The Dissertation Committee for Chao Xie
certifies that this is the approved version of the following dissertation:

High-Performance Transactional Storage

Committee:

Lorenzo Alvisi, Supervisor

Emmett Witchel

Keshav Pingali

Marcos Aguilera

High-Performance Transactional Storage

by

Chao Xie, B.E., M.S.C.S.

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

August 2016

Acknowledgments

The five years of my Ph.D. life in Austin is mixed with happiness, anxiety, excitement, disappointment and many other feelings. But it is treasure to me to have a chance to go though this unforgotten part of my life. I am really lucky to work with and learn from a group of smart, knowledgeable, and self-motivated people.

My advisor, Lorenzo Alvisi, inspires me with his deep thinking, rigorous logic and precise expression. He showed me the way to do scientific research and guided me to present research work clearly. His attention to details also helped me to realize how important a small thing can be.

My other committee members not only provided great suggestions for improving this thesis but also inspired me for future works.

I am extremely lucky to work with other fellow graduate students in LASR group. In particular, I want to thank Prince Mahajan and Chunzhi Su. As a senior student, Prince gave me a lot of guidance on building systems, and introduced me the research world of distributed transactions when I started my Ph.D. program. All these knowledge is quite useful for me when working on projects. Chunzhi worked with me closely through these years. I can always get a clearer thought through the discussions between us, though sometimes it started from disagreements. I also want to thank Manos Kapritsos, Yang

Wang, Navid Yaghmazadeh, Cody Littley, Natacha Crooks, Sebastian Angel, Trinabh Gupta, Sangmin Lee for providing great advice and help.

I want to thanks the staff of Emulab and Cloudlab. They are very kind to help me to schedule test machines for experiments and handle hardware failures quickly even on weekends. Without their help, I cannot finish my work.

Finally, I want to thank my parents and Wenqian for their understanding and supports. Their accompanying, ignoring the distance, courage me to pursue my goal, and bring me happiness.

CHAO XIE

The University of Texas at Austin
August 2016

High-Performance Transactional Storage

Publication No. _____

Chao Xie, Ph.D.
The University of Texas at Austin, 2016

Supervisor: Lorenzo Alvisi

Developers face a fundamental tension between performance and ease of programming when building complex applications. On the one hand, by freeing developers from having to worry about concurrency and failures, ACID transactions can greatly simplify the task of building applications. These benefits, however, have a cost: in a distributed setting, ACID transactions can limit performance and make systems hard to scale in the presence of contention. On the other hand, the BASE approach can achieve higher performance by providing weakened semantics. Embracing the BASE paradigm, however, exacts its own heavy price: once one renounces consistency guarantees, it is up to developers to explicitly code in their applications the logic necessary to ensure consistency in the presence of concurrency and faults, making this task complex and error-prone.

This dissertation aims to resolve this tension by building database systems that provide both the ease of programming of the ACID approach and

the performance that the BASE approach can provide. Our approach depends on the observation that different transactions affect overall performance of applications in different ways.

Traditional ACID databases ignore this diversity: they provide a single implementation of the same uniform abstraction across all transactions. This dissertation explores two different ways to leverage the previews key observation to combine performance and ease of programming, and presents two systems, Salt and Callas, inspired by them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

ACID transactions are a powerful primitive. By freeing developers from having to worry about concurrency and failures, they can greatly simplify the task of building applications. These benefits, however, have a cost: in a distributed setting, ACID transactions can limit performance and make systems hard to scale in the presence of contention. These costs have led some systems, such as Mega-Store [23], ElasTras [38], and Microsoft's Cloud SQL Server [26], to only provide limited transactional support, in the form of transactions within the same partition. Other systems, like Dynamo [42], Cassandra [61], HBase [1], and BigTable [33], have opted to give up transactional support completely. Wakening the abstractions available to developers, however, comes with its own costs. In exchange for the potential for higher performance and greater scalability, developers are left to wrestle with the challenge of guaranteeing the correctness of their application in the presence of concurrency and failures. This is such a complex and error-prone task that recently several voices have advocated moving back to the simplicity of ACID transactions, even if that means having to settle for less performance [35, 75].

Moving back and forth between these two approaches, developers actu-

ally face a fundamental tension between performance and ease of programming. This dissertation aims to resolve this tension by building database systems that provide both the ease of programming of the ACID approach and the performance that the BASE approach can provide. Our approaches depend on one key observation:

- *Not all transactions are created equal.* Different transactions affect the overall performance in different ways. For example, conflicts among transactions happen with different frequency, and when they do, they limit the concurrency of transactions to different extents.

This dissertation explores two different ways to leverage this key observation to combine performance and ease of programming, and presents two systems inspired by them.

Salt: Salt explores the notion that, since transactions have vastly different performance properties, they should export different abstractions. More precisely, Salt observes that, just as the Pareto Principle [65] would predict, only a small set of transactions actually challenge the performance limitation of the ACID paradigm. Therefore, Salt aims to only re-write these performance-critical transactions to boost their performance, while keeping other transactions unmodified.

Naively, one could simply increase concurrency by breaking down the performance-critical transactions into separate transactions. However, doing so may compromise isolation for the remaining ACID transactions. While

breaking down transactions does bring more concurrency to performance-critical transactions, it also exposes more states to *all* transactions, possibly violating their consistency invariants.

Salt solves this problem by introducing *BASE transactions*. This new abstraction allows developers to express a performance-critical transaction as a sequence of ACID subtransactions. The key to ensuring that BASE and ACID transactions safely coexist is *Salt Isolation*, a new isolation property that regulates the interactions between ACID and BASE transactions. For performance, Salt Isolation allows concurrently-executing BASE transactions to observe one another's internal states at the boundaries between consecutive substransactions; for correctness, it completely prevents ACID transactions from doing the same. As a result, BASE transactions behave differently when interacting with different transactions: to ACID transactions, BASE transactions appear like normal monolithic ACID transactions, while to other BASE transactions they expose specific internal states, increasing concurrency. This dualism is the key to achieving most of the performance that the BASE approach can provide at a fraction of the engineering effort. Our experiments shows that, by "BASE-ifying" just one out of 11 transactions in the open source ticketing application Fusion Ticket, Salt's performance is 6.5x higher than that of an ACID implementation.

Though Salt typically requires to "BASE-ify" only a handful of performance-critical transactions, this process is not trivial. Developers need to identify a way to re-write these transactions as BASE transactions and very carefully

reason about the correctness of their customized implementation. Our second system, Callas, aims to remove this extra engineering effort.

Callas: Callas aims to move beyond the ACID/BASE dilemma. Rather than trying to draw performance from weakening the abstraction offered to the developers, Callas decouples the concerns of abstraction and implementation: it unequivocally adopts the ACID paradigm, but uses a novel technique, *modular concurrency control* (MCC), to customize the mechanism though which these guarantees are provided.

MCC makes it possible to think modularly about the enforcement of any given isolation property I. It enables Callas to partition transactions in different groups, and it ensures that as long as I holds within each group, it will also hold among transactions in different groups. Separating concerns frees Callas to use within each group concurrency control mechanisms optimized for that group's transactions. Thus, Callas can find opportunities for increased concurrency where a generic mechanism might have to settle for a conservative execution. For example, Callas' in-group mechanism uses *Runtime Pipelining*, a novel technique that, by refining the static analysis approach used by transaction chopping [74], creates new chances for concurrency. In our experiments, for TPC-C, a standard database benchmark, Callas achieves an 8.2x speedup over MySQL Cluster without requiring any programming effort.

In summary, we make the following contribution:

- We introduce BASE transactions, a new abstraction that lets applica-

tions reap most of the performance of the BASE approach while at the same time limiting the complexity typically associated with it.

- We present a novel isolation property, Salt Isolation, that controls how ACID and BASE transactions interact. Salt Isolation allows BASE transactions to achieve high concurrency by observing each other's internal states, without affecting the isolation guarantees of ACID transactions.

- We build and evaluate Salt, a prototype developed on top of MySQL Cluster. Our evaluation suggests that BASE-ifying a handful of transactions can lead to close to an order of magnitude higher throughput than a fully ACID implementation.

- We propose MCC, a new, modular approach to concurrency control. By decoupling abstraction from mechanism, MCC retains the simplicity of a uniform ACID API; by separating concerns, it lets each module customize its internal concurrency control mechanism to achieve greater concurrency without sacrificing safety.

- We introduce *Runtime Pipelining*, a technique that leverages execution-time information to aggressively weaken within a group the conservative requirements of the current theory of safe transaction chopping [74] and gain, as a result, unprecedented opportunities for concurrency. The key to the effectiveness of Runtime Pipelining is the flexibility offered by

MCC, which makes this technique applicable within small groups of well-suited transactions.

- We build and evaluate Callas, a prototype, once again built on top of MySQL Cluster, that implements MCC. Our evaluation of Callas suggests that MCC can deliver performance gains comparable of Salt's to *unmodified* ACID applications.

The rest of this dissertation is organized as follows. Chapter 2 provides some necessary background. It introduces in more detail the ACID and BASE paradigms and the difficult choice between them that developers face. Chapter 3 discusses how Salt combines both the ACID and BASE paradigms in a single system. Chapter 4 describes how Callas leverages Modular Concurrency Control to boost the performance of ACID transactions. Chapter 5 discusses related work and Chapter 6 concludes.

Chapter 2

A Stark Choice: ACID vs BASE

The debate between ACID and BASE is well known [48, 52, 68]. While ACID transactions provide strong guarantees to ease the task of building applications, the BASE approach brings better performance and availability. This chapter explores and compares these two different approaches. Though each has its own advantages and disadvantages, neither currently offers both performance and ease of programming, leaving developers with a tradeoff that is hard to negotiate.

2.1 ACID transactions

Database systems organize data to help developers retrieve and analyze it easily. For this purpose, database systems need to ensure that the organized data is always in a state that satisfies the consistency constraints of applications. Though the basic read and write operations performed by the database are atomic, the database is not guaranteed to be in a consistent state after each operation. For example, Figure 2.1 shows a simple bank application that only contains one kind of requests whose purpose is to transfer money between two accounts. Consistency requires that, at the end of the transfer, the balance of

```
1  // transfer
2  Select bal into @bal from accnts where id = sndr
3  if (@bal >= amt)
4    Update accnts set bal -= amt where id = sndr
5    Update accnts set bal += amt where id = rcvr
```

Figure 2.1: A simple banking application ACID implementation

the sum of the two accounts be unchanged. In this example, during the transfer, there is a moment when the money has already been deducted from the first account and not yet added to the second account. If the system fails right at this moment, the system will be in a inconsistent state. In addition, even without failures, other users may observe the inconsistent states, and make decisions based upon them. In this example, two concurrent transfer requests may read the balance of the sender account before either of them starts to update the accounts' balance. If the sender account has enough money for either request, but not enough money for both of them, both transfer requests will pass the balance check, resulting a negative balance in the sender account.

Transactions were first proposed to prevent these uncontrolled and undesired interactions under multiuser environment, so that the database would be guaranteed to move from a consistent state to another consistent state [44]. A transaction consists of a sequence of database actions. By grouping these actions in a transaction, developers can treat all of them as a single logical unit, freeing them from worrying about concurrent executions and failures.

In order to achieve this nice guarantee, transactions must provide the following four properties known as ACID [54, 55]:

- **Atomicity** Each transaction is either applied in its entirety, or, if one action in the transaction fails, the whole transaction needs to be rolled back, so that the database is left unmodified. This property needs to be guaranteed even with failures or errors.

- **Consistency** Each committed transaction moves the database from a consistent state to another consistent state.

- **Isolation** Strictly speaking, isolation should ensure that the effect of executing concurrent transactions is the same as if they executed serially. The actions taken by one transaction should be hidden from others until the transaction commits. This guarantee comes at a non-trivial performance cost, however, so, as we will see in more detail in Section 2.1.1, over time researchers have considered weaker notions of isolation, in which some effects of an uncommitted transactions may be visible from other transactions.

- **Durability** Once a transaction commits, the system must retain the result of the transaction even if there are failures or errors.

With these four strong properties packed in a single abstraction, transactions greatly simplify the work of building an application. Figure 2.2, for example, illustrates how ACID transactions can be used to write a simple bank application. The application consists of only two transactions, `transfer` and `total-balance`, accessing the *accnts* relation. The ACID guarantees ensure

```
1  // ACID transfer transaction
2  begin txn
3    Select bal into @bal from accnts where id = sndr
4    if (@bal >= amt)
5      Update accnts set bal -= amt where id = sndr
6      Update accnts set bal += amt where id = rcvr
7  end txn

9  // ACID total-balance transaction
10 begin txn
11   Select sum(bal) from accnts
12 end txn
```

Figure 2.2: The ACID implementation of a simple banking application

that the `transfer` transaction either commits or is rolled-back automatically, despite failures or invalid inputs (such as an invalid *rcvr* id), and it is easy to add constraints (such as $bal \geq amt$) to ensure consistency invariants. In addition, developers need not worry about the interference from other transactions. For example, the `total-balance` transaction never reads the intermediate state of the `transfer` transaction, keeping the total balance as an invariant.

2.1.1 Isolation levels

To allow developers to trade performance for consistency guarantees, transactions are allowed to run under different isolation levels. Stronger isolation levels remove risks of exposing intermediate states of uncommitted transactions to other concurrent transactions. Weaker isolation levels, in turn, bring more concurrency at the cost of allowing more unexpected phenomena that

the developers need to take care of. In this section, we introduce two different definitions of isolation levels. The first definition, introduced by Berenson et al. [24], expresses isolation guarantees in terms of the unexpected phenomena they allow. The second definition is introduced by Atul Adya et al. [19]. This definition is based on dependency graphs, and it is implementation-independent. It gives more flexibility to implementations, which brings more performance potential.

Berenson's Definition

We first introduce some notation used in this definition. We write "w1[w]" to mean that transaction T_1 modifies data item x. "c1" and "a1" mean, respectively, that transaction T_1 commits or decides to rollback. If transaction T_1 reads or writes a set of records satisfying predicate P, we write it, respectively, as "r1[P]" and "w1[P]".

To define the isolation levels, Berenson et al.[24] first defines four phenomena as following. Each of these phenomena may lead to some unexpected behaviors violating database consistency.

- **P0(Dirty Write):** w1[x]...w2[x]...((c1 or a1) and (c2 or a2) in any order)

P0(Dirty Write) refers to executions where transactions T_2 modifies item x after transaction T_1 modifies it but before transactions T_1 commits or rollbacks. Assume there is a constraint between x and y (e.g., x = y), and T_1 and T_2 each maintain the constraint if run alone. If P0 is allowed, this

constraint can be easily broken if T_1 and T_2 execute concurrently, since the database may move to a state the final value of x is determined by T_1 while the final value of y is determined by T_2.

- **P1(Dirty Read):** w1[x]...r2[x]...((c1 or a1) and (c2 or a2) in any order)

P1(Dirty Read) refers to executions where transaction T_2 reads the value of x written by transaction T_1 before T_1 commits or aborts. If P1 is allowed, what may happen is that transaction T_2 reads the value x written by T_1, writes this value to another data item y, and commits. After that, T_1 rollbacks. What happens is that the result of T_2 depends on an invalid value of x.

- **P2(Fuzzy Read):** r1[x]...w2[x]...((c1 or a1) and (c2 or a2) in any order)

P2(Fuzzy Read) refers to executions where transactions T_2 modifies item x after T_1 read that item, but before $T_!$ commits or rollbacks. if P2 is allowed, transaction T_1 may get different results when reading the same item twice since T_2 may modify the value of this item.

- **P3(Phantom):** r1[P]...w2[y in P]...((c1 or a1) and (c2 or a2) in any order)

P3(Phantom) is very similar to P2 except the read request of transaction T_1 a predicate read. If P3 is allowed, transaction T_1 may get different

Isolation level	P0	P1	P2	P3
read-uncommitted	✗	✓	✓	✓
read-committed	✗	✗	✓	✓
repeatable-read	✗	✗	✗	✓
serializable	✗	✗	✗	✗

Table 2.1: Isolation levels

results when reading on the same predicate twice, since T_2 may modify one or more items that satisfy this predicate.

Based on these four phenomena, Berenson et. al. define four isolation levels shown in Table 2.1. Stronger isolation levels prevent more unexpected phenomena: serializability prevents all phenomena. Choosing the isolation level supported by a given database system then presents a familiar trade-off: stronger isolation levels make it easier for developers to write applications; weaker ones, though they allow some unexpected phenomena, have the potential of bringing more concurrency among transactions.

Adya's Definition

Berenson's definition is based on locking and, as such, it fails to meet the goal of implementation-independence. As a result, this definition rules out some implementations, such as optimistic and multiversion concurrency control schemes [28, 60]. To address these concerns, Adya et al. introduced a new way of formalizing isolation levels. Their formulation refines the classic approach of leveraging a serializability graph to express whether a history is serializable: they express necessary conditions that apply to weaker notions of isolation as requirements on the structure of a new graph they define, called

the *Direct Serialization Graph* (DSG). Each node in the DSG corresponds to a committed transaction, and each directed edge from transaction T_i to T_j indicates one of the following types of conflict between them:

- *Read dependency.* T_i installs a version x_i of an object x and T_j reads x_i, or T_j performs a predicate-based read, x_i changes the matches of T_j's read, and x_i is the same or an earlier version of x in T_j's read.
- *Write dependency.* T_i installs a version x_i of x, and T_j installs x's next version.
- *Item-anti-dependency.* T_i reads a version x_k of x, and T_j installs x's next version.
- *Predicate-anti-dependency.* T_j installs a later version of some object that changes the matches of a predicate based read performed by T_j

The DSG is central to this formulation because the occurrence of some of the phenomena proscribed by a given isolation level is equivalent to the DSG exhibiting a refinement of the following condition:

- *Circularity.* The execution history contains a directed cycle.

The refinement consists of specifying which types of edges can be used to construct the cycle: the more stringent the isolation level, the larger the set of cycles to be prevented (and of phenomena to be proscribed). For example, isolation levels that forbid reading data that has not been committed (*dirty*

Isolation level	Write dependency	Read dependency	Item anti-dependency	Predicate anti-dependency
read-uncommitted	✓	✗	✗	✗
read-committed	✓	✓	✗	✗
repeatable-read	✓	✓	✓	✗
serializable	✓	✓	✓	✓

Table 2.2: Isolation levels

reads) require the flow of information between any two transactions to be unidirectional—which can be achieved by proscribing DSG cycles consisting only of write or read dependency edges [19]. Achieving serializability, however, requires ruling out also cycles that include anti-dependency edges. Table 2.2 illustrates the types of edges considered for building DSG cycles under different isolation levels.

Though most of the phenomena can be mapped to the dependency cycles, not all of them are covered. In particular, every isolation level that at least as strong as Read Committed must also avoid the following two phenomena:

- *Aborted Reads.* A committed transaction T_2 reads some object (possibly via a predicate) modified by an aborted transaction T_1.
- *Intermediate Reads.* A committed transaction T_2 reads a version of an object x (possibly via a predicate) written by another transaction T_1 that was not T_1's final modification of x.

Adya's definition is more general than Berenson's. For example, consider the following execution:

- **H:** W1[x] R2[x] W1[y] R2[y] c1 c2

This execution is equivalent to a serial one in which T_1 executes before T_2. However, according to Berenson's definition, this execution does not satisfy the requirements of the *Serializable* isolation level, since it allows for the P1 phenomenon. On the contrary, Adya's definition treats the execution as a serializable since it leads to no dependency cycles, aborted reads, or intermediate reads.

2.2 BASE

ACID transactions provide strong consistency guarantees. These guarantees, however, do not come for free. Ensuring the consistency constrains at each step reduces the concurrency among transactions and may limit performance. The BASE approach, introduced by Brewer [32], focuses more on availability and performance. This approach attempts to loose the consistency constraints ("C" and "I" in ACID) in exchange for higher availability and performance. It allows temporary inconsistent states and gives the responsibility of tolerating these inconsistent states to applications. The name "BASE" is an acronym obtained from the following three properties:

- **Basically Available** Availability has higher priority than consistency. Therefore, every request should elicit a timely response even if that response may return an inconsistent result or a failure notification.
- **Soft state** The state of systems could change without any input.

- **Eventual consistency** If the system does not receive any more inputs, the state of the system will eventually satisfy the consistency requirements of applications.

Unlike ACID, however, BASE offers more of a set of programming guidelines (such as the use of *partition local transactions* [56, 68]) than a set of rigorously specified properties, and its instantiations take a variety of application-specific forms. Common among them, however, is a programming style that avoids distributed transactions to eliminate the performance and availability costs of the associated distributed commit protocol. There are many NoSQL systems that provide BASE style APIs [1, 17, 33, 34, 42, 61]. These systems partially or completely give up the ACID transaction paradigm.

As a concrete application of the BASE approach, consider Figure 2.3. It shows a BASE implementation of the same simple banking application shown in Figure 2.2. All the transactions used in this approach are local transactions. Now, it is up to the application to ensure consistency and atomicity despite failures that occur between the first and second transaction. And while the level of isolation offered by ACID transactions ensures that `total-balance` will compute accurately the sum of balances in *accnts*, in BASE the code needs to prevent explicitly (lines 30 and 31 of Figure 2.3) `total-balance` from observing the intermediate state after the *sndr* account has been charged but before the *rcvr*'s has been credited.

```
1  // transfer using the BASE approach
2  begin local−transaction
3    Select bal into @bal from accnts where id = sndr
4    if (@bal >= amt)
5      Update accnts set bal −= amt where id = sndr
6      // To enforce atomicity, we use queues to communicate
7      // between partitions
8      Queue message(sndr, rcvr, amt) for partition(accnts, rcvr)
9  end local−transaction

11 // Background thread to transfer messages to other partitions
12 begin transaction // distributed transaction to transfer queued msgs
13   <transfer messages to rcvr>
14 end transaction

16 // A background thread at each partition processes
17 // the received messages
18 begin local−transaction
19   Dequeue message(sndr, rcvr, amt)
20   Select id into @id from accnts where id = rcvr
21   if (@id ≠ ∅) // if rcvr's account exists in database
22     Update accnts set bal += amt where id = rcvr
23   else // rollback by sending the amt back to the original sender
24     Queue message(rcvr, sndr, amt) for partition(accnts, sndr)
25 end local−transaction

27 // total−balance using the BASE approach
28 // The following two lines are needed to ensure correctness of
29 // the total−balance ACID transaction
30 <notify all partitions to stop accepting new transfers>
31 <wait for existing transfers to complete>
32 begin transaction
33   Select sum(bal) from accnts
34 end transaction
35 <notify all partitions to resume accepting new transfers>
```

Figure 2.3: A simple banking application BAEE implementation

As we can see in this example, writing applications using the BASE approach is complex. The code is much longer, and developers are forced to reason about many corner cases, making it very easy to introduce bugs in the implementation. What is worse the complexity actually increases exponentially with the size of applications, since the number of combinations of different concurrent requests increases exponentially.

2.3 Choosing between ACID and BASE

The potential performance gains of the BASE approach are compellingand indeed, many applications over the last decade have embraced the NoSQL movement, renouncing the consistency guarantees to achieve high availability and better performance. However, the complexity of this style of programming has sparked a recent backlash against the early enthusiasm for BASE [35, 75]—as Shute et al. put it "Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains" [75].

The ACID/BASE dichotomy may appear as yet another illustration of the "no free lunch" adage: if you want performance, you must give something up. Indeed—but BASE gives virtually *everything* up: the entire application needs to be rewritten, with no automatic support for either atomicity, consistency, or durability, and with isolation limited only to partition-local transactions. Can't we aim for a more reasonable bill?

