The Dissertation Committee for Natacha Crooks
certifies that this is the approved version of the following dissertation:

# A client-centric approach to transactional datastores

Committee:

Simon Peter, Supervisor

Lorenzo Alvisi, Co-Supervisor

Emmett Witchel

Peter Bailis

# A client-centric approach to transactional datastores

by

**Natacha Crooks,**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

December 2019

# A client-centric approach to transactional datastores

Publication No. ____________________

Natacha Crooks, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Simon Peter Co-Supervisor: Lorenzo Alvisi

Modern applications must collect and store massive amounts of data. Cloud storage offers these applications simplicity: the abstraction of a failure-free, perfectly scalable black-box. While appealing, offloading data to the cloud is not without its challenges. These cloud storage systems often favour weaker levels of isolation and consistency. These weaker guarantees introduce behaviours that, without care, can break application logic. Offloading data to an untrusted third party like the cloud also raises questions of security and privacy.

This thesis seeks to improve the performance, the semantics and the security of transactional cloud storage systems. It centers around a simple idea: defining consistency guarantees from the perspective of the applications that observe these guarantees, rather than from the perspective of the systems that implement them. This new perspective brings forth several benefits. First, it offers simpler and cleaner definitions of weak isolation and consistency guarantees. Second, it enables scalable implementations of existing guarantees like causal consistency. Finally, it has applications to security: it allows us to efficienctly augment transactional cloud storage systems with obliviousness guarantees.

# Contents

# List of Tables

# List of Figures

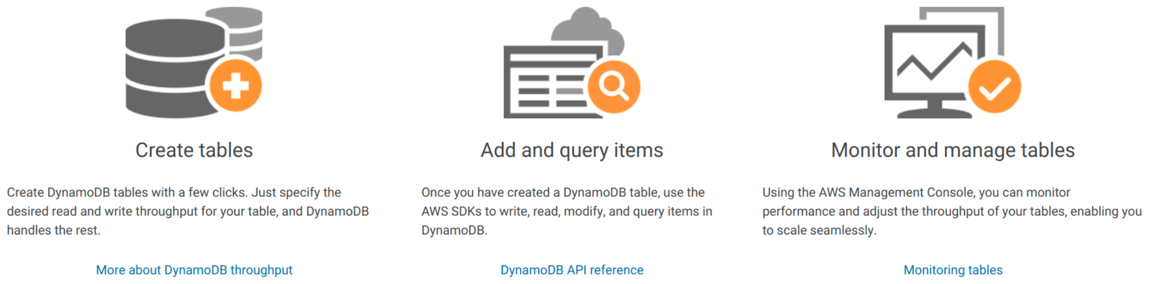
# Chapter 1

# Introduction

## 1.1 The cloud as a black-box

Traditional brick-and-mortar services are increasingly moving online and companies rely on ever larger data analytics to optimise their business logic. E-commerce sales, for instance, now represent almost 20% of all retail sales (up from 5% in 2007) [96]. Similarly, most medical practices favour electronic health records over paper documents: 84% of American hospitals store medical records electronically [127], an 8-fold increase since 2008; all aspects of our identity are increasingly stored online [21]. Even traditional industries like food distribution are affected: startups like Flexibake offer bakeries the ability to track customer food purchase to minimise food wastage [76].

In this data-driven world, data is money. It must be *collected* efficiently, even as it spans multiple heterogeneous, geo-distributed sources. Supply chains, for instance, or social networks span multiple geographical regions. Data must be stored *reliably*, even in the presence of failures. As the sensitivity of the data being stored increases, so does the need to store it *securely* in the presence of human attacks: the privacy implications of the 23AndMe [3] genomics data being stolen would be severe. There is a general concern about how collected data is used and to what purpose, as the recent Facebook Analytica scandal shows. Data must also be accessed *consistently*, even under high load. Efficiently guaranteeing consistency is challenging and applications often get it wrong, as the highly publicised DAO hack highlights. Attackers leveraged a concurrency bug on a popular



Create tables

Create DynamoDB tables with a few clicks. Just specify the desired read and write throughput for your table, and DynamoDB handles the rest.

More about DynamoDB throughput

Add and query items

Once you have created a DynamoDB table, use the AWS SDKs to write, read, modify, and query items in DynamoDB.

DynamoDB API reference

Monitor and manage tables

Using the AWS Management Console, you can monitor performance and adjust the throughput of your tables, enabling you to scale seamlessly.

Monitoring tables

*Figure 1.1: DynamoDB Front-End*

blockchain smart contract to steal fifty million dollars worth of currency [128]. Finally, data must remain *available*, even when the network is slow. Amazon once claimed that a 100ms latency spike correlates to a 1% loss in sales for the company [59].

While these problems are not inherently new, the context in which these solutions are being deployed is markedly different: application designers are rarely distributed system experts. Instead, they are domain-experts seeking to augment their business logic. These users consequently require solutions to collect and store data that demand little technical expertise. Cloud providers are thus increasingly offering storage solutions that minimise management overhead. Services like S3 [13], Dynamo [66], Cloud SQL [84], Azure Tables, [134] and many others offer applications *simplicity*. They present to applications the abstraction of a failure-free, scalable black-box; applications interact only with a narrow front-end that acts as a valve, and supposedly shields clients from the complex internal details of the system. A DynamoDB user, for example, simply has to specify the *provisioned throughput* that the system should sustain in a console window (Figure 1.1). The front-end webpage even advertises that users can, *using the AWS management console, monitor performance and adjust the throughput of their tables, enabling them to scale seamlessly*. Users do not have to worry about failures: the system is guaranteed to remain correct, even if individual failures occur. As the webpage advertises: *DynamoDB handles the rest*.

But what does correctness actually mean? Correctness in datastores usually refers to two notions: *isolation* and *consistency*.

- **Isolation** For ease of use, cloud datastores offer applications the ability to program using a

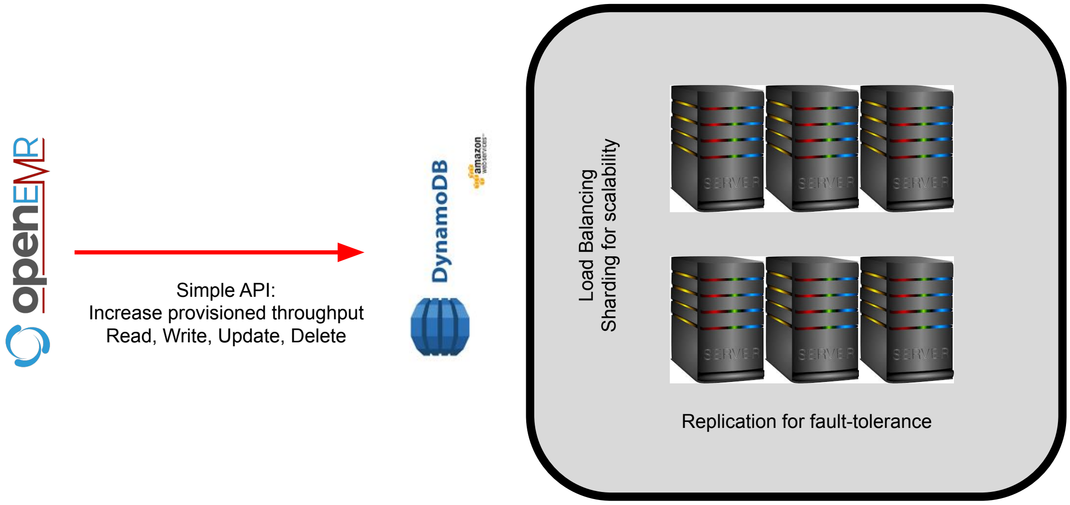

openEMR
Simple API:
Increase provisioned throughput
Read, Write, Update, Delete
DynamoDB
amazon web services
Load Balancing
Sharding for scalability
Replication for fault-tolerance

*Figure 1.2: Cloud storage as a black-box*

*transactional interface*. Transactions are groupings of operations that take effect atomically: either all operations take effect or none do. They simplify program development as they allow developers to group related operations into one single atomic unit. For performance, modern datastores allow multiple transactions to execute concurrently. *Isolation* then defines a contract that regulates the interaction between these concurrent transactions.

- **Consistency** For fault-tolerance or availability, cloud datastores usually *replicate* data on multiple servers. Consistency is then a contract that regulates the ordering of operations across replicas and places constraints on how much their state can diverge.

In summary, accessing cloud storage as a black-box shields applications from the internals of the system (Figure 1.2). The front-end "valve" limits what applications can observe to the system's *external state*. Applications thus only observe the state of the system through *the reads that they execute*. **This dissertation argues that viewing the system as a black-box represents a significant paradigm shift. This shift presents both semantic challenges and performance opportunities. This work thus proposes as contributions mechanisms and formalisms to *address* these challenges and *seize* these opportunities.**

### 1.1.1 Semantic challenges

On the one hand, viewing the system as a black-box introduces new challenges. Different correctness guarantees for datastores are usually defined in terms of low-level mechanisms, like operation ordering or timestamps [4, 31] that are not visible to applications in this black-box model. In fact, the role of the front-end is precisely designed to hide such details! Understanding correctness guarantees in this world is thus complex.

This is especially problematic as current datastores offer a myriad of different correctness guarantees (we summarise them in Chapter 2). Different systems strike different trade-offs between ease of programming and performance. Strong correctness notions like linearizability [92] or serializability [153] provide developers with the appealing illusion that transactions are executing in isolation and without concurrency or replication. These guarantees, however, tend to offer poor performance as they require extensive coordination across transactions and replicas. Many datastores, like MySQL [1], Oracle [150], or SQL Server [136], thus relax their notion of correctness; weaker guarantees offer better performance by reducing coordination [31]. In fact, almost all SQL database use weaker isolation definitions as default [4, 31, 105, 130, 136, 147, 148, 148, 158, 168, 178]. This trend poses an additional burden on the application programmer, as these weaker correctness guarantees increase program complexity. Namely, they allow for counter-intuitive application behaviours: relaxing the ordering of operations by minimising coordination yields better performance but introduces schedules and anomalies that could not arise if transactions executed in sequence on a single machine. These anomalies may break application logic: consider a bank account with a $50 dollar balance and no overdraft allowed. Under weaker isolation guarantees, the underlying database may allow two transactions to concurrently withdraw $45, incorrectly leaving the account with a negative balance (we describe this anomaly further in Chapter 3). Ensuring that the application remains correct thus requires carefully understanding what anomalies can arise under a given correctness guarantee. Unfortunately, achieving this careful understanding often requires detailed knowledge of the internals of a given system. But, as we said, the actual guarantees provided by different correctness notions are often dependent on specific (and occasionally implicit) system properties - be it properties of storage (e.g., whether it is a single or multiversioned storage [32]); of the chosen concurrency control (e.g, whether is is based on locking or timestamps [31]); or of other system features (e.g., the existence of a centralized timestamp [73]). These are specifically the details that cloud storage attempts to hide. There is consequently a fundamental mismatch between how clients perceive storage systems and how correctness guarantees are expressed.

### 1.1.2 Performance opportunities

The situation is not all bleak, however. Recognising that clients perceive cloud storage as a black-box actually brings new system opportunities. Specifically, it enables more flexible implementations of the aforementioned correctness guarantees. Why? The observation is simple: the state of individual components of the datastore does not need to be correct, as long as the external state - the state that applications actually observe, *appears indistinguishable from a state that is* correct. In effect (and less formally), this dissertation suggests following the ninth Seinfeld rule of lying[1]: *it's not a lie if you don't get caught*. Because clients only see the state of the system through their API reads, implementing a system that is indistinguishable from a correct system simply requires ensuring that the return values of operations remain correct. This observation brings forth one key benefit: it allows implementations to be contextualised for specific applications or clients according to their chosen API, and contextualisation can improve scalability. Unfortunately, such specialisation is rarely present in existing systems. As a result, these systems often implement correctness guarantees that are stronger than what applications actually require. Existing causally consistent systems [26, 118, 119], for instance, require that every replica in the datacenter store a state that is causally consistent for *any* potential client. Doing so comes at a heavy cost: these systems must often delay specific operations, which makes them subject to *slowdown cascades*. A slowdown cascade is a failure mode in which a single slow shard can cause the entire system to perform poorly, and has been mentioned by Facebook as the main barrier to adoption of current causally consistent systems [129]. This cost is not fundamental: ensuring that every component in the datastore is causally consistent is a sufficient but not a necessary invariant. It is sufficient to ensure that every client, upon requesting a set of keys, observe a state that is causally consistent for that subset of objects. Similar problems exist in systems that implement the related *parallel snapshot isolation* guarantee [58, 178].

[1] Seinfeld is a popular american TV show created by created by Larry David and Jerry Seinfeld. In one episode, the character's best friend, George Costanza, offers some advice but lying

The issue also resurfaces in the context of *privacy-preserving datastores*. There is a growing desire to share data with untrusted third parties. Medical providers, for instance, offload data to the cloud for fault-tolerance, but do not necessarily trust the cloud to keep this data private [127]. Generic solutions for accessing and manipulating data without cloud storage learning any information about the requests or dataset do exist (Private Information Retrieval [52, 109] or Oblivious RAM [82], for instance). These solutions, however, are costly: they usually incur overheads proportional to the number of objects in the system. These systems provide highly generic, clean, re-usable abstractions. The flipside however, is that they often provide guarantees that are stronger than what is required for real systems. Indeed, systems do not exist in a vacuum, but are deployed in a specific context (e.g. medical services or media streaming). This context can allow for more flexibility. Media files, for instance, need only be streamed as fast as users are viewing the content. There is no need to deliver them sooner. Contextualising the implementation of privacy-preserving mechanisms for the specific environments in which they are deployed can significantly amortise the cost of guaranteing privacy [57, 90].

## 1.2 Contributions

This dissertation argues that accessing cloud storage as a black-box requires revisiting how correctness guarantees are expressed and proposes taking a *client-centric approach to system development*. Correctness guarantees are currently expressed bottom-up, starting from the perspective of the systems that implement them. As we have argued, this approach presents both semantic and implementation drawbacks. Instead, this thesis suggests that correctness guarantees should be expressed *top down, starting from the perspective of the clients that actually use these guarantees*). Thinking of cloud storage systems as a sequence of client-observable states brings forth several benefits. From a semantic standpoint, it can make it easier to understand, compare and relate *database isolation guarantees* [58]. Second, it can simplify the handling of *write-write conflicts* in causal consistency [59]. From a performance standpoint, moving the output commit to clients for enforcing correctness has applications to both fault-tolerance and security. It is key to making causal consistency resilient to slowdown cascades [129]. It is also the core observation that allows for generic and expensive privacy algorithms to be specialised into more efficient, practical systems. We show that leveraging the streaming behaviour of Netflix clients can amortise the cost of private media streaming [90]. Similarly, taking a client-centric approach to enforcing serializability allows us to design transactional cloud storage systems that provably hide accesses to user data [57]. This thesis focuses on the following four contributions:

- **A new model for isolation** We argue that isolation guarantees should be expressed in the way they are perceived: as a contract between the storage systems and its clients, that specifies the set of observable system states. We propose a new model that directly defines isolation guarantees in terms of these high-level states. Each isolation level is expressed as constraints on the states that the application observed. Using our model, we propose the first clean hierarchy of snapshot-based guarantees, and prove that several isolation levels are equivalent. Our model is general: we provide equivalent definitions of most isolation definitions, even as we make no reference to traditional notions such as dependency graphs or histories.
- **A unified model of isolation and consistency** We extend this model to consistency definitions and provide the first unified framework for reasoning about consistency and isolation. Extending this model to consistency guarantees brings forth two benefits. First, we find that the benefits of a client-centric approach naturally extend to consistency. Specifically, it allows us to define session guarantees without making assumptions on the order of writes in the system. Second, it allow us to unify the often disparate theories of isolation and consistency and provides a structure for composing these guarantees. We leverage this modularity to extend the equivalence between causal consistency and session guarantees that previously held for single operations. Importantly, we show that this equivalence holds independently of the isolation level under which they execute.
- **Improved handling of write-write conflicts in weak consistency** Weaker consistency levels like eventual consistency or causal consistency allow replicas to issue conflicting write operations, which may cause their states to diverge. Current storage systems take a system-centric approach to resolving these conflicts. They aggressively and greedily merge them through per-object, syntactic resolution policies [118, 187]. Unfortunately, conflicting writes corrupt data in ways that the storage system cannot understand, as they create potentially incompatible

states. In effect, conflicting operations create distinct, conflicting *branches of execution* that can only be resolved with detailed knowledge of application semantics. Attempting to greedily hide these conflicting executions thus makes matters worse, not better. Merging is the responsibility of the clients or applications that use these systems. To this effect, we propose TARDiS, a replicated transactional key-value store that renounces the abstraction of sequential storage and directly exposes these conflicting branches to applications, and to the states at which the branches are created (fork points) and merged (merge points). TARDiS both simplifies semantically meaningful merging, and improves its efficiency: branches, fork points and merge points directly capture the context necessary to identify objects that must be merged and pinpoint when and how the conflict developed. We find, for example, that implementing CRDTs [172] - a library of weakly-consistent datatypes - using TARDiS cuts code size by half and improves performance by up to eight times.

- **Support for serializable and oblivious transactions** Finally, we report on the design of Obladi, the first system to provably to support oblivious, serializable transactions on top of untrusted storage. The system takes as its starting point oblivious RAM (ORAM), which hides access patterns for read and write operations. Adding support for serializable transactions on top of ORAM brings forth several challenges. Existing ORAMs cannot guarantee durability, do not support transactions, and afford only limited concurrency. The secret sauce lies in reformulating the traditional isolation definition of serializability in a client-centric way. Serializability need only hold when transactions *are observed by clients as committed*. Obladi leverages this new flexibility to delay committing transactions until the end of fixed-size epochs, enforcing consistency and durability only at epoch boundaries. Delaying operations in this way allows Obladi to securely parallelise Ring ORAM [164] and amortise the cost of expensive ORAM operations across many transactional requests. This same principle also allows Obladi to recover efficiently and securely from failures. Our results are promising: Obladi achieves within 5x-12x of the throughput of MySQL on standard OLTP benchmarks like TPC-C. Latency is higher (70×), but remains reasonable (in the hundreds of milliseconds).

