration.

To deliver on its mission, this stage's performance analysis algorithm aims for *accuracy* and *high-resolution*. The importance of accuracy is clear: if the algorithm fails to detect the actual performance bottleneck, later stages may miss optimization opportunities. Meanwhile, it is equally important to

detect the performance bottleneck with high resolution. Rather than simply identifying a set of highly-contending transaction types, we aim for the exact bottleneck *conflict edge*, i.e., the pair of transaction types whose data contention limits the performance of the workload. This fine-grained information allows later stages to fully leverage MCC's capability to fine-tune CC mechanisms for different data conflicts.

However, the complicated nature of concurrent execution in a database system makes it hard to reason about its performance and to precisely identify bottlenecks. One challenge, for example, is that a data conflict may have cascading effects on other data conflicts, hiding the root cause of the performance problem. As the following case study shows, simple profiling techniques such as the one used by Callas may fail to reliably detect the real bottleneck.

### 5.3.1 Case Study: a Latency-based Technique

Callas proposed a latency-based performance profiling technique. It leverages the observation that transactions with heavy data contention often suffer from severe queuing delays on conflicting operations. To detect these transactions, Callas increases the workload's request rate (while keeping the rest of the workload profile, such as the transaction mix ratio, unchanged), and measures how end-to-end latencies change for different transactions and operations. This technique suggests that the performance bottleneck is to be found among operations (and their corresponding transactions) whose latency increases disproportionately in this procedure.

```
1   // payment transaction
2   // input: w_id, d_id, h_amount, ...
3   begin transaction
4       warhouse[w_id].w_ytd += h_amount;
5       district[w_id, d_id].d_ytd += h_amount;
6       // some rarely conflicting operations
7   commit

9   // stock level transaction
10  // input w_id, d_id
11  begin transaction
12      o_id := district[w_id, d_id].next_order_id;
13      for (i := o_id − 20; i < o_id; ++i) {
14          ol_num := order[w_id, d_id, i].ol_num;
15          for (j := 0; j < ol_num; ++j) {
16              item := order_line[w_id, d_id, i, j].
                        ol_i_id;
17              quantity := stock[w_id, item].
                        s_quantity;
18          }
19      }
20  commit
```

Figure 5.3: Simplified logics of `payment` and `stock level`.

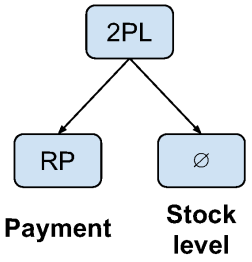



Figure 5.4: The MCC configuration under test.

Despite its appealing simplicity, this technique has several problems. First, it requires control of the incoming database workload to change its request rate; this is hard to achieve unless additional testing infrastructure, such as a workload generator, is available outside the database system. Even so, it requires additional effort from users. Second, this technique can only describe contention bottlenecks coarsely: it reports a set of highly-contending transaction types, rather than exact data conflict edges. If multiple transactions are reported, later stages in the configuration algorithm will not be able to identify the exact conflict edge that they should optimize. Even worse, this technique may miss the real cause of the data contention problem, as the following example demonstrates.

Consider the `payment` and `stock level` transactions in the TPC-C

benchmark [87], whose logic is summarized in Figure 5.3. The `payment` transaction modifies the `warehouse` and the `district` tables, and accesses some other tables that rarely conflict. The `stock level` transaction is read-only: it reads the `district` table, and then issues many reads to tables that rarely conflict. The data model of TPC-C is such that each row in the warehouse table corresponds to ten rows in the district table, i.e., each warehouse consists of ten districts. Consider running this workload under the MCC configuration of Figure 5.4, using Callas' latency-based profiling technique. Without `stock level`, runtime pipelining can efficiently handle `payment` transactions by running their operations in a pipeline. But as soon as a `payment` transaction and a `stock level` transaction conflict on the district table, 2PL will block the `payment` transaction. Indeed, the effect of this blocking can be *amplified* by a cascading effect among `payment` transactions. All other `payment` instances that access the same `warehouse` will be blocked by the stalling of the pipeline. Thus, the initial blocking on a single `district` object (caused by 2PL) is magnified to a more severe blocking on an entire `warehouse` (in the RP group). As a result, when the profiling algorithm increases the workload's request rate, only the `payment` transaction will show significant latency increases. The root cause of this performance issue—the conflict between `payment` and `stock level`—remains outside of our purview.

This observation is confirmed by the experimental results shown in Figure 5.5. We gradually increase the workload by adding more concurrent clients, and measure the latency of each transaction type. As the figure shows,

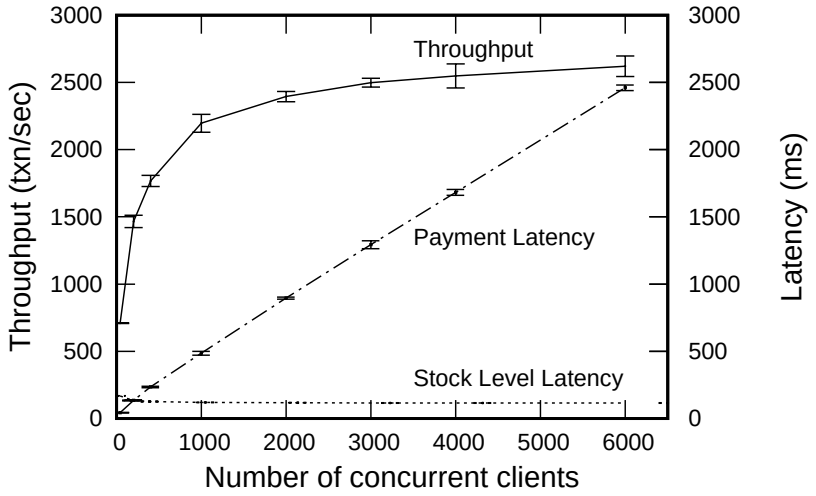



Figure 5.5: Test results of the latency-based profiling technique.

only the `payment` transaction suffers from significant latency increase, so the simple latency-based profiling algorithm will conclude that the performance bottleneck is due to conflict among `payment` transactions. But this conflict has already been optimized by runtime pipelining, and Tebaldi cannot optimize it further. To verify that the real problem is between `payment` and `stock level`, we ran another experiment using SSI at the cross-group layer to efficiently handle conflicts between these two transactions: throughput increased to around 25,000 transactions per second.

### 5.3.2 Tracking Cascading Effects of Data Contention

To reliably detect the root cause of the performance problem, we need not only to measure the severity of each data contention in the workload, but also to analyze how different instances of data contention can affect each other.

In most CC mechanisms, data contention can cause transactions to either *block* or *abort*. Consequently, two kinds of interactions can arise among multiple instances of data contention: *cascading blocking* and *cascading aborts*. Cascading blocking occurs when a blocked transaction exacerbates the blocking of other transactions waiting for it, while *cascading aborts* arise when an aborted transaction causes its depending transactions to abort if the CC mechanism exposes uncommitted states. Our new performance analysis algorithm mainly focuses on cascading blocking, since it is ubiquitous for all blocking-based CC mechanisms. Cascading abort, on the other hand, is a less pressing issue, since most CC mechanisms are already equipped with techniques to avoid or reduce cascading aborts.

To track cascading blocking, we adapt to our purposes an approach originally used to profile the performance of multi-threaded programs [25, 54, 100]. In that context, one often needs to measure the severity of blocking caused by the synchronization between different pairs of threads. This measurement is usually based on the length of time a thread waits for another, but, to improve the result's accuracy, it also takes into consideration nested waiting. If thread $A$ waits for thread $B$, and during that period $B$ also waits for thread $C$, when considering the performance implications of $B$ waiting for $C$, one should also consider its impact on the waiting between $A$ and $B$.

Our analysis algorithm adopts a similar approach. Its profiling technique consists of two parts: a sampling module distributed over all database nodes, and a centralized *performance monitor* node. The sampling module in-

struments all blocking-based CC mechanisms to log all blocking events that are caused by data contention when they execute transactions. Each log entry contains the id of the affected transaction, the id of the blocking transaction, their static transaction types, and timestamps when the blocking begins and ends. The database node batches these logs and periodically sends them to the performance monitor.

The performance monitor periodically analyzes the collected logs, and calculates a score for each *ordered* pair of transaction types: $score(< T_i, T_j >)$ is the total time spent by transactions of type $T_j$ waiting for transactions of type $T_i$ in the period being analyzed. As in profiling multi-threaded programs, we do not simply add up all blocking times between the two types of transactions, but instead adjust the score to account for nested waiting. If a transaction $t_i$ (of type $T_i$) blocks $t_j$ (of type $T_j$), we only attribute to the score of $< T_i, T_j >$ the time during which $t_i$ is not blocked by others. If $t_i$ is blocked by $t_k$ (of type $T_k$) for a time during that period, we charge that time to the conflict between $T_k$ and $T_i$ (and if $t_k$ is also blocked, we recursively analyze the inner conflict). This computation can be parallelized by multi-threading: each thread computes scores starting with a portion of log entries.