Chapter 3

Salt: Combining ACID and BASE in Distributed Database

This chapter[1] presents the design, implementation, and evaluation of Salt, a distributed database that, for the first time, allows developers to reap the complementary benefits of both the ACID and BASE paradigms within a single application. In particular, Salt attempts to dispel the false dichotomy between performance and ease of programming that fuels the ACID vs. BASE argument.

Salt aims to reclaim most of those performance gains while keeping complexity in check. The approach that we propose to resolve the tension is rooted in the Pareto principle [65]. When an application outgrows the performance of an ACID implementation, it is often due to the needs of only a handful of transactions: most transactions never test the limits of what ACID can offer. Numerous applications [2, 5, 6, 12, 13] demonstrate this familiar lopsided pattern: few transactions are performance-critical, while many others are

[1] This chapter is based on "Salt: Combing ACID and BASE in Distributed Database" [85], authored by Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi and Prince Mahajan, published in the proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. Chao designed the protocol and implemented most of the Salt prototype.

either lightweight or infrequent; e.g. administrative transactions. Our experience confirms this pattern. For example, running the TPC-C benchmark [36] on a MySQL cluster, we found that, as the load increases, only two transactions take much longer to complete—a symptom of high contention; other transactions are unaffected. Similarly, we found that the ACID throughput of Fusion Ticket [8], a popular open source online ticketing application that uses MySQL as its backend database, is limited by the performance of just *one* transaction out of 11.

Motivated by this observation, Salt is tempting to create a database where the ACID and BASE paradigms can safely coexist within the same application, so that ACID applications that struggle to meet their growing performance demands to improve their availability and scalability by incrementally "BASE-ifying" only the few ACID transactions that are performance-critical, without compromising the ACID guarantees enjoyed by the remaining transactions.

Of course, naively BASE-ifying selected ACID transactions may void their atomicity guarantees, compromise isolation by exposing intermediate database states that were previously unobservable, and violate the consistency invariants expected by the transactions that have not been BASE-ified. To enable mutually beneficial coexistence between the ACID and BASE paradigms, Salt introduces a new abstraction: *BASE transactions*.

BASE transactions loosen the tight coupling between atomicity and isolation enforced by the ACID paradigm to offer a unique combination of

features: the performance and availability benefits of BASE-style partition-local transactions together with the ability to express and enforce atomicity at the granularity called for by the application semantics.

Key to this unprecedented combination is *Salt Isolation*, a new isolation property that regulates the interactions between ACID and BASE transactions. For performance, Salt Isolation allows concurrently executing BASE transactions to observe, at well-defined spots, one another's internal states, but, for correctness, it completely prevents ACID transactions from doing the same. It limits the effects of BASE-ifying one transaction only among BASE transactions.

We have built a Salt prototype by modifying an ACID system, the MySQL Cluster distributed database [11], to support BASE transactions and Salt Isolation. Our evaluation confirms that BASE transactions and Salt Isolation together allow Salt to break new ground in balancing performance and ease of programming. For example, our experiments show that, by BASE-ifying just one out of 11 transactions in the open source ticketing application Fusion Ticket, Salt's performance is $6.5x$ higher than that of an ACID implementation.

The rest of the chapter proceeds as follows. Section 3.1 proposes a new alternative, Salt, that sidesteps the trade-off between performance and ease of programming. Section 3.2 introduces the notion of BASE transactions and Section 3.3 presents the novel notion of Salt Isolation, which allows ACID and BASE transactions to safely coexist within the same application. Sec-

tion 3.4 discusses the implementation of our Salt prototype, Section 3.5 shows an example of programming in Salt, and Section 3.6 presents the results of our experimental evaluation. Section 3.7 concludes.

3.1 A grain of Salt

Salt, as we noted in the Introduction, is based on the familiar Pareto principle: even in applications that outgrow the performance achievable with ACID solutions, not all transactions are equally demanding. While a few transactions require high performance, many others never test the limits of what ACID can offer. This raises an intriguing possibility: could one identify those few performance-critical transactions (either at application-design time or through profiling, if an ACID implementation of the application already exists) and somehow only need to go through the effort of BASE-ifying *those* transactions in order to get most of the performance benefits that come from adopting the BASE paradigm?

Realizing this vision is not straightforward. For example, BASE-ifying only the `transfer` transaction in the simple banking application of Figure 2.2 would allow `total-balance` to observe a state in which *sndr* has been charged but *rcvr*'s has not yet been credited, causing it to compute incorrectly the bank's holdings. The central issue is that BASE-ifying transactions, even if only a few, can make suddenly accessible to all transactions what previously were invisible intermediate database states. Protecting developers from having to worry about such intermediate states despite failures and concurrency, how-

ever, is at the core of the ease of programming offered by the transactional programming paradigm. Indeed, quite naturally, isolation (which regulates which states can be accessed when transactions execute concurrently) and atomicity (which frees from worrying about intermediate states during failures) are typically offered at the same granularity—that of the ACID transaction.

We submit that while this tight coupling of atomicity and isolation makes ACID transactions both powerful and attractively easy to program with, it also limits their ability to continue to deliver ease of programming when performance demands increase. For example, splitting an ACID transaction into smaller transactions can improve performance, but at the cost of shrinking the original transaction's guarantees in terms of both atomicity and isolation: the all-or-nothing guarantee of the original transaction is unenforceable on the set of smaller transactions, and what were previously intermediate states can suddenly be accessed indiscriminately by all other transactions, making it much harder to reason about the correctness of one's application.

The approach that we propose to move beyond today's stark choices is based on two propositions: first, that the coupling between atomicity and isolation should be loosened, so that providing isolation at a fine granularity does not necessarily result in shattering atomicity; and second, that the choice between either enduring poor performance or allowing indiscriminate access to intermediate states by all transactions is a false one: instead, complexity can be tamed by giving developers control over who is allowed to access these intermediate states, and when.

To enact these propositions, the Salt distributed database introduces a new abstraction: *BASE transactions*. The design of BASE transactions borrows from nested transactions [83], an abstraction originally introduced to offer, for long-running transactions, atomicity at a finer granularity than isolation. In particular, while most nested transaction implementations define isolation at the granularity of the parent ACID transaction,[2] they tune the mechanism for enforcing atomicity so that errors that occur within a nested subtransaction do not require undoing the entire parent transaction, but only the affected subtransaction.

Our purpose in introducing BASE transactions is similar in spirit to that of traditional nested transactions: both abstractions aim at gently loosening the coupling between atomicity and isolation. The issue that BASE transactions address, however, is the flip side of the one tackled by nested transactions: this time, the challenge is to provide *isolation* at a finer granularity, without either drastically escalating the complexity of reasoning about the application, or shattering atomicity.

3.2 BASE transactions

Syntactically, a BASE transaction is delimited by the familiar `begin BASE transaction` and `end BASE transaction` statements. Inside, a BASE

[2] *Nested top-level transactions* are a type of nested transactions that instead commit or abort independently of their parent transaction. They are seldom used, however, precisely because they violate the isolation of the parent transaction, making it hard to reason about consistency invariants.

transaction contains a sequence of *alkaline* subtransactions—nested transactions that owe their name to the novel way in which they straddle the ACID/BASE divide.

When it comes to the granularity of atomicity, as we will see in more detail below, a BASE transaction provides the same flexibility of a traditional nested transaction: it can limit the effects of a failure within a single alkaline subtransaction, while at the same time it can ensure that the set of actions performed by all the alkaline subtransactions it includes is executed atomically. Where a BASE transaction fundamentally differs from a traditional nested transaction is in offering *Salt Isolation*, a new isolation property that, by supporting multiple granularities of isolation, makes it possible to control which internal states of a BASE transaction are externally accessible, and by whom. Despite this unprecedented flexibility, Salt guarantees that, when BASE and ACID transactions execute concurrently, ACID transactions retain, with respect to all other transactions (whether BASE, alkaline, or ACID), the same isolation guarantees they used to enjoy in a purely ACID environment. The topic of how Salt isolation supports ACID transactions across all levels of isolation defined in the ANSI/ISO SQL standard is actually interesting enough that we will devote the entire next section to it. To prevent generality from obfuscating intuition, however, the discussion in the rest of this section assumes ACID transactions that provide the popular *read-committed* isolation level.

Independent of the isolation provided by ACID transactions, a BASE

```
1  // BASE transaction: transfer
2  begin BASE transaction
3    try
4      begin alkaline−subtransaction
5        Select bal into @bal from accnts where id = sndr
6        if (@bal >= amt)
7          Update accnts set bal −= amt where id = sndr
8      end alkaline−subtransaction
9    catch (Exception e) return // do nothing
10   if (@bal < amt) return // constraint violation
11   try
12     begin alkaline−subtransaction
13       Update accnts set bal += amt where id = rcvr
14     end alkaline−subtransaction
15   catch (Exception e) //rollback if rcvr not found or timeout occurs
16     begin alkaline−subtransaction
17       Update accnts set bal += amt where id = sndr
18     end alkaline−subtransaction
19 end BASE transaction


22 // ACID transaction: total−balance (unmodified)
23 begin transaction
24   Select sum(bal) from accnts
25 commit
```

Figure 3.1: A Salt implementation of the simple banking application

transaction's basic unit of isolation are the alkaline subtransactions it contains. Alkaline subtransactions retain the properties of ACID transactions: in particular, when it comes to isolation, no transaction (whether ACID, BASE or alkaline) can observe intermediate states produced by an uncommitted alkaline subtransaction. When it comes to observing the state produced by a *committed* alkaline subtransaction, however, the guarantees differ depending on the potential observer.

- The committed state of an alkaline subtransaction is observable by other BASE or alkaline subtransactions. By leveraging this finer granularity of isolation, BASE transactions can achieve levels of performance and availability that elude ACID transactions. At the same time, because alkaline subtransactions are isolated from each other, this design limits the new interleavings that programmers need to worry about when reasoning about the correctness of their programs: the only internal states of BASE transactions that become observable are those at the boundaries between its nested alkaline subtransactions.

- The committed state of an alkaline subtransaction is *not* observable by other ACID transactions until the parent BASE transaction commits. The internal state of a BASE transaction is then completely opaque to ACID transactions: to them, a BASE transaction looks just like an ordinary ACID transaction, leaving their correctness unaffected.

To maximize performance, we expect that alkaline subtransactions will typically be partition-local transactions, but application developers are free, if necessary to enforce critical consistency conditions, to create alkaline subtransactions that touch multiple partitions and require a distributed commit.

Figure 3.1 shows how the simple banking application of Figure 2.2 might look when programmed in Salt. The first thing to note is what has *not* changed from the simple ACID implementation of Figure 2.2: Salt does not require any modification to the ACID `total-balance` transaction; only the performance-

critical `transfer` operation is expressed as a new BASE transaction. While the complexity reduction may appear small in this simple example, our current experience with more realistic applications (such as Fusion Ticket, discussed in Section 3.6) suggests that Salt can achieve significant performance gains while leaving untouched most ACID transactions. Figure 3.1 also shows another feature of alkaline subtransactions: each is associated with an exception, which is caught by an application-specific handler in case an error is detected. As we will discuss in more detail shortly, Salt leverages the exceptions associated with alkaline subtransactions to guarantee the atomicity of the BASE transactions that enclose them.

There are two important events in the life of a BASE transaction: *accept* and *commit*. In the spirit of the BASE paradigm, BASE transactions, as in Lynx [87], are accepted as soon as their first alkaline subtransaction commits. The atomicity property of BASE transactions ensures that, once accepted, a BASE transaction will eventually commit, i.e., all of its operations will have successfully executed (or bypassed because of some exception) and their results will be persistently recorded.

To clarify further our vision for the abstraction that BASE transactions provide, it helps to compare their guarantees with those provided by ACID transactions

Atomicity Just like ACID transactions, BASE transactions guarantee that *either all the operations they contain will occur, or none will.* In particular, atomicity guarantees that all accepted BASE transactions will eventually

commit. Unlike ACID transactions, BASE transactions can be aborted only if they encounter an error (such as a constraint violation or a node crash) *before* the transaction is accepted. Errors that occur after the transaction has been accepted do not trigger an automatic rollback: instead, they are handled using exceptions. The details of our Salt's implementation of atomicity are discussed in Section 3.4.

Consistency Chasing higher performance by splitting ACID transactions can increase exponentially the number of interleavings that must be considered when trying to enforce integrity constraints. Salt drastically reduces this complexity in two ways. First, Salt does not require all ACID transactions to be dismembered: non-performance-critical ACID transactions can be left unchanged. Second, Salt does not allow ACID transactions to observe states inside BASE transactions, cutting down significantly the number of possible interleavings.

Isolation Here, BASE and ACID transactions depart, as BASE transactions provide the novel *Salt Isolation* property, which we discuss in full detail in the next section. Appealingly, Salt Isolation on the one hand allows BASE transactions to respect the isolation property offered by the ACID transactions they may execute concurrently with, while on the other yields the opportunity for significant performance improvements. In particular, under Salt Isolation a BASE transaction BT appears to an ACID transaction just like another ACID transaction, but other BASE transactions can observe the internal states that exist at the boundaries between adjacent alkaline subtransactions in BT.

Durability BASE transactions provide the same durability property of ACID transactions and of many existing NoSQL systems: *Accepted BASE transactions are guaranteed to be durable.* Hence, developers need not worry about losing the state of accepted BASE transactions.

3.3 Salt Isolation

Intuitively, our goal for Salt isolation is to allow BASE transactions to achieve high degrees of concurrency, while ensuring that ACID transactions enjoy well-defined isolation guarantees. Before taking on this challenge in earnest, however, we had to take two important preliminary steps.

The first, and the easiest, was to pick the concurrency control mechanism on which to implement Salt isolation. Our current design focuses on lock-based implementations rather than, say, optimistic concurrency control, because locks are typically used in applications that experience high contention and can therefore more readily benefit from Salt; also, for simplicity, we do not currently support multiversion concurrency control and hence snapshot isolation. However, there is nothing about Salt isolation that fundamentally prevents us from applying it to other mechanisms beyond locks.

The second step proved much harder. We had to crisply characterize what are exactly the isolation guarantees that we want our ACID transactions to provide. This may seem straightforward, given that the Berenson's definition mentioned in Section 2.1.1 already defines the relevant four isolation levels for lock-based concurrency: read-uncommitted, read-committed,

repeatable read, and serializable. Each level offers stronger isolation than the previous one, preventing an increasingly larger prefix of the following sequence of undesirable phenomena: dirty write, dirty read, non-repeatable read, and phantom [24].

Where the challenge lies, however, is in preventing this diversity from forcing us to define four distinct notions of Salt isolation, one for each of the four ACID isolation levels. Ideally, we would like to arrive at a single, concise characterization of isolation in ACID systems that somehow captures all four levels, which we can then use to specify the guarantees of Salt isolation.

The key observation that ultimately allowed us to do so is that all four isolation levels can be reduced to a simple requirement: if two *operations* in different transactions conflict, then the temporal dependency that exists between the earlier and the later of these operations must extend to the entire *transaction* to which the earlier operation belongs. Formally:

Isolation. Let $\mathcal{Q}$ be the set of operation types {read, range-read, write} and let $\mathcal{L}$ and $\mathcal{S}$ be subsets of $\mathcal{Q}$. Further, let o_1 in $transaction_1$ and o_2 in $transaction_2$, be two operations, respectively of type $T_1 \in \mathcal{L}$ and $T_2 \in \mathcal{S}$, that access the same object in a conflicting (i.e. non read-read) manner. If o_1 completes before o_2 starts, then $transaction_1$ must decide before o_2 starts.

With this single and concise formulation, each of the ACID isolation levels can be expressed by simply instantiating appropriately $\mathcal{L}$ and $\mathcal{S}$. For example, $\mathcal{L} = \{write\}$ and $\mathcal{S} = \{read, write\}$ yields read-committed isolation.

Isolation level	$\mathcal{L}$	$\mathcal{S}$
read-uncommitted	W	W
read-committed	W	R,W
repeatable-read	R,W	R,W
serializable	R,RR,W	R,RR,W

Table 3.1: Conflicting type sets $\mathcal{L}$ and $\mathcal{S}$ for each of the four isolation levels. R = Read, RR = Range Read, W = Write.

Table 3.1 shows the conflicting sets of operation types for all four isolation levels. For a given $\mathcal{L}$ and $\mathcal{S}$, we will henceforth say that two transactions are *isolated* from each other when Isolation holds between them.

Having expressed the isolation guarantees of ACID transactions, we are ready to tackle the core technical challenge ahead of us: defining an isolation property for BASE transactions that allows them to harmoniously coexist with ACID transactions. At the outset, their mutual affinity may appear dubious: to deliver higher performance, BASE transactions need to expose intermediate uncommitted states to other transactions, potentially harming Isolation. Indeed, the key to Salt isolation lies in controlling which, among BASE, ACID, and alkaline subtransactions, should be exposed to what.

Our formulation of Salt isolation leverages the conciseness of the Isolation property to express its guarantees in a way that applies to all four levels of ACID isolation.

Salt Isolation. The Isolation property holds as long as (a) at least one of $transaction_1$ and $transaction_2$ is an ACID transaction or (b) both $transaction_1$ and $transaction_2$ are alkaline subtransactions.

	ACID-R	ACID-W	alka-R	alka-W	saline-R	saline-W
ACID-R	✓	✗	✓	✗	✓	✗
ACID-W	✗	✗	✗	✗	✗	✗
alka-R	✓	✗	✓	✗	✓	✓
alka-W	✗	✗	✗	✗	✓	✓
saline-R	✓	✗	✓	✓	✓	✓
saline-W	✗	✗	✓	✓	✓	✓

Table 3.2: Conflict table for ACID, alkaline, and saline locks.

Informally, Salt isolation enforces the following constraint gradation:

- ACID transactions are isolated from all other transactions.
- Alkaline subtransactions are isolated from other ACID and alkaline subtransactions.
- BASE transactions expose their intermediate states (i.e. states produced at the boundaries of their alkaline subtransactions) to every other BASE transaction.

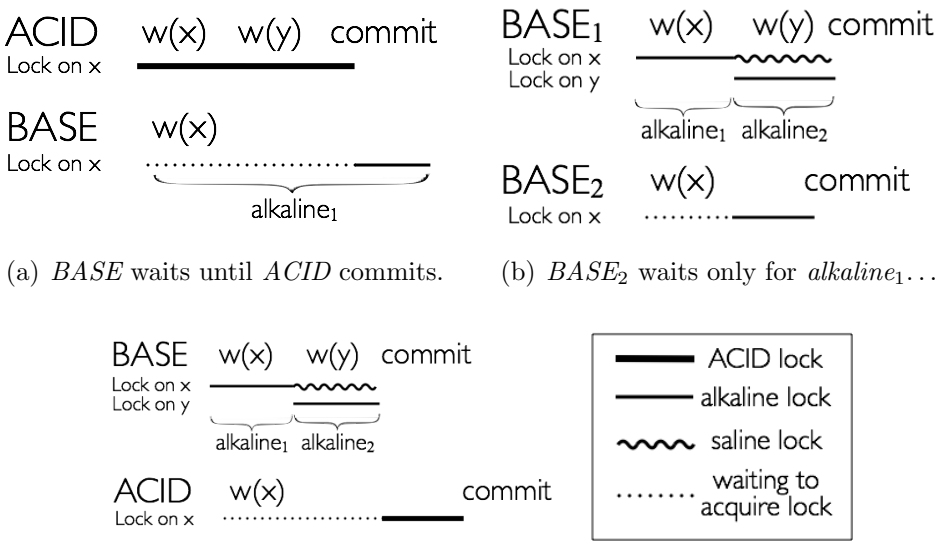
Hence, despite its succinctness, Salt isolation must handle quite a diverse set of requirements. To accomplish this, it uses a single mechanism—locks—but equips each type of transaction with its own type of lock: *ACID* and *alkaline* locks, which share the name of their respective transactions, and *saline* locks, which are used by BASE transactions.

ACID locks work as in traditional ACID systems. There are ACID locks for both read and write operations; reads conflict with writes, while writes conflict with both reads and writes (see the dark-shaded area of Table 3.2).

The duration for which an ACID lock is held depends on the operation type and the chosen isolation level. Operations in $\mathcal{L}$ require *long-term* locks, which are acquired at the start of the operation and are maintained until the end of the transaction. Operations in $\mathcal{S} \setminus \mathcal{L}$ require *short-term* locks, which are only held for the duration of the operation.

Alkaline locks keep alkaline subtransactions isolated from other ACID and alkaline subtransactions. As a result, as Table 3.2 (light-and-dark shaded subtable) shows, only read-read accesses are considered non-conflicting for any combination of ACID and alkaline locks. Similar to ACID locks, alkaline locks can be either long-term or short-term, depending on the operation type; long-term alkaline locks, however, are only held until the end of the current alkaline subtransaction, and not for the entire duration of the parent BASE transaction: their purpose is solely to isolate the alkaline subtransaction containing the operation that acquired the lock.

Saline locks owe their name to their delicate charge: isolating ACID transactions from BASE transactions, while at the same time allowing for increased concurrency by exposing intermediate states of BASE transactions to other BASE transactions. To that end, (see Table 3.2) saline locks conflict with ACID locks for non read-read accesses, but *never* conflict with either alkaline or saline locks. Once again, there are long-term and short-term saline locks: short-term saline locks are released after the operation completes, while long-term locks are held until the end of the current BASE transaction. In practice, since alkaline locks supersede saline locks, we acquire only an alkaline lock at



(a) *BASE* waits until *ACID* commits.

(b) $BASE_2$ waits only for $alkaline_1$...

(c) ...but *ACID* must wait all of *BASE* out.

Figure 3.2: Examples of concurrent executions of ACID and BASE transactions in Salt.

the start of the operation and, if the lock is long-term, "downgrade" it at the end of the alkaline subtransaction to a saline lock, to be held until after the end of the BASE transaction.

Figure 3.2 shows three simple examples that illustrate how ACID and BASE transactions interact. In Figure 3.2(a), an ACID transaction holds an ACID lock on x, which causes the BASE transaction to wait until the ACID transaction has committed, before it can acquire the lock on x. In Figure 3.2(b), instead, transaction $BASE_2$ need only wait until the end of $alkaline_1$, before acquiring the lock on x. Finally, Figure 3.2(c) illustrates the use of saline locks. When $alkaline_1$ commits, it downgrades its lock on x to a

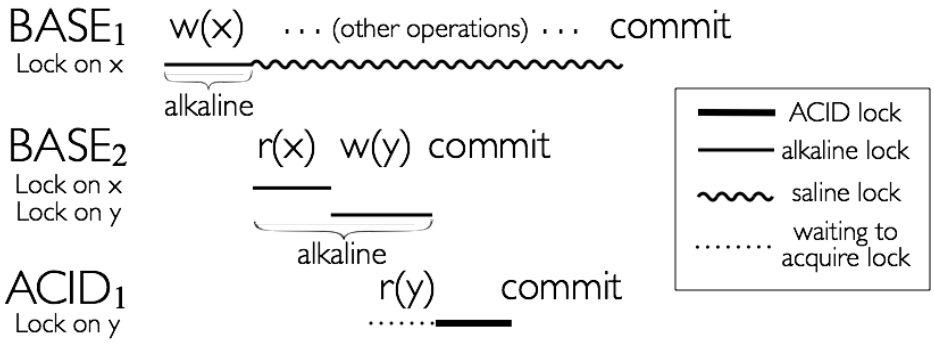


Figure 3.3: $ACID_1$ indirectly reads the uncommitted value of x.

saline lock that is kept until the end of the parent BASE transaction, ensuring that the ACID and BASE transactions remain isolated.