## 1.3 Thesis Overview

This thesis is structured as follows: Chapter 2 provides the necessary background on isolation and consistency. Chapter 3 summarises the semantic benefits of reformulating database isolation guarantees in a client-centric, state-based way. Chapter 4 extends this new model to consistency guarantees. Chapter 5 describes the benefits of organising these states in distinct branches that capture the independent executions that naturally arise in weakly consistent systems. It presents the design and implementation of TARDiS (Transactional Asynchronous Divergent Storage), a replicated key-value store that makes branching a first class primitive. Chapter 6 discusses how reformulating seriazability to be client-centric is key to supporting oblivious transactions efficiently, and presents the design of Obladi, the first cloud-based datastore that supports serializable transactions while also hiding from the cloud access patterns (when, how, and what data is accessed). Finally, Chapter 7 concludes and summarises related work not presented in this dissertation.

# Chapter 2

# Correctness Background

The previous chapter outlined two challenges faced by modern applications when offloading data to cloud storage: striking the appropriate balance between performance and correctness, and security. In this chapter, we provide the necessary background on correctness for transactional datastores. We focus specifically on the process through which weak notions of isolation and consistency have been formalised. We defer the discussion on security to Chapter 6.

## 2.1 Transactions

This thesis focuses specifically on *transactional* datastores, namely, datastores that support transactions. Transactions are a grouping of operations that appear to take effect atomically: either all of operations take effect or none do. They represent "instantaneous changes to the world" [153]. The *start* operation marks the begin point of a transaction while the *commit* or *abort* operations mark the end of the transaction (Figure 2.1). They are a popular feature of most database systems as they simplify program development. They allow programmers to group composite operations into one single atomic unit. For instance, a transaction ensures that the *increment* and *decrement* operations of a *transfer* transaction (Figure 2.1) will always either both take effect or neither will (even in the presence of crashes). This ensures that the $20 dollars that are being transferred from Alice's account $a$ to Bob's account $b$ will never disappear. Traditionally, transactions are said to guarantee the ACID properties of **A**tomicity, **C**onsistency, **I**solation, and **D**urability:

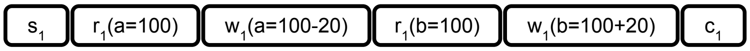


*Figure 2.1: Transaction Overview: $r_1$ stands for $T_1$ reads, while $w_1$ stands for $T_1$ writes*

- **Atomicity** states that either all the operations in a transaction take effect or none do. A transaction can either *commit*, in which case all operations will be applied to the database, or *abort*, when no operation will take effect.
- **Consistency** states that a transaction transitions the database from a consistent state to another consistent state. While this definition may appear redundant, it captures the idea that a transaction, if executed alone on a given (correct) database state, will yield another correct database state. Correctness is naturally application-dependent, but can be interpreted as the set of invariants that hold for a given application. Some of these invariants might be explicitly enforced by the database (unique key constraints, foreign key constraints) while others might be implicit in the application logic (an account balance for a bank should never go negative). While the idea of consistency might appear blurry, it is in fact what motivated the first correctness criterion for executing transactions concurrently: serializability. We describe this in more detail in the next section.
- **Isolation** defines a contract that regulates how concurrent transactions interact. It determines when the effects of a given transaction will become visible to other executing transactions. The chosen isolation consequently regulates which transaction interleavings are allowed by a given database.
- **Durability** states that the effects of committed transactions must be preserved across database failures. Otherwise said, once a transaction has committed, its effects should always remain visible in the database.

## 2.2 Isolation

Transactional datastores manage data that is shared amongst many users. They must sustain high throughput and achieve low latency, while ensuring that data accesses remain correct. To do so, most databases allow transactions to execute concurrently but regulate their interaction to allow only specific transaction interleavings. In other words, they trade-off isolation for performance. Through constraining the allowable interleavings, isolation also constrains what values can be returned by applications. Isolation thus directly defines correctness from the point of view of a user. The question becomes: how does one define isolation, and what is the correct isolation guarantee? This topic is controversial: there are many co-existing formalisms and a myriad of different isolation guarantees [4, 5, 25, 31, 32, 46, 47, 49, 61, 65, 71, 73, 78, 88, 103, 129, 153, 154, 166, 170, 178]. The rest of this section summarises this space. It places specific emphasis on understanding why the formalisms and guarantees evolved as they did and puts forward the following hypothesis: **all prior definitions remain tightly coupled with the underlying implementation/technology of systems at the time. As technology evolved, so did the necessary formalisms and definitions.**

### 2.2.1 The beautiful doctrine of serializability

The ACID semantics appear to provide two notions of correctness: 1) consistency, whereby each transaction transitions the database from a state that satisfies all application invariants to another such state, and 2) isolation, which regulates the interaction of concurrent transactions.

There is a historical reason for these two entangled definitions. Transactions were first defined for single-threaded systems. They were viewed as atomic unit of executions. Jim Gray's initial definition of transactions, for instance, omitted the notion of isolation and defined a transaction as follows [85]: a transaction is a piece of user code, some of which may execute outside of the database, some of which accesses records from the system. This piece of code is a single-threaded *contract* that guarantees the following properties:

- **Consistency**: the transaction must obey legal protocols
- **Atomicity**: it either happens or it does not; either all operations are bound by the contract or none are
- **Durability**: once a transaction is committed, it cannot be abrogated

The 1970s then saw the beginning of concurrency for database systems [35, 36, 71, 88, 166]: multiple transactions could execute simultaneously and their operations would interleave for performance. Most systems of the time agreed that the appropriate notion of correctness should be *serializabil-*

*ity* [153] or *serial reproducability* [35, 36]. They take as their starting point the statement that each transaction, when run in isolation, transitions the system from a consistent state to another consistent state. By induction, a sequence of transactions that execute in turn preserves database consistency. *A serializable execution is thus an execution that is equivalent to some serial ordering of the same set of transactions*. They coined the term of *isolation* as an additional requirement to the durability, consistency and atomicity semantics: transactions must not be affected by concurrent transactions.

This definition of serializability is appealing: it allows developers to program an application as if running in complete isolation, while still obtaining the performance benefits of transactions interleaving. To illustrate, consider again the example in which Alice is transferring money to Bob. But let us assume now that Alice's account is a shared account that can be concurrently accessed by Alice's wife, Eve. Eve wants to transfer money to Charlie and issues a transfer transaction concurrently.

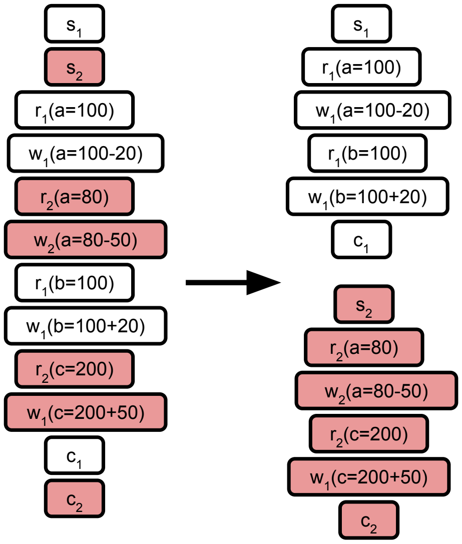


*Figure 2.2: Alice executes $T_1$, a transfer transaction from Alice to Bob of $20. Eve executes $T_2$, a transfer transaction from Alice to Charlie of $50. $T_2$'s first read sees $T_1$'s first write*

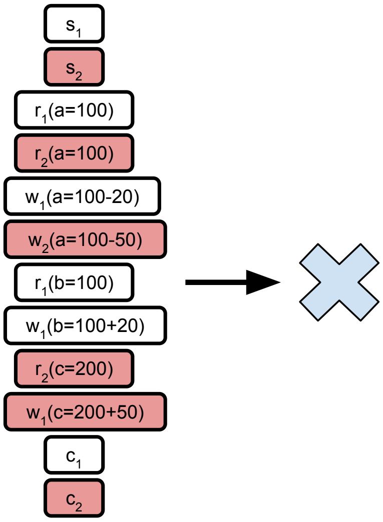


*Figure 2.3: Alice executes $T_1$, a transfer transaction from Alice to Bob of $20. Eve executes $T_2$, a transfer transaction from Alice to Charlie of $50. $T_2$'s first read does not see $T_1$'s first write*

The left-most schedule is serializable (Figure 2.2): it is *equivalent* to a schedule in which Alice's transfer operation executes fully before Eve's. One can see that the end values are correct. The system started with a total of $400 dollars and finishes with $400. Alice has $30 in her account, Bob $120, and Charlie \$250. By contrast, the schedule on the right (Figure 2.3) is not reproducible. There is no way to order $T_1$ before $T_2$ or $T_2$ before $T_1$: $T_2$ misses $T_1$'s decrement ($w_1(a = 100 - 20)$)) while $T_1$ misses $T_2$'s decrement ($w_2(a = 80 - 50)$). Moreover, the end result is not correct: the total balance has grown to \$420 dollars. Unfortunately, just as money does not grow on trees, it does not grow in serializable schedules.

As appealing as the notion of serializability is, there is a catch: the previous paragraph intentionally under-defined the notion of *equivalence*. This definition turns out to be subtle, and can vary across systems and formalisms. The implicit reliance on system constructs means that the elegant definition of serializability can fall prey to inconsistencies.

### 2.2.2 The gritty reality of serializability

While all systems agree with the spirit of serializability, their proposed formalisms differ: they are primarily tied to the architecture of the system for which they define correctness. As new, more efficient algorithms were proposed (ex: the introduction of replication, of multiversioning), new definitions were introduced. These differences persist to this day. We summarise them here.

Most treatments of concurrency control use similar notations for transactions and refer to *histories* as denoting the execution of a set of transactions. For the purpose of this chapter, we use the formalism defined in Bernstein et al. [33] and we formulate other frameworks accordingly.

A **transaction** $T_i$ is a partial order of operations where:

- $T_i \subset \{r_i(x), w_i(x)|\text{x is a data item}\} \bigcup \{a_i, c_i\}$
- $a_i \in T_i$ iff $c_i \notin T_i$
- if t is $c_i$ or $a_i$ (whichever is in $T_i$), for any other operation $p \in T_i$, $p <_i t$
- if $r_i(x), w_i(x) \in T_i$, then either $r_i(x) <_i w_i(x)$ or $w_i(x) <_i r_i(x)$

Intuitively, this definition states that a transaction must contain exclusively read or write operations, and will necessarily either commit or abort after having executed all other operations.

A **history** then summarises the ordering of operations in an execution of transactions. Histories are

| Lock Mode | X | S |
|---|---|---|
| X | ✗ | ✗ |
| S | ✗ | ✓ |

*Table 2.1: Lock compatibility matrix*

defined as partial orders as some of these operations may execute concurrently. Histories constrain ordering in two ways. First, if a transaction $T_i$ specifies the order of two operations, these operations must appear in that order in the history. Second, all histories must specify the order of *conflicting operations*. Two operations are said to conflict if they access the same object and at least one of them is a write operation. More formally, given a set of transactions $T = \{T_1, T_2, ..., T_n\}$, a history $H$ is a partial order with ordering relation $<_H$ where:

- $H = \bigcup_{i=1}^{n} T_i$
- $<_H \subset \bigcup_{i=1}^{n} <_i$
- for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_h p$.

For simplicity, we assume that all transactions either commit or abort[1].

**Serializability through locking** Jim Gray's seminal 1975 paper [88] defines the notion of serial schedule exclusively through locking. He specifies different *modes of locking* that restrict transaction interleavings to serializable executions. Specifically, the paper defines two modes of locking *Shared (S)* mode and *Exclusive (X)* and a *compatibility matrix* (Table 2.1).[2] *Compatible* locks can be acquired concurrently, while incompatible locks cannot. Transactions must acquire a shared lock for read operations and an exclusive lock on write operations, and release all locks at commit or abort time.

**Conflict Serializability** While the aforementioned approach is sufficient to ensure serializability for lock-based systems, it is implementation-specific and is consequently unsuitable for other concurrency control mechanisms (for instance, it cannot be applied to Kung et al's optimistic concurrency control protocols [107]). Several authors thus proposed a more general theory of serializability, based on the notion of conflict equivalence and serializability graphs [34–36]. Recall that Gray et al.'s defined

[1] The full formalisation includes the notion of committed projection to handle histories in which transactions do not terminate

[2] The full paper includes Intention mode locking for hierarchical datastructures. We ignore it here.

serializability to mean *equivalent to a serial schedule*. Conflict serializability formalises this notion as follows: two histories $H$ and $H'$ are *conflict equivalent* if:

- they are defined over the same set of transactions and have the same operations
- they order conflicting operations of non-aborted transactions in the same way; that is, for any conflicting operations $p_i$ and $p_j$, belonging to transactions $T_i$ and $T_j$ (where $a_i$, $a_j \notin H$): if $p_i <_H q_j$ then $p_i <_{H'} q_j$.

Intuitively, this definition states that two histories are conflict equivalent as long as the ordering of conflicting operations is identical; non-conflicting operations can be reordered. Serializability is then defined as follows: *a history is serializable if it is conflict equivalent to a serial history*. The authors further define an equivalent serializability condition in terms of *serialization graphs*. A serialization graph (SG) for H, denoted SG(H), is a directed graph whose nodes are committed transactions and whose edges are $T_i \rightarrow T_j$ such that one of $T_i$'s operation precedes and conflicts with one of $T_j$'s operations in H. A history H is then serializable iff SG(H) is acyclic. Looking at the serializability graphs of the executions previously shown in Figure 2.2, we indeed see that the equivalent serialization graph of the serializable execution is acyclic (Figure 2.4). In contrast, the serialization graph of the non-serializable execution displays a cycle (Figure 2.5). The two transactions conflict on object $a$ and object $b$. For the serializable schedule, transaction $T_1$ always executes the conflicting operations before $T_2$, there is thus no cycle in the graph. In contrast, the schedule in Figure 2.5 executes $r_2$ followed by $w_1$ followed again by $w_2$: this creates a cycle in the graph. The definition of conflict serializability is not equivalent to the definition of lock-based serializability. There exists schedules which are conflict serializable but not lock-serializable. Consider again the execution shown in Figure 2.4. The corresponding serialization graph is acyclic; the execution is consequently conflict-serializable. It is however, not lock-serializable as $w_2(x)$ could not execute before $T_1$ commits (due to the read locks that transactions acquire).

**View Serializability** Yannakakis et al.[202] defines an alternative definition of equivalence called view equivalence. View equivalence is premised on the assumption that if each transaction's reads observe the same value in two histories, then their writes will also produce identical values. A history is thus view-equivalent to a serial history if its reads return the same value in both executions. More

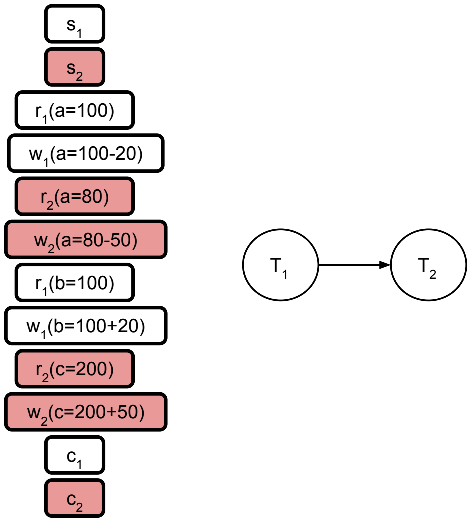


*Figure 2.4: Acyclic serialization graph for serializable history*

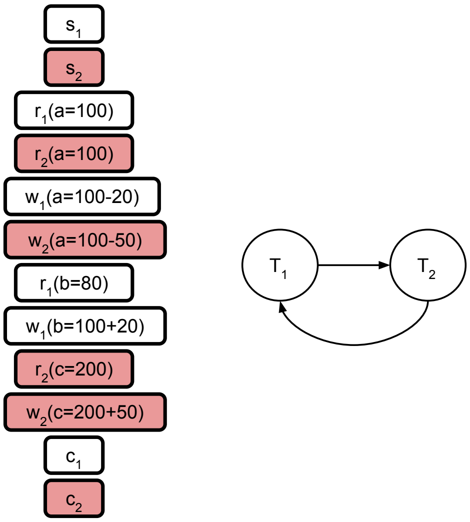


*Figure 2.5: Cyclic serialization graph for non-serializable history*

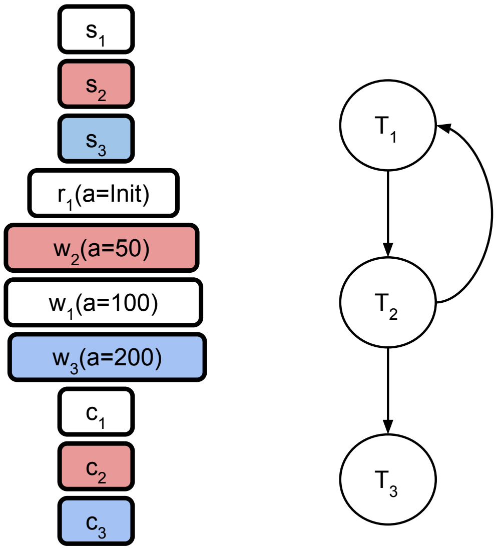


*Figure 2.6: An execution that is view-serializable but not conflict- serializable*

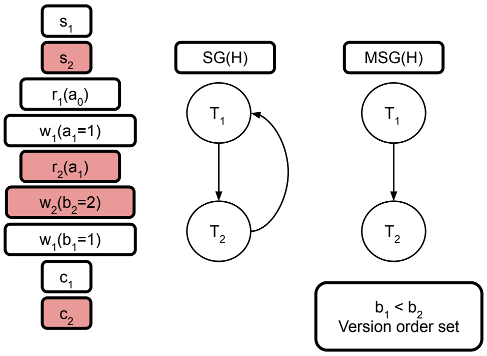


*Figure 2.7: An execution that is multiversioned-serializable but not conflict or view-serializable*

formally, two histories $H$, and $H'$ are view-equivalent if

- they are over the same set of transactions and have the same operations
- for any object x, if $T_i$ reads value $x_j$ from $T_j$ in $H$ then $T_i$ reads $x_j$ from $T_j$ in $H'$. A transaction

reads a value from a transaction $T_j$ if it $T_j$ is the last transaction to write $x$ before $T_i$'s read.