To be more concrete, consider the example in Figure 5.6. We use yellow intervals to indicate blocking events caused by data contention, and green intervals to indicate the rest of time. In this example, transaction $t_2$ blocks transaction $t_1$ twice. The first time, there is no nested waiting, and our algorithm simply increases $score(< T_2, T_1 >)$ by 4 milliseconds. The second time, $t_1$ waits

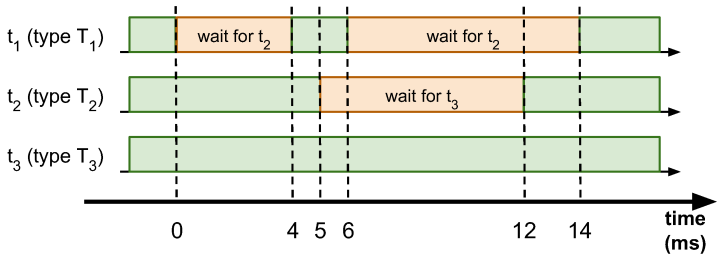



Figure 5.6: An example for Tebaldi's performance analysis algorithm.

for $t_2$ for 8 milliseconds, but during this time, $t_2$ is itself blocked by $t_3$. Our algorithm only increases $score(< T_2, T_1 >)$ by the time when $t_2$ is running (2 milliseconds), and increases $score(< T_3, T_2 >)$ by 6 milliseconds. All together, the three blocking events contribute 6 milliseconds to $score(< T_2, T_1 >)$, and 13 milliseconds to $score(< T_3, T_2 >)$ (7 milliseconds from $t_2$ directly waiting for $t_3$, and 6 milliseconds from the nested waiting).

The end objective is to calculate a score for each *conflict edge*, which is an *unordered* pair of transaction types. The score for the conflict edge between $T_i$ and $T_j$ is the sum of $score(< T_i, T_j >)$ and $score(< T_j, T_i >)$ over the period being analyzed (if $T_i = T_j$, it equals to $score(< T_i, T_i >)$). We identify as the performance bottleneck the conflict edge with the highest score.

This new performance profiling algorithm tracks more accurately the root cause of performance issues, and because it does not require tuning the workload's request rate, it can serve as an *online* algorithm to detect the performance bottleneck in production environments. On the other hand, it needs more computation, and in a distributed setting, it requires clocks of all

participating database nodes to be well synchronized. Our system currently relies on the Network Time Protocol (NTP) [63] to synchronize clocks. As Callas and Tebaldi are designed to work within a single data center, we are able to achieve fairly accurate results: NTP can achieve sub-millisecond precision in most of our experiments.

## 5.4 Optimization Stage

After determining which conflict edge limits performance, the next step is to find potential ways to optimize it. We achieve this by adjusting MCC's configuration so that the target conflict is handled by a better-suited CC mechanism. The key question is then *how* to adjust MCC's configuration; an additional challenge is to integrate such adjustments with the preprocessing logics, if any, required by various CC mechanisms (e.g., the static analysis needed by transaction chopping [80, 99] and runtime pipelining [95]) so that the entire reconfiguration can take place automatically.

### 5.4.1 Proposing New Configurations

In Callas, adjusting MCC's configuration is relatively simple, since the limited flexibility of its two-layer architecture leaves only a couple of possible options: one can either move transactions to a different group, or create a new group for them. With Tebaldi's hierarchical MCC, things become more complicated, as the new model brings both new opportunities and new challenges. On the one hand, it allows more flexibility to fine-tune CC mechanisms for

different conflict edges, and thus more opportunities for optimization. On the other hand, the new model's larger set of options introduces more complexity in determining the new configuration. It also raises the question of how to properly leverage this additional flexibility without inadvertently *hurting* performance: in the end, hierarchical MCC is unable to independently assign CC mechanisms for each individual conflict edge, and changing how one conflict edge is handled may affect other edges.

Depending on where the target conflict lies in the current configuration, our algorithm takes three different strategies to optimize it, but all of these strategies follow a single overarching criterion: changes in MCC's configuration should be kept as local as possible with respect to the bottleneck conflict. The intuition behind this policy is that, ideally, we only want to change how MCC handles the bottleneck conflict without affecting any other conflicts in the workload. Unfortunately, we cannot always achieve this ideal case; even so, we strive to minimize side effects to other conflict edges, e.g., by limiting them only to edges that involve one of the two transactions in the bottleneck edge.

**Case 1: Multiple Instances of the Same Transaction** We start with a simple but quite common case, shown on the left side of Figure 5.7: the bottleneck conflict is among transactions of the same type, say $T_1$. In this case, we are able to find an optimization strategy that makes purely local adjustments, as shown on the right side of the figure. The new configuration splits the leaf node containing $T_1$ by moving $T_1$ to a new leaf node managed

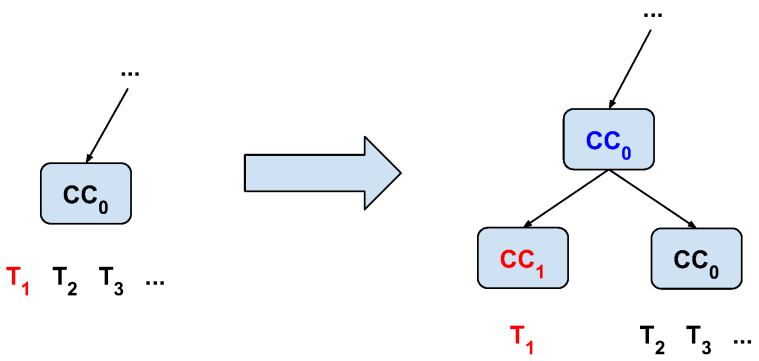



Figure 5.7: Adjustment for single-transaction bottleneck.

by a better-suited CC mechanism (shown in red). It then adds a non-leaf node with the original CC mechanism to regulate data conflicts between $T_1$ and other transactions (shown in blue). Thus, the new configuration only changes the CC mechanism regulating conflicts among transactions of type $T_1$, while handling all other conflicts, including those between $T_1$ and other transaction types, as in the original configuration. To determine which concurrency control mechanism should be associated with the new leaf (i.e., the best mechanism to regulate conflicts among $T_1$ transactions), Tebaldi iterates over all concurrency control mechanisms, forming multiple candidate configurations.

**Case 2: Transactions from the Same Group** We can generalize the previous strategy for handling single-transaction bottleneck to deal with bottlenecks due to conflicts between two transaction types currently mapped to

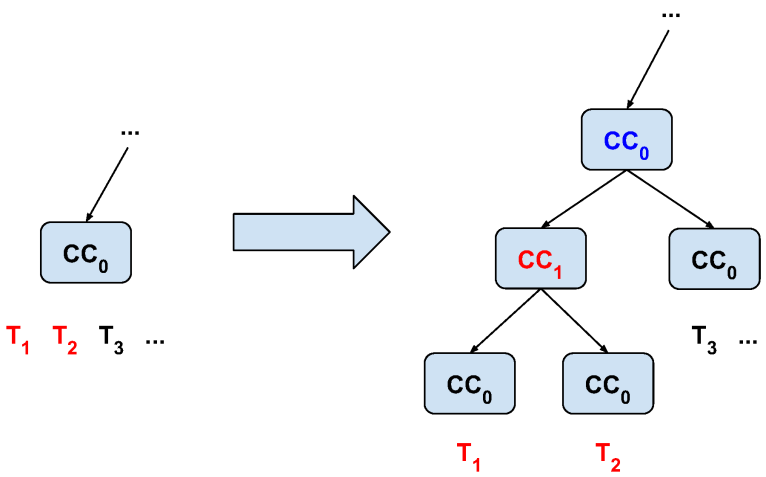
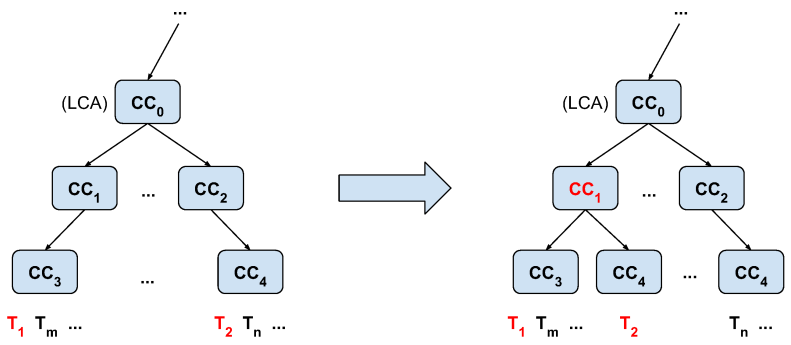


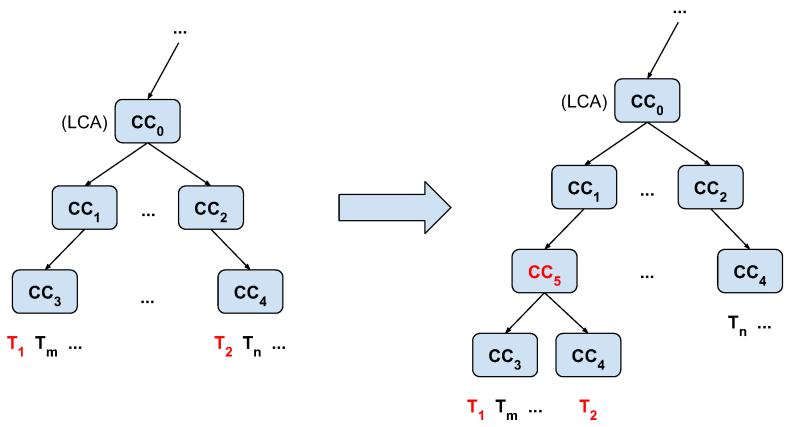

Figure 5.8: Adjustment for transactions from the same group.

the same group, such as $T_1$ and $T_2$ in Figure 5.8. The adjusted configuration, shown on the right, introduces a new CC mechanism, $CC_1$ (shown in red), to handle conflicts between $T_1$ and $T_2$, without changing how other conflicts are handled. The only difference from the single-transaction case is that $T_1$ and $T_2$ are placed in individual groups, so conflicts among multiple instances of transactions of type $T_1$ (or respectively, $T_2$) can be handled by the original CC mechanism, $CC_0$.