Indirect dirty reads In an ACID system the Isolation property holds among any two transactions, making it quite natural to consider only direct interactions between pairs of transactions when defining the undesirable phenomena prevented by the four isolation levels. In a system that uses Salt isolation, however, the Isolation property covers only *some* pairs of transactions: pairs of BASE transactions are exempt. Losing Isolation's universal coverage has the insidious effect of introducing indirect instances of those undesirable phenomena.

The example in Figure 3.3 illustrates what can go wrong if Salt Isolation is enforced naively. For concreteness, assume that ACID transactions require a read-committed isolation level. Since Isolation is not enforced between $BASE_1$ and $BASE_2$, $w(y)$ may reflect the value of x that was written by $BASE_1$.

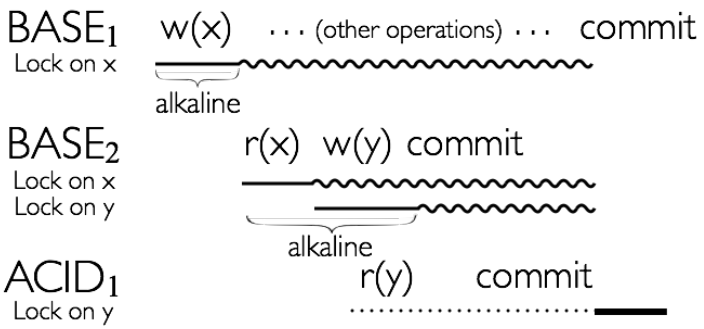


Figure 3.4: How Salt prevents indirect dirty reads.

Although Isolation is enforced between $ACID_1$ and $BASE_2$, $ACID_1$ ends up reading x's uncommitted value, which violates that transaction's isolation guarantees.

The culprit for such violations is easy to find: dirty reads can indirectly relate two transactions ($BASE_1$ and $ACID_1$ in Figure 3.3) without generating a direct conflict between them. Fortunately, none of the other three phenomena that ACID isolation levels try to avoid can do the same: for such phenomena to create an indirect relation between two transactions, the transactions at the two ends of the chain must be in direct conflict.

Our task is then simple: we must prevent indirect dirty reads. Salt avoids them by restricting the order in which saline locks are released, in the following two ways:

Read-after-write across transactions A BASE transaction B_r that reads a value x, which has been written by another BASE transaction B_w,

cannot release its saline lock on x until B_w has released its own saline lock on x.

Write-after-read within a transaction An operation o_w that writes a value x cannot release its saline lock on x until *all* previous read operations within the same BASE transaction have released their saline locks on their respective objects.

The combination of these two restrictions ensures that, as long as a write remains uncommitted (i.e. its saline lock has not been released) subsequent read operations that observe that written value and subsequent write operations that are affected by that written value will not release their own saline locks. This, in turn, guarantees that an ACID transaction cannot observe an uncommitted write, since saline locks are designed to be mutually exclusive with ACID locks. Figure 3.4 illustrates how enforcing these two rules prevents the indirect dirty read of Figure 3.3. Observe that transaction BASE_2 cannot release its saline lock on x until BASE_1 commits (read-after-write across transactions) and BASE_2 cannot release its saline lock on y before releasing its saline lock on x (write-after-read within a transaction).

Theorem 1. [Correctness] Given isolation level $\mathcal{A}$, all ACID transactions are protected (both directly and, where applicable, indirectly) from all the undesirable phenomena prevented by $\mathcal{A}$.

Proof of Theorem 1

Lemma 3.3.1. *Under salt read uncommitted or higher isolation levels, all ACID transactions are protected from dirty write.*

Formally, suppose $w_1(x_1) \in transaction_1$ *and* $w_2(x_1) \in transaction_2$*, if either* $transaction_1$ *or* $transaction_2$ *is an ACID transaction and* $w_1(x_1) \rightarrow w_2(x_1)$*, then* $transaction_1 \rightarrow w_2(x_1)$ *(*$a \rightarrow b$ *denotes that* a *completes before* b *starts.* a *and* b *can be an operation in a transaction or a whole transaction.).*

Proof If both $transaction_1$ and $transaction_2$ are ACID, the property is guaranteed by the original ACID implementation. We will prove the following two cases:

- Case 1: $transaction_1$ is a BASE transaction and $transaction_2$ is an ACID transaction. In this case, $w_1(x_1)$ grabs the saline write lock on object x_1 at it starts and holds it until the end of $transaction_1$. $w_2(x_1)$ of $transaction_2$ needs to acquire the ACID write lock on object x_1 before it can start. And since the ACID write lock conflicts with the saline write lock held by $w_1(x_1)$ and $w_1(x_1) \rightarrow w_2(x_1)$, $w_2(x_1)$ can not start before $transaction_1$ completes, which means $transaction_1 \rightarrow w_2(x_1)$.

- Case 2: $transaction_1$ is an ACID transaction and $transaction_2$ is a BASE transaction. The proof is similar as that in Case 1.

Because there is no indirect dirty write, salt read uncommitted avoid dirty write for ACID transactions.

Lemma 3.3.2. *Under salt read committed or higher isolation levels, all ACID transactions are protected from dirty read.*

Formally, suppose $w_1(x_1) \in transaction_1$, $r_2(x_1)$ *and* $w_2(x_2) \in transaction_2$, $r_3(x_2)$ *and* $w_3(x_3) \in transaction_3$, ..., *and* $r_N(x_{N-1}) \in transaction_N$, *if either* $transaction_1$ *or* $transaction_N$ *is an ACID transaction, and* $w_1(x_1) \rightarrow r_2(x_1) \rightarrow w_2(x_2) \rightarrow r_3(x_2) \rightarrow ... \rightarrow w_{N-1}(x_{N-1}) \rightarrow r_N(x_{N-1})$, *then* $transaction_1 \rightarrow r_N(x_{N-1})$

Proof We only need to prove that this property holds in two cases—$transaction_N$ is ACID and all others are BASE; and $transaction_1$ is ACID and all others are BASE— because these two cases can combine to form any kinds of sequence except the all ACID sequence, which is trivial to prove.

- Case 1: $transaction_N$ is ACID and all others are BASE. In this case, the *read-after-write across transactions* property guarantees that if $w_1(x_1) \rightarrow r_2(x_1)$, then $r_2(x_1)$ releases the saline lock on object x_1 after $w_1(x_1)$ releases it. And the *write-after-read within a transaction* property guarantees that if $r_2(x_1) \rightarrow w_2(x_2)$, $w_2(x_2)$ releases the saline lock on object x_2 after $r_2(x_1)$ releases its saline lock on object x_1. By induction, we can prove that $w_1(x_1)$ releases the saline lock on object x_1 before $w_{N-1}(x_{N-1})$ releases its saline lock on object x_{N-1}. And since $transaction_N$ is ACID, $r_N(x_{N-1})$ needs to grab the ACID lock, which is conflicting with the saline lock on object x_{N-1}. This means that $r_N(x_{N-1})$ can not start before $w_{N-1}(x_{N-1})$ releases its saline lock on object x_{N-1}. And since $w_1(x_1)$

releases the saline lock on object x_1 before $w_{N-1}(x_{N-1})$ releases its saline lock on object x_{N-1}, $r_N(x_{N-1})$ can not start before $w_1(x_1)$ releases the saline lock on object x_1. And since a write saline lock is released at the end of a BASE transaction, $r_N(x_{N-1})$ can not start before $transaction_1$ completes, which means $transaction_1 \rightarrow r_N(x_{N-1})$.

- Case 2: $transaction_1$ is ACID and all others are BASE. Since $transaction_1$ is an ACID transaction, $w_1(x_1)$ holds the ACID write lock on object x_1 until the end of $transaction_1$. And since $transaction_2$ is BASE, $r_2(x_1)$ needs to acquire the saline read lock on x_1 and the saline read lock conflicts with the ACID write lock, so $transaction_1 \rightarrow r_2(x_1)$. And since $r_2(x_1) \rightarrow w_2(x_2) \rightarrow r_3(x_2) \rightarrow ... \rightarrow w_{N-1}(x_{N-1}) \rightarrow r_N(x_{N-1})$, we can get $transaction_1 \rightarrow r_N(x_{N-1})$.

To conclude, salt read committed or higher isolation levels avoid dirty read.

Lemma 3.3.3. *Under salt repeatable read or higher isolation levels, all ACID transactions are protected from fuzzy read.*

Formally, suppose $r_1(x_1) \in transaction_1$ *and* $w_2(x_1) \in transaction_2$, *if either* $transaction_1$ *or* $transaction_2$ *is an ACID transaction and* $r_1(x_1) \rightarrow w_2(x_1)$, *then* $transaction_1 \rightarrow w_2(x_1)$.

Proof If both $transaction_1$ and $transaction_2$ are ACID, this property is guaranteed by the original ACID implementation, so we only need to prove

the following two cases:

- Case 1: $transaction_1$ is ACID and $transaction_2$ is BASE. In this case, $r_1(x_1)$ holds the ACID read lock of object x_1 to the end of $transaction_1$. As since $transaction_2$ is BASE, $w_2(a)$ needs to acquire the saline write lock of x_1, which conflicts with the ACID read lock. Therefore, $transaction_1 \rightarrow w_2(x_1)$.

- Case 2: $transaction_1$ is BASE and $transaction_2$ is ACID. The proof is similar as that in Case 1.

In conclusion, salt repeatable read of higher salt isolation level avoid fuzzy read.

Lemma 3.3.4. *Under salt serializable level, all ACID transactions are protected from phantom read.*

Formally, suppose $r_1(P) \in transaction_1$ *and* $w_2(x\ in\ P) \in transaction_2$, *if either* $transaction_1$ *or* $transaction_2$ *is an ACID transaction and* $r_1(P) \rightarrow w_2(x\ in\ P)$, *then* $transaction_1 \rightarrow w_2(x\ in\ P)$. *(P denotes a predicate).*

Proof The proof is similar to that of Lemma 3 and the only difference is that $r_1(P)$ holds a range lock on predicate P, which conflicts with the write lock that $w_2(x\ in\ P)$ needs to acquire for object x.

Proof for Theorem According to Lemmas 1-4, Theorem 1 is proved.

Clarifying serializability The strongest lock-based isolation level, *locking-serializable* [24], not only prevents the four undesirable phenomena we mentioned earlier, but, in ACID-only systems, also implies the familiar definition of serializability, which requires the outcome of a serializable transaction schedule to be equal to the outcome of a serial execution of those transactions.

This implication, however, holds only if *all* transactions are isolated from all other transactions [24]; this is not desirable in a Salt database, since it would require isolating BASE transactions from each other, impeding Salt's performance goals.

Nonetheless, a Salt database remains true to the essence of the locking-serializable isolation level: it continues to protect its ACID transactions from all four undesirable phenomena, with respect to both BASE transactions and other ACID transactions. In other words, even though the presence of BASE transactions prevents the familiar notion of serializability to "emerge" from universal pairwise locking-serializability, ACID transactions enjoy in Salt the same kind of "perfect isolation" they enjoy in a traditional ACID system.

3.4 Implementation

We implemented a Salt prototype by modifying MySQL Cluster [11], a popular distributed database, to support BASE transactions and enforce Salt Isolation. MySQL Cluster follows a standard approach among distributed databases: the database is split into a number of partitions and each partition uses a master-slave protocol to maintain consistency among its replicas, which

are organized in a chain. To provide fairness, MySQL Cluster places operations that try to acquire locks on objects in a per-object queue in lock-acquisition order; Salt leverages this mechanism to further ensure that BASE transactions cannot cause ACID transactions to starve.

We modified the locking module of MySQL Cluster to add support for alkaline and saline locks. These modifications include support for (a) managing lock conflicts (see Table 3.2), (b) controlling when each type of lock should be acquired and released, as well as (c) a queuing mechanism that enforces the order in which saline locks are released, to avoid indirect dirty reads. Our current prototype uses the read-committed isolation level, as it is the only isolation level supported by MySQL Cluster.

3.4.1 Early commit for availability

To reduce latency and improve availability, Salt supports *early commit* [87] for BASE transactions: a client that issues a BASE transaction is notified that the transaction has committed when its first alkaline subtransaction commits. To ensure both atomicity and durability despite failures, Salt logs the logic for the entire BASE transaction before its first transaction commits. If a failure occurs before the BASE transaction has finished executing, the system uses the log to ensure that the entire BASE transaction will be executed eventually.

3.4.2 Failure recovery

Logging the transaction logic before the first alkaline subtransaction commits has the additional benefit of avoiding the need for managing cascading rollbacks of other committed transactions in the case of failures. Since the committed state of an alkaline subtransaction is exposed to other BASE transactions, rolling back an uncommitted BASE transaction would also require rolling back any BASE transaction that may have observed rolled back state. Instead, early logging allows Salt to roll uncommitted transactions forward.

The recovery protocol has two phases: *redo* and *roll forward*. In the first phase, Salt replays its redo log, which is populated, as in ACID systems, by logging asynchronously to disk every operation after it completes. Salt's redo log differs from an ACID redo log in two ways. First, Salt logs both read and write operations, so that transactions with write operations that depend on previous reads can be rolled forward. Second, Salt replays also operations that belong to partially executed BASE transactions, unlike ACID systems that only replay operations of committed transactions. During this phase, Salt maintains a *context* hash table with all the replayed operations and returned values (if any), to ensure that they are not re-executed during the second phase.

During the second phase of recovery, Salt rolls forward any partially executed BASE transactions. Using the logged transaction logic, Salt regenerates the transaction's query plan and reissues the corresponding operations.

Of course, some of those operations may have already been performed during the first phase: the *context* hash table allows Salt to avoid re-executing any of these operations and nonetheless have access to the return values of any read operation among them.

3.4.3 Transitive dependencies

As we discussed in Section 3.3, Salt needs to monitor transitive dependencies that can cause indirect dirty reads. To minimize bookkeeping, our prototype does not explicitly track such dependencies. Instead it only tracks *direct* dependencies among transactions and uses this information to infer the order in which locks should be released.

As we mentioned earlier, MySQL Cluster maintains a per-object queue of the operations that try to acquire locks on an object. Salt adds for each saline lock a pointer to the most recent non-ACID lock on the queue. Before releasing a saline lock, Salt simply checks whether the pointer points to a held lock—an O(1) operation.

3.4.4 Local transactions

Converting an ACID transaction into a BASE transaction can have significant impact on performance, beyond the increased concurrency achieved by enforcing isolation at a finer granularity. In practice, we find that although most of the performance gains in Salt come from fine-grain isolation, a significant fraction is due to a practical reason that compounds those gains: alkaline

subtransactions in Salt tend to be small, often containing a single operation.

Salt's *local-transaction* optimization, inspired by similar optimizations used in BASE storage systems, leverages this observation to significantly decrease the duration that locks are being held in Salt. When an alkaline subtransaction consists of a single operation, each partition replica can locally decide to commit the transaction—and release the corresponding locks—immediately after the operation completes. While in principle a similar optimization could be applied also to single-operation ACID transactions, in practice ACID transactions typically consist of many operations that affect multiple database partitions. Reaching a decision, which is a precondition for lock release, typically takes much longer in such transactions: locks must be kept while each transaction operation is propagated along the entire chain of replicas of each of the partitions touched by the transaction and during the ensuing two-phase commit protocol among the partitions. The savings from this optimization can be substantial: single-operation transactions release their locks about one-to-two orders of magnitude faster than non-optimized transactions.[3] Interestingly, these benefits can extend beyond single operation transactions—it is easy to extend the local-transaction optimization to cover also transactions where all operations touch the same object.

[3]This optimization applies only to ACID and alkaline locks. To enforce isolation between ACID and BASE transactions, saline locks must still be kept until the end of the BASE transaction.

```
1  begin BASE transaction
2    Check whether all items exist. Exit otherwise.
3      Select w_tax into @w_tax from warehouse where w_id =
           : w_id;
4    begin alkaline−subtransaction
5          Select d_tax into @d_tax, next_order_id into @o_id
               from district where w_id = : w_id and d_id =
               : d_id;
6          Update district set next_order_id = o_id + 1 where
               w_id = : w_id AND d_id = : d_id;
7    end alkaline−subtransaction
8      Select discount into @discount, last_name into @name,
           credit into @credit where w_id = : w_id and d_id =
           : d_id and c_id = : c_id
9      Insert into orders values (: w_id, : d_id, @o_id, ...);
10     Insert into new_orders values (: w_id, : d_id, o_id);
11     For each ordered item, insert an order line, update stock
           level, and calculate order total
12 end BASE transaction
```

Figure 3.5: A Salt implementation of the *new-order* transaction in TPC-C. The lines introduced in Salt are shaded.

3.5 Case Study: BASE-ifying *new-order*

We started this project to create a distributed database where performance and ease of programming could go hand-in-hand. How close does Salt come to that vision? We will address this question quantitatively in the next section, but some qualitative insight can be gained by looking at an actual example of Salt programming.

Figure 3.5 shows, in pseudocode, the BASE-ified version of *new-order*, one of the most heavily run transactions in the TPC-C benchmark (more about TPC-C in the next section). We chose *new-order* because, although its logic

is simple, it includes all the features that give Salt its edge.

The first thing to note is that BASE-ifying this transaction in Salt required only minimal code modifications (the highlighted lines 2, 4, and 7). The reason, of course, is Salt isolation: the intermediate states of *new-order* are isolated from all ACID transactions, freeing the programmer from having to reason about all possible interleavings. For example, TPC-C also contains the *deliver* transaction, which assumes the following invariant: if an order is placed (lines 9-10), then all order lines must be appropriately filled (line 11). Salt does not require any change to *deliver*, relying on Salt isolation to ensure that *deliver* will never see an intermediate state of *new-order* in which lines 9-10 are executed but line 11 is not.

At the same time, using a finer granularity of isolation between BASE transactions greatly increases concurrency. Consider lines 5-6, for example. They need to be isolated from other instances of *new-order* to guarantee that order ids are unique, but this need for isolation does not extend to the following operations of the transaction. In an ACID system, however, there can be no such distinction; once the operations in lines 5-6 acquire a lock, they cannot release it until the end of the transaction, preventing lines 8-11 from benefiting from concurrent execution.

3.6 Evaluation

To gain a quantitative understanding of the benefits of Salt with respect to both ease of programming and performance, we applied the ideas of Salt to

two applications: the TPC-C benchmark [36] and Fusion Ticket [8].

TPC-C is a popular database benchmark that models online transaction processing. It consists of five types of transactions: *new-order* and *payment* (each responsible for 43.5% of the total number of transactions in TPC-C), as well as *stock-level*, *order-status*, and *delivery* (each accounting for 4.35% of the total).

Fusion Ticket is an open source ticketing solution used by more than 80 companies and organizations [4]. It is written in PHP and uses MySQL as its backend database.

Unlike TPC-C, which focuses mostly on performance and includes only a representative set of transactions, a real application like Fusion Ticket includes several transactions—from frequently used ones such as *create-order* and *payment*, to infrequent administrative transactions such as *publishing* and *deleting-event*—that are critical for providing the required functionality of a fully fledged online ticketing application and, therefore, offers a more accurate view of the programming effort required to BASE-ify entire applications in practice.

Our evaluation tries to answer three questions:

- What is the performance gain of Salt compared to the traditional ACID approach?
- How much programming effort is required to achieve performance comparable to that of a pure BASE implementation?

- How is Salt's performance affected by various workload characteristics, such as contention ratio?

We use TPC-C and Fusion Ticket to address the first two questions. To address the third one, we run a microbenchmark and tune the appropriate workload parameters.

Experimental setup In our experiments, we configure Fusion Ticket with a single event, two categories of tickets, and 10,000 seats in each category. Our experiments emulate a number of clients that book tickets through the Fusion Ticket application. Our workload consists of the 11 transactions that implement the business logic necessary to book a ticket, including a single administrative transaction, *delete-order*. We do not execute additional administrative transactions, because they are many orders of magnitude less frequent than customer transactions and have no significant effect on performance. Note, however, that executing more administrative transactions would have incurred no additional programming effort, since Salt allows unmodified ACID transactions to safely execute side-by-side the few performance-critical transactions that need to be BASE-ified. In contrast, in a pure BASE system, one would have to BASE-ify *all* transactions, administrative ones included: the additional performance benefits would be minimal, but the programming effort required to guarantee correctness would grow exponentially.

In our TPC-C and Fusion Ticket experiments, data is split across ten partitions and each partition is three-way replicated. Due to resource limita-

tions, our microbenchmark experiments use only two partitions. In addition to the server-side machines, our experiments include enough clients to saturate the system.

All of our experiments are carried out in an Emulab cluster [18, 84] with 62 Dell PowerEdge R710 machines. Each machine is equipped with a 64-bit quad-core Xeon E5530 processor, 12GB of memory, two 7200 RPM local disks, and a Gigabit Ethernet port.

3.6.1 Performance of Salt

Our first set of experiments aims at comparing the performance gain of Salt to that of a traditional ACID implementation to test our hypothesis that BASE-ifying only a few transactions can yield significant performance gains.

Our methodology for identifying which transactions should be BASE-ified is based on a simple observation: since Salt targets performance bottlenecks caused by contention, transactions that are good targets for BASE-ification are large and highly-contented. To identify suitable candidates, we simply increase the system load and observe which transactions experience a disproportionate increase in latency.

Following this methodology, for the TPC-C benchmark we BASE-ified two transactions: *new-order* and *payment*. As shown in Figure 3.6, the ACID implementation of TPC-C achieves a peak throughput of 1464 transactions/sec. By BASE-ifying these two transactions, our Salt implementation achieves a throughput of 9721 transactions/sec—6.6x higher than the ACID

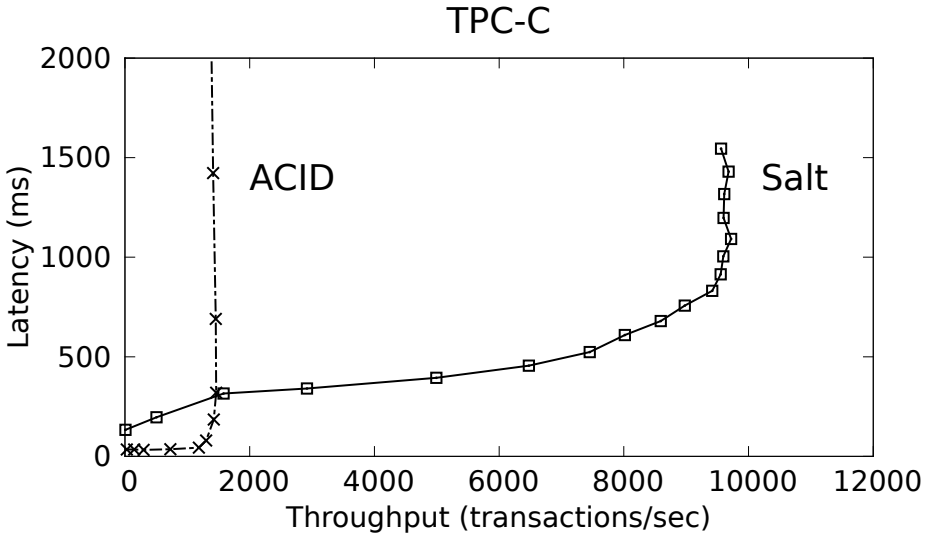


Figure 3.6: Performance of TPC-C.

throughput.