- for each $x$, if $w_i(x)$ is the final write of $x$ in $H$ then it is also the final write of x in $H'$.

A history is view-serializable if it is view-equivalent to a serial history. While all conflict-serializable histories are view-serializable, the converse is not true, as we illustrate in Figure 2.6. The execution is not conflict-serializable as there is a cycle in the serialization graph. It is however view-serializable and corresponds to the serial schedule $T_1 \rightarrow T_2 \rightarrow T_3$ as $T_1$ reads the initial value of $a$ in both schedules and $T_3$ performs the final write in both executions.

**Multiversioned Serializability** Conflict serializability and view-serializability both implicitly assume that a read operation must read the latest value of an object. Both notions were defined at a time in which database systems stored a single value of every object in the system. However, the 1980s demonstrated the potential performance gains of storing *old values* of every object in the system. To take advantage of this new flexibility, Bernstein et al.[32] developed a new serializability theory of multiversioned databases. As their name suggests, multiversioned databases introduce the notion of versions for a given data item. A write on object x for transaction $T_i$ creates a new version $x_i$ of object x (written $w_i(x_i)$). A read of transaction $T_j$ is said to read-from $T_i$ if it reads the version of $x$ that $T_i$ writes (written $r_j(x_i)$). Bernstein et al. further introduces the notion of a version order set $\ll$. A version order for an object $x$ is a total order over all of the versions of x written in $H$. A version order set is for H is the union of the version orders for all data items. A multiversioned serialization graph (MSG) for H, denoted MSG(H), is a directed graph whose nodes are committed transactions and whose edges are $T_i \rightarrow T_j$ such that either $T_j$ reads from $T_i$ or $x_i < x_j$ in the version order. A history H is then multiversioned serializable iff there exists a version order set $\ll$ such that MSG(H) is acyclic.

Consider Figure 2.7, the execution is multiversioned serializable as there exists an assignment of versions ($a_1 < a_2$, $b_1 < b_2$) such that MSG(H) is acyclic. This execution is not conflict serializable: the SG(H) contains a cycle. It is also not view-serializable as $T_1$ and $T_2$ cannot be reordered without changing the final database state. In effect, multiversioned serializability allows for write operations to be re-ordered, unlike its view or conflict counterparts.

### 2.2.3 Living on the edge: weak isolation and anomalies

In spite of the algorithmic and implementation progress made over several decades, serializability remains slow: it requires extensive coordination between concurrent transactions. As a result, the majority of databases actually provide, as default, a guarantee that is strictly weaker than serializability [132, 136, 148–150, 158, 168]; some databases do not offer serializability at all [148, 150, 168]. Instead, they weaken the *isolation* property in ACID.

*Figure 2.8: Weak Isolation - a metaphor*

Weak isolation guarantees are not unlike cool racing bikes (Figure 2.8). On the one hand, they are fast: weaker isolation levels reduce the amount of coordination necessary and hence provide better performance. On the other hand, relaxing isolation is dangerous: weak isolation levels admit *anomalies* which are non-serializable behaviours that can break application logic. Application developers must thus defensively program against these anomalies to ensure that the application logic remains correct. In this subsection, we briefly summarise the most common types of anomalies and illustrate their consequences.

**Dirty-read anomaly** Serializability ensures that no committed transaction will read values of an uncommitted transaction. Consider what would happen if *dirty reads* were allowed (Figure 2.10), that is, if a transaction were allowed to read uncommitted values. In our example, transaction $T_1$ transfers $20 dollars from Alice to Bob's account. It then aborts, canceling the deposit and reverting Bob's balance to $100. $T_2$, however, reads the new balance and mistakenly believes that the deposit is successful.

**Lost-update anomaly** Serializability ensures that no concurrent transactions can write the same value. It thus prevents the *lost update* anomaly. To illustrate, consider a schedule in which two

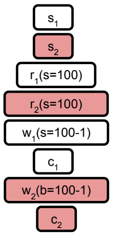


*Figure 2.9: Lost-update anomaly*

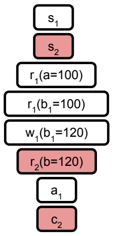


*Figure 2.10: Dirty-read anomaly*

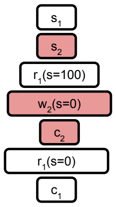


*Figure 2.11: Non-repeatable read anomaly*

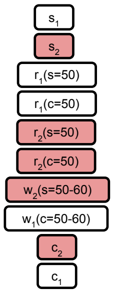


*Figure 2.12: Write-skew anomaly*

transactions attempt to decrement stock from a warehouse (Figure 2.9: both transactions read the initial value (100) of the stock object, and both decrement it by one. $T_1$ writes 99 back to the *stock* object, and $T_2$ does the same. Two transactions decremented the stock object, yet the final value is only 99, which is incorrect (it should be 98).

**Non-repeatable read anomaly** Serializability gives transactions the illusion that they are executing in isolation and without concurrency. They are guaranteed, upon reading the same value multiple times of a given object, that the value will remain the same. This is no longer true when allowing for non-repeatable reads (Figure 2.11).

**Phantom-read anomaly** The phantom read anomaly follows the same logic as the non-repeatable read anomaly: it allows for scans or aggregate queries (like max or min) to return different values within the same transaction, betraying the fact that the transaction is not executing in isolation.

**Write-skew anomaly** Finally, the write-skew anomaly is an anomaly that arises exclusively in multiversioned systems. Write-skew arises when two conflicting transactions miss each other's updates and can consequently not be ordered with respect to each other. To illustrate, consider the following example (Figure 2.12). Alice and Bob share a checking and savings account and can withdraw money from either as long as the joint balance is not negative. This execution is clearly not serializable as $T_1$ misses $T_2$'s update while $T_2$ misses $T_1$'s update (which introduces a cycle). Moreover, the application invariant, namely that the joint balance of the two accounts should never be negative, is now broken.

| Isolation | Read lock duration | Write lock duration |
|---|---|---|
| Degree 0 | None | Short write lock |
| Degree 1 | None | Long write lock |
| Degree 2 | Short read lock | Long write lock |
| Degree 3 | Long read lock | Long write lock |

*Table 2.2: Gray - Lock Modes*

### 2.2.4 Formalising weak isolation

These anomalies are not artificial examples and arise in practice. A recent paper by Warszawski et al. [195] observes that a majority of E-Commerce frameworks exhibit application-level bugs when running under weaker isolation guarantees (recall that most databases do not actually provide serializability). They allowed for inventories to become negative, gift vouchers to be overspent, and cart checkout totals to become out of sync with the actual price of items in the cart. Programming correctly against weak isolation levels thus requires a detailed understanding of semantics that is currently lacking. As such, there has been a long quest to try and formalise isolation guarantees. Unfortunately, much like the initial definitions of serializability, the various formalisms proposed over the years remain tightly coupled to underlying system implementations. These formalisms evolved as technology improved: from lock-based systems, to optimistic concurrency control, to more aggressive multiversioned-based systems.

**Degrees of Isolation** The notion of weak isolation guarantees was first introduced by Gray et al. [86] in 1976 with the goal of improving performance at the cost of more complex programming semantics. The authors define four *degrees of consistency* (Degrees 0 to 3) that provide increased "protection" from concurrent transactions. The author summarises each degree as follows:

- **Degree 0** protects others from your update: a transaction T sees degree 0 consistency if T does not overwrite dirty data of other transactions.

- **Degree 1** additionally provides protection from losing updates: T sees degree 1 consistency if T does not overwrite dirty data of other transactions *and* T does apply writes before commit time.

- **Degree 2** additionally provides protection from incorrect data items: T sees degree 2 consistency if T does not overwrite dirty data of other transactions *and* T does not apply writes before commit time *and* T does not read dirty data of other transactions.

- **Degree 3** additionally provides protection from reading incorrect relationships among data items (it is equivalent to serializability): T sees degree 3 consistency if T does not overwrite dirty data of other transactions *and* T does not apply writes before commit time *and* T does not read dirty data of other transactions *and* other transactions do not dirty any data read by T before T completes.

The paper then formalises these degrees in terms of different types of shared (read) locks and exclusive (write) locks (Table 2.2). To ensure that a transaction does not overwrite the dirty data of other transactions, one must use short write locks that are released after the operation finishes (degree 0 consistency). To additionally ensure that a transaction does not commit any writes before commit time (ie. to guarantee degree 1 consistency), these locks must be upgraded to long write locks (that are held until commit time). Degree 2 consistency then places constraints on read operations: it mandates the use of short read locks to ensure that a transaction does not read the dirty data of other transactions. Finally, degree 3 consistency upgrades these locks to long read locks to ensure that other transactions do not write data read by a transaction $T$ before $T$ completes.

**ANSI-SQL: Non-locked based formulation** Gray's proposed degrees of isolation are, however, limited to lock-based system. In 1992, the ANSI/ISO SQL-92 specification thus set out as goal to define an industry standard for weak isolation that would be implementation-independent and support different concurrency control schemes. Each ANSI level is defined as proscribing specific *anomalies* or *phenomena*. The specification (informally) defines three such phenomena:

- **Dirty read** Transaction $T_1$ modifies object $x$. Another transaction $T_2$ then reads $x$ before $T_1$ commits or aborts. If $T_1$ aborts, $T_2$ has read an object that was never committed and so never really existed.

- **Fuzzy or non-repeatable read** Transaction $T_1$ reads object $x$ and then $T_2$ modifies $x$ and commits. If $T_1$ then attempts to reread $x$, it receives a modified value.

- **Phantom read** Transaction $T_1$ reads a set of objects satisfying some `<search`

| Phenomenon | Strict (Anomaly) Interpretation | Broad (Phenomena) Interpretation |
|---|---|---|
| Dirty Read | **A1:** $w_1(x)$...$r_2(x)$... ($a_1$ and $c_2$ in either order) | **P1:** $w_1(x)$...$r_2(x)$ (all commit/abort) |
| Fuzzy Read | **A2:** $r_1(x)$...$w_2(x)$...$c_2$...$r_1(x)$...$c_1$ | **P2:** $r_1(x)$...$w_2(x)$(all commit/abort) |
| Phantom | **A3:** $r_1(P)$...$w_2(y \in P)$...$c_2$...$r_1(P)$...$c_1$ | **P3:** $r_1(P)$...$w_2(y \in P)$(all commit/abort) |

*Table 2.3: Broad and strict intepretations of the ANSI SQL phenomena*

> `condition>`. Transaction $T_2$ then creates data items that satisfy $T_1$'s `<search condition>` and commits. If $T_1$ then repeats its read with the same `<search condition>`, it gets a set of data items different from the first read.

The authors then define three isolation guarantees (intended to be equivalent to aforementioned degrees 1-3). *Read uncommitted* disallows no phenomena. *Read committed* disallows dirty reads, *repeatable read* disallows both dirty reads and fuzzy reads, while serializable disallows all three phenomena.

**Formalising ANSI** The ANSI SQL definitions are informally defined in plain English and offer multiple possible interpretations. Berenson et al. [31] shows that these definitions could be interpreted in ways that result in inconsistencies. They instead propose a new set of correct isolation definitions that are precise. More specifically, the authors identify two possible interpretations of the informal English definitions. The first interpretation applies the definition literally: it disallows only executions in which the anomaly actually occurs. The authors refer to this as the strict interpretation of ANSI SQL or as the anomaly-based interpretation. An alternative interpretation of the ANSI definitions disallows all executions that *may* lead to those undesirable behaviours. The authors refer to this as the broad interpretation or as the phenomena-based interpretation. The anomaly-based interpretation naturally admits more executions than the phenomena based interpretation: some of these executions turn out to be inconsistent.

To illustrate, consider the *dirty read phenomenon*: it can be interpreted to disallow one of two schedules (see Table 2.3). In its most literal interpretation, it disallows only schedule **A1** (where $T_1$ aborts but $T_2$ commits). Otherwise said, a transaction $T_2$ that reads from an aborted transaction $T_1$ must not be allowed to commit. In its broadest interpretation, the definition disallows any history where $T_2$ reads uncommitted data (**P1**). The same ambiguity is present for fuzzy reads and phantoms. The strict, anomalous interpretation (**A2**) of *fuzzy reads* prevents a transaction $T_1$ from reading an

| Isolation level | P0 Dirty write [4] | P1 Dirty read | P2 Fuzzy read | P3 Phantom |
|---|---|---|---|---|
| ANSI Read uncommitted | Not Possible | Possible | Possible | Possible |
| ANSI Read committed | Not Possible | Not Possible | Possible | Possible |
| ANSI Repeatable Read | Not Possible | Not Possible | Not Possible | Possible |
| ANSI Serializable | Not Possible | Not Possible | Not Possible | |

*Table 2.4: ANSI SQL isolation levels defined in terms of the four phenomena*

object twice if another transaction $T_2$ overwrites $x$ between the two reads and commits. In contrast, the broad interpretation prevents any transaction from overwriting an object that has been read by a concurrent transaction. The same logic applies to *phantoms* but extended to predicate reads and writes. The question becomes: are both interpretations valid? Unfortunately, no. The authors show that preventing the three phenomena A1, A2, and A3 in their strict interpretation does not necessarily yield executions that are serializable (equivalent to a serial schedule). In contrast, preventing the three phenomena in their broad interpretation is sufficient to yield serializable executions[3]. Consider the execution in Figure 2.13, which involves a \$40 transfer between bank balance rows $x$ and $y$. $T_1$ is thus transferring \$40 from $x$ to $y$ and maintains a total balance of 100. $T_2$, however, reads the total balance to be \$60. This execution does not exhibit any of the anomalies A1, A2 or A3. It does however, exhibit phenomenon P1, and would thus be rejected in the broad interpretation. We summarise the final, correct definitions of the ANSI SQL isolation levels in terms of the four phenomena in Table 2.4.

Is the problem solved? Not quite. The original goal of the ANSI-SQL specification was to provide definitions independent of the underlying concurrency control mechanism. Yet, the authors acknowledge that the preventative definitions of the ANSI phenomena are a disguised redefinition of the earlier lock-based characterisation of isolation. The definitions they propose are *equivalent* to the lock-based schemes proposed by Gray et al. [86]. As Adya et al. [4] points out, these definitions are, as a result, overly restrictive. They fail to allow executions that one would think of as serializable. Consider the following three executions $H_0$, $H_1$ and $H_2$ in Figures 2.14, 2.15 and 2.16 respectively. $H_0$ does not even satisfy ANSI read uncommitted as it violates P0 (dirty writes). Recall that Bernstein et al. [32] defined a history as multiversioned-serializable if its serialization graph is acyclic. If we

[3] as the authors acknowledge, this statement only holds under the assumption that there is a single version of every object in the system (we describe this in more detail in the next chapter)

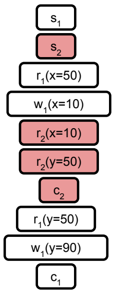


*Figure 2.13: Anomaly approach error*

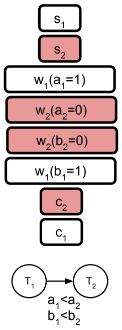


*Figure 2.14: Rejected serializable schedule H0*

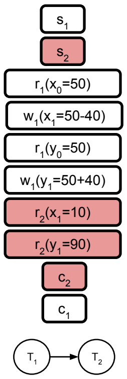


*Figure 2.15: Rejected serializable schedule H1*

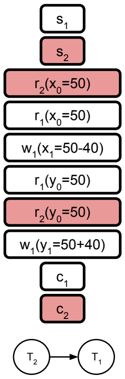


*Figure 2.16: Rejected serializable schedule H2*

look at the matching serialization graph of $H_0$, we see that it is acyclic; hence $H_0$ is serializable. It is simply necessary to re-order its writes (which both conflict-serializability and view-serializability disallows). The same logic applies to $H_1$ and $H_2$: they respectively violate P1 ($T_2$ reads uncommitted values) and P2 ($T_1$ overwrites $x$ and $y$ that have already been read by uncommitted $T_2$), yet they are multiversion-serializable as their serialization graphs are acyclic.

**Adding optimism** As Adya et al. [4] point out, the preventative interpretation rules out any execution in which conflicting operations execute concurrently. In effect, it rules out any implementation that could not arise in a lock-based implementation. Namely, it disallows executions that could arise in a multiversioned or optimistic concurrency control. Adya and his co-authors propose to address these shortcomings in a framework that extends the theory of multiversioned serialization graphs to weak isolation guarantees.