In our experience, these two cases already cover many bottleneck scenarios: much data contention originates from conflicts among multiple instances of the same transaction, as they share the same data access pattern. Moreover, as our initial configuration puts all update transactions in a single group regulated by two-phase locking, many bottlenecks arise from conflicts between two different transaction types from this group.




(a) Move $T_2$ to a node along the path from LCA to $T_1$'s group...




(b) Or create a node along the path from LCA to $T_1$'s group.

Figure 5.9: Strategies to handle conflicts across different groups.

**Case 3: Transactions from Distinct Groups** In this more general case, shown in the left side of Figure 5.9, perfectly localized adjustments are not usually possible. Nevertheless, we can minimize side-effects to other conflict edges by only changing the structure of the subtree rooted at the lowest common ancestor (LCA) of the two groups that include the two transactions in

the bottleneck edge. Specifically, we split the group that holds one of these transactions and move that transaction beneath one of the nodes along the path from the LCA to the other bottleneck transaction. For example, in Figure 5.9a, $T_2$ is moved beneath a node on the path from the LCA to $T_1$. This approach gives us the flexibility to choose between any of the different CC mechanisms along the two paths to handle conflicts between $T_1$ and $T_2$. Alternatively, we can create a new node along one of the two paths and add a new CC mechanism to handle these conflicts, as shown in red in Figure 5.9b. The location of the new node does not affect how we handle conflicts between $T_1$ and $T_2$, but changes the handling of the conflicts between $T_1$ (or respectively, $T_2$) and other transactions in the subtree.

**Filtering Candidate Configurations** In theory, MCC can work with any configuration, but in practice, not all candidate configurations are equally likely to bring good performance. There are several reasons for this. First, not all CC mechanisms are designed to optimize heavy data contention (e.g., 2PL). Second, not all CC mechanisms, and not all combinations of parent/child CCs, can enforce *efficiently* the consistent ordering that MCC requires of participating CC mechanisms. For example, we mentioned in the previous chapter (Section 4.3.2 and Section 4.4.3) that serializable snapshot isolation needs to adopt an inefficient batching technique for consistent ordering, unless all update transactions are placed in a single child group (as in our initial configuration).

These reasons motivate the introduction of CC-specific filters to remove candidate configurations that are unlikely to perform well. At this time, we support two types of filters. The first checks whether a CC mechanism is designed to perform well under heavy data contention; the other checks whether CC mechanisms on non-leaf nodes can efficiently support consistent ordering. Only candidates that pass these checks are considered in the next stage.

### 5.4.2 Cooperating with CC-specific Preprocessing

Some CC mechanisms require special preprocessing steps that make necessary changes to transaction codes and their own configurations (e.g., runtime pipelining performs a static analysis to reorder transactions and chop them into steps).

Our system supports two kinds of CC-specific preprocessing interfaces that can be invoked as necessary during the optimization stage. The first one allows a CC mechanism to perform a static analysis and make necessary changes to the code of the transactions in its group, such as reordering operations and adding CC-specific markers. For example, runtime pipelining (Section 4.4.2) uses this interface to implement the aforementioned static analysis; similarly, timestamp ordering (Section 4.4.4) analyzes transactions' read and write sets, and applies the *promise* optimization.

The second type of preprocessing is more interesting: it allows a CC mechanism to locally change the proposed MCC configuration at its node, thus allowing a CC mechanism to *refine* the candidate configuration with its own

knowledge. In particular, this allows our system to support a useful feature: *a limited form of partition-by-instance configurations.*

By default, our configuration algorithm only generates partition-by-type configurations that assign an entire transaction type to the same CC node. With CC-specific preprocessing, a CC node can choose to split itself into several identical copies, and provide a new partitioning function that remaps transactions in the original group to different copies on a per-transaction-instance basis. This effectively creates hybrid configurations that partition transactions first by types, and then by instances.

As an example, timestamp ordering (TSO) can often benefit from this partition-by-instance feature. TSO orders transactions using timestamps that are assigned at the beginning of transactions. But to enforce consistent ordering in Tebaldi, it needs to commit transactions in timestamp order (Section 4.4.4). This can cause false conflicts among transactions: at commit time, a transaction has to wait until all transactions with smaller timestamps finish, even if it does not have actual data conflicts with some or any of these transactions. Partition-by-instance feature helps relieve this problem by further partitioning transactions in a TSO group, separating transactions that are unlikely to conflict into different groups.

**Supporting Stored Procedures** Our automatic configuration framework does not require knowledge about the code run by each transaction, but those concurrency control mechanisms that analyze or change transaction code do

need such information. Therefore, our system supports both standard interactive transactions and stored procedures. We provide a basic programming language for stored procedures that supports if/else, for loops, local variables and arrays. We don't require all transactions to be implemented in stored procedures. However, transactions that benefit from CC mechanisms that need access to transaction code should be implemented as stored procedures.

**Conflicts on Reordering Operations** If multiple CC nodes in the same path of the tree reorder operations, they may cause a conflict: different CC mechanisms may prefer to order operations in different ways. We address this problem by prioritizing CC nodes placed higher in the path: when a CC node reorders operations in a transaction, it must not violate any existing decisions made by its parent nodes. This is because higher-level CC nodes manage more transactions than lower-level ones, so they may have more global restrictions on how to order operations.

## 5.5 Testing Stage

The testing stage switches MCC's configuration to each of the candidate configurations proposed by the optimization stage, and measures their end-to-end throughput. It then compares the best-performing candidate with the original configuration to decide the final configuration of the current iteration of the automatic configuration process.

Unlike Callas, which configures its concurrency control federation stati-

cally when the system starts. Tebaldi keeps evolving its configuration and does so dynamically, without requiring the database to be restarted for each reconfiguration. During the transition, some transactions may be still running under the old configuration, while others may have transitioned to the new one; we need to ensure proper isolation between them. To this effect, our system supports two different reconfiguration protocols with different applicability and performance: the *partial restart protocol* and the *online update protocol*.

### 5.5.1 The Partial Restart Protocol

The partial restart protocol is, in many ways, similar to a full database restart, but it avoids some major sources of overhead, such as the cost of retrieving logs from persistent storage and performing a full failure recovery. This protocol leverages Tebaldi's design choice that separates the concurrency control module from the storage module (Section 4.3), and only restarts the former module.

In this protocol, Tebaldi first stops accepting new transactions, and waits for ongoing transactions to finish. It then completely re-initializes the concurrency control module with the new configuration, and resumes the system. The protocol consists of three phases: clean-up, prepare, and apply.

**Clean-up Phase** The reconfiguration protocol is initiated and managed by a center node called the *reconfiguration server*. It starts the reconfiguration by broadcasting a clean-up message to all the transaction coordinators. Upon

receiving the message, a transaction coordinator buffers new transactions in a queue, and waits for all the ongoing transactions that it manages to finish. Optionally, a force-abort can be used to speed up reconfiguration after a timeout. The reconfiguration server also sends clean-up messages to all management nodes that manage background tasks, such as the garbage collection manager, to pause their activities. A clean-up confirmation is sent to the reconfiguration server when ongoing activities on a database node have finished.

**Prepare Phase** When the reconfiguration server receives all clean-up confirmations, it starts the prepare phase. The server broadcasts the new configuration and all CC-specific preprocessing instructions to all database nodes. These nodes then re-initialize their concurrency control module with the new configuration. All the old CC mechanism instances, together with all CC-specific in-memory states, are destroyed or recycled, and new CC mechanism instances are constructed. The data module is not affected in this procedure.

Tebaldi then populates the new concurrency control mechanisms' internal state, including indices, version maps, and lock tables. Here, Tebaldi follows the recovery protocol described in Section 4.5.4; but since the partial recovery does not affect the database state in the storage module, Tebaldi can skip the first two steps of the recovery protocol (retrieving logs and reconstructing database states), and directly reconstruct the concurrency control module's states. Since this is a fully local and in-memory procedure, the partial restart protocol is much cheaper than performing a full restart.

**Apply Phase** After the prepare phase completes on all the involved database nodes, the reconfiguration server broadcasts an apply request to all transaction coordinators and other management nodes in the clean-up phase to resume their execution. Transaction coordinators resume all pending new transactions and handle them with the new configuration.

Tebaldi separates the prepare and apply phase in reconfiguration to prevent a race condition: if we combine the two phases into a single reconfiguration request, different database nodes may receive the combined request at different times, and one node may start to issue transactions in the new configuration before other participating nodes know of it, causing an error.

### 5.5.2 The Online Update Protocol

Although the partial restart protocol avoids the recovery cost of a full restart, it can still cause temporary service interruption. In certain cases, we can use an online update protocol to mitigate such interruptions. The basic idea is that, since our optimization algorithm often makes only local changes to MCC's configuration, we may be able to substitute only a part of the concurrency control tree, rather than replacing the entire tree.