For the Fusion Ticket benchmark, we only BASE-ify one transaction, *create-order*. This transaction is the key to the performance of Fusion Ticket, because distinct instances of *create-order* heavily contend with each other. As Figure 3.7 shows, the ACID implementation of Fusion Ticket achieves a throughput of 1088 transactions/sec, while Salt achieves a throughput of 7090 transactions/sec, $6.5x$ higher than the ACID throughput. By just BASE-ifying *create-order*, Salt can significantly reduce how long locks are held, greatly increasing concurrency.

In both the TPC-C and Fusion Ticket experiments Salt's latency under low load is higher than that of ACID. The reason for this disparity lies in how requests are made durable. The original MySQL Cluster implemen-

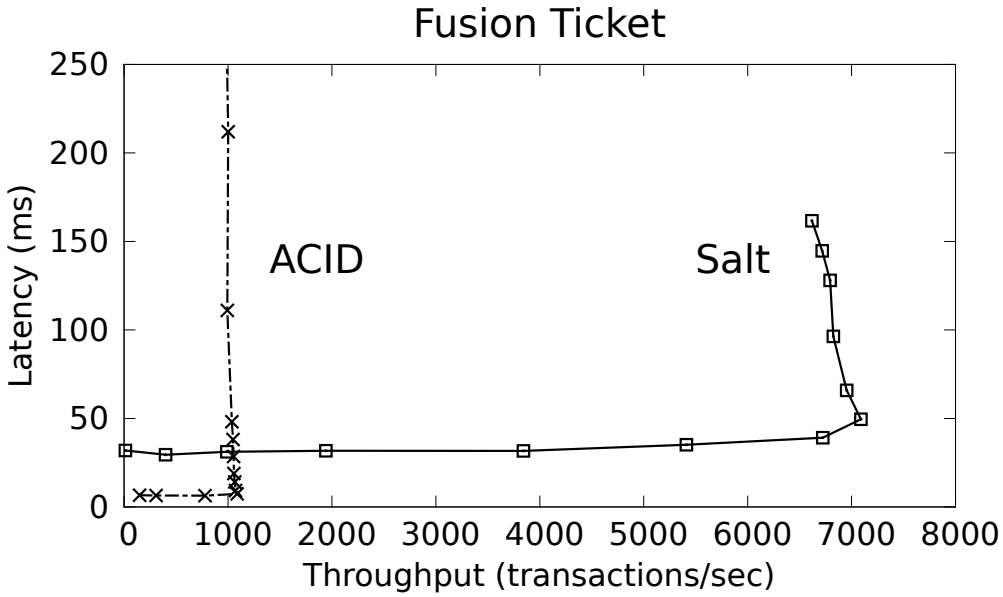


Figure 3.7: Performance of Fusion Ticket.

tation returns to the client *before* the request is logged to disk, providing no durability guarantees. Salt, instead, requires that all BASE transactions be durable before returning to the client, increasing latency. This increase is exacerbated by the fact that we are using MySQL Cluster's logging mechanism, which—having been designed for asynchronous logging—is not optimized for low latency. Of course, this phenomenon only manifests when the system is under low load; as the load increases, Salt's performance benefits quickly materialize: Salt outperforms ACID despite providing durability guarantees.

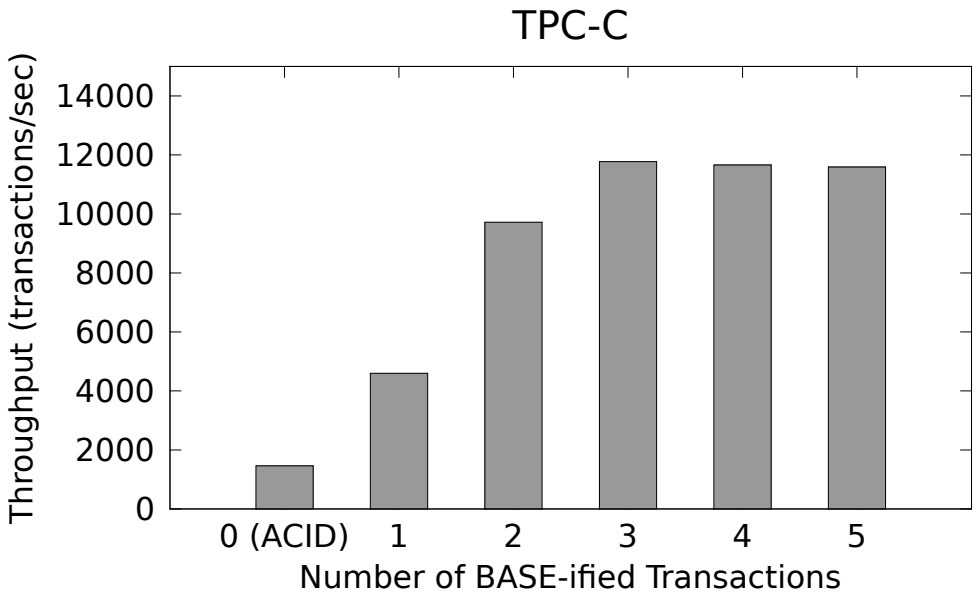


Figure 3.8: Incremental BASE-ification of TPC-C.

3.6.2 Programming effort vs Throughput

While Salt's performance over ACID is encouraging, it is only one piece of the puzzle. We would like to further understand how much programming effort is required to achieve performance comparable to that of a pure BASE implementation—i.e. where all transactions are BASE-ified. To that end, we BASE-ified as many transactions as possible in both the TPC-C and Fusion Ticket codebases, and we measured the performance they achieve as we increase the number of BASE-ified transactions.

Figure 3.8 shows the result of incrementally BASE-ifying TPC-C. Even with only two BASE-ified transactions, Salt achieves 80% of the maximum throughput of a pure BASE implementation; BASE-ifying three transactions

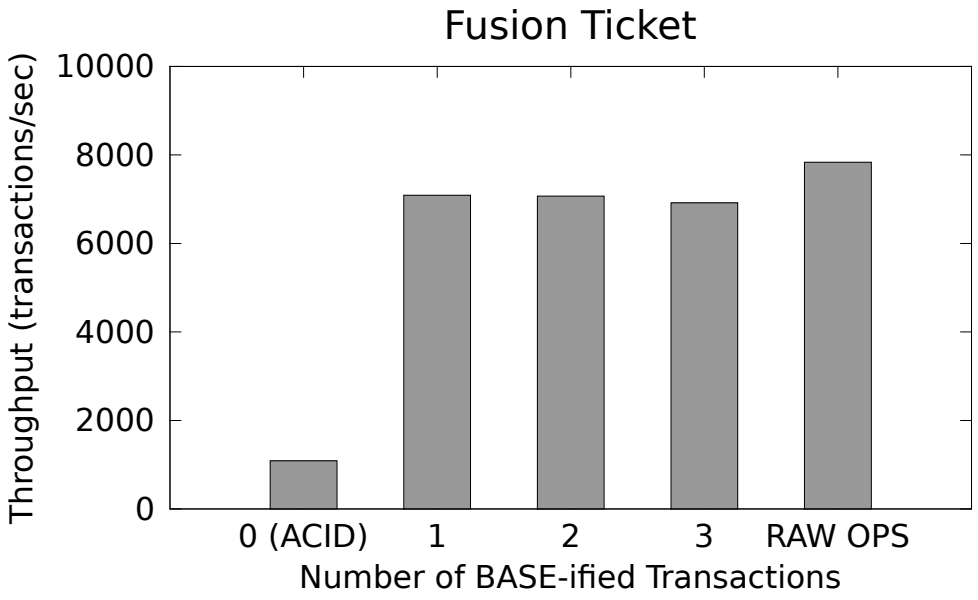


Figure 3.9: Incremental BASE-ification of FT.

actually reaches that throughput. In other words, there is no reason to BASE-ify the remaining two transactions. In practice, this simplifies a developer's task significantly, since the number of state interleavings to be considered increases exponentially with each additional transactions that need to be BASE-ified. Further, real applications are likely to have proportionally fewer performance-critical transactions than TPC-C, which, being a performance benchmark, is by design packed with them.

To put this expectation to the test, we further experimented with incrementally BASE-ifying the Fusion Ticket application. Figure 3.9 shows the results of those experiments. BASE-fying one transaction was quite manageable: it took about 15 man-hours—without prior familiarity with the code—

and required changing 55 lines of code, out of a total of 180,000. BASE-ifying this first transaction yields a benefit of $6.5x$ over ACID, while BASE-ifying the next one or two transactions with the highest contention does not produce any additional performance benefit.

What if we BASE-ify more transactions? This is where the aforementioned exponential increase in state interleavings caught up with us: BASE-ifying a fourth or fifth transaction appeared already quite hard, and seven more transactions were waiting behind them in the Fusion Ticket codebase! To avoid this complexity and still test our hypothesis, we adopted a different approach: we broke down all 11 transactions into raw operations. The resulting system does not provide, of course, any correctness guarantees, but at least, by enabling the maximum degree of concurrency, it lets us measure the maximum throughput achievable by Fusion Ticket. The result of this experiment is labeled RAW OPS in Figure 3.9. We find it promising that, even by BASE-ifying only one transaction, Salt is within 10% of the upper bound of what is achievable with a BASE approach.

3.6.3 Contention

To help us understand how contention affects the performance of Salt, we designed three microbenchmarks to compare Salt, with and without the local-transaction optimization, to an ACID implementation.

In the first microbenchmark, each transaction updates five rows, randomly chosen from a collection of N rows. By tuning N, we can control the

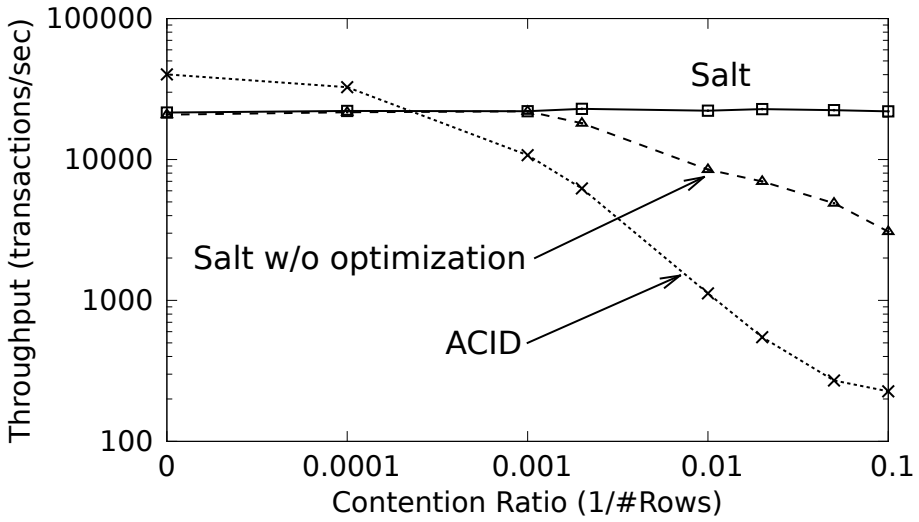


Figure 3.10: Effect of contention ratio on throughput.

amount of contention in our workload. Our Salt implementation uses BASE transactions that consist of five alkaline subtransactions—one for each update.

Figure 3.10 shows the result of this experiment. When there is no contention, the throughput of Salt is somewhat lower than that of ACID, because of the additional bookkeeping overhead of Salt (e.g., logging the logic of the entire BASE transaction). As expected, however, the throughput of ACID transactions quickly decreases as the contention ratio increases, since contending transactions cannot execute in parallel. The non-optimized version of Salt suffers from this degradation, too, albeit to a lesser degree; its throughput is up to an order of magnitude higher than that of ACID when the contention ratio is high. The reason for this increase is that BASE transactions contend on alkaline locks, which are only held for the duration of the current alkaline

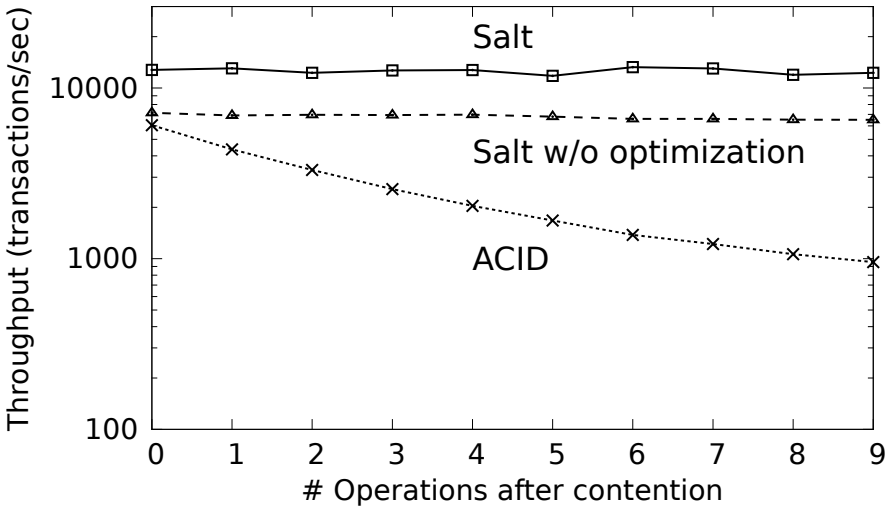


Figure 3.11: Effect of contention position on throughput.

subtransactions and are thus released faster than ACID locks. The optimized version of Salt achieves further performance improvement by releasing locks immediately after the operation completes, without having to wait for the operation to propagate to all replicas or wait for a distributed commit protocol to complete. This leads to a significant reduction in contention; so much so, that the contention ratio appears to have negligible impact on the performance of Salt.

The goal of the second microbenchmark is to help us understand the effect of the relative position of contending operations within a transaction on the system throughput. This factor can impact performance significantly, as it affects how long the corresponding locks must be held. In this experiment, each transaction updates ten rows, but only one of those updates contends

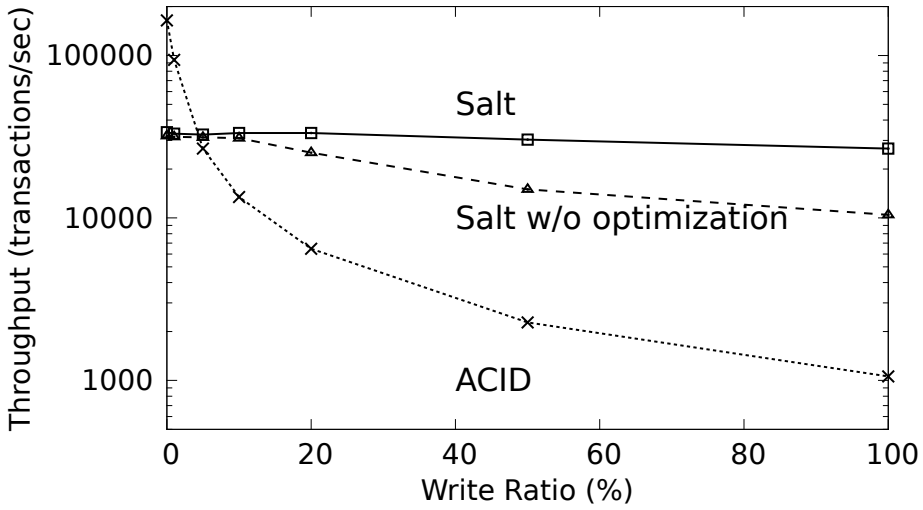


Figure 3.12: Effect of read-write ratio on throughput.

with other transactions by writing to one row, randomly chosen from a collection of ten shared rows. We tune the number of operations that follow the contending operation within the transaction, and measure the effect on the system throughput.

As Figure 3.11 shows, ACID throughput steadily decreases as the operations that follow the contending operation increase, because ACID holds an exclusive lock until the transaction ends. The throughput of Salt, however, is not affected by the position of contending operations because BASE transactions hold the exclusive locks—alkaline locks—only until the end of the current alkaline subtransaction. Once again, the local-transaction optimization further reduces the contention time for Salt by releasing locks as soon as the operation completes.

The third microbenchmark helps us understand the performance of Salt under various read-write ratios. The read-write ratio affects the system throughput in two ways: (i) increasing writes creates more contention among transactions; and (ii) increasing reads increases the overhead introduced by Salt over traditional ACID systems, since Salt must log read operations, as discussed in Section 3.4. In this experiment each transaction either reads five rows or writes five rows, randomly chosen from a collection of 100 rows. We tune the percentage of read-only transactions and measure the effect on the system throughput.

As Figure 3.12 shows, the throughput of ACID decreases quickly as the fraction of writes increases. This is expected: write-heavy workloads incur a lot of contention, and when transactions hold exclusive locks for long periods of time, concurrency is drastically reduced. The performance of Salt, instead, is only mildly affected by such contention, as its exclusive locks are held for much shorter intervals. It is worth noting that, despite Salt's overhead of logging read operations, Salt outperforms ACID even when 95% of the transactions are read-only transactions.

In summary, our evaluation suggests that, by holding locks for shorter times, Salt can reduce contention and offer significant performance improvements over a traditional ACID approach, without compromising the isolation guarantees of ACID transactions.

3.7 Conclusion

The ACID/BASE dualism has to date forced developers to choose between ease of programming and performance. Salt shows that this choice is a false one. Using the new abstraction of BASE transactions and a mechanism to properly isolate them from their ACID counterparts, Salt enables a tenable middle ground between ACID and BASE paradigms; a middle group where performance can be incrementally attained by gradually increasing the programming efforts required.

Chapter 4

Callas

Salt achieves most of the ease of programming of ACID and most of performance of BASE by rewriting performance-critical transactions incrementally. However, the extra programming effort involved is still non-trivial and can easily introduce bugs. Callas[1] aims to move beyond the ACID/BASE dilemma. Rather than trying to draw performance from weakening the abstraction offered to the programmers, Callas unequivocally adopts the familiar abstraction offered by the ACID paradigm and set its sight on finding a more efficient way to implement that abstraction.

The key observation that motivates the architecture of Callas is simple. While ease of programming requests that ACID properties hold uniformly across all transactions, when it comes to the mechanisms used to enforce these properties, uniformity can actually hinder performance: a concurrency control mechanism that must work correctly for *all* possible pairs of transactions will necessarily have to make conservative assumptions, passing up opportunities

[1]This chapter is based on "High-Performance ACID via Modular Concurrency Control" [86], authored by Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos and Yang Wang, published in the proceedings of the 25th ACM Symposium on Operating Systems Principles. Chao designed the protocol and implemented most of the Callas prototype.

for optimization.

Callas then decouples the concerns of abstraction and implementation: it offers ACID guarantees uniformly to all transactions, but uses a novel technique, *modular concurrency control* (MCC), to customize the mechanism through which these guarantees are provided.

MCC makes it possible to think modularly about the enforcement of any given isolation property I. It enables Callas to partition transactions in separate groups, and it ensures that as long as I holds within each group, it will also hold among transactions in different groups. Separating concerns frees Callas to use within each group concurrency control mechanisms optimized for that group's transactions. Thus, Callas can find opportunities for increased concurrency where a generic mechanism might have to settle for a conservative execution. For example, taking advantages of the flexibility offered by MCC, Callas in-group concurrency control mechanism, *Runtime Pipelining*, leverages execution-time information to aggressively weaken within a group the conservative requirements of the current theory of safe transaction chopping [74] and gains, as a result, unprecedented opportunities for concurrency.

We have built Callas by modifying MySQL Cluster distributed database. Our evaluation of Callas suggests that MCC can deliver significant performance gains to unmodified ACID applications. For example, we find that, for TPC-C, Callas achieves an 8.2x speedup over MySQL Cluster without requiring any programming effort.

The rest of Chapter is organized as follows. The next sections describe the criteria to articulate correctness. After Section 4.1 discusses why an undifferentiated concurrency control mechanism is undesirable, Section 4.2 introduces MCC and specifies the correctness conditions that any valid instantiation of MCC must meet. Section 4.3 and Section 4.4 present the mechanisms Callas uses to ensure isolation across groups and within each group, respectively. The implementation of Callas is the topic of Section 4.5, while Section 4.6 presents the results of our experimental evaluation. Section 4.7 concludes.

4.1 The cost of uniformity

The power of the ACID paradigm lies in its simplicity. Developers need only wrap their code in an ACID transaction, and it is guaranteed to be executed atomically, to leave the database in a consistent state, to be isolated from any other transactions, and to be durable. One of the great assets of this abstraction is that it applies uniformly to all transactions, independent of the internal logic of other transactions, thus freeing the developer from having to worry about transaction interleavings.

Current ACID databases support this uniform abstraction with an equally uniform mechanism. Notwithstanding its simplicity, when it comes to enforcing isolation this choice can become an obstacle to performance and scalability. Whether using locking or optimistic concurrency control (OCC), current ACID databases rely on one-size-fits-all—and thus fundamentally conservative —mechanisms to ensure isolation.

```
1  // transfer_balance
2  begin transaction
3    bal_dest = bal_dest + val
4    bal_orig = bal_orig − val
5  commit


8  // sum_balance (infrequent)
9  begin transaction
10   return bal_orig + bal_dest
11 commit
```

Figure 4.1: A simple banking application.

For example, when a transaction accesses an object (e.g., a database row), a lock-based mechanism must acquire a lock that is held until the end of the transaction, preventing all other transactions from observing intermediate states of that transaction. Perhaps surprisingly, given their name, OCC mechanisms are, in their own way, equally conservative. Although they allow transactions to speculatively execute in parallel without acquiring locks, they do not refine the criteria for determining contention, but simply delay the check: if contention is detected at commit time, they force all but one of the contending transactions to rollback.

By treating all transactions equally, one-size-fits-all mechanisms cannot take advantage of workload-specific optimizations. Isolation is uniformly enforced to prevent all other transactions from observing intermediate states. In some circumstances, however, such precautions are excessive: it is quite common to find transactions that can safely expose *some* of their intermediate

states to *some* other transactions.

Consider, for example, how a lock-based mechanism would handle the previous simple banking application in Figure 4.1: *transfer_balance* deducts some amount from the *bal_orig* account and adds it to *bal_dest*; *sum_balance* computes the total assets across the two accounts.

When these transactions can execute concurrently, uniformly enforcing isolation requires *transfer_balance* to keep the lock on *bal_dest* until the transaction commits, to prevent *sum_balance* from computing the wrong total by observing the intermediate state where *bal_dest* has been credited but *bal_orig* not yet charged. Keeping the locks for so long, however, also prevents other instances of the *transfer_balance* transaction from executing concurrently, even though they could do so safely, since their operations commute.[2] Ideally, one would like to release the lock on *bal_orig* after the amount is deducted from it, but only for other *transfer_balance* transactions; *sum_balance* should still be prevented from observing the intermediate state.

Salt shows that leveraging this insight can yield significant performance benefits. To extract them, Salt forgoes the simplicity of a uniform ACID paradigm and instead introduces BASE transactions. The cost, as we have seen, is added complexity for the programmer.

In Salt, abandoning a uniform concurrency control mechanism to in-

[2]Commutativity is only one example of the missed opportunities for greater concurrency that constitute the cost of uniformity—we discuss the performance implications for transaction chopping [74] in Section 4.4.

crease performance has led, as if by necessity, to also surrendering the benefits of a uniform ACID abstraction. In its own attempts to leverage the same insight that motivated Salt, Callas strives to stay clear of the pitfall of tightly coupling the scope of mechanism and abstraction, and comes to a fundamentally different conclusion.