Adya's model is expressed in terms of *histories*, which consist of two parts: a partial order of events that reflect the operations of a set of transactions, and a version order that imposes a total order on committed object versions. Every history is associated with a *directed serialization graph* DSG(H) [32], whose nodes consist of committed transactions and whose edges mark the conflicts (read-write, write-write, or write-read) that occur between them:

- **Write-write conflict** $T_i$ writes a version of $x$, and $T_j$ writes the next version of $x$, denoted as $T_i \xrightarrow{ww} T_j$

- **Write-read conflict** $T_i$ writes a version of $x$, and $T_j$ reads the version of $x$ $T_i$ writes, denoted as $T_i \xrightarrow{wr} T_j$
- **Read-write conflict** $T_i$ reads a version of $x$, and $T_j$ writes the next version of $x$, denoted as $T_i \xrightarrow{rw} T_j$

For specific isolation levels, Adya further augments the model with logical start and commit timestamps for transactions, leading to *start-ordered serialization graphs* ($SSG(H)$) that add start-dependency edges to the nodes and edges of the corresponding DSG(H) (two transactions $T, T'$ are start-ordered if the commit timestamp of one precedes the start timestamp of the other).We note that Berstein's theory of multiversioning applies to Adya's cycle-based framework: a non-serializable history has a cyclic serialization graph while a serializable execution's DSG is acyclic. Berstein's framework, however, cannnot capture the differences between weak isolation guarantees.

Adya's framework instead can express the subtle differences between isolation levels in terms of *specific* cycles that are proscribed at each level. The model defines several phenomena:

- **G0** A cycle consisting of write-write edges
- **G1(a)** (Aborted read) A transaction $T_1$ reads a value produced by an aborted transaction $T_2$
- **G1(b)** (Intermediate read) A transaction $T_1$ reads a version of an object *x* written by a transaction $T_2$ that $T_2$ subsequently overwrites.
- **G1** A cycle consisting of any write-write/write-read edges
- **G2** A cycle consisiting of any write-write/write-read/read-write edges
- **G-SI(a)** A write-write/write-read edge without a corresponding start-edge
- **G-SI(b)** A cycle consisting of any number of write-write/write-read/start edges and a single read-write edge.

As before, an execution satisfies a given isolation level if it disallows specific phenomena. Read-uncommitted then disallows cycles consisting only of write-write edges in the DSG(H) (G0)[5]. It

[5] We will use Adya's shorthand for this and other phenomena in §3.2, when we prove that our new state-based definitions of isolation guarantees are equivalent to his.

| Isolation | Proscribed phenomena |
|---|---|
| Read uncommitted | G0 |
| Read committed | G0, G1 |
| Snapshot isolation | G0, G1, G-SI |
| Serializable | G0, G1, G2 |

*Table 2.5: Adya - Proscribed phenomena*

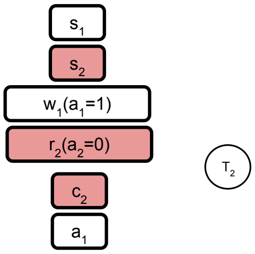


*Figure 2.17: Valid under read-uncommitted*

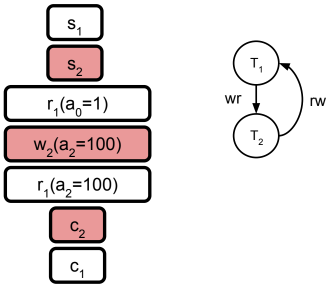


*Figure 2.18: Valid under read-committed*

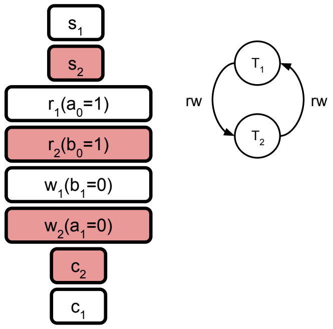
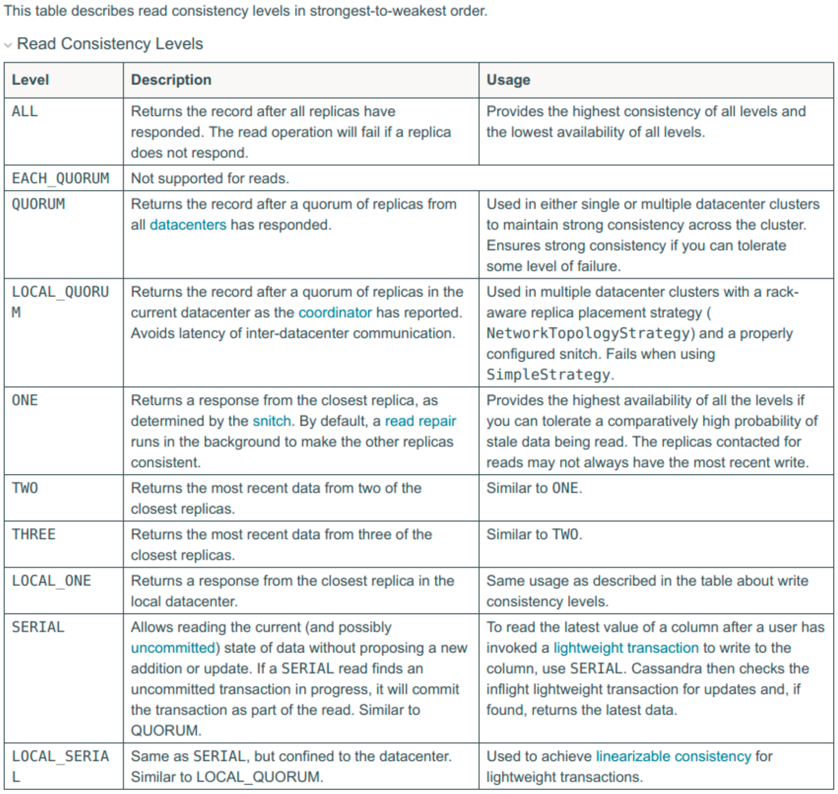


*Figure 2.19: Valid under snapshot isolation*

places no constraints on reads. All remaining ANSI SQL isolation levels disallow cycles consisting of write-write/write-read edges, as well as intermediate reads and aborted reads (phenomenon G1). Serializability also disallows cycles that include read-write edges (G2). In contrast, snapshot isolation disallows write-write/write-read edges without corresponding start edges (G-SI(a)) as well as cycles containing a single read-write edge in the SSG(H) (G-SI(b)). We summarise the model in Table 2.5.

To provide more intuition, consider the following three executions in Figures 2.17, 2.18 and 2.19. The execution in Figure 2.17 is allowed by read-uncommitted but disallowed by read-committed (it reads an aborted read - this is an instance of the). $T_1$ aborts and is consequently not shown in the serialization graph. The execution violates G1(a), the aborted read property. The execution in Figure 2.18 is allowed by read-committed but disallowed by snapshot isolation: it exhibits phenomenon G-SI(b), a write-write/write-read cycle with a single read-write dependency edge. This an instance of the non-repeatable read anomaly illustrated in §2.2.3. Finally, the schedule in Figure 2.19 is allowed by snapshot isolation but disallowed by serializability (it exhibits a cycle consisting of rw edges - this is an instance of the write-skew anomaly illustrated in §2.2.3).

Adya's formalism, like its other existing counterparts, specifies isolation guarantees as constraints on



This table describes read consistency levels in strongest-to-weakest order.

⌄ Read Consistency Levels

| Level | Description | Usage |
|---|---|---|
| ALL | Returns the record after all replicas have responded. The read operation will fail if a replica does not respond. | Provides the highest consistency of all levels and the lowest availability of all levels. |
| EACH_QUORUM | Not supported for reads. | |
| QUORUM | Returns the record after a quorum of replicas from all datacenters has responded. | Used in either single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure. |
| LOCAL_QUORUM | Returns the record after a quorum of replicas in the current datacenter as the coordinator has reported. Avoids latency of inter-datacenter communication. | Used in multiple datacenter clusters with a rack-aware replica placement strategy ( NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy. |
| ONE | Returns a response from the closest replica, as determined by the snitch. By default, a read repair runs in the background to make the other replicas consistent. | Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write. |
| TWO | Returns the most recent data from two of the closest replicas. | Similar to ONE. |
| THREE | Returns the most recent data from three of the closest replicas. | Similar to TWO. |
| LOCAL_ONE | Returns a response from the closest replica in the local datacenter. | Same usage as described in the table about write consistency levels. |
| SERIAL | Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the transaction as part of the read. Similar to QUORUM. | To read the latest value of a column after a user has invoked a lightweight transaction to write to the column, use SERIAL. Cassandra then checks the inflight lightweight transaction for updates and, if found, returns the latest data. |
| LOCAL_SERIAL | Same as SERIAL, but confined to the datacenter. Similar to LOCAL_QUORUM. | Used to achieve linearizable consistency for lightweight transactions. |

*Figure 2.20: Read Consistency Levels - Cassandra*

the ordering of the read and write operations that the storage system performs, and relies on low-level implementation details like timestamps or version order. Unfortunately, applications cannot directly observe this ordering: to them, the storage system is a black box. All they can observe are the values returned by the read operations they issue: they experience the storage system as if it were going through a sequence of atomic state transitions, of which they observe a subset. To make it easier for applications to reason about different levels of isolation, this dissertation adopts the viewpoint of the applications that must ultimately use their guarantees and introduce a new formalization of isolation based on application-observable states. We present our new model of isolation in Chapter 3.

## 2.3 But what about distributed system consistency?

The introduction to this thesis defined two notions of correctness for transactional datastore:

- **Isolation** defines a contract that regulates the interaction between concurrent transactions.

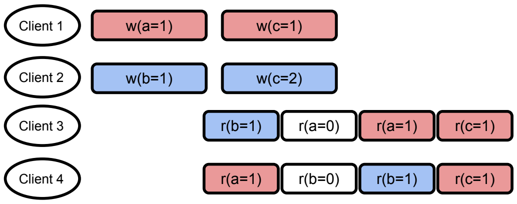


*Figure 2.21: Write anomalies*

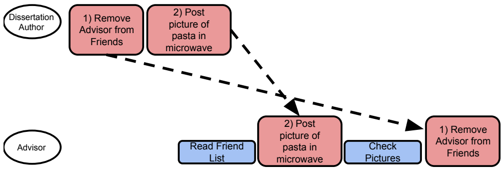

Dissertation Author
1) Remove Advisor from Friends
2) Post picture of pasta in microwave
Advisor
Read Friend List
2) Post picture of pasta in microwave
Check Pictures
1) Remove Advisor from Friends

*Figure 2.22: Causal ordering anomaly*

- **Consistency** defines a contract that regulates the ordering of operations across replicas and places constraints on how much their state can diverge[6]

Isolation guarantees thus typically do not regulate how transactions should be ordered; they simply regulate how their operations become visible to concurrent transactions. Consistency guarantees, in contrast, exclusively constrain the ordering of operations with respect to all other operations. They cannot, however, reason about transactional constructs. The reason behind these orthogonal approaches to correctness is mainly historical. Isolation was defined in the context of centralised database systems, where it was assumed that transactions from the same client would be processed in the order in which they were executed. Consistency, instead, was primarily defined in the context of shared multiprocessor systems that had no notion of multiword atomic operations (in other words, no transactions), but whose different caches could cause operations to read stale values, or take effect out-of-order at different processors. The notion of multiple, possibly incoherent caches, translates well to large-scale distributed systems, and the distributed system community adopted that terminology.

In spite of these apparent ideological differences, the process through which consistency has been formalised mirrors almost exactly that of isolation. The physical constraints imposed by network latencies and network partitions in large-scale distributed systems has also fostered the growth of a myriad of consistency guarantees [192]. The CAP theorem [39, 79] highlights that one must make a trade-off between coordination (and consequently how much replicas can diverge) and partition-tolerance. In light of the CAP theorem, many wide-area services and applications [44] choose to renounce strong consistency and focus instead on prodiving the ALPS properties [118] of **A**vailability,

[6] Unfortunately, the term consistency is overloaded: the database community defines consistency (the C in ACID) as a database satisfying application invariants. The distributed system community in contrast, uses the term consistency to specify constraints on operation ordering. For clarity, we will refer to the distributed system notion of consistency as "consistency", and to the database notion of consistency as "database consistency".

low **L**atency, **P**artition tolerance and high **S**calability. Cassandra alone, for instance, offers ten possible consistency levels for read operations (Figure 2.20). Systems like Yahoo's PNUTS [54] or Microsoft's DynamoDB [66] offer similar choices. Much like weak isolation, these guarantees balance ease of programming with performance. Strong consistency guarantees provide users with the abstraction that they are executing on a logically centralised replica (much like serializability provides users with the abstraction that each transaction is executing serially) but are slow. Weaker consistency levels offer lower latency and higher throughput, but introduce consistency anomalies, which are ordering of operations that are inconsistent with the execution of a non-replicated system.

This section does not attempt to provide a full summary of all existing definitions (Vukolic et al. elegantly summarises the vast majority of them in [192] - we reproduce it in Figure 2.23). Instead, the section summarises the possible types of anomalies that can arise in weakly consistent systems and outlines three main issues with current formalisms.

### 2.3.1 Consistency Anomalies

Consistency definitions, unlike isolation guarantees, center around the notion of a client. Each client executes a totally ordered sequence of (read or write) operations in what is known as program order (or client order) [111]. We note that some consistency guarantees center around the notion of a replica. We describe later in the next section why this is suboptimal and choose the client-based formalism here.

**Real-Time Ordering** Isolation guarantees allow for a transaction executed on a Wednesday to be serialized before a transaction executed on the following Friday. A system without real-time ordering constraints can return arbitrarily stale data. Distributed systems do not operate in a vacuum and interact with the outside world, they must somehow remain synchronised with real time; a weather forecast system that always returns the weather from January 1st, 2000 only has limited utility. Consistency guarantees like linearizability [92] or bounded staleness [] prevent these anomalies. Guarantees like sequential consistency [111] or causal consistency [9] do not.

**Total Ordering** The goal standard for consistency is to give users the abstraction that they are executing with a single logical computer that processes all operations. However, many distributed systems allow writes to be processed independently at different sites and replicated asynchronously. This can cause different clients to observe writes as taking effect in different orders. Consider the example in Figure 2.21. Client 1 issues a write to $a$ concurrently with Client 2's write to $b$. Client 3 perceives the write to $a$ as taking effect before $b$. The opposite is true for Client 4. Consistency guarantees like linearizability [92] or sequential consistency [111] prevent these anomalies. Others like causal consistency [9] or session guarantees do not.

**Causal Ordering** Asynchronous replication may cause updates from the same client to reach other clients in reverse order. This can cause correctness bugs if the clients' updates are related. To illustrate, consider the example in Figure 2.22. This dissertation's author, who connects to a photo-sharing application through one replica, wants, before posting pictures of herself pasta cooking in the microwave, to defriend her advisor, who accesses the application through a different replica [24, 54, 118]. Accordingly, this dissertation's author first defriends her advisor and only then uploads the photos. Unfortunately, these operations, originally performed on replica A, are replicated on B in the opposite order: site B first receives the photos update, and then the defriending update, allowing the advisor to see the photos nonetheless[7]. Guarantees like causal consistency prevent these anomalies. Others like session guarantees or eventual consistency do not.

**Write-write conflicts** Traditionally, guarantees that impose a total order of operations (linearizability, sequential consistency) are considered "strong" consistency guarantees while those that do not are considered "weak". Weak consistency levels usually allow write operations to be processed independently at different sites. Without systematic coordination, geographically distinct replicas can issue conflicting operations that may cause replicas' states to diverge, or operations to be lost. Consider again the example in Figure 2.21: clients 1 and 2 issue a write to object $c$ concurrently. What should happen to these operations? Which write is the "correct" one? In the example, we apply the same strategy as the one used in COPS [118] of arbitrarily choosing one update (client 1's update here), causing the loss of client 2's update.

**System-based Ordering** System-based ordering notions use shards of replicas as the basic unit of ordering in the system. They guarantee, for instance, that all operations within a partition or replica will be totally ordered [60, 118, 178].

[7] and one should not underestimate the dangers of showing pasta cooked in the microwave to one's Italian advisor

### 2.3.2 Issues with current formalisms

We identify three primary issues with current consistency formalisms in large-scale distributed systems:

**System-specificity** First, many consistency guarantees rely on the system-based ordering we described above [60, 118, 135, 178]. This approach is suboptimal for three reasons: first, it reveals a system detail that the client is not necessarily aware of. It requires knowledge of system internals that is not necessarily available. We highlight in Chapter 4 that understanding the specific guarantee provided by DocumentDB requires knowledge of how writes are ordered in the system. This information is not readily available in the documentation. In fact, this dissertation's author asked the question on StackOverflow, and the DocumentDB developers could not answer for proprietary reasons. Second, it constrains functionality: system-based ordering based on shards or replicas prevents resharding or requires clients to be sticky [129] (clients must remain attached to the same replica). Finally, it often precludes more optimistic implementations of certain definitions [129].

**Consistency for transactions** Second, cloud storage systems are increasingly adding support for transactions [16, 66, 134, 138, 155], it is currently not clear what correctness guarantee the resulting systems actually offer, as there lacks a unified framework for understanding consistency and isolation. This lack of unified framework has also led to confusion in the database community: new isolation guarantees defined in the context of replicated systems have incorporated limited notions of consistency in ad-hoc ways. For instance, the new parallel snapshot isolation definition introduced by Sovran et al. [178] turned out to be equivalent to the PL-2+ definition previously introduced by Adya [4] (more on this in Chapter 3).

**Write-write conflicts** Consistency guarantees constrain the ordering of read and write operations. They remain silent, however, on how to handle the write-write conflicts that naturally arise in distributed systems. The previous section illustrated one such anomaly: Alice's update was lost. There is currently no clean way to reason about such conflicts. As we describe further in Chapter 5, current systems suffer from two main limitations. First, to maintain the abstraction of sequential storage, systems often use fixed, syntactic resolution policies to reconcile write-write conflicts. This is the case for instance in Lloyd et al's COPS system [118]. The system relies on a *deterministic writer* *wins* strategy. These policies are ill-suited to deal with real-world semantics as they can arbitrarily lose updates. Second, current systems ignore the cross-object semantics that naturally arise in real applications. Systems like Bayou [68], Dynamo [62], Ficus [91] or Coda [100], even though they support more flexible merging policies, resolve conflicts at the granularity of a single object. This is insufficient to fully resolve the effects of write-write conflicts, as we detail in Chapter 5.