In this alternative protocol, we compare the old and the new configuration and find the lowest node in the old tree that is root of the subtress that includes all the changes. If that node is not the root of the entire MCC tree, we can perform reconfiguration without interrupting the database execution in two steps, as shown in Figure 5.10. First, we merge the two configurations

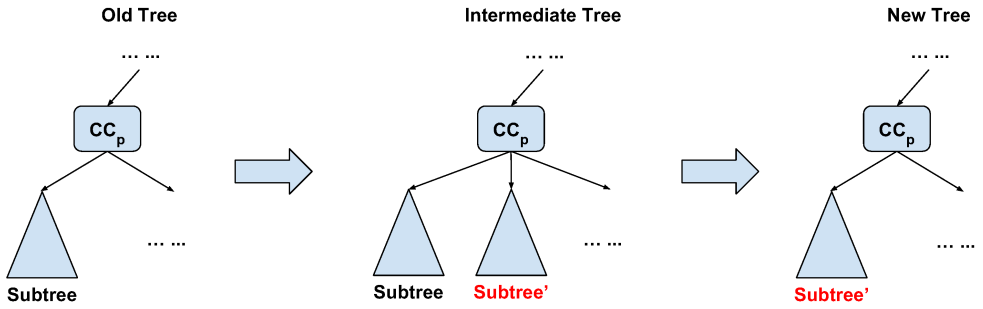



Figure 5.10: The Online Update Protocol.

by adding the new subtree to the parent node. Since the old subtree is not removed, ongoing transactions can continue running within the old subtree, while new transactions are handled by the new subtree. The concurrency control between transactions in the old and new configurations is handled by the parent node [1]. Second, once all transactions running within the old configuration complete, the old subtree can be (optionally) removed.

This protocol has an additional requirement: the parent CC node must allow changes in its children groups. Not all concurrency control mechanisms can do this. For example, such changes are not supported by the special version of SSI with only one read-write group (e.g., the root CC node of our initial configuration), since the intermediate configuration may introduce a second read-write group. In these cases, we fall back to the partial restart protocol.

[1]Note that the parent node may not be able to *efficiently* handle these data contentions, so this protocol can still cause visible performance degradation during reconfiguration.

## 5.6 Evaluation

This section reports on the performance of Tebaldi's automatic configuration algorithm. Its purpose is to answer the following questions:

- How much performance benefit can the automatic configuration algorithm reach?
- How close is that performance to that attainable from a manually configured MCC database?
- How does each component of Tebaldi's configuration algorithm perform, and what is its contribution to the performance results?

**System Implementation** The automatic configuration algorithm is implemented on top of the Tebaldi codebase; it is driven by an automatic configuration manager that also serves as the performance monitor in our new profiling algorithm (Section 5.3), and the reconfiguration server in our runtime reconfiguration algorithms (Section 5.5).

Comparing to the codebase used to evaluate hierarchical MCC in the previous chapter (Section 4.6), the new codebase also includes the durability protocol I described in Section 4.5.4. All experiments in this chapter are carried out with the durability feature enabled, and we use RocksDB [19] as the underlying persistent storage. The new codebase also comes with many performance optimizations. For example, when deploying multiple data nodes on

the same machine, we pin the working thread of each data node to a separate core. Because of these and other reasons discussed later, such as changes in CloudLab cluster and the use of stored procedures, performance numbers in this chapter are not directly comparable with those in previous chapters, even when they refer to the same experiment.

**Experimental Setup** All experiments in this section are carried out in a CloudLab [5] cluster of C8220 machines. Each machine is equipped with two Intel E5-2660 CPUs (20 physical cores in total), 256GB of memory, and a 10Gb Ethernet. Unless otherwise noted, our database system is distributed among 20 machines, and each machine runs 10 instances of transaction coordinators and 10 instances of data servers. An additional machine hosts the database's management nodes, including the automatic configuration manager.

Although these experiments use the same type of hardware as those in the previous chapter (Section 4.6), by the time we conducted these new experiments, CloudLab had changed the BIOS settings on these machines. Specifically, they set the machines' power and performance profile to maximize performance, and enabled I/O Acceleration Technology. These new BIOS settings yield better performance in baseline experiments—an additional reason why performance in this chapter are not directly comparable with those in the previous chapter.

**Benchmarks** We test our system with the same two benchmarks used in the previous chapter: TPC-C [41], and SEATS benchmark [46]. We modify these benchmarks in the same way described in Section 4.6 to adapt them to the key-value store interface. We also use the same sets of contention-heavy workloads to demonstrate MCC's benefit in handling high-contention workloads. Specifically, we populate TPC-C benchmark with ten warehouses, and run its test clients in a closed-loop. For SEATS, the workload was modified to simulate ticket selling for sporting events rather than airline flights: there are at most 50 "flights" at any single time, and each of them has 30,000 seats. Workloads for both the TPC-C and SEATS benchmarks come with a large numbers of concurrent clients (2,000) that saturate our database system (because of either contention, or resource bottlenecks), allowing us to measure the peak throughput of our database system.

We use stored procedures to implement transactions that are optimized by runtime pipelining or timestamp ordering. They are `new order`, `payment`, and `delivery` transactions in TPC-C, and `new` / `update` / `delete reservation` transactions in SEATS. These transactions always use stored procedures throughout the experiments we report in this chapter, even if they are not optimized in some of the tests [2]. Other transactions are left as interactive transactions.

[2]Using stored procedures reduces network roundtrips, which in turn reduces the severity of data contention. This is yet another reason why performance numbers in this chapter are often higher than those in previous chapter, especially for baseline configurations.

Unless otherwise specified, we run each experiment three times, and report the average number and the 95% confidence interval.

**Baselines** For TPC-C and SEATS, we compare the performance of our system with three baselines. The first baseline is a simple, stand-alone two-phase locking mechanism. The second one consists of the initial configuration of our iterative algorithm, as shown in Figure 5.2. Comparing against these two baselines tells us how much performance benefit our automatic configuration algorithm can bring to database applications. The third baseline instead serves as a performance upper bound: it runs Tebaldi with manually configured MCC hierarchies as described in Section 4.6. All three baselines are implemented within Tebaldi's framework by configuring MCC manually.

### 5.6.1 Performance of the TPC-C Benchmark

There are five types of transactions in TPC-C: `stock level` and `order status` are read-only transactions, while `new order`, `payment` and `delivery` both read and write. When running under serializable isolation, this benchmark exhibits many different types of data contentions, making it hard to manually identify performance bottlenecks and propose an appropriate MCC configuration.

**Baseline Performance** First, we look at the performance of TPC-C with two-phase locking. Figure 5.11 shows that stand-alone two-phase locking, at

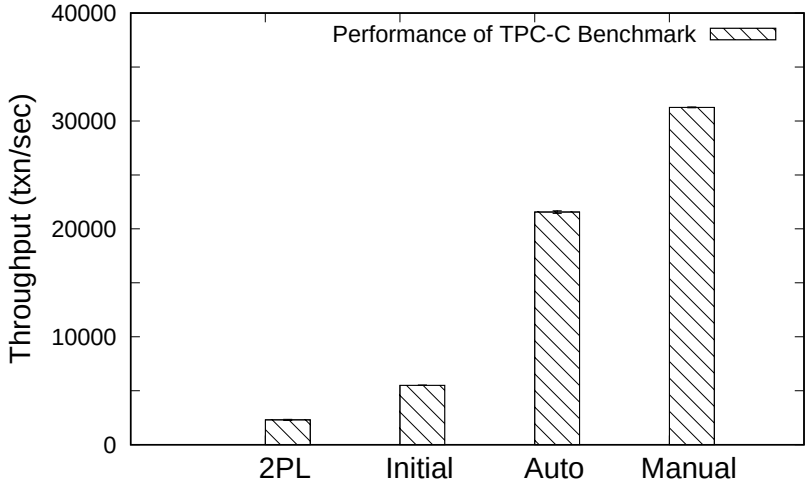



Figure 5.11: Performance of automatic configuration on TPC-C benchmark.

2,295 transactions per second, has the lowest throughput among all the configurations we consider. The reason for its low performance is that 2PL, as a conservative mechanism, cannot efficiently handle the heavy data contention in TPC-C's workload.

The initial MCC configuration of our optimization algorithm optimizes read-only transactions using a multi-versioning technique similar to multi-version 2PL. In TPC-C, this is enough to resolve the heavy conflict between `stock level` and the update transactions. The throughput of this configuration is 5,511 transactions per second (the second bar from the left of Figure 5.11): it is 2.4× higher than basic 2PL, but still relatively low, because of the heavy data contention among update transactions.