4.2 A modular approach to isolation

Callas aims to offer to unmodified, transactional ACID applications the kind of performance that previously could only be achieved by rewriting applications in a BASE/NoSQL style. Such coding exercises are notorious for being error-prone and time consuming [75], even when backed by Object-Relational Mapping systems [22]. Callas' goal is to do away with them completely, with only a negligible cost in performance.

The design of Callas is based on a simple proposition: that the key for combining performance and ease of programming is to *decouple* the ACID abstraction—which should hold identically for all transactions—from the mechanism used to support it—which should instead adapt to the unique characteristics of different transactions. The approach that we propose is rooted in three main observations.

First, no existing programming paradigm approaches the simplicity offered by ACID. Such is its superiority on this front to bring into question whether any performance benefit that a BASE alternative can deliver is actually worth the trouble [35, 75].

Second, significant improvements to the performance of ACID are unlikely to come from techniques that rely on properties that must hold for *all* of the transactions in a given application. A case in point is transaction chopping [74], an elegant technique that can yield greater concurrency while maintaining serializability, but only if a specific property (which can be formalized as the absence of SC-cycles[3] [74]) holds across the entire set of an application's transactions. In practice, enforcing this property can often significantly limit opportunities for concurrency in applications that suffer from high contention.

Third, as Salt has demonstrated, the potential performance gains to be had by allowing individual transactions to export multiple granularities of isolation can be substantial.

The architecture of Callas leverages these observations by supporting a modular approach to regulating concurrency, realized through a novel technique we call *modular concurrency control* (MCC). The vision that motivates MCC is simple. Instead of relying on a single concurrency control mechanism for all transactions, MCC partitions transactions in groups and enables the flexibility to assign to each group its own private concurrency control mechanism; being charged with regulating concurrency only for the transactions within their own groups, these mechanisms can be much more aggressive while still upholding safety. Finally, MCC offers a mechanism to properly handle

[3]We will discuss SC-cycles in more detail in Section 4.4.

conflicts among transactions in different groups.

An attractive feature of this approach is its generality. First, it imposes no restrictions on the types of transactions it can handle. In particular, it does not require to predefine all transactions that will be run in the system: interactive or external transactions can always be handled by placing them in a separate group that uses a standard, conservative concurrency control mechanism. Second, although our current implementation of Callas leverages modularity only within the context of lock-based mechanisms, MCC does not, in principle, depend on whether concurrency control is implemented using locks or OCC, or on whether the targeted isolation level relies on a single version or a multiversion database—we leave a thorough exploration of the performance opportunities offered by this generality to future work.

To succeed, this high-level plan must address two complementary concerns: performance and correctness.

The key factor for performance is to group transactions appropriately, in order for each group to best exploit opportunities for optimizations. Not all grouping choices are equally sensitive, however: optimizing grouping for transactions that run infrequently or are lightweight is less critical. Callas therefore heuristically assigns those transactions to a single group, and instead focuses on determining the most favorable grouping for the transactions that are primarily shaping the performance profile of a given application. We discuss the policy and mechanism used by Callas to group transactions in Sections 4.3 and 4.5.

Establishing correctness involves a two step process: given any of the traditional ACID isolation guarantees, first prove that each group, separately, satisfies the guarantee; and then, under the assumption that all groups do, that the isolation guarantee is upheld globally.

The theoretical underpinnings that Callas uses to discharge these obligations are found in the Adya's general definition for expressing isolation levels as we mentioned in Section 2.1.1. According to this definition, for any given isolation level, an instantiation of the Callas architecture must satisfy the following conditions to guarantee correctness:

- **Within each group** The concurrency control mechanism for group G must prevent prevent Circularity (as defined for the targeted isolation level) when all transactions on the cycle are in G. In addition, for isolation levels except Read Uncommitted, The concurrency control mechanism for group G must prevent Aborted Reads and Intermediate Reads if T_1 and T_2 are both in G.

- **Across groups** Circularity (as defined for the targeted isolation level) must be prevented if at least two transactions on the cycle are from different groups. Further, for isolation levels except Read Uncommitted, Aborted Reads and Intermediate Reads must be prevented if T_1 and T_2 are from different groups.

The next two sections describe the design of Callas along the two axes we have used to articulate correctness. Section 4.3 describes how Callas lever-

ages a new class of locks, called *nexus locks*, to prevent Circularity and proscribe Aborted and Intermediate reads across groups.

Section 4.4 introduces a new in-group concurrency control mechanism, called *Runtime Pipelining*, designed to leverage the modularity of Callas: since it regulates concurrency for only a small number of transactions, it can afford to apply aggressive optimizations. Runtime Pipelining owes much of its performance—as well as its name—to its integration of static analysis with novel run-time checks that guarantee safety while increasing opportunities for concurrency.

4.3 Enforcing isolation across groups

The design of the mechanism Callas uses to guarantee inter-group isolation is driven by several considerations. Foremost, of course, is safety: the mechanism should enforce the correctness conditions identified in Section **??**. Not far behind, however, are performance and liveness. First, we would like the inter-group mechanism to disrupt as little as possible the ability of the group-specific mechanisms to extract concurrency from the transactions they regulate. Second, we would like to guarantee fairness: the eagerness of exploiting performance opportunities within a group should not cause transactions from a less fortunate group to starve.

Callas meets these requirements using a simple lock-based approach. This choice is pragmatic: although there is nothing in Callas' architecture that would prevent the use of OCC, the MySQL Cluster distributed database

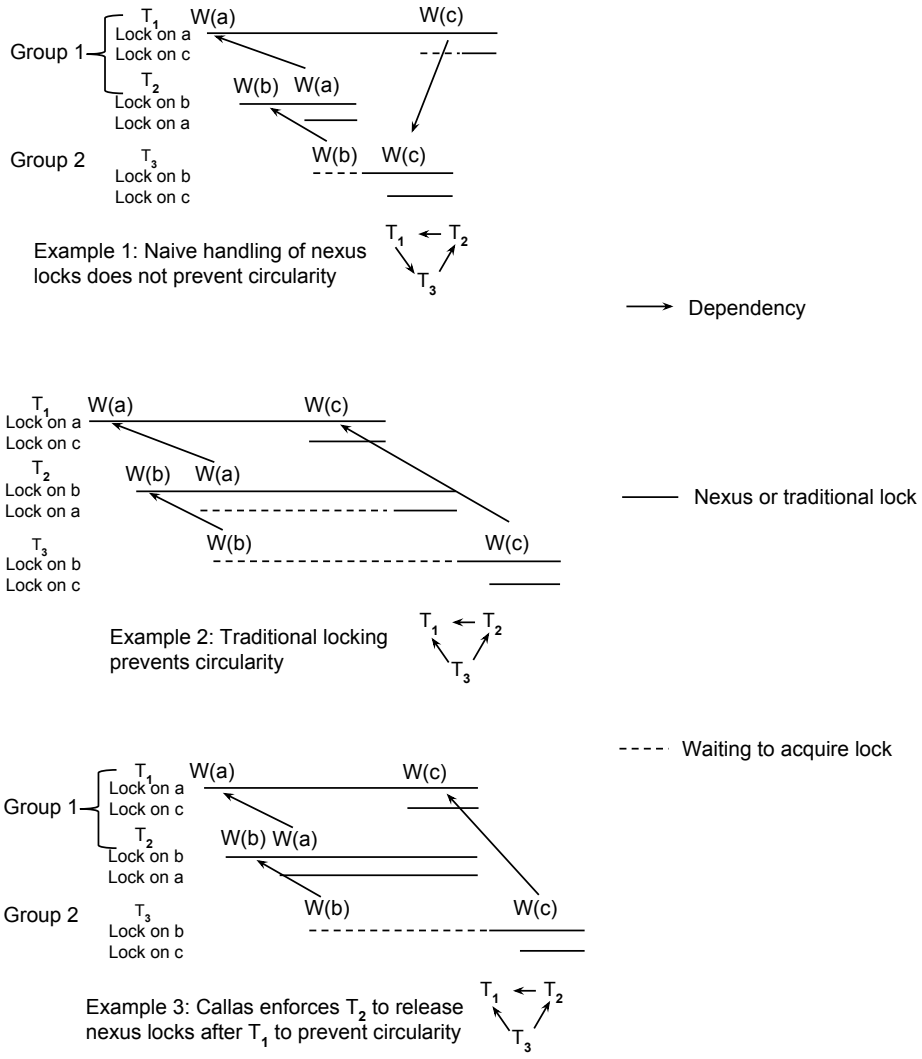


Figure 4.2: Circularity can occur if Callas does not regulate the order in which transactions from the same group release their nexus locks.

we modify to implement Callas does not support it.[4] Indeed, any reasonable implementation of the inter-group mechanism that meets the above requirement of minimal disruption will do.

At the core of Callas' inter-group mechanism are *nexus locks*, a new type of lock whose role is to regulate conflicts between transactions that belong to different groups while leaving transactions within each group relatively unconstrained. Nexus locks in Callas are ubiquitous: any transaction, before being allowed to perform a read or write operation on a database row, must acquire the corresponding nexus lock. This demand may seem to run contrary to the requirement of making the inter-group mechanism inconspicuous to the concurrency control mechanisms specific to each group. The key to resolving this apparent tension lies in the flexibility of nexus locks. When two transactions in different groups try to acquire a nexus lock on a row, the lock functions as an enforcer: unless both transactions are *reading* the row, one of them will have to wait until the other releases the lock. If the transactions belong to the same group, however, the nexus lock imposes no such constraints: both transactions can acquire the nexus lock simultaneously.

Forcing transactions to acquire nexus locks on the rows they access prevents Aborted Reads and Intermediate Reads from occurring across groups. If two transactions from different groups access the same row and one of them is a write, only the first will acquire the row's nexus lock, while the other will

[4]MySQL Cluster supports only two versions of each object. While this feature guarantees that reads never block, it falls short of full multiversion concurrency control (MVCC).

not be able to acquire the lock until the earlier transaction completes. It is thus impossible for the later transaction to read aborted or intermediate states from the earlier one.

Simply acquiring nexus locks, however, is not sufficient to prevent Circularity. Consider the first example of Figure 4.2: it focuses on write dependencies, since write dependency cycles are forbidden by all ANSI isolation levels. Assume, in the spirit of MCC, that the concurrency control mechanism of Group 1 guarantees no dependency cycles between T_1 and T_2. Although nexus locks prevent dependency cycles between T_1 and T_3 (and similarly between T_2 and T_3), a dependency cycle spanning T_1, T_2, and T_3 can still form.

We extend the enforcement power of nexus locks by refining the way in which traditional locking prevents Circularity. With traditional locking (Example 2 of Figure 4.2), the "depends on" relation between transactions is tied to the "completes before" relation: if T_2 depends on T_1, then T_2 must wait for T_1 to release its lock at the end of its execution, ensuring that T_2 will not start until T_1 completes. Since "completes before", unlike "depends on", is inherently acyclic, by tying the two relations traditional locking guarantees that "depends on" will be acyclic too.

If we now go back to the first example of Figure 4.2, what went wrong there is clear: Circularity can arise because nexus locks tie "depends on" to "completes before" *only* for transactions that belong to different groups. Although T_2 depends on T_1, since they are both in the same group T_2 is allowed to start before T_1 completes. Were nexus locks to do otherwise, however, and

delay T_2, they would curb concurrency within Group 1.

To solve this puzzle, Callas refines the condition used by traditional locking to avoid circularity. Rather than tying "depends on" to "completes before", Callas binds it to the weaker (and yet provably sufficient [**?**]) "releases locks before" and enforces the following rule:

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

The third example of Figure 4.2 illustrates how this rule, which is implied by the stronger "completes before", prevents dependency cycles without hampering concurrency. Now, we prove its correctness as following:

Definition 1. We use $T_i \rightarrow T_j$ to denote T_j depends on T_i.

Lemma 4.3.1. *If $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$, then T_n cannot release its nexus lock until T_1 does.*

Proof. First, we prove that for any $T_i \rightarrow T_j$, T_j cannot release its nexus locks until T_i does. Case 1: T_i and T_j are from the same group. In this case, Nexus Lock Release Order ensures that T_j cannot release its nexus lock until T_i does. Case 2: T_i and T_j are from different groups. In this case, since T_j cannot acquire the nexus lock until T_i releases it, of course T_j cannot release the lock until T_i does. Then one can easily prove Lemma 1 by induction.

Theorem 4.3.2. *Nexus locks prevent dependency cycles spanning multiple groups.*

Proof. We prove by contradiction. Assume there exists a dependency cycle spanning mutiple groups $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$. Since it spans multipe groups, there must exist T_{j-1} and T_j such that $T_{j-1} \rightarrow T_j$ and they are from different groups. This means T_{j-1} must release the nexus lock before T_j can acquire it to finish execution (Fact 1: $t_{release}(T_{j-1}) < t_{finish}(T_j)$). On the other hand, the assumption that $T_j \rightarrow T_{j+1} \rightarrow ...T_{j-1}$ means T_{j-1} cannot release its its nexus lock until T_j does (Lemma 1), and T_j only releases its nexus locks after it finishes (Fact 2: $t_{release}(T_{j-1}) > t_{finish}(T_j)$). Facts 1 and 2 contradict with each other and thus it is impossible to form a dependency cycle spanning multiple groups.

To ensure that every transaction's nexus locks are eventually released, Callas makes the simple choice of maintaining a FIFO queue for each nexus lock. Note that, thanks to MCC, the release of nexus locks is completely decoupled from the act of committing the transaction that holds those locks, which Callas leaves to the concurrency control mechanism of the group to which the transaction belongs. As soon as a transaction commits, any resource that the transaction held to control concurrency *within* its group can be released, even as the transaction may hold onto its nexus locks in order to release them in the correct order.

Nexus locks, ACID locks, and latency An unobtrusive inter-group mechanism is essential to achieving the potential for greater performance of MCC. A key feature of nexus locks is that any latency overhead they introduce, when compared with ACID locks, is due solely to their implementation, and not inherent to their semantics. Indeed, ignoring implementation overheads, if T_2 wants to acquire and then release a nexus lock held by T_1, it can always do so no later than if the lock had been ACID. The reason is simple: if T_1 and T_2 are from different groups, then a nexus lock behaves exactly like an ACID lock; if T_1 and T_2 are from the same group, then T_2 is always allowed to acquire a nexus lock while T_1 still holds it, while instead access to ACID locks is exclusive unless both T_1 and T_2 seek a read lock. The Nexus Lock Release Order rule can delay the release of T_2's locks if T_2 depends on T_1, but never more than if the locks had been ACID—in which case, T_2 would not even be allowed to acquire the locks until T_1 released them.

Of course, implementation overheads cannot in practice be ignored. However, we find them to be low in most cases (§4.6.4), in particular when compared with the substantial performance gains nexus locks enable by making it possible to safely deploy the kind of aggressive in-group concurrency control mechanisms we are discussing next.

4.4 Enforcing isolation within groups

While the inter-group mechanism's main goal is to do no harm, the key to unlocking the performance potential of MCC is in the group-specific concur-

rency control mechanisms that it enables. Fulfilling that potential involves two steps: grouping transactions appropriately, and identifying mechanisms that can yield greater concurrency within each group, while maintaining safety.

The first of these steps appears hard to complete, as the number of possible groupings to consider is exponential. In practice, our experience building Callas is significantly more encouraging. As we already pointed out, the transactions that shape the performance of an application tend to be few [85] and we found that even just one or two simple specialized mechanisms can produce significant performance gains (§4.6): with such small numbers, systematically exploring all interesting groupings becomes a tractable problem (§4.5).

The additional concurrency called for by the second step demands transactions to expose more intermediate states. This could be done, for instance, by weakening their isolation properties [85], but to do so within a group would violate our requirement to offer all transactions the same ACID abstraction.

Transaction chopping (and its limitations) An attractive alternative is to turn, as several recent systems have done [66, 87], to an elegant theory that increases concurrency by chopping transactions—but in a way guaranteed to maintain serializability [74].

To prevent Aborted Reads and guarantee Atomicity, the theory requires transactions to be *rollback-safe*, meaning that any rollback statement must lie in the first subtransaction produced by a valid chopping. Since for

serializability the absence of Circularity implies no Intermediate Reads, the theory focuses on preventing the former. It uses static analysis to construct an *SC-graph*, whose vertices are candidate transaction pieces, and whose edges, which are undirected, are of two kinds: S-edges connect the pieces within a transaction; C-edges connect pieces of different transactions that access the same object, when at least one of the accesses is a write. The theory shows that if a candidate chopping gives rise to an *SC-cycle*, then Circularity *might* arise during an execution. Hence, a candidate chopping of a set of transactions is considered safe (i.e., guarantees serializability) if (i) it is rollback-safe and (ii) it contains no SC-cycles.

Unfortunately, in practice these two conditions tend to produce choppings too conservative to result in much additional concurrency. To satisfy rollback safety, the first piece of each transaction must be large enough to include all rollback statements, limiting the opportunity for new interleavings.[5]

Relying on SC-cycles for safety has even more significant performance implications. Applications typically contain so many dependency cycles among their transactions that the only safe choppings, if any, are very coarse. One might expect grouping to help here, since it restricts the requirement of being free of SC-cycles only to the transactions within each group—and it does (§4.6), but only to a limited extent. We find that SC-cycles tend to arise quite com-

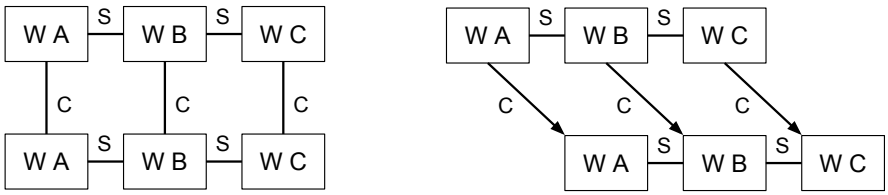
[5]This problem could be solved by asking application developers to rewrite their transactions to explicitly account for rollbacks at the application level. Our goal, however, is to achieve high performance with no additional programming effort.

monly among the very performance-critical transactions that, if they could be more finely chopped, would most benefit the application's performance. An extreme but quite common case of this phenomenon occurs when a performance-critical transaction cannot be aggressively chopped because multiple instances of it may conflict with each other if executing concurrently.

Consider, for example, the *new_order* transaction in TPC-C. In first approximation, it roughly follows the access pattern of Figure 4.3(a): first, it inserts rows into the *order* table, then it inserts rows into the *item* table, and finally it updates the *order_line* table. As Figure 4.3(a) shows, SC-cycle analysis would conclude that it is not possible to split this transaction into subtransactions, as any two instances of the *new_order* transaction have three dependency edges (C-edges) between them.

Enter MCC These limitations motivate us to explore how to leverage the modularity of MCC to move beyond the opportunities for concurrency offered by the current theory of safe transaction chopping. To that end, Callas introduces *Runtime Pipelining*, a new in-group mechanism whose aggressive approach to concurrency control proves particularly effective within small groups. Runtime Pipelining relies on two new techniques: it leverages at execution time a refinement of the static analysis approach used by traditional transaction chopping to allow concurrency when SC-cycles would prevent it; and it prevents Aborted Reads and guarantees atomicity while avoiding, whenever possible, the performance downsides of enforcing rollback safety.

Similar to transaction chains [87], Runtime Pipelining assumes that



(a) SC-cycle analysis cannot chop (b) Runtime Pipelining

Figure 4.3: Runtime Pipelining for create_order transaction in TPC-C. A=order table, B=item table, C=order_line table

the tables (though not necessarily the rows) accessed by each transaction are known prior to execution. The *scope* of this assumption, however, is much weaker than in transaction chains, since it applies only to the transactions in the target group. In practice, this assumption needs only to hold for the few transactions that, being performance critical, can most take advantage of a more aggressive in-group concurrency control mechanism.

4.4.1 Runtime Pipelining

Shasha et al. prove [74] that their static analysis technique produces the finest transaction chopping *guaranteed* to be safe: any more refined chopping has the potential to create Circularity and violate serializability. This sobering fact, however, does not imply that renouncing any further concurrency need be the price of safety. The key insight behind Runtime Pipelining is that, rather than preemptively inhibiting the *possibility* of Circularity, it may be feasible in some circumstances to allow for that possibility, relying instead on run-time techniques to prevent it from becoming an actuality.

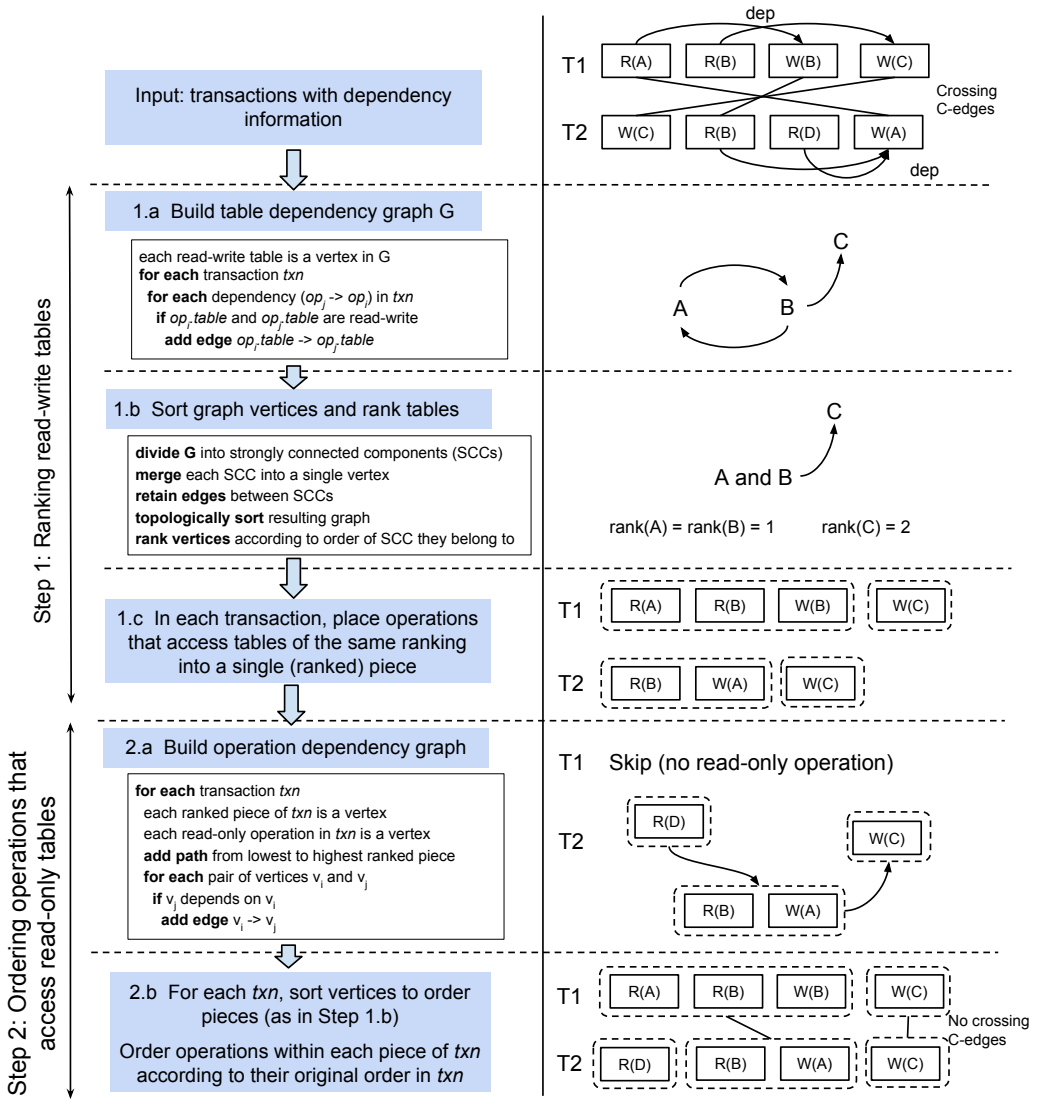


Figure 4.4: Pseudocode of Callas' transaction chopping algorithm (left) and its effects on a simple example (right).