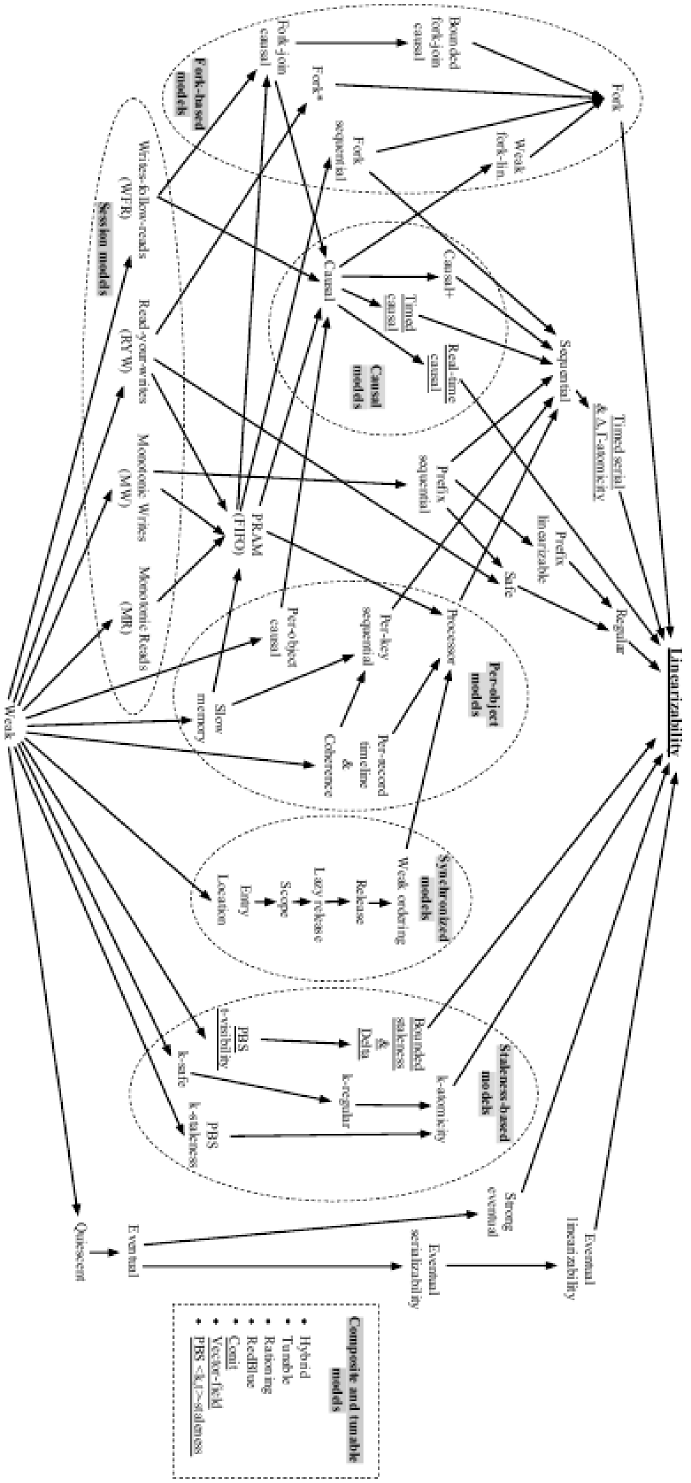


*Figure 2.23: Hierarchy of consistency models (reproduced from Vukolic et al. [192])*

# Chapter 3

# A new model of isolation

**The context** As highlighted in Chapter 1, modern applications increasingly offload the managing of data to cloud-based replicated and/or distributed systems. These systems, which often span multiple regions or continents, must sustain high-throughput, guarantee low-latency, and remain available across failures. To mitigate the increased programming complexity that comes from managing geo-replication, scalability and fault-tolerance, commercial databases and distributed storage systems [29, 83, 84, 132, 133, 135, 148, 150, 158] interact with applications through a front-end that gives applications the illusion of querying or writing to a logically centralized, failure-free node. This node will simply scale as much as one's wallet will allow [83, 84, 132, 133, 150]. PaaS (Platform as a Service) cloud-based storage systems [83, 133], databases [84] or webservers [12], for example, simply require applications to pay for guaranteed throughput [84, 133].

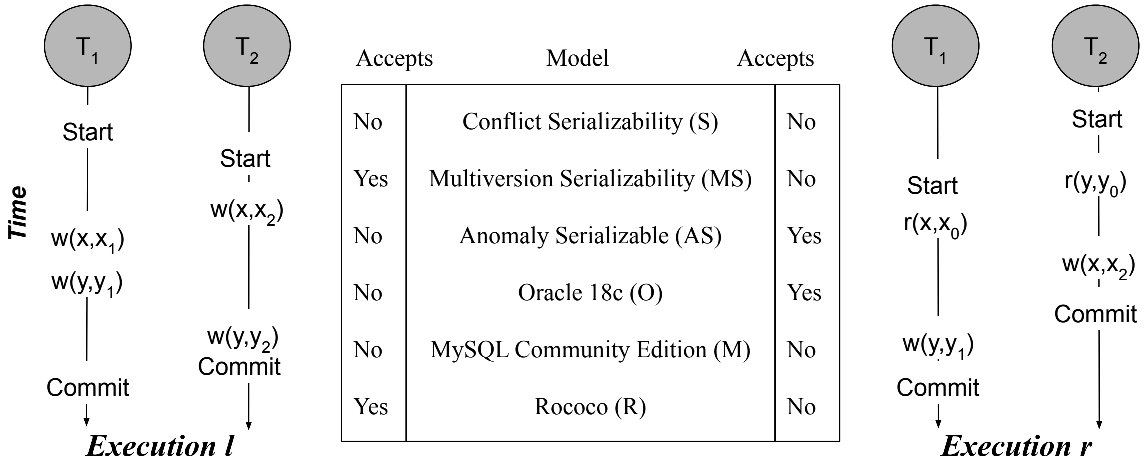
Chapter 2 summarised how, to meet performance and scalability demands, commercial databases or distributed storage systems like MySQL [148], Oracle [150], or SQL Server [136] often give up serializability [153] and instead privilege weaker but more scalable correctness criteria [4, 31, 105, 148, 157, 158, 178, 204] (referred to as *weak isolation*) such as snapshot isolation [31] or read committed [31]. In fact, to the best of our knowledge, almost all SQL databases use read committed as their default isolation level [132, 136, 148, 150, 158, 168], with some only supporting read-committed or snapshot isolation [148, 168][1].

[1] As of June 2019

While necessary for performance, this trend poses an additional burden on the application programmer, as these weaker isolation guarantees allow for counter-intuitive application behaviors: relaxing the ordering of operations yields better performance, but introduces schedules and anomalies that could not arise if transactions executed atomically and sequentially. These anomalies may affect application logic; in §2.2.3, this thesis sketched out a number of executions that could arise under weak isolation but not serializability. These anomalies (dirty reads, non-repeatable reads, write skew, etc.) could break application logic.

Ensuring that the application remains correct with weak isolation is challenging. A careful understanding of the system that implements a given isolation level is oftentimes *necessary* to determine which anomalies the system will admit and how these will affect application correctness. The supposedly clean abstraction that the front-end sought to expose is in fact leaky. Worse, it actually obscures details that are necessary to understand what guarantees a given isolation definition offers.

Indeed, the guarantees provided by isolation levels are often dependent on specific and occasionally implicit system properties—be it properties of storage (e.g., whether it is single or multiversioned [32]); of the chosen concurrency control (e.g., whether it is based on locking or timestamps [31]); or other system features (e.g., the existence of a centralized timestamp [73]).



| Accepts | Model | Accepts |
|---|---|---|
| No | Conflict Serializability (S) | No |
| Yes | Multiversion Serializability (MS) | No |
| No | Anomaly Serializable (AS) | Yes |
| No | Oracle 18c (O) | Yes |
| No | MySQL Community Edition (M) | No |
| Yes | Rococo (R) | No |

*Figure 3.1: Serializability. Abbreviations refer to: S[153], MS[32], AS[31] O[150], M[148], R[139].*

Recall for example serializability [153]: as we summarised in Chapter 2, the original ANSI SQL specification states that guaranteeing serializability is equivalent to preventing four phenomena [31].

This equivalence, however, only holds for lock-based, single version databases. In a multi-versioned system, preventing the four phenomena is insufficient to guarantee serializability. Consider for example the schedule Figure 3.1(r): $T_1$ missed $T_2$'s write while $T_2$ misses $T_1$'s write. It is thus clearly not serializable, yet admits none of the four phenomena. Alternatively, consider the schedule Figure 3.1(l): conflict and view serializability disallow it while multiversioned serializability admits the execution. Such implicit dependencies continue to have practical consequences: current multiversioned commercial databases that prevent these four phenomena, such as Oracle 18c, claim to implement serializability, when they in fact implement the weaker notion of snapshot isolation [22, 73, 150]. In contrast, a majority reject the (serializable) schedule of Figure 3.1(l) because, for performance reasons, these systems choose not reorder writes.

**The problem** We submit that the root of this complexity is a fundamental semantic gap between how application programmers experience isolation guarantees and how they are currently formally defined. From a client's perspective, isolation guarantees are contracts between the storage systems and its clients, specifying the set of behaviors that clients can expect to observe—i.e., the set of admissible values that each read is allowed to return. When it comes to formally defining these guarantees, however, the current practice is to focus on the *mechanisms* that can produce those values—i.e., histories capturing the relative ordering of low-level read and write operations. Instead, the focus should be on the values that the clients can observe.

Expressing isolation in a system-centric way and at such a low level of abstraction has significant drawbacks. First, it requires application programmers to reason about the ordering of operations that they cannot directly observe. Second, it makes it easy, as we have seen, to inadvertently contaminate what should be system-independent guarantees with system-specific assumptions. Third, by relying on operations that are only meaningful within one of the layers in the system's stack, it makes it hard to reason end-to-end about the system's guarantees.

**The secret sauce** To address these issues, this chapter proposes a new model that, for the first time, expresses isolation guarantees exclusively as *properties of states that applications can observe*, without relying on traditional notions—such as dependency graphs, histories, or version orders—that are instead invisible to applications. This new client-centric foundation comes at no cost in terms of generality or expressiveness: this thesis offers state-based and client-centric definitions of most modern isolation definitions, and prove that they are equivalent to their existing counterparts. It does, however, result in greater clarity, which yields significant benefits.

**The benefits** First, this model makes clear to developers what anomalies, if any, their applications can expect to observe, thus bridging the semantic gap between how isolation is experienced and how it is formalized. For example, we show (§3.3.1) how a state-based and client-centric definition brings immediately into focus the root cause of the write-skew anomaly, which distinguishes snapshot isolation from serializability.

Second, by removing the distorting effects of implementation artefacts, our approach makes it easy to compare the guarantees of distinct, but semantically close, isolation guarantees. The results are sometimes surprising. We prove (§3.3.2) that several well-known flavors of isolation in fact provide the same guarantees: *parallel snapshot isolation* (PSI) [46, 178] is equivalent to *lazy consistency* (PL-2+) [4, 5]; similarly, *generalized snapshot isolation* (GSI) [154] is actually equivalent to ANSI snapshot isolation (ANSI SI) [31], though GSI was proposed as a more scalable alternative to ANSI SI. Likewise, we also show that the lesser known *strong session SI* [61] and *prefix-consistent SI* [154] are also equivalent. Ultimately, the insights offered by state-based definitions enable us to organize in a clean hierarchy (§3.3.2) what used to be incomparable flavors of snapshot isolation [4, 19, 31, 61, 129, 154, 178].

Finally, by focusing on how clients perceive a given isolation guarantee, rather than on the mechanisms currently used to implement it, a state-based formalization can lead to a fresh, end-to-end perspective on how that guarantee should be implemented. Specifically, a state-based definition of parallel snapshot isolation (PSI) makes clear that the requirement of totally ordering transactions at each datacenter, which is baked into its current definition [178], is only an implementation artefact. Removing it offers the opportunity of an alternative implementation of PSI that makes it resilient to *slowdown cascades* [129], a common failure scenario in large-scale datacenters that has inhibited the adoption of stronger isolation models in industry [10].

**Roadmap** Chapter 2 reviewed the current approaches to formalizing isolation guarantees. We introduce our state-based, client-centric model in Section 3.1, and use it in Section 3.2 to define

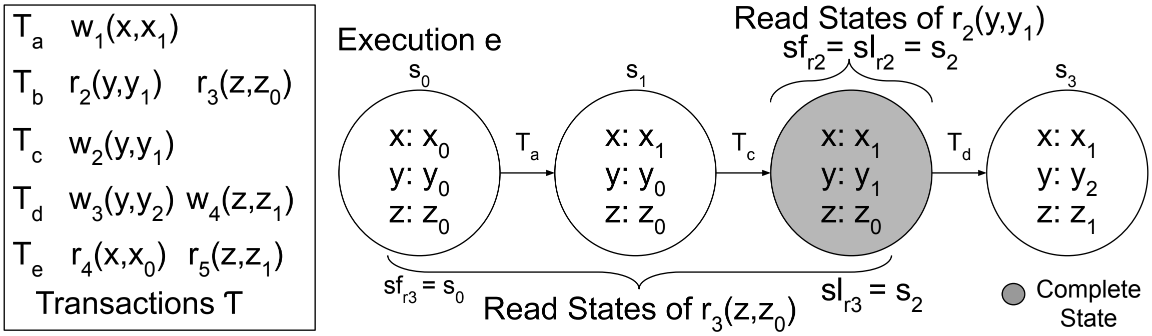


*Figure 3.2: Read States and execution.*

several isolation guarantees. We highlight the benefits of our approach in Section 3.3 and summarize related work in Section 3.4, before outlining our work's limitations in Section 3.5[2].

## 3.1 A State-based Model

### 3.1.1 Towards a new formalism

Chapter 2 summarised existing approaches to formalise isolation guarantees. It highlighted that isolation guarantees have been formalized in many different ways: as a function of schedule equivalence [153], using implementation-oriented operational specifications [25, 31, 178], or by relating the order in which transactions commit with the values that they observe [46, 47, 170]. The most prevalent approach, however, has been to formulate isolation guarantees as dependency graphs. Adya's formalism [4] was the first to define weak isolation guarantees in the context of these dependency graphs. Adya's specification has since been adopted as the de-facto language for specifying isolation [69, 129, 183, 195]. We select Adya's model as a baseline and prove our definitions equivalent to his in §3.2.

While popular, Adya's formalism, like its other existing counterparts, specified isolation guarantees as constraints on the ordering of the read and write operations that the storage system performs. His framework relies on low-level implementation details like timestamps or version order. Unfortunately, applications cannot directly observe this ordering: to them, the storage system is a black box. All they

[2] This work revises the previously published paper: *Seeing is Believing: A client-centric approach to isolation*, published at PODC 2017. Youer Pu contributed to defining and proving the isolation definitions

can observe are the values returned by the read operations they issue: they experience the storage system as if it were going through a sequence of atomic state transitions, of which they observe a subset.

To make it easier for applications to reason about different levels of isolation, we instead adopt the viewpoint of the applications that must ultimately use their guarantees; we introduce a new formalization of isolation based on *application-observable states*. To the best of our knowledge, our model is the first to specify isolation in an exclusively client-centric fashion, without relying on some notion of history. Instead, it associates with each transaction the set of candidate states (called *read states*) from which the transaction may have retrieved the values it read during its execution. Read states perform a role similar to Kripke structures [106]: they inform the application of the set of possible worlds (i.e., states) consistent with what a transaction observed during its execution.

### 3.1.2 Model Overview

Intuitively, a storage system guarantees a specific isolation level $I$ if it can produce an *execution* (a sequence of atomic state transitions) that satisfies two conditions. First, the execution must be consistent with the values observed by each transaction $T$; in our model, this requirement is expressed by associating every transaction $T$ with a set of read states, representing the states that the storage *could* have been in when the application executed $T$'s operations. Second, the execution must be valid, in that it must satisfy the constraints imposed by $I$; $I$ effectively narrows down which transactions' read states can be used to build an acceptable execution. If no read state proves suitable for some transaction, then $I$ does not hold.

### 3.1.3 Definitions

We now formally define the necessary concepts. They can be grouped as follows: our model refers to a *storage system* that executes *transactions* consisting of *operations*. This storage system transitions through a series of *states* as a result of executing these transactions. Transactions observe a subsect of those *states*. Different states have differerent properties, that our model will subsequently use to define the various isolation guarantees.

**Storage system** We define a *storage system S* with respect to a set $\mathcal{K}$ of keys and $\mathcal{V}$ of values; a *system state s* is a unique mapping from keys to values produced by writes from aborted or committed transactions. For simplicity, we assume that each value is uniquely identifiable, as is common practice both in existing formalisms [4, 32] and in practical systems (ETags in Azure [133] and S3 [15], timestamps in Cassandra [16]). There can thus be no ambiguity, when reading an object, as to which transaction wrote its content. In the initial system state, all keys have value $\bot$; later states similarly include every key, possibly mapped to $\bot$.

**Transactions** As is common in database systems, we assume that applications modify the storage system's state using transactions. A *transaction* $T$ is a tuple $(\Sigma_T, \xrightarrow{to})$, where $\Sigma_T$ is the set of *operations* in $T$, and $\xrightarrow{to}$ is a total order on $\Sigma_T$[3]. Operations can be either reads or writes. *Read* operation $r(k, v)$ retrieves value *v* by reading key *k*; *write* operation $w(k, v)$ updates $k$ to its new value $v$. The *read set* of $T$ comprises the keys read by $T$: $\mathcal{R}_T = \{k | r(k, v) \in \Sigma_T\}$. Similarly, the *write set* of $T$ comprises the keys that $T$ updates: $\mathcal{W}_T = \{k | w(k, v) \in \Sigma_T\}$. For simplicity of exposition, we assume that a transaction only writes a key once.