The rightmost bar in Figure 5.11 shows the throughput of our third baseline, which runs TPC-C with the 3-layer MCC configuration that we pro-

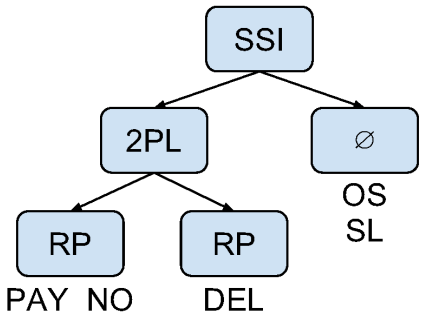
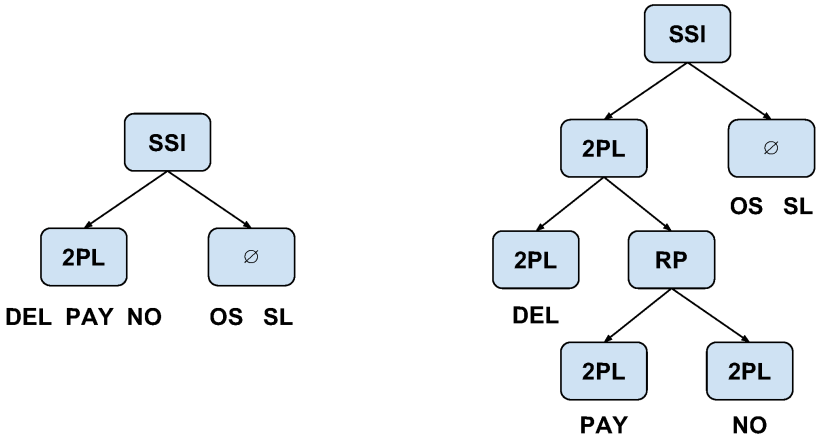



Figure 5.12: Manual configuration for TPC-C. Leaf nodes are labeled with transactions: `payment` (PAY), `new order` (NO), `delivery` (DEL), `order status` (OS), and `stock level` (SL).

posed in the previous chapter (Figure 5.12). The throughput is 31,264 transactions per second, which is 5.7× higher than the second baseline, and 13.6× higher than 2PL. This result shows the significant performance potential of Modular Concurrency Control. The key question is then, can we still get such benefits without requiring users to configure MCC manually?

**Performance of Automatic Configuration** Our automatic configuration algorithm takes four iterations to improve the performance of the TPC-C workload. The final configuration is shown in Figure 5.13d, and its throughput reaches 21,556 transactions per second (the third bar from the left in Figure 5.11). This is 3.9× higher than our initial configuration, 9.4× higher than 2PL, and 69% of what can be achieved through manual configuration.

We show the evolution of our automatically-generated configuration in Figure 5.13. In the first iteration, our system determines that the most severe bottleneck in the initial configuration (Figure 5.13a) is data contention




(a) Initial Configuration

(b) After the first round

(c) After the second round

(d) After the third round

Figure 5.13: Automatic configuration in TPC-C. Leaf nodes are labeled with transactions: `payment` (PAY), `new order` (NO), `delivery` (DEL), `order status` (OS), and `stock level` (SL).

between `new order` and `payment` transactions. This is indeed the case: both transaction types access the `warehouse` table, which has only ten rows, and with 2PL, `payment` needs to acquire a write lock here, blocking all other concurrent accesses. Our optimization algorithm proposes the candidate configuration shown in Figure 5.13b, which optimizes the bottlenecking conflict with runtime pipelining. The new configuration increases the throughput to 9,031 transactions per second.

In the second iteration, the conflicts among `payment` transactions become the next major bottleneck. This is because the first round of optimization did not change how conflicts within `payment` transactions are handled, and two-phase locking cannot efficiently deal with that. To address this problem, our system proposes to optimize the `payment` group with runtime pipelining, as shown in Figure 5.13c, increasing the throughput to 16,870 transactions per second.

The next major bottleneck, then, become the conflicts among `new order` transactions. As in the second iteration, our optimization algorithm chooses to optimize it with runtime pipelining. After this round, the end-to-end throughput reaches the final number of 21,556 transactions per second.

In the last iteration, our system detects that once again, it is conflicts among `new order` transactions that have become the most significant contention bottleneck. However, this conflict edge has already been optimized by runtime pipelining, and Tebaldi cannot find better optimizations, so the automatic configuration algorithm stops.

**Comparing with the Manual Configuration** The final MCC configuration derived by our algorithm (Figure 5.13d) is similar to the manual configuration of Figure 5.12. There are two differences: first, the algorithm does not optimize `delivery` transactions with runtime pipelining, since it is not the most significant data contention in the last round of iteration; second, the algorithm does not merge the three runtime pipelining groups between `new order` and `payment` transactions into a single group. In this particular case, having three runtime pipelining groups does not bring better performance; instead, the additional computation from these groups can degrade the performance.

In the end, our automatic configuration algorithm only reaches 69% performance of the manual configuration. There are two major reasons for this performance difference. First, as I have mentioned above, using three runtime pipelining groups to handle data contentions among `new order` and `payment` transactions degrades performance. Second, the quality of runtime pipelining's static analysis differs between the automatic and the manual configurations. In particular, the manual configuration also performs the static analysis required by runtime pipelining manually; this enables the *uniqueness* optimization [95], which can remove concurrency control for operations that are guaranteed to access different rows. Unfortunately, this optimization is hard to implement with fully automatic static analysis, since it requires solving hard problems in programming languages, such pointer and loop analysis.

### 5.6.2 Seats benchmark

Like TPC-C, the SEATS benchmark has different sources of data contentions, such as those between read-only and update transactions, and among `new reservation` and `delete reservation` transactions.

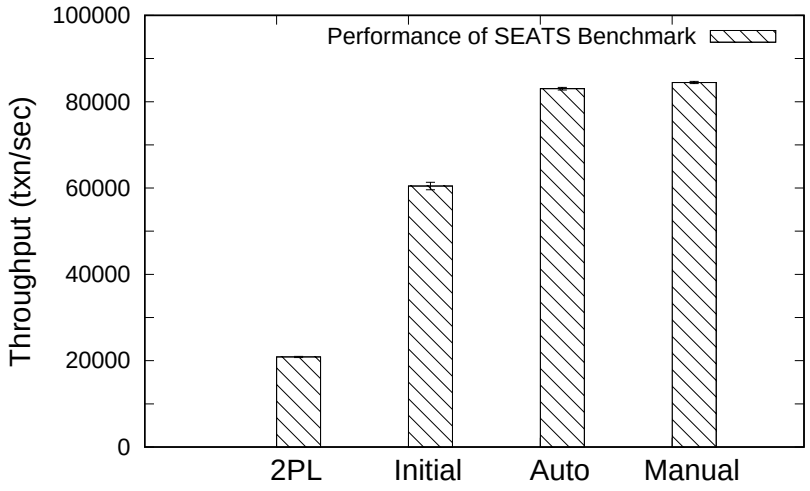



Figure 5.14: Performance of automatic configuration on SEATS benchmark.

**Baseline Performance** The first baseline experiment measures the throughput of the SEATS benchmark with monolithic two-phase locking. As shown by the left most bar of Figure 5.14, the throughput is 20,864 transactions per second. The conservative locking mechanism can become a major bottleneck when data contention is high in the workload, which is exactly the case in the SEATS benchmark.

The second baseline adopts the initial MCC configuration used in our automatic configuration algorithm. This configuration efficiently handles data

conflicts between read-only and update transactions using a multi-versioned mechanism, and its performance already reaches 60,452 transactions per second (the second bar from the left in Figure 5.14), which is 2.9× higher than 2PL. But this configuration does not optimize update transactions, so its performance is still limited by data contentions among these transactions.

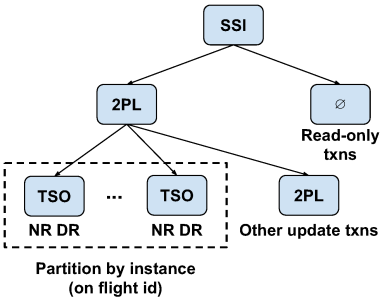



Figure 5.15: Manual configuration for SEATS. Leaf nodes are labeled with transactions: `new reservation` (NR) and `delete reservation` (DR).

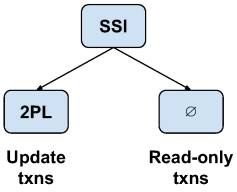
The third baseline measures the performance of the manually configured MCC federation for the SEATS benchmark. In Section 4.6.2 we proposed a manual configuration that places `new`, `update` and `delete reservation` in TSO groups, and uses a partition-by-instance technique to dispatch transactions that operate on different "flights" to different TSO group instances. But one can actually do better, by observing that `update reservation` transactions in fact conflict much less often than the other two transactions. Both `new` and `delete reservation` transactions modify the `flight` table, so they conflict as long as two transactions are on the same *flight*. `Update reservation` transactions, however, do not modify the `flight` table: they only conflict when

two transactions access the same *seat*. Hence optimizing only `new` and `delete reservation` transactions with TSO (Figure 5.15) can actually result in better performance, since it avoids the unnecessary overhead of TSO for `update reservation`. With this new configuration, the throughput reaches 84,453 transactions per second (the rightmost bar of Figure 5.14), which is 4× higher than 2PL, and 40% higher than the initial configuration used in our configuration algorithm.

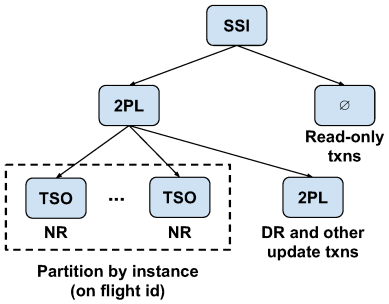
**Performance of Automatic Configuration** Starting with the initial configuration, our algorithm takes three iterations to optimize the SEATS workload. The final configuration is the same as our manual configuration, and it also leverages the partition-by-instance optimization (Figure 5.16c). Therefore, it achieves a similar performance of (83,008 transactions per second, third bar from the left in Figure 5.14). Figure 5.16 shows the evolution of the configuration generated automatically by our algorithm.