Figure 4.3(a) illustrates the opportunity that Runtime Pipelining targets. Note how, as long as one can ensure that, during the execution, the top transaction accesses each table before the bottom one does, all C-edges acquire the same direction: all cycles are broken, and the transactions can be safely chopped in three pieces (Figure 4.3(b)). This finer chopping enables a form of pipelining: while the top transaction accesses the *item* table, the bottom one can concurrently access the *order* table and so forth.

The example suggests a way forward to safely extract greater concurrency from transaction chopping: rather than eliminating *all* SC-cycles, allow, intuitively, those where C-edges do not cross, since they can be neutralized at run time by controlling the order of execution of conflicting transaction pieces. To carry out this plan, we use a combination of static analysis and run-time mechanisms.

A new static analysis algorithm What prevents C-edges to cross and makes it safe to chop more aggressively in the example of Figure 4.3(b) is that both transactions access read-write tables in the same order. Generalizing from that example, assume that there exists a total ranking of each of the read-write tables accessed by the transactions in a group.[6] Then, the goal of our new static analysis algorithm is to produce choppings that satisfy the following two golden rules.

GR1: *Operations within a transaction piece are only allowed to access*

[6]If a table is read-only, accessing it does not create C-edges

read-write tables of the same rank (read-only tables, which by definition have no rank, can be also accessed).

GR2: *For any pair of pieces p_1 and p_2 of a given transaction that access read-write tables, if p_1 is executed before p_2, then p_1 must access tables of smaller rank than p_2* (as in GR1, read-only tables can be also accessed).

Then, by construction, the only C-edges that remain are those that connect pieces of different transactions that access tables of the same rank.

The two-step algorithm that achieves this goal and an example that illustrates its unfolding are shown in Figure 4.4. The first step totally ranks the read-write tables accessed by any of the transactions, and, within each transaction, groups together in a single piece all operations that access tables of the same rank. At the end of this step, the relative order of execution of the operations that, in each transaction, access read-write tables, is set. The second step then determines the execution order of the read-only operations of each transaction.

Step 1: Ranking read-write tables. Under the aggressive assumption that all operations in a transaction can be safely reordered, there is no constraint on the rank of read-write tables, and finding the finest chopping that does not have crossing C-edges is simple: we can assign a unique rank to each read-write table, sort operations in each transaction according to the rank of the table they access (operations that access read-only tables can be placed anywhere), and merge in the same piece those operations that access the same table. One

can easily prove this chopping satisfies our two golden rules.

In practice, however, there often exist data or control dependencies that compel the ordering of operations within a transaction and constrain the ranking of tables. In Figure 4.4, for example, R(A) must happen before W(B) in T_1, which forces $rank(A) \leq rank(B)$ (GR2), while for T_2, R(B) must happen before W(A), implying $rank(B) \leq rank(A)$. This means we must assign the same rank to tables A and B and merge all operations that touch them into a single piece.

Concretely, we achieve this result with the help of a *table-dependency graph*. The graph's nodes are read-write tables: we consider operations that access read-only tables in the next step. To add edges to the graph, we proceed as follows. For every transaction in the group, if there exists a data or control dependency between two operations op_1 and op_2 of the transaction, we add a directed edge between the tables they access (Figure 4.4, Step 1.a), indicating $rank(Table_{op_1}) \leq rank(Table_{op_2})$; we then assign the same rank to all the tables in the same strongly connected component, and assign ranks to all read-write tables according to their topological order in the resulting graph (Figure 4.4, Step 1.b). Within each transaction, operations that access tables with the same rank are merged into a single piece (Figure 4.4, Step 1.c).

Step 2: Ordering operations that access read-only tables. Transactions that access read-only tables contain operations that have not yet been ordered. To do so, we create a new graph for each transaction T, adding a vertex for each of the ranked pieces of T produced by Step 1 (such vertices acquire the rank

of their corresponding piece) and a vertex, with no assigned rank, for each operation of T that accesses read-only tables. To encode the outcome of Step 1, we create a path that connects ranked vertices, from the least to the highest ranked; in addition, we add a directed edge between two vertices if there exists a data or control flow dependency between them (Figure 4.4, Step 2.a). Next, we proceed as we did in Step 1.b of Figure 4.4: we evolve the graph so that each strongly connected component is represented as a single new vertex, joining in parallel the corresponding transaction pieces, and topologically sort the resulting graph to obtain the definitive order of execution of the pieces that comprise each transaction. Within each piece, operations are executed in the order in which they appeared in their transaction, prior to its chopping (Figure 4.4, Step 2.b).

Next we formally prove that the previous static analysis algorithm can generate a valid chopping that achieves our two golden rules.

Lemma 4.4.1. *Step 2 does not change the chopping generated in Step 1. Formally, for op_1 and op_2 that access read-write tables, 1) if op_1 and op_2 are placed in a single piece in Step 1, they will also be placed in a single piece in Step 2; 2) if op_1 and op_2 are placed in different pieces in Step 1, they will also be placed in different pieces in Step 2; 3) if op_1 is placed before op_2 in Step 1, op_1 will also be placed before op_2 in Step 2.*

Proof. 1) is true because Step 2 never breaks any pieces created by Step 1. 3) is true because in Step 2.a, when the algorithm constructs the graph,

it adds a path from lowest to highest ranked piece, ensuring that the order calculated by Step 1 is respected in Step 2.

We prove 2) by contradiction. Assume in Step 1, op_1 is placed in piece p_1, op_2 is placed in piece p_2, and p_1 is placed before p_2; in Step 2, op_1 and op_2 are placed in the same piece. This means in the graph generated by Step 2.a, p_1 and p_2 are in the same strongly connected component, and thus there must be a path from p_2 to p_1. This means some operation (op'_1) in p_1 must depend on some operation (op'_2) in p_2. In this case, in the graph generated by Step 1.a, there must be an edge from the table accessed by op'_2 to the table accessed by op'_1 and thus it is impossible for Step 1.b to place p_1 before p_2 and this contradicts our assumption.

Lemma 4.4.2. *The result chopping is valid. Formally, for any two operations op_1 and op_2 in a transaction, if there is a data or control dependency from op_1 to op_2, then in the result chopping, op_1 appears before op_2.*

Proof. First note that since there is a dependency from op_1 to op_2, op_1 must appear before op_2 in the original transaction. Then we prove by cases:

Case 1: both op_1 and op_2 access read-write tables. In this case, Step 1.a places a directed edge from the table T_1 accessed by op_1 to the table T_2 accessed by op_2 and orders them based on the topological order of the graph. There are two possible subcases:

Case 1.1. T_1 and T_2 are not in the same strongly connected component. Step 1.b gives T_1 a lower rank than T_2 in this case and thus Step 1.c places

op_1 before op_2. Step 2 will not change this result (Lemma 2) and thus finally op_1 will be placed before op_2.

Case 1.2. T_1 and T_2 are in the same strongly connected component. Step 1.b gives the same rank to T_1 and T_2 in this case and thus Step 1.c places op_1 and op_2 in the same piece. Step 2 will not change this result (Lemma 2) and Step 2.b orders op_1 and op_2 according to their original order in the transaction and thus places op_1 before op_2.

Case 2: at least one of op_1 and op_2 accesses a read-only table. In this case, Step 2.a places a directed edge from op_1 to op_2 and orders them based on the topological order of the graph. There are also two possible subcases:

Case 2.1: if op_1 and op_2 are not in the same strongly connected component, Step 2.a places op_1 before op_2.

Case 2.2: if op_1 and op_2 are in the same strongly connected component, Step 2.a places them in the same piece and Step 2.b orders them according to their original order in the transaction and thus places op_1 before op_2.

Lemma 4.4.3. *The resulting chopping satisfies GR1: operations within a transaction piece are only allowed to access read-write tables of the same rank.*

Proof. By construction, Step 1.c merges operations that access tables with the same rank into a piece. Therefore, the property holds at the end of Step 1. Step 2 does not change the chopping for operations that access read-write tables (Lemma 2). Therefore, the property still holds.

Lemma 4.4.4. *The resulting chopping satisfies GR2: for any pair of pieces p_1 and p_2 of a given transaction that access read-write tables, if p_1 is executed before p_2, then p_1 must access tables of smaller rank than p_2.*

Proof. By construction, Step 1.c sorts operations based on the rankings of tables they access. Therefore, the property holds at the end of Step 1. Step 2 does not change the chopping for operations that access read-write tables (Lemma 2). Therefore, the property still holds.

Theorem 4.4.5. *The static analysis algorithm generates a valid chopping that satisfies GR1 and GR2.*

Proof. Combining Lemmas 3, 4, and 5, we can prove this theorem.

Enforcing safety at run time Once static analysis produces choppings that satisfy our golden rules, neutralizing the remaining SC-cycles at run time is easy. Consider a piece of transaction T_i that accesses a table that involves a C-edge. If in so doing T_i becomes (anti-)dependent on some uncommitted transaction T_j that has already accessed that table, then that C-edge and every subsequent C-edge between T_i and T_j become (logically) directed: thenceforth, T_i cannot commit until T_j does, and every piece of T_i that accesses tables with ranking r must wait until T_j either has executed a piece that accesses tables with ranking at least r, or commits.

In practice, Runtime Pipelining is even more aggressive in pursuing opportunities for concurrency. It only declares a dependency between T_i and

T_j if they access the same row at run time (this is easy for T_i to verify in Callas by checking if T_j has acquired a nexus lock on the row). If not, Runtime Pipelining imposes no restrictions on execution ordering.

Although our discussion has focused on enforcing serializability, Runtime Pipelining can be easily applied to other notions of isolation by simply weakening the conditions under which it declares a dependency. For example, were Runtime Pipelining tuned to enforce read committed isolation, anti-dependencies would not trigger ordered execution.

We prove its correctness as following:

Theorem 4.4.6. *Runtime Pipelining prevents Intermediate Reads.*

Proof. If a transaction writes to the same table more than once, static analysis algorithm puts those operations into a single piece, thus it is impossible for another piece to observe states made by the earlier writes.

Theorem 4.4.7. *Runtime Pipeling prevents dependency cycles.*

Proof. We prove by contradiction. Suppose there exists a dependency cycle $T_1 \to T_2 \to ... \to T_n \to T_1$.

By construction, only pieces that access tables with the same rank can generate a dependency edge. Suppose r is the maximum rank of tables that are involved in any dependency edges in the cycle. We define p_i as the first piece of T_i that accesses a table with a rank not smaller than r, or the commit

operation if such a piece does not exist. We define t_i as the completion time of p_i.

First, we prove that any transaction T_k must have executed p_k when the cycle forms and $t_{k-1} < t_k$ if $T_{k-1} \rightarrow T_k$. Based on our assumption, there must exist at least one transaction T_i that has already executed a piece that accesses a table with rank r. Then let's consider T_{i-1}. There are two possible cases:

Case 1: The dependency edge $T_{i-1} \rightarrow T_i$ occurs on a table with rank smaller than r. In this case, T_i has already depended on T_{i-1} and thus Runtime Pipelining ensures that T_i cannot execute p_i until T_{i-1} completes p_{i-1}. Therefore, T_{i-1} must have executed p_{i-1} and $t_{i-1} < t_i$.

Case 2: The dependency edge $T_{i-1} \rightarrow T_i$ occurs on a table with rank r. In this case, p_{i-1} and p_i must both access tables with rank r. Since the dependency edge already occurs, T_{i-1} must have already finished p_{i-1} before T_i executes p_i. Therefore, T_{i-1} must have executed p_{i-1} and $t_{i-1} < t_i$.

By induction, any transaction T_k must have executed p_k and $t_{k-1} < t_k$.

Then by induction, we can get $t_1 < t_2 < ... < t_n < t_1$, which is impossible. Therefore, our assumption—a dependency cycle can form—is wrong.

Beyond rollback safety Runtime Pipelining takes an equally aggressive approach when it comes to avoiding the Aborted Reads and Atomicity violations that chopping introduces. Rather than settling for either the loss of concurrency or programming effort that rollback safety may cause, Runtime

Pipelining adopts an optimistic approach: it allows a transaction T_1 to read uncommitted states from T_2, but it does not allow T_1 to commit until T_2 commits. If T_2 is rolled back, then T_1 must also roll back.

While optimism pays off in finer chopping, no programming effort, and greater performance in groups when aborts and rollbacks are rare, it raises the possibility of performance loss in the presence of cascading rollbacks.

To avoid this danger, Runtime Pipelining takes two steps. First, it leverages MCC to prevent rollbacks from propagating outside of a group. Thus, misplaced optimism only affects performance in groups that are guilty of it. Second, it dynamically responds to an unexpected incidence of rollbacks by becoming increasingly more conservative. When the rollback rate crosses a threshold, Runtime Pipelining goes temporarily back to enforcing rollback safety; since we expect high rollback rates to be infrequent, however, it periodically tries to revert to its original optimistic approach.

The option of enforcing rollback safety on demand allows Runtime Pipelining to enjoy the full benefits of optimism when the rollback rate is reasonably low and avoid long-term damage from misplaced optimism.

Further, to ensure liveness in the face of rollbacks, Runtime Pipelining limits the depth of dependency chains composed of uncommitted transactions, and prevents a transaction that has been rolled back from performing uncommitted reads on retry.

4.5 Implementation

The current prototype of Callas is built upon the MySQL Cluster distributed database [11]. To implement Runtime Pipelining, we detect conflicts at the MySQL Cluster locking module and notify the transaction coordination module to enforce ordering between subtransactions, when necessary; to ensure isolation across groups, we modify the locking module of MySQL Cluster to support nexus locks and enforce their release order. Relying on MySQL Cluster, however, means that the current prototype of Callas must use the *read-committed* isolation level, the only one that MySQL Cluster supports.

To combine performance with simplicity, we developed tools that automate the process of grouping transactions and chopping them into subtransactions.

4.5.1 Automated chopping

The automated-chopping tool closely follows the Runtime Pipelining algorithm (§4.4) to statically analyze the transaction code and add markers to indicate the subtransaction boundaries to the run-time system, but introduces three additional optimizations: (i) it performs static analysis over columns rather than tables to produce finer choppings; (ii) it removes unnecessary C-edges; and (iii) it identifies better performing subtransaction orderings. We discuss the two latter optimizations in greater detail below.

Removing redundant C-edges Since commutative operations can be executed in any order without violating isolation, our tool, like Lynx [87], removes

C-edges between them.

Additionally, it searches for instances of *runtime uniqueness*, where multiple transaction instances modify the same table, but each is guaranteed to operate on a different row. For example, in TPC-C, the *new_order* transaction acquires a unique order ID by incrementing a *nextOrderID* object, and then proceeds to modify the corresponding row. Such opportunities are identifiable by searching for "monotonic" objects, i.e., objects, such as counters, that all transactions modify monotonically before using them as a key in a query. Runtime uniqueness is yet another example of an optimization whose effectiveness can be magnified by the modularity of MCC, since runtime uniqueness is less likely to hold in large groups of transactions. In TPC-C, for example, uniqueness does not hold globally, as other transactions (e.g., *delivery*) do not use *nextOrderID* and may therefore access the same row as *new_order*.

Identifying more performant orderings Given a transaction T, any topological order of the pieces of T produced by Step 2.b of the algorithm in Figure 4.4 yields a safe way to execute T. We then have some freedom in choosing the order in which T's pieces should execute. We leverage this freedom by having the larger pieces—classified heuristically by the number of queries they contain—execute as early as possible. The rationale behind this optimization is that Runtime Pipelining only enforces ordering between transactions once a dependency manifests at run time. By executing large subtransactions early, we decrease the chance that they will be subject to ordering, thus increasing parallelism.

4.5.2 Automated grouping

The goal of our grouping tool is to identify groups of transactions that contend heavily with each other. The user need only provide her workload of choice; the tool analyzes the performance of the workload using various groupings and returns the grouping that yields the best performance. Our current tool does not explore all possible groupings, but rather uses heuristics to identify groupings that are more likely to increase concurrency. Our evaluation suggests that this heuristic approach is enough to provide significant performance benefits (e.g., 8.2x speedup for TPC-C).

The tool works in iterations. In each iteration it runs the workload and creates a profile for this iteration's performance measurements. Based on these measurements, it tries to identify the most prominent source of contention and suggests a grouping that could alleviate it. It then runs our chopping tool on this grouping, and proceeds to measure the performance of this new configuration in the next iteration. This process terminates if an iteration does not yield any performance improvement.

To identify sources of contention, we use as a hint the latency of individual operations. As the load on the system grows, the latency of highly contending operations tends to increase disproportionately. The corresponding transactions are then our primary candidates for optimization. If there are only few such transactions, our tool enumerates all possible groupings; otherwise, it focuses on those that hold locks on contended items for long intervals.

4.6 Evaluation

The goal of Callas is to provide unmodified database applications with the level of performance that was previously only achievable by manually modifying all or part of the application code. To assess whether Callas achieves this goal, we evaluate the performance of Callas using various applications and workloads. In particular, our evaluation answers the following questions:

- What is the performance gain of Callas over a traditional ACID database? (§4.6.1)
- How does the performance of Callas compare against that of other approaches that aim to improve database throughput? (§4.6.1)
- How do various optimizations, groupings, and workload parameters affect the performance of Callas? (§4.6.2, §4.6.3, §4.6.5)
- What is the overhead of nexus locks? (§4.6.4)
- As the rate of rollbacks changes, how effective is it to optimistically renounce rollback safety to extract performance? (§4.6.6)

We answer these questions by measuring the performance of Callas using microbenchmarks and three applications: TPC-C [36], Fusion Ticket [8], and Front Accounting [7].

TPC-C is a database benchmark that models online transaction processing. It contains three highly-contending read-write transactions and two read-only transactions.

Fusion Ticket is an open source software solution for online ticketing and advanced sales. To perform a fair comparison with Salt [85], we run the same workload used in that paper, which includes several transactions critical to the performance and functionality of an online shop.

Front Accounting is an open source accounting and Enterprise Resource Planning (ERP) program. It allows a company to manage its sales, purchases, and stock levels. Our workload includes 17 transactions that simulate the workload of a retail company: the company purchases goods from suppliers at a low price and sells them to customers at a higher price. It includes five read-write transactions: *create-order*, *payment*, *delivery*, *pay-supplier*, and *stock-adjustment*, and 12 read-only transactions to query order information.

Experimental setup In TPC-C, we populate ten warehouses, and assign each warehouse to a separate partition. Our Fusion Ticket setup mirrors that of Salt: there is one event and two categories of tickets, with 10,000 seats in each category. Finally, in Front Accounting, we configure the retail company to operate on 100 different types of goods, and on average to make a bulk purchase for every 1,000 sale orders.

For all experiments, we use ten database partitions, each of which is three-way replicated. All our throughput numbers were calculated while the system is saturated.

Our experiments are carried out on Dell PowerEdge R320 machines in CloudLab [3]. Each machine is equipped with a Xeon E5-2450 processor,

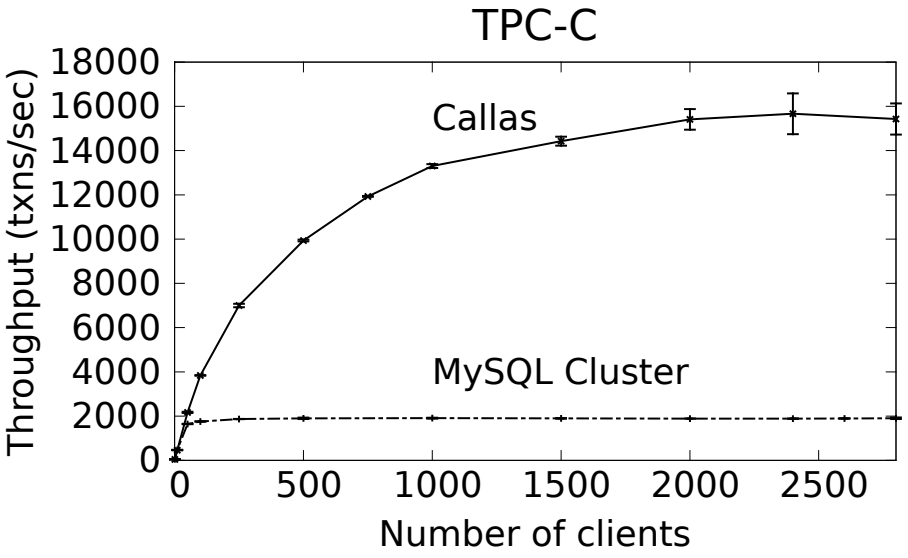


Figure 4.5: Throughput of TPC-C

16 GB of memory, four 7200 RPM SATA disks, and 1 Gb Ethernet.

4.6.1 Callas' performance

Our first set of experiments uses TPC-C, Fusion Ticket, and Front Accounting to compare the throughput of Callas to that of MySQL Cluster—the system from which Callas descends.

As shown in Figure 4.5, the performance of Callas on the TPC-C benchmark is about 8.2x higher than that of the original MySQL Cluster.