**Time** Finally, we assume the existence of a time oracle $\mathcal{O}$ that assigns distinct real-time *start* and *commit* timestamps ($T.start$ and $T.commit$) to every transaction $T \in \mathcal{T}$. A transaction $T_1$ time-precedes $T_2$ (we write $T_1 <_s T_2$) if $T_1.commit < T_2.start$.

**State Transition** Applying a transaction $T$ to a state *s* transitions the system to a state $s'$ that is identical to *s* in every key except those written by $T$. Formally,

**Definition 1** $s \xrightarrow{T} s' \equiv \Big(\big([(k, v) \in s' \wedge (k, v) \notin s] \Rightarrow k \in \mathcal{W}_T\big) \wedge \big(w(k, v) \in \Sigma_T \Rightarrow (k, v) \in s'\big)\Big).$

We refer to *s* as the *parent state* of $T$ (denoted as $s_{p,T}$) [4]; to the transaction that generated *s* as $T_s$; and to the set of keys in which *s* and $s'$ differ as $\Delta(s, s')$.

**Execution** An *execution* $e$ for a set of transactions $\mathcal{T}$ is a totally ordered set defined by the pair $(\mathcal{S}_e, \xrightarrow{T \in \mathcal{T}})$, where $\mathcal{S}_e$ is the set of states generated by applying, starting from the system's initial state, a permutation of all the transactions in $\mathcal{T}$. We write $s \xrightarrow{*} s'$ (respectively, $s \xrightarrow{+} s'$) to denote a sequence of zero (respectively, one) or more state transitions from *s* to $s'$ in *e*. For example, in

[3] Bernstein et al. in his formalism uses a partial order to specify a transaction. For simplicity, we consider a linearisation of the total order here

[4] Henceforth, we will drop the subscripted $T$ unless there is ambiguity.

Figure 3.2, $\mathcal{T}$ comprises five transactions, operating on a state that consists of the current version of keys $x$, $y$, and $z$.

**Read States** Note that while $e$ identifies the state transitions produced by each transaction $T \in \mathcal{T}$, it does not specify from which states in $\mathcal{S}_e$ each operation in $T$ reads. In particular, reading a key in replicated distributed systems will not necessarily return the value produced by the latest write to that key, as writes may become visible in different orders at different replicas. In general, multiple states in $\mathcal{S}_e$ may be compatible with the value returned by any given operation. We call this subset the operation's *read states*. To prevent operations from *reading from the future*, we restrict the valid read states for the operations in $T$ to be no later than $s_p$. Further, once an operation in $T$ writes $v$ to $k$, we require all subsequent operations in $T$ that read $k$ to return $v$ [4]: in this case, their set of read states by convention includes all states in $\mathcal{S}_e$ up to and including $s_p$.

**Definition 2** *Given an execution $e$ for a set of transactions $\mathcal{T}$, let $T \in \mathcal{T}$ and let $s_p$ denote $T$'s parent state. The read states for a read operation $o = r(k,v) \in \Sigma_T$ define the set of states* $\mathcal{RS}_e(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p \wedge \big((k,v) \in s \vee (\exists w(k,v) \in \Sigma_T : w(k,v) \xrightarrow{to} r(k,v))\big)\}$.

Figure 3.2 illustrates the notion of read states for the operations executed by transaction $T_b$. Since $r_2$ returns $y_1$, its only possible read state is $s_2$, i.e., the only state containing $y_1$. When it comes to $r_3$, however, $z_0$ could have been read from any of $s_0$, $s_1$, or $s_2$: from the perspective of the client executing $T_b$, these read states are indistinguishable. By convention, write operations have read states too: for a write operation in $T$, they include all states in $\mathcal{S}_e$ up to and including $T$'s parent state. It is easy to prove that the read states of any operation $o$ define a subsequence of contiguous states in the total order that $e$ defines on $\mathcal{S}_e$. We refer to the first state in that sequence as $sf_o$ and to the last state as $sl_o$. For instance, in Figure 3.2, $sf_{r_3}$ is $s_0$ (the first state that contains $z_0$) and $sl_{r_3}$ is $s_2$ ($z_0$ is overwritten in $s_3$). When the predicate $\textsc{preread}_e(\mathcal{T})$ holds, then such states exist for all transactions in $\mathcal{T}$:

**Definition 3** *Let* $\textsc{preread}_e(T) \equiv \forall o \in \Sigma_T : \mathcal{RS}_e(o) \neq \emptyset$.
*Then* $\textsc{preread}_e(\mathcal{T}) \equiv \forall T \in \mathcal{T} : \textsc{preread}_e(T)$.

**Complete States** We say that a state $s$ is *complete* for $T$ in $e$ if every operation in $T$ can read from $s$. We write:

**Definition 4** *COMPLETE*$_{e,T}(s) \equiv s \in \bigcap_{o \in \Sigma_T} \mathcal{RS}_e(o)$.

Looking again at Figure 3.2, $s_2$ is a complete state for transaction $T_b$, as it is in the set of candidate read states of both $r_2(y, y_1)$ ($\{s_2\}$) and $r_3(z, z_0)$ ($\{s_0, s_1, s_2\}$). A complete state is not guaranteed to exist: no such state exists for $T_e$, as the sole candidate read states of $r_4$ and $r_5$ (respectively, $s_0$ and $s_3$) are distinct. As we will see in §3.2, complete states are key to determining whether transactions read from a consistent snapshot.

## 3.2 Formalising Isolation

| | |
|---|---|
| Serializability (CT$_{SER}(T, e)$) | COMPLETE$_{e,T}(s_p)$ |
| Snapshot Isolation (CT$_{SI}(T, e)$) | $\exists s \in S_e$.COMPLETE$_{e,T}(s) \wedge$ NO-CONF$_T(s)$ |
| Read Committed (CT$_{RC}(T, e)$) | PREREAD$_e(T)$ |
| Read Uncommitted (CT$_{RU}(T, e)$) | True |
| Parallel Snapshot Isolation (CT$_{PSI}(e, T)$) | PREREAD$_e(T) \wedge \forall T' \ \triangleright T : \forall o \in \Sigma_T : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$ |
| Strict Serializability (CT$_{SSER}(e, T)$) | COMPLETE$_{e,T}(s_p) \wedge \forall T' \in \mathcal{T} : T' <_s T \Rightarrow s_{T'} \xrightarrow{*} s_T$ |
| Read Atomic (CT$_{RA}(e, T)$) | $\forall r_1(k_1, v_1), r_2(k_2, v_2) \in \Sigma_T \wedge k_2 \in \mathcal{W}_{T_{sf_{r_1}}} \Rightarrow sf_{r_1} \xrightarrow{*} sf_{r_2}$ |

*Table 3.1: Commit Tests*

Isolation guarantees specify the valid set of executions for a given set of transactions $\mathcal{T}$. We show that it is possible to formalize these guarantees solely in terms of each transaction's read and parent states, without relying on histories of low-level operations or on implementation details such as timestamps. The underlying reason is simple: ultimately, it is through the visible states produced during an execution that the storage system can prove to its users that a given isolation guarantee holds. Histories are just the mechanism that generates those probatory states; indeed, multiple histories can map to the same execution.

In a state-based model, isolation guarantees constrain each transaction $T \in \mathcal{T}$ in two ways. First, they limit which states, among those in the candidate read sets of the operations in $T$, are admissible. Second, they restrict which states can serve as parent states for $T$. We express these constraints by means of a *commit test*: for an execution $e$ of a set $\mathcal{T}$ of transactions to be valid under a given isolation level $\mathcal{I}$, each transaction $T$ in $e$ must satisfy the commit test CT$_{\mathcal{I}}(T, e)$ for $\mathcal{I}$.

**Definition 5** *Given a set of transactions $\mathcal{T}$ and their read states, a storage system satisfies an isolation* *level $\mathcal{I}$ iff* $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{\mathcal{I}}(T, e)$.

Table 3.1 summarizes the commit tests that define the isolation guarantees most commonly-used in research and industry: the ANSI SQL isolation levels (serializability, read committed, read uncommitted, and snapshot isolation) as well as parallel snapshot isolation [46, 178], strict serializability [153], and the recently proposed read atomic [25] isolation level. Though our state-based definitions make no reference to histories, we prove that they are equivalent to those in Adya's classic treatment. As the proofs follow a similar structure, we provide an informal proof sketch only for serializability and snapshot isolation, deferring a more complete and formal treatment to Appendices A, B and E.

**Serializability** Serializability requires the values observed by the operations in each transaction $T$ to be consistent with those that would have been observed in a sequential execution. The commit test enforces this requirement through two complementary conditions on observable states. First, all of $T$'s operations must read from the same state $s$ (i.e., $s$ must be a complete state for $T$). Second, $s$ must be the parent state of $T$, i.e., the state that $T$ transitions from. These two conditions suffice to guarantee that $T$ will observe the effects of all transactions that committed before it. This definition is equivalent to Adya's cycle-based definition. Specifically, we prove that (a more formal proof can be found in Appendix A.2):

**Theorem 1** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{SER}(T, e) \equiv \neg \mathit{G1} \land \neg \mathit{G2}$.

*Proof sketch.* ($\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{SER}(T, e) \Rightarrow \neg \mathrm{G1} \land \neg \mathrm{G2}$). By definition, $e$ is a totally-ordered execution where the parent state of every transaction $T$ is a complete state for $T$. Considering the order of transactions in $e$, we make three observations. First, all write-write edges in the DSG point in the same direction, as they map to state transitions in the totally-ordered execution $e$. Second, all write-read edges point in the same direction as write-write edges: given any transaction $T$, since all operations in $T$ read from $T$'s parent state, all write-read edges that end in $T$ must originate from a transaction that precedes $T$ in $e$'s total order. Finally, all read-write dependency edges point in the same direction as write-write and write-read edges: as all read operations in $T$ read from $T$'s parent state, the value they return cannot be later overwritten by a transaction $T'$ ordered before $T$ in $e$. Since all edges point in the same direction, no cycle can form in the DSG.

($\exists e : \forall T \in \mathcal{T} : \text{CT}_{SER}(T, e) \Leftarrow \neg\text{G1} \wedge \neg\text{G2}$). If no cycle exists in the DSG, we can construct an execution $e'$ such that the parent state $s_p$ of each transaction $T$ is a complete state for $T$. We construct $e'$ by topologically sorting the DSG (it is acyclic) and by applying every transaction in the resulting order. Thus, if a transaction $T'$ writes a value that $T$ subsequently reads (write-read edge), the state associated with $T'$ is guaranteed to precede $T$'s state in the execution $e'$. Moreover, as there are no backpointing read-write edges, no other transaction in $e'$ will update an object read by $T$ between the state produced by $T'$ and $s_p$. $s_p$ is therefore a valid read state for every operation in $T$ and, consequently, a complete state for $T$. □

**Snapshot isolation (SI)** Like serializability, SI prevents transaction $T$ from seeing the effects of concurrently running transactions. The commit test enforces this requirement by having all operations in $T$ read from the same state $s$, produced by a transaction that precedes $T$ in the execution $e$. However, SI no longer insists on that state $s$ being $T$'s parent state $s_p$: other transactions, whose operations $T$ will not observe, may commit in between $s$ and $s_p$. The commit test only forbids $T$ from modifying any of the keys that changed value as the system's state progressed from $s$ to $s_p$. Denoting the set of keys in which $s$ and $s'$ differ as $\Delta(s, s')$, we express this as $\text{NO-CONF}_T(s) \equiv \Delta(s, s_p) \cap \mathcal{W}_T = \emptyset$. We prove that this definition is equivalent to Adya's (a more formal proof can be found Appendix A.3):

**Theorem 2** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{SI}(T, e) \equiv \neg$*G1* $\wedge \neg$*G-SI.*

*Proof sketch.* ($\exists e : \forall T \in \mathcal{T} : \text{CT}_{SI}(T, e) \Leftarrow \neg\text{G1} \wedge \neg\text{G-SI}$). We construct a valid execution for any history satisfying $\neg$G1 $\wedge$ $\neg$G-SI using the time-precedes partial order introduced by Adya. First, we topologically sort transactions according to their commit point, and apply them in the resulting order to generate an execution $e$. Next, we prove that every transaction $T$ satisfies the commit test: we first show that the state created by the last transaction $T_{rs}$ on which $T$ start-depends is a complete state. As $T$ start-depends on $T_{rs}$, it must also start-depend on all transactions that precede $T_{rs}$, since, by construction, these transactions have a commit timestamp smaller than $T_{rs}$. Moreover, as $T_{rs}$ is the last transaction that $T$ starts-depend on, all subsequent transactions will either be concurrent with $T$, or start-depend on $T$. Adya's ($\neg$G-SI) requirement (formally, that there cann be neither a write-read /write-write edge without also a start dependency edge nor a cycle including a single read-write edge) implies that $T$ can only read or overwrite a value written by a transaction $T'$ if

$T$ start-depends on $T'$. Any such $T'$ must either be $T_{rs}$ or precede $T_{rs}$ in $e$. Similarly, if another transaction $T''$ overwrites a value that $T$ reads, $T$ cannot start-depend on $T''$ as it would otherwise create a cycle with a start-edge and a single read-write edge. $T''$ is therefore ordered after $T_{rs}$ in $e$. We conclude that $s_{T_{rs}}$ necessarily contains all the values that $T$ reads: it is a complete state. Next, we show that $\Delta(s_{T_{rs}}, s_T) = \emptyset$. By construction, $T$ cannot start-depend on any transaction $T'$ that follows $T_{rs}$ in the execution but precedes $T$. By G-SI, there cannot be a write-write dependency edge from $T'$ to $T$, and their write-sets must therefore be distinct. Consequently: $\Delta(s_{T_{rs}}, s_T) = \emptyset$.

($\exists e : \forall T \in \mathcal{T} : \text{CT}_{SI}(T, e) \Rightarrow \neg\text{G1} \wedge \neg\text{G-SI}$). We show that the serialization graph $SSG(H)$ corresponding to $e$ does not exhibit phenomena G1 or G-SI. Every transaction in $e$ reads from some previous state and commits in the total order defined by $e$. It follows that all write-write and write-read edges follow the total order introduced by $e$: there can be no cycle consisting of write-write/write-read dependencies. ¬G1 is thus satisfied. To show that $SSG(H)$ does not exhibit G-SI, we first select the start and commit point of each transaction. We assign commit points to transactions according to their order in $e$. We assign the start point of each transaction $T$ to be directly after the commit point of the first transaction $T_{rs}$ in $e$ whose generated state satisfies $\text{COMPLETE}_{e,T}(s) \wedge (\Delta(s, s_p) \cap \mathcal{W}_T = \emptyset)$. It follows that $T_{rs}$ (and all the transactions that precede it in $e$) start-precede $T$. Proving ¬G-SIa is then straightforward: any transaction $T'$ that $T$ write-read/write-write depends on precedes $T_{rs}$ in the execution, and consequently start-precedes $T$. Proving ¬G-SIb requires a little more care. By ¬G-SIa, there necessarily exists a corresponding start-depend edge for any write-read or write-write edge between two transactions $T$ and $T'$: if there exists a cycle with exactly one read-write edge in the $SSG(H)$, there must exist a cycle with exactly one read-write edge and only start-depend edges in $SSG(H)$. Assuming by contradiction that G-SIb holds and that there exists a cycle with one read-write edge and multiple start-depend edges (we reduce this cycle to a single start-depend edge as start-edges are transitive). Let $T$ read-write depend on $T'$: $s_{T'}$ is ordered after $s_{T_{rs}}$ in $e$ (otherwise $s_{T_{rs}}$ cannot be a valid read state for $T$). However, as previously mentioned, $T$ only has start-depend edges with transactions that precede $T_{rs}$ (included) in $e$. $T'$ thus does not start-depend on $T$, a contradiction. □

Unlike Adya's, however, the correctness of our state-based definition does not rely on using start and commit timestamps. This is a crucial difference. Including these low-level attributes in the definition has encouraged the development of variations of SI that differ in their use of timestamps, whose fundamental guarantees are, as a result, difficult to compare. In §3.3.2 we show that, when expressed in terms of application-observable states, several of these variations, thought to be distinct, are actually equivalent![5]

**Read committed** Read committed allows $T$ to see the effects of concurrent transactions, as long as they are committed. The commit test therefore no longer constrains all operations in $T$ to read from the *same* state; instead, it only requires each of them to read from a state that precedes $T$ in the execution $e$. We prove in Appendix A.4 that:

**Theorem 3** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{RC}(T, e) \equiv \neg$*G1.*

**Read uncommitted** Read uncommitted allows $T$ to see the effects of concurrent transactions, whether they have committed or not. The commit test reflects this permissiveness, to the point of allowing transactions to read arbitrary values. Still, we prove in Appendix A.5 that:

**Theorem 4** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{RU}(T, e) \equiv \neg$*G0.*

The reason behind the laxity of the state-based definition is that isolation models in databases consider only committed transactions and are therefore unable to distinguish values produced by aborted transactions from those produced by future transactions. This distinction, however, is not lost in environments, such as transactional memory, where correctness depends on providing guarantees such as opacity [89] for all live transactions. We discuss this further in Section 3.5.

**Strict Serializability** Strict serializability guarantees that the real-time order of transactions will be reflected in the final history or execution. It can be expressed by adding the following condition to the serializability commit test: $\forall T' \in \mathcal{T} : T' <_s T \Rightarrow s_{T'} \xrightarrow{*} s_T$.