The most severe bottleneck in the initial configuration (Figure 5.16a) is the conflict within instances of the `new reservation` transactions. The conflict between `new` and `delete reservation` transactions is also severe, but it scores lower in our profiling algorithm, so in the first iteration, the optimization stage only focuses on the conflict within `new reservation` transactions.

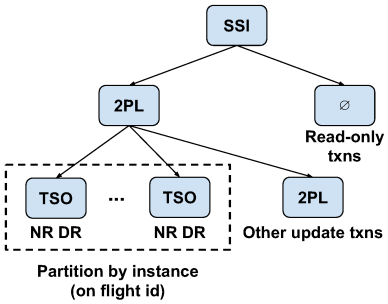
Our algorithm attempts to optimize this bottleneck by placing `new reservation` transactions into a new group and by adopting a better-suited CC mechanism. Two candidates are proposed: runtime pipelining and times-




(a) Initial Configuration




(b) After the first round




(c) After the second round

Figure 5.16: Automatic configuration in SEATS. Leaf nodes are labeled with transactions: `new reservation` (NR) and `delete reservation` (DR).

tamp ordering (TSO). Furthermore, in the case of TSO, the CC-specific preprocessing algorithm detects that `new reservation` transactions can be further partitioned in a by-instance manner, using the `flight id` input as the partitioning key (Figure 5.16b). After evaluating both candidates, our system finds that TSO gives better performance. After this iteration, throughput increases to 64,978 transactions per second. This is only slightly better than the initial configuration, since another critical bottleneck, data conflicts between `new` and `delete reservation` transactions, is not optimized.

In the second iteration, the most severe bottleneck become the con-

flicts between `new` and `delete reservation` transactions. This conflict is challenging, since it involves transactions from two different groups. Nonetheless, our optimization algorithm proposes two candidates following the strategies described in Section 5.4: it either moves `delete reservation` to the `new reservation`'s TSO group, or vice-versa. After testing both options, it determines that the best one is to move `delete reservation` to the group of `new reservation` and to use TSO to handle all conflicts among those transactions (Figure 5.16c).

Our algorithm also considers adding a non-leaf CC node somewhere along the path from the two transactions' lowest common ancestor to one of the group. However, all such candidates do not pass the CC-specific filters. In particular, the CC mechanism of the new non-leaf node cannot be TSO, since TSO is not efficient when serving as a non-leaf mechanism. It cannot be runtime pipelining either, since runtime pipelining requires a child node to report in-group dependency information after each operation, which is hard to achieve in TSO.

In the last iteration, Tebaldi finds that the performance bottleneck once again become the conflicts among `new reservation` transactions. The algorithm stops here, since it cannot further optimize this bottleneck.

### 5.6.3 Overhead of the Profiling Algorithm

Tebaldi's performance profiling algorithm gathers information about individual data contention events at runtime. Doing so raises the concern of

the profiling overhead; further, if profiling is too costly, it can even change the performance of the workload.

To address this concern, we run the TPC-C and SEATS benchmarks and compare the their performance with the profiling logic turned on and turned off, using a macro to remove the performance profiling code and data structures at compile time. We evaluate four testcases, running TPC-C and SEATS benchmarks under Tebaldi's initial MCC configuration (Figure 5.13a and Figure 5.16a), and under the manual configurations (Figure 5.12 and Figure 5.15). With Tebaldi's initial MCC configuration, both benchmarks suffer from severe contention bottleneck. We use these workloads to verify if the profiling logic increases the size of critical sections in concurrency control. With the manual configurations, these benchmarks have much higher throughput, consume more computation resources, and generate more profiling logs. We use these workloads to verify if the profiling logic has significant computational costs.

Figure 5.17 shows the performance of the four testcases. Workloads with the "-I" suffix use Tebaldi's initial MCC configuration, and workloads with the "-M" suffix use manual configuration. Results are normalized with the throughput when performance profiling is turned off. We find that, in all testcases, enabling performance profiling reduces throughput by less than 2%, since the actual performance analysis does not take place on transaction coordinator and data server nodes, where data and CC mechanisms reside. Instead, these nodes only need to generate logs for data conflict events, and

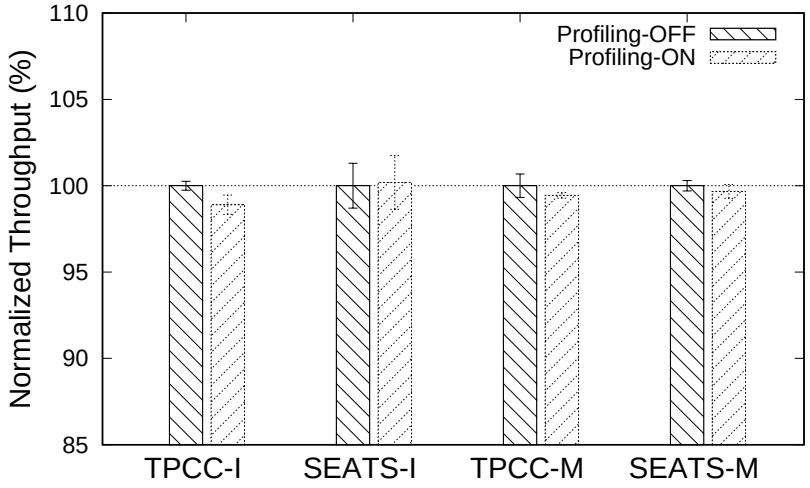



Figure 5.17: Overhead of Performance Profiling.

batch them before sending them to a separate performance monitor node for further analysis.

### 5.6.4 Benefit of Supporting Partition-by-instance

Although our algorithm mainly focuses on MCC configurations that partition transactions by their application-level types, it also supports a limited form of partition-by-instance configurations by cooperating with CC-specific preprocessing algorithms (see Section 5.4). We now measure the benefit of this feature.

Consider again the SEATS benchmark (Section 5.6.2). In this workload, our configuration algorithm attempts to optimize the performance of `new` and `delete reservation` transactions with timestamp ordering (TSO). With the help of the partition-by-instance feature, the proposed MCC config-

| Configuration | Partition-by-instance | Throughput (Txn/sec) |
|---|---|---|
| After the first iteration | ON | 64,978 |
| | OFF | 15,566 |
| After the second iteration | ON | 83,008 |
| | OFF | 13,131 |

Table 5.1: Performance of the SEATS benchmark with and without the partition-by-instance optimization.

uration can separate transactions about different flights to different groups. This additional flexibility in federating CC mechanisms prevents TSO from unnecessarily ordering transactions that rarely conflict, and it is the key to get any performance benefit from TSO.

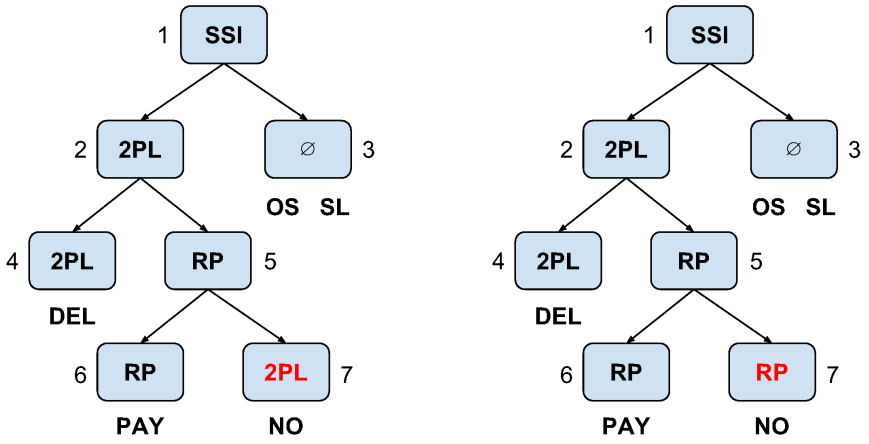
Table 5.1 shows the results of running SEATS benchmark using the MCC configuration proposed by the first and second iteration of our configuration algorithm, but without using the partition-by-instance feature. The performance of the first configuration (putting `new reservation` transactions into the TSO group) drops drastically from 64,978 to 15,566 transactions per second. The performance of the second configuration (putting both `new` and `delete reservation` transactions into the TSO group) drops even further, from 83,008 to 13,131 transactions per second. The reason is that, with a single TSO group, all transactions in the group are totally-ordered by an increasing timestamp (even if two transactions don't have data conflict), and they must commit in this timestamp order to enforce consistent ordering. The unnecessary cost from these additional ordering constraints overwhelms TSO's benefit for the transactions that actually conflict with each other, so much

that the performance is even worse than for monolithic 2PL. Adding `delete reservation` transactions to the group further increases such cost, and further reduces performance.