Callas' performance improvement is partly due to the ability of the automated grouping tool to identify highly contending transactions and group them accordingly. In this case, the tool placed the *new_order* and *payment* transactions in one group; the *delivery* transaction in a second group; and the

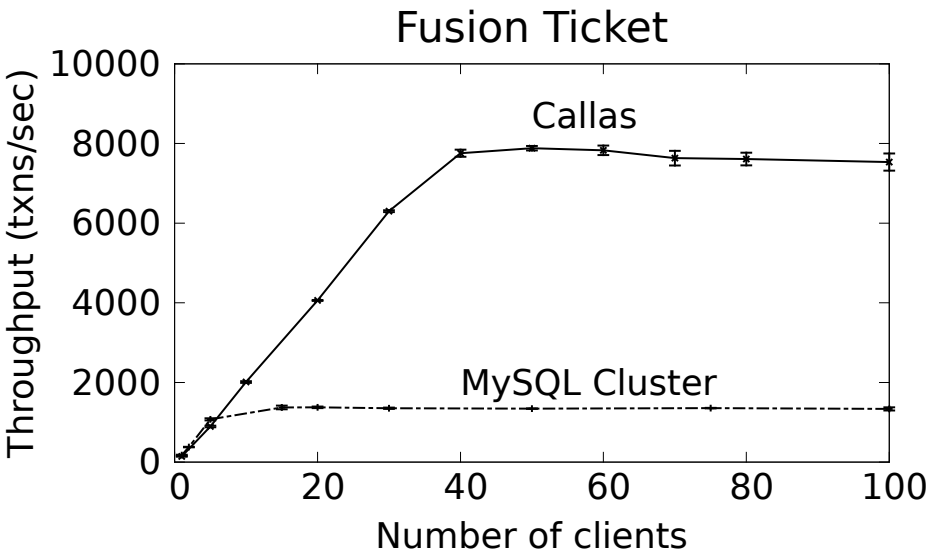


Figure 4.6: Throughput of Fusion Ticket

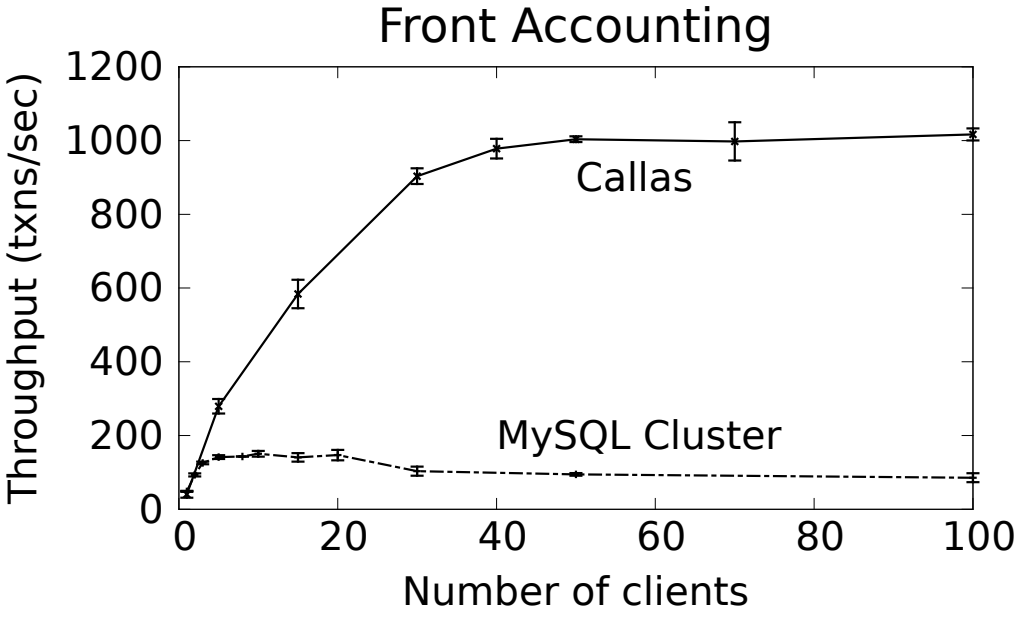


Figure 4.7: Throughput of Front Accounting

remaining transactions in a third group. This grouping reflects the contention pattern of these transactions: *new_order* and *payment* contend heavily for the *warehouse* and *district* tables, but Runtime Pipelining is effective in allowing them to release their locks early, after acquiring a unique ID. Moreover, although both *new_order* and *payment* update the *district* table, they update different columns, which allows our static analysis to remove the C-edge between them. This makes them ideal candidates for belonging to the same group: they contend for the same row lock, but do not have any C-edges between them, and therefore can be grouped together without introducing any SC-cycles. Callas uses Runtime Pipelining to chop transactions in the first two groups, while the third group is left unoptimized, as it does not contain performance-critical transactions.

Figure 4.6 shows the performance of Callas and MySQL Cluster for the Fusion Ticket application. Callas outperforms MySQL Cluster by a factor of 5.7x. For this application, our tool generates two groups: the first contains the *checkout* transaction and uses Runtime Pipelining for chopping, while the second contains the remaining transactions and is left unoptimized.

As shown in Figure 4.7, when running the Front Accounting application, Callas outperforms MySQL Cluster by a factor of 6.7x. For this application, our tool generated four groups: transactions *create-order*, *delivery*, and *payment* are each placed in their own groups and use Runtime Pipelining for chopping, while the rest of the transactions are placed in a fourth, unoptimized, group.

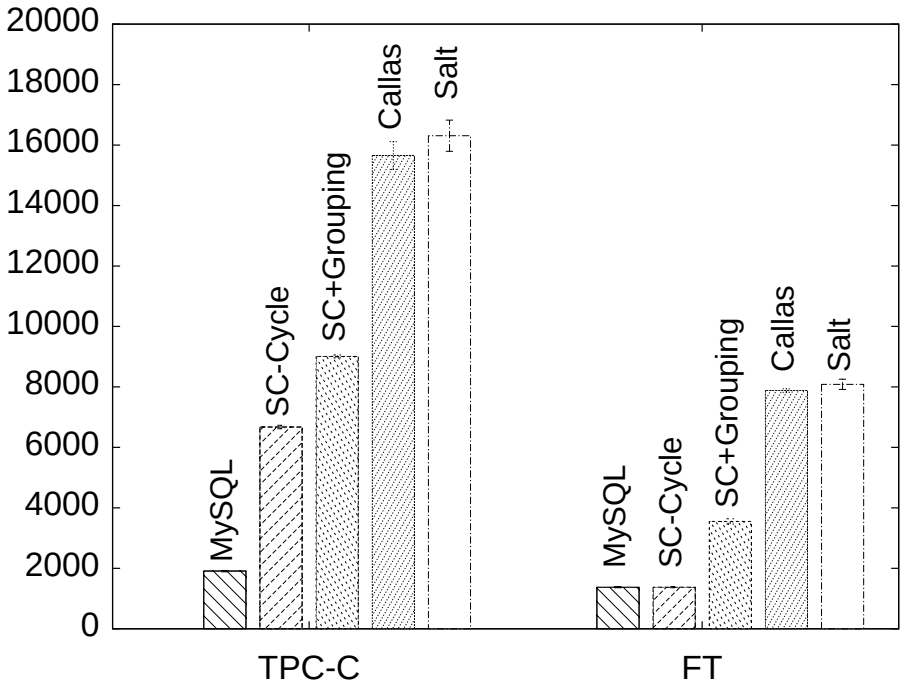


Figure 4.8: Effect of different techniques

Comparison with other techniques The next experiment compares the performance of Callas for TPC-C and Fusion Ticket[7] with that of other techniques. We first consider applying the SC-cycle static analysis of traditional transaction chopping to the entire application (i.e., without grouping). This step boosts the performance of TPC-C to 3.5x of the MySQL baseline, but does not help Fusion Ticket at all, since traditional transaction chopping cannot safely chop the performance-critical transaction of Fusion Ticket. Next, we combine traditional transaction chopping with our grouping mechanism:

[7]These were the applications used to evaluate Salt [85].

Latency(ms)	MySQL		Callas	
	Quantile		Quantile	
	50th	99th	50th	99th
new_order (TPC-C)	26	51	28	50.5
checkout (FT)	12	25.3	12	25
delivery (FA)	36.3	69	36.6	66

Table 4.1: Latency under low throughput.

we split transactions into groups and use SC-cycle analysis to chop transactions within each group. This approach further improves the throughput of TPC-C by 35%, and raises the throughput of Fusion Ticket to 2.6x of the baseline. Callas, using Runtime Pipelining instead of standard SC-cycle analysis, achieves a further 74% and 120% throughput boost, respectively. Remarkably, the performance of Callas is within 5% of that of Salt [85]. We find it encouraging that, despite staying true to the ACID paradigm, Callas can achieve performance similar to approaches that require manual modification of the application code.

Latency Table 4.1 presents the request latency for our three applications, when the system is under low load. In all cases, the latency of Callas is similar to that of the unmodified MySQL Cluster.

4.6.2 Performance impact of various optimizations

Figure 4.9 breaks down the contribution of each optimization to the performance of Callas (using the grouping produced by our heuristic algorithm) for TPC-C and Fusion Ticket.

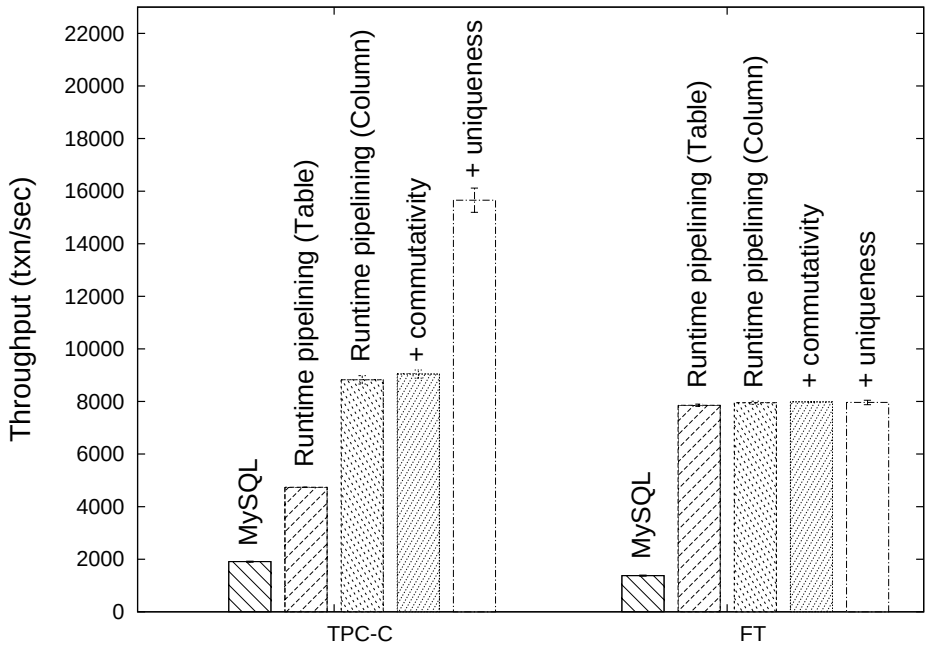


Figure 4.9: Effect of different optimizations.

The effectiveness of the different optimizations is application-dependent. In Fusion Ticket, Runtime Pipelining alone, even naively applied at the granularity of tables, is enough to achieve almost all of Callas' performance improvement. Not so in TPC-C, where Callas gets a significant performance boost from performing static analysis at the column—rather than the table—level by identifying several columns that are accessed in read-only mode (even as the table they belong to is accessed in read-write mode). This allows Callas to remove conflict edges between transactions and achieve finer-grained chopping. Leveraging commutativity yields only a minor performance improvement

in TPC-C, because it only applies to a few individual statements. Runtime uniqueness instead provides another big boost in the performance of TPC-C by removing several critical conflict edges in the *new_order* transaction, leading to finer-grained chopping.

Note that, although it does not explicitly appear in Figure 4.9, MCC is essential to Callas' performance gains, because many of its optimizations would simply not be applicable without MCC. Runtime uniqueness, for example, could not be leveraged in TPC-C, since it does not hold for all TPC-C's transactions; and Runtime Pipelining itself would prove virtually ineffective if applied across the *entire* set of Fusion Ticket's often-complex transactions.

4.6.3 Performance impact of different groupings

To demonstrate the importance of grouping transactions appropriately, we measure Callas' throughput when running TPC-C using different transaction groupings. TPC-C has five transactions: three are read-write transactions and two are read-only. We first compare our heuristic grouping to two naive groupings. The first puts all transactions in a single group, while the second puts each of the five transactions in a separate group. Our heuristic grouping—the result of running the heuristic algorithm of Section 4.5.2—consists of three groups: *new_order* and *payment* are in one group, *delivery* is in a second, and the two read-only transactions in a third.

Figure 4.10 shows the results of this experiment. Even with all transactions in the same group, Runtime Pipelining yields a significant performance

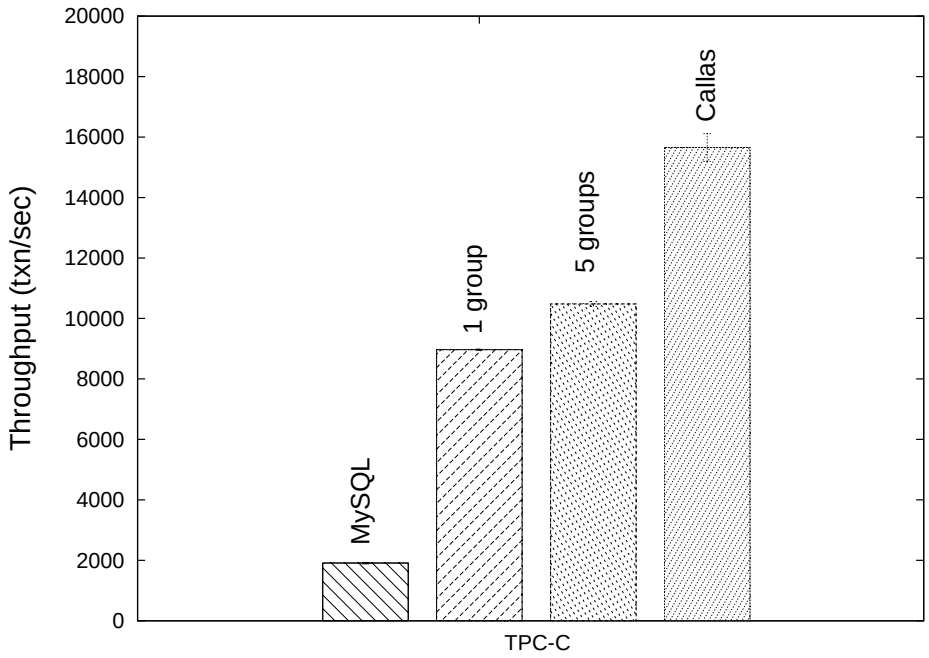


Figure 4.10: Effect of choosing different groupings.

benefit compared to a traditional ACID implementation. Placing each transaction in a separate group further improves performance by easing the bottleneck caused by contending read-write transactions. In particular, Runtime Pipelining can now apply optimizations, such as runtime uniqueness, that were not applicable when all three read-write transactions were grouped together.

Having each transaction in a separate group, however, is not ideal. Since *new-order* and *payment* conflict frequently, it is preferable to place them in the same group so that their conflicts can be regulated using a custom in-group mechanism, instead of the coarse inter-group locks. Indeed, the grouping

returned by our heuristic algorithm outperforms this grouping by at least 50%.

To get a sense of how the grouping produced by our algorithm compares to an optimal grouping, we iterated over all the possible grouping strategies: we found that, at least for TPC-C, no other grouping achieved a higher throughput.

4.6.4 Overhead of nexus locks

Two factors contribute to the overhead of nexus locks: the cost of maintaining an additional lock and that of correctly enforcing the Nexus Lock Release Order rule (§4.3). The latter cost is only incurred when transactions conflict, while the former is always present. To measure separately their effect on throughput, we designed two microbenchmarks, one with no contention and one with high contention.

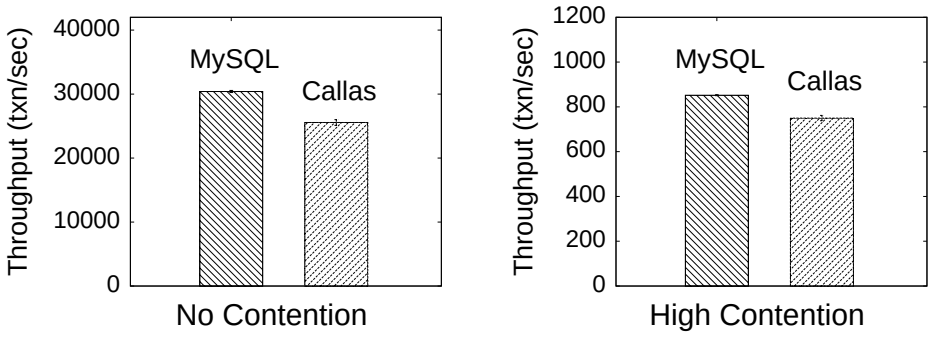


Figure 4.11: Overhead of nexus locks

To eliminate any benefit that may come from using Callas, we run both microbenchmarks with each transaction instance in a separate group,

and enforce isolation within each group using the default MySQL Cluster locking mechanism. We run both experiments with two shards, each three-way replicated. In the no-contention experiment, each transaction has exclusive access to five rows. In the high-contention experiment, all transactions touch the same two rows, with each row in one shard.

As shown in Figure 4.11, in the no-contention experiment, MySQL Cluster outperforms Callas by about 19%. Our profiling shows the bottleneck lies in the additional demands on the CPU to acquire, maintain, and release nexus locks. In the high-contention experiment, the throughput of MySQL Cluster is about 13.6% higher than that of Callas.

The CPU overhead of nexus locks is of course still there, but it becomes relatively less prominent because contention increases the execution time of transactions. Instead, the additional message exchanges Callas requires to enforce the Nexus Lock Release Order rule become the dominant factor.

4.6.5 Effect of contention rate on performance

The design of Callas focuses on optimizing in-group contention while being conservative about contention between transactions in different groups. While our experience with real applications suggests that it is typically possible to partition transactions, so that inter-group contention is minimized, we would like to understand how robust the performance of Callas is to increased levels of inter-group contention. We design two microbenchmarks, each exploring a different factor of inter-group contention: execution frequency and

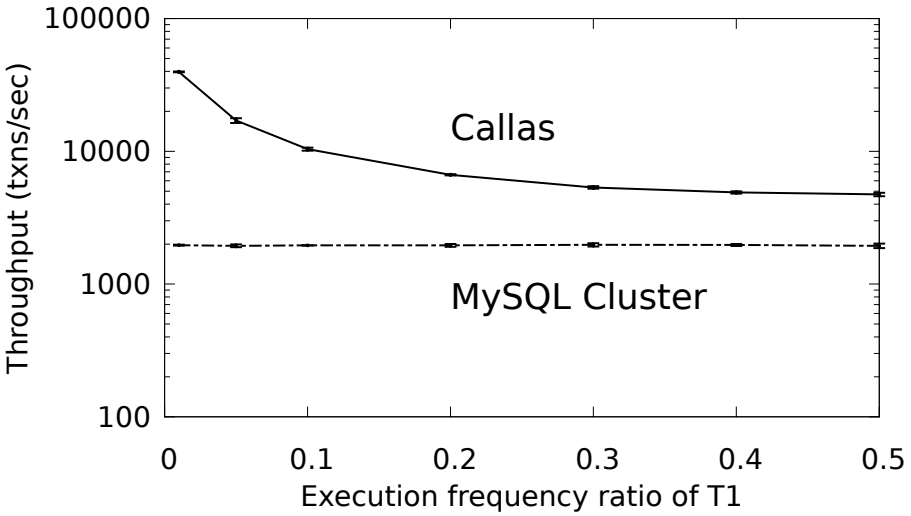


Figure 4.12: Effect of execution frequency on performance.

contention rate. Both microbenchmarks start by executing operations that cause conflicts (across groups or inside a group) and end with a sequence of operations that cause no conflicts. Unlike MySQL Cluster, Callas can chop contending and non-contending operations in separate pieces and release contending locks early: hence, the longer the sequence of non-conflicting operations at the end of a transaction, the greater the performance benefits that Callas can bring.

To separate sufficiently the performance of Callas and that of MySQL Cluster, in order for us to study the effects of inter-group contention on the former, both our microbenchmarks use a sequence of five non-conflicting operations.

The first microbenchmark explores the performance repercussions of

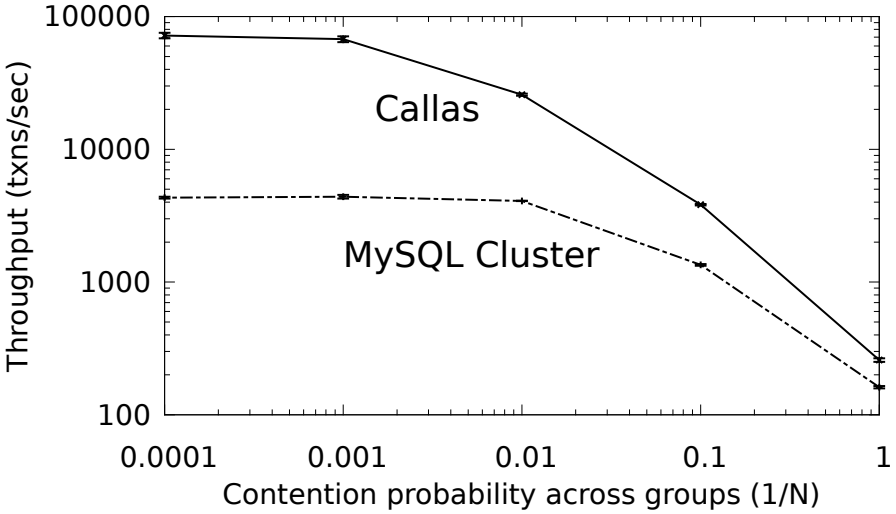


Figure 4.13: Effect of contention probability across groups.

having two frequently executing transactions in different groups. The microbenchmark includes two types of transactions, T_1 and T_2. Each transaction contains six $(1+5)$ operations: the first operation updates one row, randomly chosen out of ten rows, thus fixing the contention rate between T_1 and T_2 at 10%. The remaining five operations update non-conflicting rows (i.e., rows that are private to each transaction instance). We place T_1 and T_2 in separate groups that use Runtime Pipelining for chopping, and we tune the relative execution frequency of these two transactions.

The second microbenchmark explores the effect of inter-group contention rate on the performance of Callas. This microbenchmark is similar to the first: we use two types of transactions, T_1 and T_2, each in its own group, only this time they have the same execution frequency. Each trans-

action contains seven $(2 + 5)$ operations: the first operation updates one row at random, chosen out of N rows, where N is a parameter that controls the inter-group contention rate. The second operation modifies a random row, chosen out of ten rows, from a table private to each transaction type, thus introducing a 10% in-group contention rate. The remaining five operations update non-conflicting rows (private to each transaction instance).

Figures 4.12 and 4.13 show the results for these two microbenchmarks. Both experiments show the same trend: when the inter-group contention is low, the performance of Callas far exceeds that of a traditional ACID database. For example, when the execution frequency ratio of T_1 to T_2 is 100:1 (Figure 4.12), the throughput of Callas is 20.2x that of MySQL Cluster. Similarly, when T_1 and T_2 have a low conflict rate of 0.01% (Figure 4.13), the throughput of Callas is 16.6x that of MySQL.

As the inter-group contention rate increases—because both transactions run frequently or contend heavily— the benefit of Callas decreases. This is to be expected, as Callas is effectively attempting to regulate heavy contention using traditional locking.

Note, however, that even when the inter-group contention is as high as the in-group contention, the performance benefit of Callas is still substantial. In Figure 4.12, even when T_1 and T_2 are executed with the same frequency, Callas' throughput is more than twice that of MySQL's; in Figure 4.13, when N=1 (i.e., contention is at 100%), Callas achieves a 60% throughput gain. The reason behind this performance increase is that, even when the workload

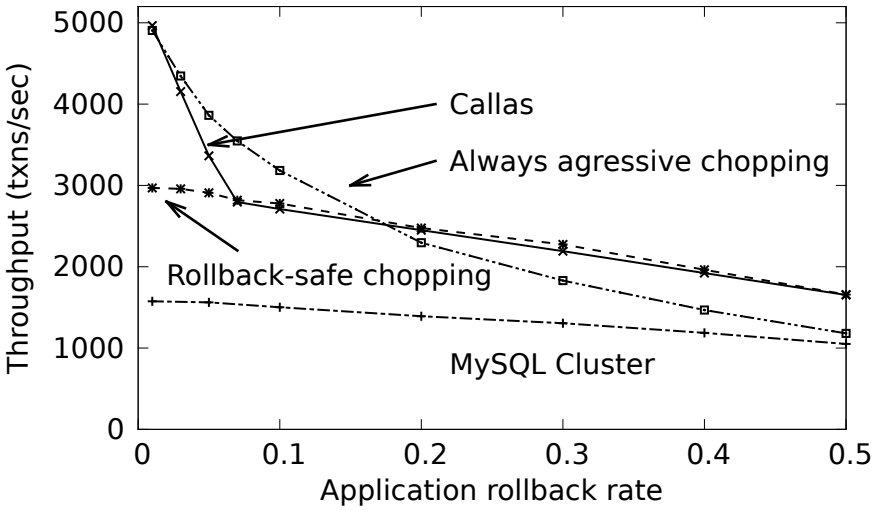


Figure 4.14: Effect of application rollback rate on performance.

is uniform (e.g., both transactions have the same frequency), this does not mean that a T_1 is always followed by a T_2 (and vice versa). As long as two T_1s (or T_2s) are executed consecutively, Runtime Pipelining can optimize their execution. Interestingly, increasing in-group concurrency implicitly increases inter-group concurrency as well, since transactions hold their locks for shorter times.