**Parallel Snapshot Isolation** Parallel snapshot isolation (PSI) was recently proposed by Sovran et al. [178] to address SI's scalability issues in geo-replicated settings. Snapshot isolation requires transactions to read from a snapshot (a *complete state* in our parlance) that reflects a single commit

[5] As proofs follow a similar structure, we defer all subsequent proofs to Appendices.

ordering of transactions. The coordination implied by this requirement is expensive to carry out in a geo-replicated system and must be enforced even when transactions do not conflict. PSI aims to offer a scalable alternative by allowing distinct geo-replicated sites to commit transactions in different orders. The specification of PSI is given as an abstract specification code that an implementation must emulate. Specifically, a PSI execution must enforce three properties. First, *site snapshot read*: all operations read the most recent committed version at the transaction's origin site as of the time when the transaction began (P1). Second, *no write-write conflicts*: the write sets of each pair of somewhere-concurrent committed transactions must be disjoint (two transactions are somewhere-concurrent if they are concurrent on site($T_1$) or site($T_2$)) (P2). And finally, *commit causality across sites*: if transaction $T_1$ commits at a site A before transaction $T_2$ starts at site A, then $T_1$ cannot commit after $T_2$ at any site.

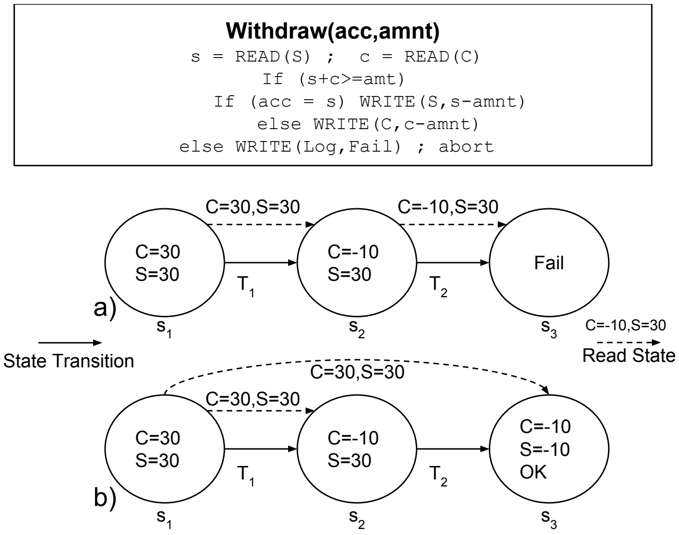
Our first step towards a state-based definition of PSI is to populate, using solely client-observable states, the *precede-set* of each transaction $T$, i.e., the set of transactions after which $T$ must be ordered. A transaction $T'$ is in $T$'s precede-set if (*i*) $T$ reads a value that $T'$ wrote; or (*ii*) $T$ writes an object modified by $T'$ and the execution orders $T'$ before $T$; or (*iii*) $T'$ precedes $T''$ and $T''$ precedes $T$. Formally: $\textsc{d-prec}_e(\hat{T}) = \{T | \exists o \in \Sigma_{\hat{T}} : T = T_{sf_o}\} \cup \{T | s_T \xrightarrow{+} s_{\hat{T}} \wedge \mathcal{W}_{\hat{T}} \cap \mathcal{W}_T \neq \emptyset\}$. We write $T_i \blacktriangleright T_j$ if $T_i \in \textsc{d-prec}_e(T_j)$ and $T_i \triangleright T_j$ if $T_i$ transitively precedes $T_j$. PSI guarantees that the state observed by a transaction $T$'s operation includes the effects of all transactions that precede it. We can express this requirement in PSI's commit test as follows:

**Definition 6** $\mathrm{CT}_{PSI}(T, e) \equiv \textit{PREREAD}_e(T) \wedge \forall T' \triangleright T : \forall o \in \Sigma_T : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$.

This client-centric definition of PSI makes immediately clear that the state which operations observe is not necessarily a complete state, and hence may not correspond to a snapshot of the database at a specific time. We prove the following theorem in Appendix E.3:

**Theorem 5** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T, e) \equiv$ *PSI.*

**Read Atomic** Read atomic [25], like PSI, aims to be a scalable alternative to snapshot isolation. It preserves atomic visibility (transactions observe either all or none of a committed transaction's effects) but does not preclude write-write conflicts nor guarantees that transactions will read from a causally consistent prefix of the execution. These weaker guarantees allow for efficient implementations and



**Withdraw(acc,amnt)**

```
s = READ(S) ;  c = READ(C)
      If (s+c>=amt)
   If (acc = s) WRITE(S,s-amnt)
      else WRITE(C,c-amnt)
else WRITE(Log,Fail) ; abort
```

*Figure 3.3: Simple Banking Application. Alice and Bob share checking and savings accounts. Withdrawals are allowed as long as the sum of both account is greater than zero.*

nonetheless ensure *synchronization independence*: one client's transactions cannot cause another client's transactions to fail or stall. Read atomic can be expressed in our state-based model as follows:

**Definition 7** $\mathrm{CT}_{RA}(T, e) \equiv \forall r_1(k_1, v_1), r_2(k_2, v_2) \in \Sigma_T \wedge k_2 \in \mathcal{W}_{T_{sf_{r_1}}} \Rightarrow sf_{r_1} \xrightarrow{*} sf_{r_2}$.

Intuitively, if an operation $o_1$ observes the writes of a transaction $T_i$'s, all subsequent operations that read a key included in $T_i$'s write set must read from a state that includes $T_i$'s effects. We prove the following theorem in Appendix B:

**Theorem 6** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{RA}(T, e) \equiv$ *RA*.

## 3.3 Benefits of a state-based approach

Specifying isolation using client-observable states rather than histories is not only equally expressive, but brings forth several benefits: it gives application developers a clearer intuition for the implications of choosing a given isolation level (§3.3.1), brings additional clarity to how different isolation levels relate (§3.3.2), and opens up opportunities for performance improvements in existing implementations (§3.3.3).

| | |
|---|---|
| Strong SI | C-ORD$(T_{s_p}, T) \land \exists s \in S_e$ : COMPLETE$_{e,T}(s) \land$ NO-CONF$_T(s) \land (T_s <_s T) \land (\forall T' <_s T : s_{T'} \xrightarrow{*} s)$ |
| Strong Session SI/PC-SI | C-ORD$(T_{s_p}, T) \land \exists s \in S_e$ : COMPLETE$_{e,T}(s) \land$ NO-CONF$_T(s) \land (T_s <_s T) \land (\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s)$ |
| ANSI SI /GSI | C-ORD$(T_{s_p}, T) \land \exists s \in S_e$ : COMPLETE$_{e,T}(s) \land$ NO-CONF$_T(s) \land (T_s <_s T)$ |
| Adya SI | $\exists s \in S_e$ : COMPLETE$_{e,T}(s) \land$ NO-CONF$_T(s)$ |
| PSI/PL-2+ | PREREAD$_e(T) \land \forall T' \ \triangleright T$ : CAUS-VIS$(e, T)$ |

*Table 3.2: Commit tests for snapshot-based protocols*

### 3.3.1 Minimizing the intuition gap

A state-based model makes it easier for application programmers to understand the anomalies allowed by weak isolation levels, as it precisely captures the *root cause* of these anomalies. Consider, for example, snapshot isolation: it allows for a non-serializable behavior called write-skew, illustrated in the simple banking example of Figure 3.3. Alice and Bob share checking (C) and savings (S) accounts, each holding \$30, the sum of which should never be negative. Before performing a withdrawal, they check that the total funds in their accounts allow for it. They then withdraw the amount from the specified account, using the other account to cover any overdraft. Suppose Alice and Bob try concurrently to withdraw \$40 from, respectively, their checking and savings account, and issue transactions $T_1$ and $T_2$. Figure 3.3(a) shows an execution under serializability. Because transactions read from their parent state (see Table 3.1), $T_2$ observes $T_1$'s withdrawal and, since the balance of Bob's accounts is below \$40, aborts. In contrast, consider the execution under snapshot isolation in Figure 3.3(b). As it is is legal for both $T_1$ and $T_2$ to read from a complete but stale state $s_1$, Alice and Bob can both find that the combined funds in the two accounts exceed \$40, and, unaware of each other, proceed to generate an execution whose final state $s_3$ is illegal. The state-based definitions of snapshot isolation and serializability make both the causes and the danger of write-skew immediately clear: to satisfy snapshot isolation, it suffices that both transactions read from the same complete state $s_1$, even though this behavior is clearly not serializable, as $s_1$ is not the parent state of $T_2$. The link is, arguably, less obvious with the history-based definition of snapshot isolation, which requires "disallowing all cycles consisting of write-write and write-read dependencies and a single anti-dependency".

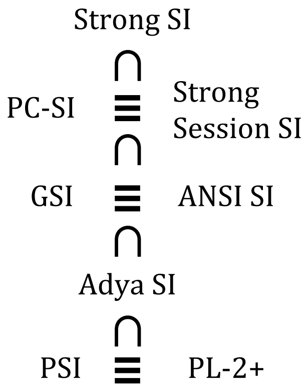


*Figure 3.4: Snapshot-based isolation guarantees hierarchy. (ANSI SI [31, 61], Adya SI [4], Strong SI [61], GSI [154], PSI [178], Strong Session SI [61], PL-2+ [5], PC-SI [61]).*

### 3.3.2 Removing implementation artefacts

By cleanly separating high-level properties from low-level implementation details, a state-based model makes the plethora of isolation guarantees introduced in recent years easier to compare. We leverage this newfound clarity below to systematize snapshot-based guarantees, including ANSI SI [31], Adya SI [4], Weak SI [61], Strong SI [61], generalized snapshot isolation (GSI) [154], parallel snapshot isolation (PSI) [178], Strong Session SI [61], PL-2+ (Lazy Consistency) [5], Prefix-Consistent SI (PC-SI) [61]. We find that several of these isolation guarantees, previously thought to be distinct, are in fact *equivalent* from a client's perspective, and establish a clean *hierarchy* that encompasses them.

Snapshot-based isolation guarantees, broadly speaking, are defined as follows. A transaction is assigned both a start and a commit timestamp; the first determines the database snapshot from which the transaction can read (it includes all transactions with a smaller commit timestamp), while the second maintains the "first-committer-wins" rule: no conflicting transactions should write to the same objects. The details of these protocols, however, differ. Each strikes a different performance trade-off in how it assigns timestamps and computes snapshots that influences its high-level guarantee in ways that can only be understood by applications with in-depth knowledge of the internals of the underlying systems. As a result, it is hard for application developers and researchers alike to compare and contrast them.

In contrast, formulating isolation in terms of client-observable states *forces* definitions that specify guarantees according to how they are *perceived by clients*. It then becomes straightforward to understand what guarantees are provided, and to observe their differences and similarities. Specifically, it clearly exposes the three dimensions along which snapshot-based guarantees differ: (*i*) *time* (whether timestamps are logical [4, 129, 178] or based on real-time  [31, 61, 154]); (*ii*) *snapshot recency* (whether the computed snapshot contains *all* transactions that committed before the transaction start time  [4, 61] or can be stale [19, 61, 154, 178]); and *state completeness* (in our parlance, whether snapshots must correspond to a complete state  [4, 31, 61, 154] or whether a causally consistent [9] snapshot suffices [5, 19, 129, 178]).

Grouping isolation guarantees in this way highlights a clean hierarchy between these definitions, and suggests that many of the newly proposed isolation levels proposed are in fact equivalent to prior guarantees. We summarize the different commit tests in Table 3.2 and the resulting hierarchy in Figure 3.4. As the existence of the hierarchy follows straightforwardly from the commit tests, we defer the proof of its soundness to Appendix F, along with proofs of the corresponding equivalences.

At the top of the hierarchy is Strong SI [61]. It requires that a transaction $T$ observe the effects of all transactions that have committed (in real-time) before $T$ (in other words, read from the most recent database snapshot) and obtain a commit timestamp greater than any previously committed transaction. We express this (Table 3.2, first row) by requiring that the last state in the execution generated by a transaction that happens before $T$ in real time must be complete ($\forall T' <_s T : s_{T'} \xrightarrow{*} s$), and that the total order defined by the execution respects commit order ($\text{C-ORD}(T, T') \equiv T.commit < T'.commit$). We prove that this formulation is equivalent to its traditional implementation specification in Appendix C.4:

**Theorem 7** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{StrongSI}(T, e) \equiv Strong\ SI.$

Skipping for a moment one level in the hierarchy, we consider next ANSI SI [31]. ANSI SI weakens Strong SI's requirement that the snapshot from which $T$ reads include *all* transactions that precede $T$ in real-time (including those that access objects that $T$ does not access). This weakening, which improves scalability by avoiding the coordination needed to generate Strong SI's snapshot, can be expressed in our state-based approach by relaxing the requirement that the complete state be the most recent in real-time (Table 3.2, third line). An attractive consequence of this new formulation is that it clarifies the relationship between ANSI SI and *generalized snapshot isolation* [154], a refinement of ANSI SI for lazily replicated databases. We prove that these two decade-old guarantees are actually equivalent in Appendices C.2 and D.2:

**Theorem 8** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{\textit{ANSI SI}}(T, e) \equiv \textit{GSI} \equiv \textit{ANSI SI}.$

This is not an isolated case: we find that the two less popular notions of isolation that occupy the level of the hierarchy between Strong SI and ANSI SI (Strong Session SI (SSessSI) [154] and Prefix-Consistent SI (PC-SI) [61]) are also equivalent These guarantees seek to prevent *transaction inversions* [61] (a client $c_1$ executes a transaction $T_1$ followed, in real-time by $T_2$ without $T_2$ observing the effects of $T_1$) that can arise when transactions read from a stale snapshot—but without requiring all transactions to read from the most recent snapshot. To this effect, they strike a balance between ANSI SI and Strong SI: they introduce the notion of *sessions* and require a transaction $T$ to read from a snapshot more recent than the commit timestamp of all transactions that precede $T$ in a session (formally: a session *se* is a tuple $(T_{se}, \xrightarrow{se})$ where $\xrightarrow{se}$ is a total order over the transactions in $\mathcal{T}_{se}$ such that $T \xrightarrow{se} T' \Rightarrow T <_s T'$). Our model straightforwardly captures this definition (Table 3.2, second row) by requiring that the complete state from which a transaction reads follow the commit state of all transactions in a session. We prove the following theorem in Appendices C.3 and D.3:

**Theorem 9** $\exists e : \forall t \in \mathcal{T} : \text{CT}_{Session\ SI}(t, e) \equiv SSessSI \equiv PC\text{-}SI.$

Though ANSI SI or Strong Session SI are both more scalable than Strong SI, their definitions still include several red flags for efficient large-scale implementations. First, they require a total order on transactions ($\textsc{c-ord}(s, s_p)$), forcing developers to implement expensive coordination mechanisms, even as transactions may access different objects. Second, they limit a transaction $T$ to reading only from complete states that do not include transactions that committed in real time *after* $T$'s start timestamp. This implementation choice often forces transactions to read further in the past than necessary, making them more prone to write-write conflicts with concurrent transactions. Moreover, it prevents transactions from reading uncommitted operations, precluding efficient implementations for high-contention workloads [72, 201]. Adya's reformulation of SI [4] side-steps many of these baked-in implementation decisions by removing the dependence on real-time, instead allocating *logical* timestamps consistent with the transactions' observations. Our model can capture this distinction by simply removing the two aforementioned clauses from the commit test (Table 3.2, fourth row), allowing for maximum flexibility for how snapshot isolation can be implemented without affecting client-side guarantees.

The lowest level of the hierarchy covers snapshot-based isolation guarantees intended for large-scale geo-replicated systems. When transactions may be asynchronously replicated for performance and availability, it is challenging to require that transactions read a database snapshot that corresponds to a single moment in time (and hence read from a *complete state*) as it would require transactions to become visible atomically across all (possibly distant) datacenters. PSI [178] (introduced in §3.2) and PL-2+ [4, 5] consequently weaken Adya's SI to address these new challenges: PSI requires that transactions read from a committed snapshot but allows concurrent transactions to commit in a different order at different sites, while PL-2+ disallows cycles consisting of either write-write/write-read dependencies, or containing a single anti-dependency edge. Unlike what these widely different low-level definitions suggest, taking a client-centric view of these guarantees indicates that PSI and PL-2+ in fact weaken Adya's snapshot isolation in an identical fashion: they no longer require transactions to read from a complete state, and instead require that operations read from a (possibly different) state that includes the effects of all previously observed transactions. Our model cleanly captures the shared guarantee provided by PL-2+/PSI: that a transaction $T$ must observe the effects of all transactions that it is not concurrent with (Table 3.2, fifth line). We write: for every transaction $T'$ that a transaction $T$ depends on: $\forall o \in \Sigma_T : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o \equiv \text{CAUS-VIS}(e, T)$. From this client-centric formulation, we prove the following theorem in Appendix E:

**Theorem 10** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T, e) \equiv$ *PSI* $\equiv$ *PL-2+*.

### 3.3.3 Identifying performance opportunities

Beyond improving clients' understanding, defining isolation guarantees in terms of client-observable states helps prevent them from subjecting transactions to stronger requirements than what these guarantees require end-to-end. Indeed, by removing all implementation-specific details (timestamps, replicas) present in system-centric formulations, our model gives full flexibility to how these guarantees can be implemented. We illustrate this danger, and highlight the benefits of our approach, using the specific example of PSI/PL-2+.

In its original specification, the definition of parallel snapshot isolation [178] requires datacenters to enforce snapshot isolation, even as it globally only offers (as we prove in Theorem 10) the guarantees of lazy consistency/PL-2+. This baked-in implementation decision makes the *very definition* of PSI unsuitable for large-scale partitioned datacenters as it makes the definition (and therefore any system that implements it) susceptible to slowdown cascades. Slowdown cascades (common in large-scale systems [10]) arise when a slow or failed node/partition delays operations that do not access that node itself, and have been identified by industry [10] as the primary barrier to adoption of stronger consistency guarantees. By enforcing SI on every site, the history-based definition of PSI creates a total commit order across all transactions within a datacenter, even as they may access different keys. Transactions thus become dependent on all previously committed transactions on that datacenter, and cannot be replicated to other sites until all these transactions have been applied. If a single partition is slow, all transactions that artificially depend on transactions on that node will be unnecessarily delayed, creating a cascading slowdown.