### 5.6.5 Overhead of Different Reconfiguration Protocols

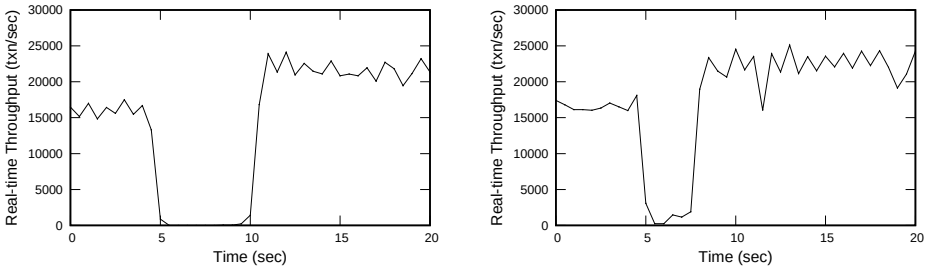
Finally, we measure the performance of Tebaldi's two reconfiguration protocols: partial restart and online update (Section 5.5). We adopt each of them in the TPC-C benchmark, and measure the real-time performance when reconfigurations occur. As we have seen in Section 5.6.1, there are three reconfigurations in the TPC-C benchmark. Among them, the first one cannot be handled by the online update protocol, since the SSI root node does not allow children groups to change (see the limitation of the online update protocol in Section 5.5); Tebaldi then falls back to the partial restart protocol. The other two reconfigurations can be handled by both protocols. For simplicity, we only present the last reconfiguration from Figure 5.18a to Figure 5.18b (CC nodes are labeled with numbers for reference, and changes are marked in red). The performance of the second reconfiguration is similar.

Figure 5.19a shows the real-time performance during reconfiguration with the partial restart protocol. Reconfiguration starts at about the 5-second mark in the figure, and takes 6 seconds to complete, from the first performance drop to regaining full throughput. Within this interval, the clean-up phase takes about 0.5 seconds to wait for ongoing transactions to finish (while the throughput gradually drops to zero). Then, the prepare phase takes 3.5 seconds




(a) The old MCC configuration (b) The new MCC configuration

Figure 5.18: The third reconfiguration in TPC-C.




(a) Partial restart protocol (b) Online update protocol

Figure 5.19: Performance of the third reconfiguration in TPC-C.

to switch to the new configuration and recover the root CC's internal states (the throughput stays at zero). Finally, the database spends another 2 seconds to resume the workload and ramp up.

In the online update protocol, we substitute the entire runtime pipelin-

ing subtree rooted at node 5. This is achieved by appending the new subtree in Figure 5.18b to the parent 2PL node (node 2) in the old configuration, and by redirecting new transactions to the new subtree. Note that this reconfiguration only changes node 7, but its parent, the runtime pipelining node 5, does not allow change of children groups. So the online update protocol moves one level up along the MCC hierarchy and changes the entire subtree at node 5.

Figure 5.19b shows the real-time performance of the online update protocol. There are two major differences with the partial restart protocol. First, the entire reconfiguration only takes about 3 seconds to complete, which is half the time of the partial recovery protocol. This is because the online update protocol does not need to reconstruct CC's internal state, as it seamlessly transforms the old MCC hierarchy to the new one. Second, at no time the throughput is zero (even in the one second period immediately following the beginning of reconfiguration, when the throughput is lowest, Tebaldi still processes about 230 transactions per second), since the online update protocol itself does not block ongoing or incoming transactions—ongoing `payment` and `new order` transactions continue to run in the old subtree, while new ones are handled by the new subtree; conflicts between them are handled by the parent 2PL node. However, throughput during the reconfiguration is low, since 2PL cannot handle those conflicts efficiently.

Finally, we compare our reconfiguration protocols with a straw-man approach that simply restarts Tebaldi to change its configuration. We shut down Tebaldi as soon as the third reconfiguration starts, and find that Tebaldi

takes about 19 seconds to reload logs and recover from the failure. Therefore, even the partial restart protocol is significantly more efficient than a full system restart.

### 5.6.6 Comparing against Single-Machine Databases

Tebaldi's hierarchical MCC technique is most effective in distributed databases, where network roundtrips can increase the severity of data contention bottlenecks. Therefore, this dissertation has mainly focused on a distributed setting, and compared Tebaldi's performance against monolithic CCs in that setting.

Indeed, there are many factors that can lead to choosing a distributed database rather than a single-machine one. For example, the dataset may be too large to fit in a single machine; the I/O or computational cost of a high workload request rate may overwhelm what a single machine can provide; further, if a database needs to be replicated, running transactions necessarily involves network roundtrips that can increase the severity of data contention for simple concurrency controls like 2PL.

Further, even when using a single-machine database is possible, it may not solve the data contention problem, as it is hard to implement all transactions in stored procedures, since doing so can largely increase the complexity of programming and maintaining the application code [71]. So, even for a single-machine database system, running interactive transactions can involve one roundtrip for each operation (between the application and the database

system). This can reduce the efficiency of simple concurrency control techniques, similar (though at a smaller scale) to what happens in a distributed setting.

Even so, it is fair to ask whether running simple CCs (like 2PL and SSI) in a single-machine setting can outperform Tebaldi cluster (with hierarchical MCC) by avoiding the cost of network roundtrips. To answer this question, we compare Tebaldi cluster's performance against that of monolithic CC mechanisms in *single-machine* and *single-threaded* settings, including against the widely-used single-machine database: PostgreSQL [16].

**Experimental Settings** We measure the performance of the TPC-C benchmark. As in Section 5.6.1, we implement `new order`, `payment` and `delivery` transactions as stored procedures, and the rest as interactive transactions. We compare the performance of Tebaldi in a cluster (which we have seen in Section 5.6.1) against three single-machine / single-threaded baselines. In the first baseline, we deploy Tebaldi on a single machine to simulate a single-machine database, and use stand-alone 2PL as the concurrency control mechanism. The second baseline is similar to the first one, but constrains Tebaldi to use only one transaction coordinator and one data server, simulating a "single-threaded" setting.

Of course, as Tebaldi is designed as a distributed system, it is not optimized for a single-machine setting. For example, Tebaldi's message-passing framework can still cause delays when passing messages between two database

nodes on the same machine. To provide more credible results, the third baseline measures TPC-C's performance on a widely-used single-machine database system, PostgreSQL [16]. We use the latest version (11.0) of PostgreSQL, and configure it to run transactions at the serializable isolation level. To achieve a fair comparison with Tebaldi, we disable PostgreSQL's replication and its synchronous flushing (durability) features. PostgreSQL eventually needs to write data to disk, so we configure it to run on top of an in-memory file system (tmpfs) to prevent disk I/O from becoming a bottleneck. We also port the TPC-C benchmark to the SQL language, and try to optimize it with SQL features (such as marking `order status` and `stock level` as read-only transactions, and using the `UPDATE... RETURNING...` clause to perform read and modify in a single query). Unlike the previous two baselines, PostgreSQL uses serializable snapshot isolation as its concurrency control mechanism [72].

For each of these baselines, we run TPC-C clients either on the same machine as the database system, or on a different machine. We report the configuration that gives the highest throughput.

| Setting | Throughput (txn/sec) |
|---|---|
| Tebaldi Single-Machine (2PL) | $2{,}065 \pm 48$ |
| Tebaldi Single-Thread (2PL) | $1{,}073 \pm 68$ |
| PostgreSQL (SSI) | $6{,}964 \pm 486$ |
| Tebaldi Single-Machine (3-layer) | $4{,}005 \pm 44$ |

Table 5.2: TPC-C's performance in single-machine settings.

**Experiment Results** Table 5.2 summarizes the performance of the three baselines. Running 2PL in Tebaldi in a single-threaded setting only handle about 1,000 transactions per second, as the CPU bottlenecks performance. Running 2PL in Tebaldi on a single-machine actually achieves a performance similar to running 2PL in Tebaldi on a cluster (about 2,300 transactions per second, see Section 5.6.1). The bottleneck here is data contention. Since Tebaldi's design and implementation (especially, its message-passing framework) is not optimized for a single-machine setting, running 2PL on a single machine in Tebaldi does not improve throughput.

Running TPC-C in PostgreSQL gives about 7,000 transactions per second. This is higher than running 2PL in Tebaldi. This is expected: PostgreSQL uses SSI for concurrency control, which, as we have seen in Section 4.6.1, can handle TPC-C workload more efficiently than 2PL. Still, this number is much lower than what hierarchical MCC can achieve when Tebaldi is run on a cluster: more than 21,000 transactions per second with automatic configuration, and more than 31,000 transactions per second with manual configuration (see Section 5.6.1).

We also measure how Tebaldi's hierarchical MCC performs on a single machine: Tebaldi's 3-layer federation (Figure 5.12) can process about 4,000 transactions per second. So, in a single-machine setting, hierarchical MCC still outperforms 2PL, but fares worse than PostgreSQL (since hierarchical MCC uses more CPU resources than a single CC, and Tebaldi is not optimized for the single-machine setting). However, as we have seen in Section 5.6.1, hierarchical

MCC allows us to easily scale out Tebaldi to a 20-machine cluster, which yields about 8× higher performance despite the heavy data contention and network roundtrips.

# Chapter 6

# Conclusion

This dissertation takes a major step toward achieving high performance ACID transactions without sacrificing its benefit of ease-of-programming. The starting point of our work is Modular Concurrency Control, a recent approach to federate concurrency control mechanisms for better performance. This dissertation presents Tebaldi, a new distributed key-value store that addresses two major problems in the current embodiment of MCC, bringing it to the next level.

Chapter 4 introduces hierarchical MCC, Tebaldi's new approach to harness the performance opportunity of combining different specialized concurrency controls within the same database. With this new model, Tebaldi can partition data conflicts at a fine granularity and match them to a wide variety of concurrency control mechanisms, within a framework that is modular and extensible.