4.6.6 Beyond rollback safety

Our final set of experiments measure the performance of Runtime Pipelining's adaptive approach for preventing Aborted Reads and Atomicity violations.

We design a microbenchmark that can trigger cascading rollbacks in a

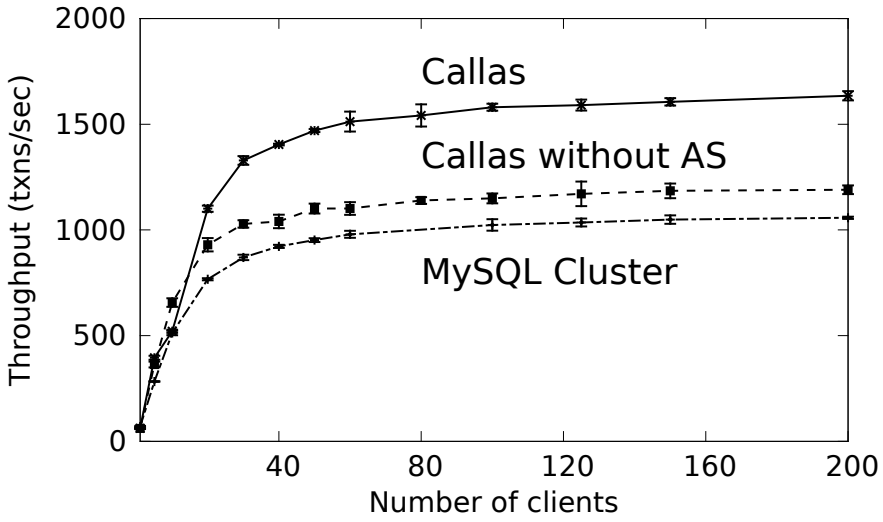


Figure 4.15: Effect of Callas adaptive response to high rollback rates.

controlled manner. It uses three tables—A, B, and C—with ten rows each, and contains one transaction with 11 operations. The first operation picks a number i at random between 1 and 10, and checks a condition on the i^{th} row in table A and the second operation updates that row if the check succeeds. The third and fourth operation do the same for the i^{th} row of table B; and the fifth and sixth do the same for the i^{th} row of table C. The last five operations update non-conflicting rows (private to that transaction instance). Runtime Pipelining splits this transaction into the following eight subtransactions: $\langle 1, 2\rangle$ $\langle 3, 4\rangle$ $\langle 5, 6\rangle$ $\langle 7\rangle$ $\langle 8\rangle$ $\langle 9\rangle$ $\langle 10\rangle$ $\langle 11\rangle$, the first subtransaction containing the first two operations, etc. The check of the fifth operation has a probability to fail and cause a rollback, triggering a cascading rollback if other transactions already depend on this transaction.

In our first experiment, we tune the probability of the third subtransaction triggering a rollback. As shown in Figure 4.14, the throughput of both Callas and MySQL Cluster decreases as the rollback rate increases. The throughput of Callas is always higher than that of MySQL Cluster, but the improvement decreases from 2.9x to 60%. When the rollback rate is low, Callas can execute all eight subtransactions in a pipeline, but when the rollback probability increases, Callas' adaptive control mechanism falls back to "safe mode" by merging the first three subtransactions—thus placing the rollback statement in the first subtransaction. Even in safe mode, however, Callas can still parallelize the execution of the last five subtransactions. In Figure 4.14, the switch to safe mode happens when the rollback rate is higher than 7%. For reference, we also measured the throughput of Callas with safe mode always on, and with safe mode always off. In the former case we lose parallelism when the rollback rate is low, whereas in the latter we incur significant overhead when the rollback rate is high. Thanks to Runtime Pipelining's adaptive mechanism, Callas comes close to the best of both worlds.

In practice, since real applications have low rollback rates most of the time, we expect safe mode to be triggered only infrequently; the rest of the time Callas would still be aggressively optimizing transactions.

Figure 4.15 takes a closer look at the performance of Callas under stress. We fix the rollback rate to a high value (50%) and increase the load of the system until we reach saturation. We observe that under low load, it is not critical for Runtime Pipelining to adaptively fall back to safe mode; in

fact, an always-aggressive version of Callas performs slightly better. As the load—and, hence, parallelism—increases, however, the adaptivity of Runtime Pipelining prevents cascading rollbacks from causing a performance collapse, while allowing Callas to continue leveraging some parallelism.

4.7 Conclusion

Separating concerns and decoupling abstraction from mechanism are basic tenets of sound system design—and for good reasons. We confirm their benefits yet again, by applying them to the long-standing problem of improving the performance of ACID applications. The flexibility of the modular concurrency control architecture at the core of Callas allows the applications we have tested, to obtain, unmodified, the kind of performance previously achievable only by manually rewriting all or part of the applications' code.

Chapter 5

Related Work

Many database systems [9–11, 14–16, 67] provide ACID guarantees to greatly simplify applications' development. On the other hand, in order to achieve higher performance and availability, other systems [1, 17, 33, 34, 42, 61] give up the ACID paradigm, and instead, adopt the BASE approach [32]. Several efforts have tried to relieve the tension between ease of programming and performance by finding alternatives to the ACID/BASE duality. We review them in this chapter.

5.1 Optimizing ACID transactions

Frustrated with ACID performance, researchers have tried different approaches to improve the performance of ACID databases.

5.1.1 Optimizing certain transaction types

A popular approach is to optimize certain types of transactions. By so doing, systems can leverage the characteristics of these transactions to find more efficient mechanisms. Read-only transactions is a target. These transactions are quite common in applications, and, because they do not include write

operations, they need not to pay the overhead of the mechanisms required to handle general transactions. Multiversion two-phase locking [29], for example, avoids conflicts between read-only transactions and read-write transactions by allowing read-only transactions to read committed versions consistently according to a timestamp assigned at the beginning of the execution. Spanner [35], for another example, can avoid the two-phase commit(2PC) protocol for read-only transactions, assuming clocks are well synchronized.

Other systems have explored ways to optimize other types of transactions. H-Store [77], for example, optimizes transactions that contain only queries that can be executed on just one site in the cluster. H-Store leverages this feature to avoid unnecessary network communication across different sites and improve its performance. H-Store also optimizes one-shot transactions, in which, though different queries may touch different sites, each individual query executes on just one site. Besides transactions that operate on a single site, Granola [37] also optimizes *independent* distributed transactions, i.e., transactions where each site can reach the same decision even without communication. For these transactions, Granola can eliminate the 2PC protocol.

Though these optimizations are helpful for specific types of transactions, they can't, unlike Salt and Callas, provide more opportunities for concurrency to generic transactions.

5.1.2 Optimizing under certain workload conditions

Another approach to improving ACID performance is to optimize transactions when certain conditions hold for the workload. For example, Sagas [50] lets developers chop long-running transactions into pieces when such chopping does not affect the application semantics. Transaction chopping [74] and Lynx [87] use SC-cycles discussed in Section 4.4 to identify transactions eligible for chopping.

Like Callas, Lynx observes that executing transaction pieces in a well-defined order can avoid conflicts: its *origin ordering* technique ensures that if two transactions T_1 and T_2 start on the same server, and T_1 starts before T_2, then, to guarantee safety, T_1 pessimistically executes before T_2 at every server where they both execute. However, since it is hard in practice to anticipate the specific servers where user transactions will execute, origin ordering can only prevent conflicts among the predictable internal transactions used for updating secondary indexes and joint tables. In contrast, Callas' Runtime Pipelining is widely applicable, since it relies on information (the order in which transactions access tables) that can be easily established through static analysis, and only enforces ordering if it detects an actual conflict at run time, leaving significantly greater opportunities for concurrency.

Rococo [66] relaxes Lynx's eligibility condition by reordering transaction operations and applying additional run-time mechanisms. Calvin [78] avoids using 2PC by predefining an execution schedule for transactions, but

again under the assumption that the system can predict which server a transaction will access when it is executed.

In general, such optimizations have the potential to yield significant performance improvements, but the assumptions on which they rely are usually hard to satisfy in real applications. Modular Concurrency Control can help increase the applicability of these techniques by requiring those assumptions to only hold within each group, rather than globally.

5.1.3 Leveraging new hardware

The development of new hardware also provides new opportunities to improve the performance of ACID transactions. These improvements are orthogonal to the work presented in this dissertation, which focuses on reducing the cost of handling contention. Indeed, the techniques used in the system surveyed below could be leveraged in combination with Salt and Callas.

Remote Direct Memory Access (RDMA) is a hardware feature that allows users to access other machines in the same cluster with low latency and CPU cost. It gives developers a chance to reduce the messaging overhead of distributed transactions. For example, DrTM[81] first uses restricted transactional memory (RTM), a hardware feature provided by Intel that allows users to execute a group of memory accesses in an atomic way, to handle local transactions. Then, leveraging RDMA, it extends RTM to handle distributed transactions efficiently. Combining these two new hardware features, DrTM can achieve higher performance and scalability. There are, however, limita-

tions. First, DrTM must know the read/write set and the memory transactions touch before the execution to properly implement its locking protocol. Second, because RDMA makes it hard to maintain replicas consistent, DrTM forgoes replications, thus reducing the availability of the system. Similarly, FaRM [43] uses RDMA to reduce the overhead of handling distributed transactions. In addition, to efficiently support durability, FaRM takes advantage of non-volatile DRAM. However, due to their optimistic approach, both DrTM and FaRM cannot handle well workloads with high contention. In contrast, Salt and Callas can bring significant performance improvement for high-contention workloads.

Given the trend of having more cores and larger memory in one machine, Silo[79], a single-node database, aims to fully use the resources provided by a single machine. To do it, Silo designs a new protocol to reduce the local conflicts. It demonstrates that a machine with many cores and large memory can actually achieve comparable performance compared to a cluster.

5.2 Providing limited transaction

A further way for applications to balance ease of programming and performance is to limit the scope of transactional guarantees. For example, ElasTraS [38], MegaStore [23], G-Store [39], and Microsoft's Cloud SQL Server [26], only provide ACID transactions within a single partition or key group. G-Store and ElasTraS further allow dynamic modification of such key groups, so that users can change data partition at runtime to support more transactions.

When compared to the pure BASE approach, partition-local transactions can simplify the applications; at the same time, avoiding distributed transactions can increase their performance. Unfortunately, it is not always easy to partition data, so that applications can simply avoid distributed transactions. For requests that touch multiple partitions, these systems rely on the developers to ensure correctness, which is tedious and error-prone [35, 75].

With a different take to address the same challenges, Sinfonia [20] introduces minitransactions. In minitransactions, all actions that the coordinator requires a site to perform are required to not depend on any actions that are executed on other sites. Therefore, it is possible to piggyback the entire set of actions that the coordinator requires a site to perform into the first phase of 2PC. As a result, Sinfonia can start, execute, and commit a minitransaction with two network round-trips, significantly reducing the overhead of distributed transactions. As the authors of Sinfonia point out, however, minitransactions do not easily fit general applications, since the requirements that minitransactions depend upon are not easy to satisfy.

5.3 Related Techniques

Some of the techniques used by Salt and Callas are similar to those used in other systems. In this section, we discuss these techniques and the unique ways in which Salt and Callas use them to resolve the tension between ease of programming and performance.

5.3.1 Combining Different Concurrency Control Mechanisms

Researchers have explored several approaches to apply different concurrency control mechanisms in the same system. For example, transactional federated database systems [31, 41, 51, 70, 72] support *global transactions* that touch different database systems. Each database in the federated system can use its own concurrency control mechanisms. Federated systems then guarantee the serializability for these global transactions. Unlike Callas, the performance of federated systems is typically worse than that of the database systems since they focus on achieving more functionality (such as supporting global transactions) rather than greater performance.

Local atomicity properties [82] have been proposed as a way to enable developers to enforce serializability of transactions by enforcing serializability at each shared data object. This work makes it possible to combine different concurrency control mechanisms that satisfy the same atomicity property. By encapsulating the synchronization in the implementations of shared objects, the system can improve modularity, and like Callas, increases the chance of finding more efficient mechanisms specialized for each data object. This approach, however, does not allow developers to combine arbitrary concurrency control mechanisms. For example, using two-phase locking and multi-version concurrency control for different shared data objects may violate the requirements of cross-object serializability. Further, this approach still requires *all* conflicts that involve a given data object to be handled using the same concurrency control mechanism. Callas, instead, allows different conflicts on the

same data item to be regulated by different concurrency control mechanisms, creating opportunities for greater concurrency.

Bernstein introduces mixed concurrency control mechanisms, in which write-write conflicts and read-write conflicts can be handled by different concurrency control mechanisms [27]. This flexibility can potentially achieve higher concurrency. However, the conflicts in this model can be only divided into two types. The same type of conflicts, such as all write-write conflicts, have to use the same concurrency control mechanism. In addition, this approach only allows specific combinations, such as *multiversion two-phase commit*. Instead, MCC allows us to combine arbitrary concurrency control mechanisms.

5.3.2 Group Mutual Exclusion

The goal of Callas' cross-group mechanism is to prevent dependency cycles that span multiple groups. To accomplish it, the cross-group mechanism enforces two properties. First, Nexus locks allow transactions from the same group to access the same data object concurrently while preventing transactions from different groups from doing so. Second, the cross-group mechanism enforces *Nexus Lock Release Order*, the order in which the locks can be released. Nexus locks solve the problem of *group mutual exclusion*, first formalized by Joung [58]. Group mutual exclusion is a generalization of the mutual exclusion problem. Similar to Nexus locks, group mutual exclusion allows different processes from the same group to enter the critical section

concurrently while preventing processes that belong to different groups from doing so. Note however that group mutual exclusion is only one of the two properties that Callas' cross-group mechanism enforces: *Nexus Lock Release Order* is key to preventing circularity, while the group mutual exclusion specification puts no requirement on the order in which processes must leave the critical section.

5.3.3 Transaction Group

To reduce the overhead of determining whether two transactions may conflict at run time, SDD-1 [30] introduces the notion of *transaction classes*, which bear an intriguing but ultimately passing similarity to Callas' *transaction groups*. In SDD-1, each transaction class is defined statically by the database administrator, and it is formally identified by a logical read and write set. A transaction T fits in any class whose read and write set are a superset of the corresponding sets for T: the class to which a specific instance of T is actually assigned is not decided until run time. SSD-1 simplifies concurrency control by first using static analysis to identify conflicts within classes (rather than transactions), and by then leveraging the observation that transactions that are assigned at run time to different classes can conflict only if their classes conflict. In Callas, transaction groups are instead the key mechanism that enables the separation of concerns that is at the core of MCC. By delimiting the scope of each in-group concurrency control mechanism, they allow them to aggressively seek opportunities for greater concurrency.

5.3.4 Nested Transactions

Nested transactions, transactions that consisting of a sequence of subtransactions, were first proposed by Davies under the name *spheres of control* [40]. Typically, nested transactions continue to provide isolation at the granularity of the whole transaction; however, each subtransaction can be aborted independently. Therefore, if one subtransaction fails, the transaction can simply restart from the last committed subtransaction. Though BASE transactions are structured as nested transactions, they use their inner structure to address different issues. Nested transactions tune the granularity of atomicity so that errors occurring within a nested transaction do not require undoing the entire parent transaction, but only the affected subtransaction. BASE transactions, instead, tune the granularity of isolation. They use subtransactions to define the states that could be exposed among BASE transactions to increase concurrency.

Several other works use the structure of nested transaction to increase concurrency. [46, 49, 63]. These works weaken the usual notion of serializability by permitting additional user-defined interleavings. However, as in the BASE approach, when defining these interleavings, users need to reason about the consistency constrains that applications require by considering all transactions. Salt, instead, uses BASE transactions only for performance critical transactions, and guarantees that the remaining ACID transactions are not affected, limiting complexity. In addition, since BASE transactions are accepted as soon as their first alkaline subtransaction commits and will eventually com-

mits, BASE transactions also improve availability of applications.

Chapter 6

Conclusion

This dissertation shows that, by specializing the mechanisms used to handle different transactions, database systems can achieve both performance and ease of programming.

In Chapter 3, we explored the implications of specializing the ACID abstraction. Salt introduces BASE transactions to express performance critical transactions, and creates the conditions under which they can safely coexist with other ACID transactions.

In Chapter 4, we instead explored the implications of maintaining the same, unchanged ACID transaction for all transactions, while instead specializing its implementation. Callas' MCC allows developers to provide isolation properties by enforcing these properties within each group modularly. This modularity enables Callas to customize the concurrency control mechanisms for different sets of transactions.

Although the vision of modular concurrency control is compelling and Callas makes a strong case for it, the implementation of this vision in Callas is still partial. Callas can customize the concurrency control mechanism for in-group conflicts, but all cross-group conflicts in Callas can only be han-

dled by a single cross-group mechanism based on two-phase locking. This limitation prevents us from seeking aggressive optimizations for cross-group conflicts, limiting the ability to leverage opportunities for concurrency. Realizing the vision of modular concurrency control in its fullness remains an open challenge. One possible approach we are currently exploring is to combine different concurrency control mechanisms in a hierarchical structure. This way, each cross-group mechanism need only handle the cross-group conflicts among its subgroups in the hierarchy, allowing us to seeking specialized optimizations also for these cross-group conflicts.

Bibliography

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

[2] AuctionMark. http://hstore.cs.brown.edu/projects/auctionmark/.

[3] Cloud Lab. http://www.cloudlab.us/.

[4] Current users of Fusion Ticket. http://www.fusionticket.com/hosting/our-customers.

[5] Dolibarr. http://www.dolibarr.org/.

[6] E-venement. http://www.e-venement.org/.

[7] Front Accounting. http://frontaccounting.com/.

[8] Fusion ticket. http://www.fusionticket.org/.

[9] MemSQL. http://www.memsql.com/.

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

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

[12] Ofbiz. http://ofbiz.apache.org/.

[13] Openbravo. http://www.openbravo.com/.

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

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

[16] SAP Hana. http://www.saphana.com/.

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

[18] Utah Emulab. http://www.emulab.net/.

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

[20] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 159–174, New York, NY, USA, 2007. ACM.

[21] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[22] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical

Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, 2015. ACM.

[23] 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.

[24] 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.

[25] A. J. Bernstein, P. M. Lewis, and S. Lu. Semantic conditions for correctness at different isolation levels. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 57–66, 2000.

[26] Philip A Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for Cloud Computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1255–1263. IEEE, 2011.

[27] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[28] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control-theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[29] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. 1987.

[30] Philip A Bernstein, David W Shipman, and James B Rothnie Jr. Concurrency Control in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 5(1):18–51, 1980.

[31] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–240, October 1992.

[32] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[33] 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, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[34] 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.

[35] 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.

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

[37] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 2012. USENIX.

[38] 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.

[39] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 163–174. ACM, 2010.

[40] Charles T. Davies, Jr. Recovery semantics for a db/dc system. In *Proceedings of the ACM Annual Conference*, ACM '73, pages 136–141, New York, NY, USA, 1973. ACM.

[41] A. Deacon, H. J. Schek, and G. Weikum. Semantics-based multilevel transaction management in federated systems. In *Data Engineering, 1994. Proceedings.10th International Conference*, pages 452–461, Feb 1994.

[42] 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.

[43] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. ACM.

[44] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.

[45] M. J. Bertin et al. *Pisot and Salem Numbers*. user Verlag, Berlin, 1992.

[46] Abdel Aziz Farrag and M. Tamer Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Trans. Database Syst.*, 14(4):503–525, December 1989.

[47] Alan Fekete. Allocating isolation levels to transactions. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 206–215, New York, NY, USA, 2005. ACM.

[48] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 78–91, New York, NY, USA, 1997. ACM.

[49] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[50] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD*, 1987.

[51] D. Georgakopoulos, M. Rusinkiewicz, and A. P. Sheth. Using tickets to enforce the serializability of multidatabase transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):166–180, Feb 1994.

[52] Seth Gilbert and Nancy Ann Lynch. Perspectives on the CAP Theorem. Institute of Electrical and Electronics Engineers, 2012.

[53] James N Gray. *Notes on data base operating systems*. Springer, 1978.

[54] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[55] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[56] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *Third Biennial Conference on Innovative Data Systems Research*, pages 132–141, 2007.

[57] George Roy Hill and William Goldman. Butch Cassidy and the Sundance Kid. Clip at `https://www.youtube.com/watch?v=1IbStIb9XXw`, October 1969.

[58] Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13:51–60, 1998.

[59] Donald K. Knuth. *The TEXbook*. Addison-Wesley, 1984.

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

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

[62] Leslie Lamport. *LATEX: A document preparation system*. Addison-Wesley, 2nd edition, 1994.

[63] Nancy A. Lynch. Multilevel atomicity—a new correctness criterion for database concurrency control. *ACM Trans. Database Syst.*, 8(4):484–502, December 1983.

[64] F Mittelbach M Goosens and A Samarin. *The LATEXCompanion*. Addison-Wesley, 1994.

[65] Michael McLure. Vilfredo Pareto, 1906 Manuale di Economia Politica, Edizione Critica, Aldo Montesano, Alberto Zanni and Luigino Bruni (eds). *Journal of the History of Economic Thought*, 30(01):137–140, 2008.

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

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

[68] Dan Pritchett. Base: An acid alternative. *Queue*, 6:48–55, May 2008.

[69] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[70] W. Schaad, H. J. Schek, and G. Weikum. Implementation and performance of multi-level transaction management in a multidatabase environment. In *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings. RIDE-DOM '95. Fifth International Workshop on*, pages 108–115, Mar 1995.

[71] Ralf Schenkel and Gerhard Weikum. Integrating snapshot isolation into transactional federation. In *Proceedings of the 7th International Conference on Cooperative Information Systems*, CooplS '02, pages 90–101, London, UK, UK, 2000. Springer-Verlag.

[72] Ralf Schenkel and Gerhard Weikum. *Integrating Snapshot Isolation into Transactional Federations*, pages 90–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[73] Lui Sha, John P. Lehoczky, and E. Douglas Jensen. Modular concurrency control and failure recovery. *IEEE Trans. Computers*, 37:146–159, 1988.

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

[75] 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.

[76] Michael Spivak. *The joy of TEX*. American Mathematical Society, Providence, R.I., 2nd edition, 1990.

[77] 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, 2007.

[78] 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, 2012.

[79] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[80] Alf J. van der Poorten. Some problems of recurrent interest. Technical Report 81-0037, School of Mathematics and Physics, Macquarie University, North Ryde, Australia 2113, August 1981.

[81] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[82] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–282, April 1989.

[83] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[84] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. pages 255–270, Boston, MA, December 2002.

[85] 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.

[86] 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, New York, NY, USA, 2015. ACM.

[87] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

Vita

Chao Xie was born in GuiYang, a beautiful city in Southeast China. He lived there until he graduated from Guiyang No.1 high school in 2004. Then he attended Tsinghua University, and received his bachelor's degree of Computer Science and Technology in 2011. He jointed the Department of Computer Science at University of Texas at Austin as a Ph.D. student in the fall of 2011.

Email address: xiechao89@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.