Chapter 5 systematically explores how to automatically manage Tebaldi's MCC configuration, and presents an algorithm that iteratively adjust Tebaldi's configuration to improve its performance. This feature allows Tebaldi to hide the complexity of configuring MCC from database users, making sure that such

complexity will not nullify the initial motivation of MCC: improving ACID's performance without sacrificing ease-of-programming.

As a research prototype, Tebaldi has limitations. At this time, it only supports four CC mechanisms—we design Tebaldi with the goal of making it modular and extensible, but adding new CC mechanisms requires them to undergo non-trivial changes to enforce consistent ordering, and not all CC mechanisms (and their combinations) can achieve this easily. Tebaldi uses heuristics to manage its own configuration, but as performance profiling and optimization are, in general, hard, these heuristics may not always work well.

There are many design aspects in a database system that can affect its performance, such as the data model, the concurrency control mechanism, the replication protocol, and the techniques to enforce atomicity and durability... While this dissertation only addresses one of these aspects, as data contention is ubiquitous and concurrency control can be costly, we believe this work is an important building block towards achieving high performance for ACID transactions.

# Bibliography


[1] Apache HBase. http://hbase.apache.org/.

[2] AWS databases. http://aws.amazon.com/products/databases/.

[3] Azure SQL database. http://azure.microsoft.com/en-us/services/sql-database/.

[4] BerkeleyDB. https://www.oracle.com/database/berkeley-db.

[5] Cloud Lab. https://www.cloudlab.us/.

[6] Cloud spanner. https://cloud.google.com/spanner/.

[7] CockroachDB. https://www.cockroachlabs.com/.

[8] Microsoft SQL Server. http://www.microsoft.com/sqlserver/.

[9] Microsoft Transact-SQL Document. https://docs.microsoft.com/en-us/sql/t-sql/.

[10] MySQL. https://www.mysql.com.

[11] MySQL 5.7 Reference Manual - MySQL NDB Cluster 7.5 and NDB Cluster 7.6. https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster.html.

[12] MySQL 8.0 Reference Manual. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[13] MySQL Cluster. http://www.mysql.com/products/cluster/.

[14] Oracle 18c Document. http://docs.oracle.com/en/database/oracle/oracle-database/18/.

[15] Oracle Database. http://www.oracle.com/database/.

[16] Postgres SQL. http://www.postgresql.org/.

[17] PostgreSQL Manual. https://www.postgresql.org/docs/10/static.

[18] Redis. https://redis.io/.

[19] RocksDB. https://rocksdb.org/.

[20] SimpleDB. http://aws.amazon.com/simpledb/.

[21] *American National Standard for Information Systems : database language : SQL*. ANSI X3.135-1992. American National Standards Institute, 1992.

[22] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering*, pages 67–78. IEEE, 2000.

[23] Atul Adya and Barbara H Liskov. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis,

Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[24] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[25] Thomas E Anderson and Edward D Lazowska. *Quartz: A tool for tuning parallel program performance*, volume 18. ACM, 1990.

[26] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[27] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[28] Catriel Beeri, Hans-Jörg Schek, and Gerhard Weikum. Multi-Level Transaction Management, Theoretical Art or Practical Need? In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '88, pages 134–154, London, UK, 1988. Springer-Verlag.

[29] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[30] P. A. Bernstein, W. S. Wong, and D. W. Shipman. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 5:203–216, 1979.

[31] Philip A Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[32] Philip A Bernstein and Nathan Goodman. Multiversion Concurrency Control-Theory and Algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[33] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[34] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of Multidatabase Transaction Management. In *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research*, volume 2, pages 23–56. IBM Press, 1992.

[35] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, Vancouver, Canada, 2008. ACM.

[36] Arvola Chan, Stephen Fox, Wen-Te K Lin, Anil Nori, and Daniel R Ries. The Implementation of an Integrated Concurrency Control and Recovery Scheme. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pages 184–191. ACM, 1982.

[37] Arvola Chan and Robert Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, (2):205–212, 1985.

[38] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, Berkeley, CA, USA, 2006. USENIX Association.

[39] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[40] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[41] Transaction Processing Performance Council. TPC benchmark C, Standard Specification Version 5.11, 2010.

[42] James Cowling and Barbara Liskov. Granola: Low-overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[43] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: an elastic transactional data store in the cloud. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[44] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Pro-*

*ceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[45] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[46] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment*, 7(4), 2013.

[47] Deborah DuBourdieux. Implementation of Distributed Transactions. In *Berkeley Workshop*, pages 81–94, 1982.

[48] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[49] Jose M. Faleiro and Daniel J. Abadi. Rethinking Serializable Multiversion Concurrency Control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, July 2015.

[50] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and

Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005.

[51] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

[52] Jim Gray. Notes on data base operating systems. In *Advanced Course: Operating Systems*, pages 393–481, 1978.

[53] Jim N Gray, Raymond A Lorie, Gianfranco R Putzolu, and Irving L Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[54] Robert J Hall. Cpprofj: Aspect-capable call path profiling of multi-threaded java applications. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pages 107–116. IEEE, 2002.

[55] Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *Third Biennial Conference on Innovative Data Systems Research*, pages 132–141, 2007.

[56] Evan P. C. Jones, Daniel J Abadi, and Samuel Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In

*Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 603–614. ACM, 2010.

[57] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Mchoppinoadden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[58] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[59] Hsiang-Tsung Kung and John T Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[60] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, April 2010.

[61] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the 2016 International*

*Conference on Management of Data*, SIGMOD '16, pages 1659–1674, San Francisco, California, USA, 2016. ACM.

[62] Sharad Mehrotra, Henry F Korth, and Avi Silberschatz. Concurrency Control in Hierarchical Multidatabase Systems. *The VLDB Journal - The International Journal on Very Large Data Bases*, 6(2):152–172, 1997.

[63] David Mills, Jim Martin, Jack Burbank, and William Kasch. Network time protocol version 4: Protocol and algorithms specification. Technical report, 2010.

[64] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[65] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[66] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 479–494, Broomfield, CO, October 2014. USENIX Association.

[67] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 517–532, 2016.

[68] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-memory Transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 511–524, Broomfield, CO, 2014. USENIX Association.

[69] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 677–689, Melbourne, Victoria, Australia, 2015. ACM.

[70] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[71] Andrew Pavlo. What are we doing with our lives?: nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 3–3. ACM, 2017.

[72] Dan R. K. Ports and Kevin Grittner. Serializable Snapshot Isolation

in PostgreSQL. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, August 2012.

[73] Dan Pritchett. BASE: An Acid Alternative. *Queue*, 6:48–55, May 2008.

[74] Calton Pu. Superdatabases for Composition of Heterogeneous Databases. In *Proceedings of the 4th International Conference on Data Engineering*, pages 548–555. IEEE, 1988.

[75] Yoav Raz. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Mangers Using Atomic Commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 292–312. Morgan Kaufmann Publishers Inc., 1992.

[76] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[77] Werner Schaad, Hans-J Schek, and Gerhard Weikum. Implementation and Performance of Multi-Level Transaction Management in a Multidatabase Environment. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering, 1995: Distributed Object Management*, pages 108–115. IEEE, 1995.

[78] Ralf Schenkel and Gerhard Weikum. Integrating Snapshot Isolation into Transactional Federations. In *Cooperative Information Systems*, pages 90–101. Springer, 2000.

[79] Lui Sha, John P Lehoczky, and Douglas E Jensen. Modular Concurrency Control and Failure Recovery. *IEEE Transactions on Computers*, 37(2):146–159, 1988.

[80] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[81] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littleeld, and Phoenix Tong. F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 777–778. ACM, 2012.

[82] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, Cascais, Portugal, 2011. ACM.

[83] Michael Stonebraker. Stonebraker on nosql and enterprises. *Communications of the ACM*, 54(8):10–11, 2011.

[84] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural

Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[85] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. Bringing Modular Concurrency Control to the Next Level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 283–297. ACM, 2017.

[86] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, Scottsdale, Arizona, USA, 2012. ACM.

[87] Transaction Processing Performance Council. The TPC-C home page. http://www.tpc.org/tpcc.

[88] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1643–1658. ACM, 2016.

[89] William E Weihl. Distributed Version Management for Read-Only Actions. *IEEE transactions on Software Engineering*, (1):55–64, 1987.

[90] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(2):249–282, 1989.

[91] Gerhard Weikum. A Theoretical Foundation of Multi-level Concurrency Control. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 31–43, Cambridge, Massachusetts, USA, 1986. ACM.

[92] Gerhard Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[93] Gerhard Weikum and Hans-J. Schek. Database Transaction Models for Advanced Applications. pages 515–553. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[94] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. SALT: Combining ACID and BASE in a Distributed Database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014. USENIX Association.

[95] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 279–294, Monterey, California, 2015. ACM.

[96] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 231–243. ACM, 2018.

[97] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, San Francisco, California, USA, 2016. ACM.

[98] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, Monterey, California, 2015. ACM.

[99] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geodistributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

[100] Fang Zhou. wPerf: Identifying Critical Waiting in Multi-threaded Applications. In *SOSP Student Research Competition 2017 (SOSP'17 SRC)*.

# Vita

Chunzhi Su was born in Zibo, Shandong, China, and he lived there during his childhood. In 2005, he went to Shanghai to attend No.2 High School Attached to East China Normal University. After graduating from high school, he attended Shanghai Jiao Tong University in 2008, and received bachelor's degree in Computer Science. In 2012, he joined the University of Texas at Austin as a Ph.D. student in Computer Science.

Permanent email: suchunzhi@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.