An approach based on client-observable states, in contrast, makes no such assumptions: the depend-set of a transaction is computed using client observations and read states only, and thus consists exclusively of transactions that the application itself can *perceive* as ordered with respect to one another. Every dependency created stems from an actual *observation*: the number of dependencies that a client-centric definition creates is consequently minimal (and the fewer dependencies a system creates, the less likely it will be subject to slowdown cascades). To illustrate this potential benefit, we simulated the number of transactional dependencies created at each datacenter by the traditional definition of PSI as compared to the "true" dependencies generated by the proposed client-centric definition, using an asynchronously replicated transactional key-value store, TARDiS [59] (described in Chapter 5). On a workload consisting of read-write transactions (three reads, three writes) accessing data uniformly over 10,000 objects (Figure 3.5), we found that a client-centric approach decreased dependencies, per transaction, by two orders of magnitude (175×), a reduction that can yield significant dividends in terms of scalability and robustness.

State-based specifications of isolation guarantees can also benefit performance, as they abstract away

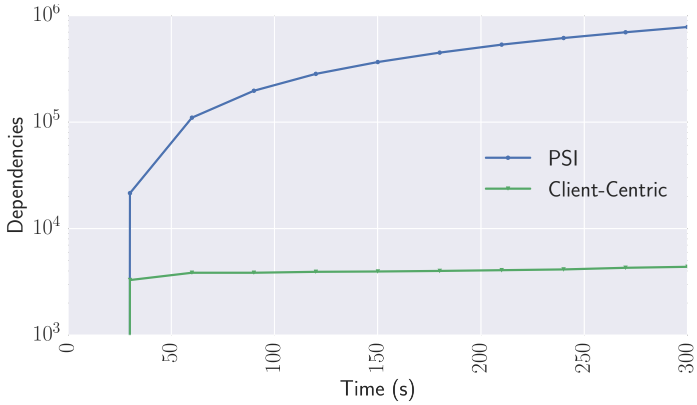


*Figure 3.5: Number of dependencies per transaction as a function of time. TARDiS [59] runs with three replicas on a shared local cluster (2.67GHz Intel Xeon CPU X5650, 48GB memory and 2Gbps network).*

the details of specific mechanisms used to enforce isolation, and instead focus on how different flavors of isolation constrain permissible read states. A case-in-point is Ardekani et al.'s non-monotonic snapshot isolation (NMSI) [19]: NMSI logically moves snapshots forward in time to minimize the risk of seeing stale data (and consequent aborts due to write-write conflicts), without violating any consistency guarantees. This technique is premised on the observation that, given the values read by the client, the states at the earlier and later snapshot are indistinguishable.

## 3.4 Related work

Most past definitions of isolation and consistency [4, 19, 31, 32, 32, 40, 73, 92, 153, 178, 185] refer to specific orderings of low-level operations and to system properties that cannot be easily observed or understood by applications. To better align these definitions with what clients perceive, recent work [20, 46, 123, 192] distinguishes between *concrete* executions (the nuts-and-bolts implementations details) and *abstract* executions (the system behaviour as perceived by the client). Attiya et al., for instance, introduce the notion of observable causal consistency [20], a refinement of causal consistency where causality can be inferred by client observations. Likewise, Cerone et al. [46, 47] introduce the dual notions of visibility and arbitration to define, axiomatically, a large number of existing isolation levels. The simplicity of their formulation, however, relies on restricting their model to consider only isolation levels that guarantee atomic visibility [25], which prevents them from expressing guarantees like read-committed, the default isolation level of most common database systems [132, 136, 148, 148, 149, 158, 168, 195], and the only supported level for some [148][6]. Shapiro and Ardekani [170] adopt a similar approach to identify three orthogonal dimensions (total order, visibility, and transaction composition) that they use to classify consistency and isolation guarantees. All continue, however, to characterize correctness by constraining the ordering of read and write operations and often let system specific details (e.g., system replicas) leak through definitions. Our model takes their approach a step further: it *directly defines* consistency and isolation in terms of the observable states that are routinely used by developers to express application invariants [11, 23, 59]. Finally, several practical systems have recognized the benefits of taking a client-centric approach to system specification and development. These systems target very different concerns, from file I/O [143] to cloud storage [131], and from Byzantine fault-tolerance [104] to efficient Paxos implementations [157]. In the specific context of databases and key-value stores, in addition to Ardekani et al.'s work [19], Mehdi et al. [129] recently proposed a client-centric implementation of causal consistency that is both scalable and resilient to slowdown cascades (§6.2.2).

## 3.5 Limitations

TThe model presented in this chapter currently has two main limitations, which we plan to address in future work. First, it does not constrain the behavior of ongoing transactions. It thus cannot express consistency models, like opacity [89] or virtual world consistency [93] , designed to prevent STM transactions from accessing an invalid memory location. This limitation is consistent with the assumption, made in most isolation and consistency research, that applications never make externally visible decisions based on uncommitted data, so that their actions can be rolled back if the transaction aborts. Second, our model focuses on the traditional transactional/read/write model, predominant in database theory and modern distributed storage systems. To support semantically rich operations, abstract data types, and commutativity, we will start from Weikum et al's theory of multi-level serializability [196], which maps higher-level operations to reads and writes.

[6] as of June 2019

## 3.6 Conclusion

This chapter presents a new, client-centric way to reason about isolation based on application-observable states and proves it to be as expressive as prior approaches based on histories. We present evidence suggesting that this approach (*i*) maps more naturally to what applications can observe and illuminates the anomalies allowed by distinct isolation/consistency levels; (*ii*) makes it easy to compare isolation guarantees, leading us to prove that distinct, decade-old guarantees are in fact equivalent; and (*iii*) facilitates reasoning end-to-end about isolation guarantees, enabling new opportunities for performance optimization.

# Chapter 4

# Extending our model to consistency

Chapter 3 introduced a new state-based, client-centric model of isolation guarantees. It demonstrated that such an approach minimised the gap between how isolation guarantees are defined and how they are perceived.

**The context** Isolation guarantees, however, as we mentioned in Chapter 2 regulate only the interaction between concurrently executing transactions. For instance, serializability allows the transactions of the same client to be reordered. In contrast, consistency guarantees explicitly regulate the ordering of individual operations; sequential consistency, session guarantees [50, 185], or causal consistency [9] thus explicitly regulate the ordering of individual operations that pertain to individual clients. These guarantees ensure that, as operations are replicated across different replicas, clients observe a state that remains coherent. As with isolation, "coherent" depends on the specific consistency guarantee.

**The problem** We identify two primary problems with how consistency guarantees are currently defined. First, as with isolation, large scale distributed systems offer a myriad of consistency guarantees to applications. The strongest consistency notion, linearizability [92], provides the clean abstraction of a centralised, non-replicated system. While easy to program, the coordination necessary to ensure that replicas operate in lock-step implies that linearizable systems provide only limited throughput and cannot make progress during network partitions. The CAP theorem [39, 79] highlights that one must make a trade-off between coordination (and consequently how much replicas can diverge) and partition-tolerance. In light of this theorem, many wide-area services and applications [44] choose to renounce strong consistency and focus instead on prodiving the ALPS properties [118] of **A**vailability, low **L**atency, **P**artition tolerance and high **S**calability. Some systems provide only eventual consistency [138, 167, 193], which provides little to no guarantees on far replicas can diverge. Guarantees like session guarantees [185] or causal consistency [9, 118, 135] provide a balance between these two extremes. Unfortunately, as with isolation, understanding the precise meaning of these guarantees requires specific knowledge of how the system implements them (for example, whether writes are totally ordered across sessions): yet, these details are not available to applications. For example, the exact meaning of *session guarantees* ( Bayou [187], Corba [50] and, more recently, in Pileus [186] and Microsoft's DocumentDB [135]) depends on whether the system implements a total order of write operations across client sessions. Consider the execution in Figure 4.1: does it satisfy the session guarantee *monotonic reads*, which calls for reads to reflect a monotonically increasing set of writes? The answer depends on whether the underlying system provides a total order of write operations across client sessions, or just a partial order based on the order of writes in each session. The specification of monotonic reads, however, is silent on this issue.

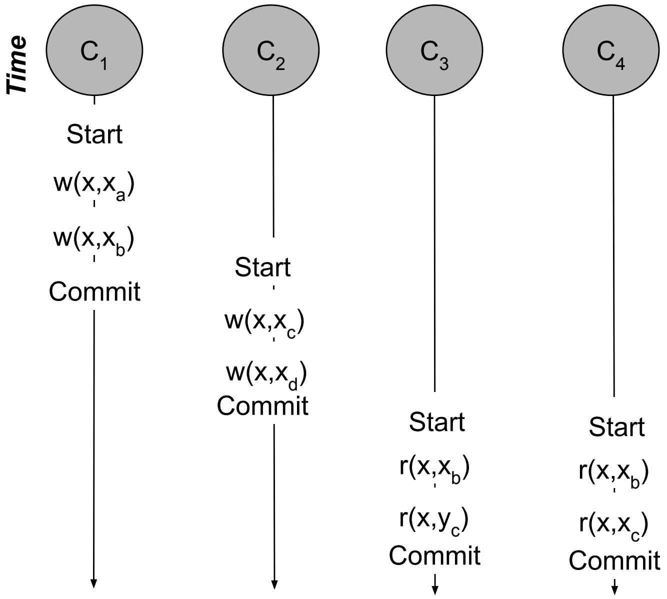


*Figure 4.1: Monotonic Read Execution*

Second, cloud storage systems are increasingly adding support for transactions [16, 66, 134, 138, 155], it is currently not clear what correctness guarantee the resulting systems actually offer, as there lacks a unified framework for understanding consistency and isolation.

**The benefits** We consequently extend our state-based model of isolation to consistency. Extending this client-centric approach to consistency guarantees brings forth two benefits. First, it allows us to define session guarantees [50, 185] without making assumptions on the order of writes in the system. Second, it allow us to unify the often disparate theories of isolation and consistency and provides a structure for composing their guarantees. We leverage this modularity to extend to transactions the equivalence between causal consistency and session guarantees that Chockler et al. had proved for single operations [50], showing, moreover, that it holds independently of the isolation level under which they execute.

**Roadmap** Chapter 2 reviewed the current approaches to formalizing consistency guarantees. We introduce our state-based, client-centric model in Section 4.1, and use it in Section 4.2 to define several consistency guarantees. We highlight the benefits of our approach in Section 4.3, before outlining our work's limitations in Section 4.4.

## 4.1 A State-based Model

Chapter 3 introduced a client-centric model for isolation that does not on notions of history or implementation like timestamps or write versions. It associates with each transaction the set of candidate states (called *read states*) from which the transaction may have retrieved the values it read during its execution. Read states inform the application of the set of possible worlds (i.e., states) consistent with what a transaction observed during its execution. To unify our treatment of consistency and isolation, we also assume here that applications modify the storage system's state using transactions.

Isolation guarantees typically do not regulate how transactions from a given client should be ordered [1]. They instead tacitly assume that transactions from the same client will execute in client-order, as they would in a centralized or synchronously replicated storage system. In weakly consistent systems however, where transactions can be replicated asynchronously, this assumption no longer hold. To

[1] Strict serializability is the exception to this rule.

reestablish order, distributed systems introduce the notion of *sessions*. Sessions encapsulate the sequence of operations performed by each entity (thread, client, or application) and provide each with a view of the system consistent with its own actions. We thus introduce this notion of sessions to the model defined in Chapter 3. Formally, a session *se* is a tuple $(T_{se}, \xrightarrow{se})$ where $\xrightarrow{se}$ is a total order over the transactions in $\mathcal{T}_{se}$. The set of all sessions is denoted by $SE$.

We additionally introduce the notion of *internal read consistency*: the predicate $\textsc{irc}_e(T)$ t states that operations that follow each other in the transaction order should *appear to* read from a monotonically increasing state.

## 4.2 A new model for consistency

Session guarantees are today's most popular consistency guarantees, but have so far only been defined for operations [40, 50, 185]: to build the foundation of a common theory of isolation and consistency, we proceed to define session-based consistency guarantees for transactions. We start by offering a state-based definition of sequential consistency (*SC*) [112]. *SC* requires that read operations within each transaction observe monotonically increasing states and have non-empty candidate read sets; further, like previously defined isolation levels, it demands that all sessions observe a single execution. Unlike isolation levels, however, *SC* also requires transactions to take effect in the order specified by their session. We define the commit test ($CT$) for *SC* as follows:

**Definition 1** $\text{CT}_{SC}(e, T) \equiv \textit{PREREAD}_e(T) \wedge \textit{IRC}_e(T) \wedge (\forall se \in SE : \forall T_i \xrightarrow{se} T_j : (s_{T_i} \xrightarrow{+} s_{T_j} \wedge \forall o \in \Sigma_{T_j} : s_{T_i} \xrightarrow{*} sl_o))$

Guaranteeing the existence of a single execution across all clients is often prohibitively expensive if sites are geographically distant. Many systems instead allow clients in different sessions to observe *distinct* executions. Clients consequently perceive the system as consistent with their own actions, but not necessarily with those of others. To this effect, we reformulate the *commit test* into a *session test*: for an execution $e$ to be valid under a given session $se$ and session guarantee $SG$, each transaction *T* in $\mathcal{T}_{se}$ must satisfy the session test for $SG$, written $\text{SESSION}_{SG}(se, T, e)$.

**Definition 2** *A storage system satisfies a session guarantee* $SG \equiv \forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{SG}(se, T, e)$.

Intuitively, session tests invert the order between the existential qualifier for execution and the universal quantifier for sessions. Table 4.1 shows the session tests for the most common session guarantees. We informally motivate their rationale below.

**Read-My-Writes** RMW states that a client will read from a state that includes any preceding writes *in its session*. RMW is a fairly weak guarantee: it does not constrain the order in which writes take effect, does not provide any guarantee on the reads of a client who never writes or on the outcome of reads performed in other sessions (as it limits the scope of PREREAD only to the transactions of its session). The session test simply requires that the read state of every operation in a transaction's session be after the commit state of all preceding update transactions in that session.

**Monotonic Reads** MR, instead, constrains a client's reads to observe increasingly up-to-date states: this applies to transactions in a session, and to operations within the transaction (by IRC). The notion of "up-to-date" varies by client, as the storage is free to arrange transactions differently for each session's execution. Violations of MR can thus only be detected by clients reading the same value three times: reading the initial value, a new value, and the initial value again. Moreover, a client is not guaranteed to see the effects of its own write: MR allows reading from monotonically increasing but stale states.

**Monotonic Writes** In contrast, MW constrains the ordering of writes that belong to the same session: the sequence of state transitions in each execution must be consistent with the order of update transactions in *every session*. Unlike MR and RMW, monotonic writes is a global guarantee. The PREREAD requirement for read operations, instead, continues to apply only within each session.

**Writes-Follow-Reads** Like MW, WFR is a global guarantee, this time covering reads as well as writes. It states that, if a transaction reads from a state $s$, all transactions that follow in that session must be ordered after $s$ in the execution.

**Causal Consistency** Finally, causal consistency guarantees that any execution will order transactions in a causally consistent order: read operations in a session will see monotonically increasing read states, and commit in session order. Likewise, transactions that read from a state $s$ will be ordered after $s$ in all sessions. This relationship is transitive: every transaction that reads $s$ (or that follows in the session) will also be ordered after $s$.

| | |
|---|---|
| Read-My-Writes (RMW) | $\text{PREREAD}_e(\mathcal{T}_{se}) \wedge \forall o \in \Sigma_T : \forall T' \xrightarrow{se} t : \mathcal{W}_{T'} \neq \emptyset \Rightarrow s_{T'} \xrightarrow{*} sl_o$ |
| Monotonic Reads (MR) | $\text{PREREAD}_e(\mathcal{T}_{se}) \wedge \text{IRC}_e(T) \wedge \forall o \in \Sigma_T : \forall T' \xrightarrow{se} t : \forall o' \in \Sigma_{T'} : \neg(sl_o \xrightarrow{+} sf_{o'})$ |
| Monotonic Writes (MW) | $\text{PREREAD}_e(\mathcal{T}_{se}) \wedge \forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : (\mathcal{W}_{T_i} \neq \emptyset \wedge \mathcal{W}_{T_j} \neq \emptyset) \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$ |
| Writes-Follow-Reads (WFR) | $\text{PREREAD}_e(\mathcal{T}) \wedge \forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : \forall o_i \in \Sigma_{T_i} : \mathcal{W}_{T_j} \neq \emptyset \Rightarrow sf_{o_i} \xrightarrow{+} s_{T_j}$ |
| Causal Consistency (CC) | $\text{PREREAD}_e(\mathcal{T}) \wedge \text{IRC}_e(T) \wedge (\forall o \in \Sigma_T : \forall T' \xrightarrow{se} t : s_{T'} \xrightarrow{*} sl_o)$<br>$\wedge(\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j})$ |

*Table 4.1: Session test for session guarantees*

## 4.3 From Session Guarantees to Causal Consistency

Unifying consistency with isolation makes it straightforward to compose these guarantees and understand how they interact. Using this model, we generalize a seminal result by Chockler et al's [50]: we prove the four session guarantees, taken together, remain equivalent to causal consistency when using transactions. Interestingly, we note that this result holds independently of the transactions' isolation level, as it enforces no relationship between a transaction's parent state and the read states of the operations of that transaction.

**Theorem 1.** *Let* $\mathcal{G} = \{RMW, MR, MW, WFR\}$*, then*
$\forall se \in SE : \exists e : \forall T \in T_{se} : \text{SESSION}_{\mathcal{G}}(se, T, e) \equiv \forall se \in SE : \exists e : \forall T \in T_{se} : \text{SESSION}_{CC}(se, T, e)$

We prove this result in Appendix G. The proof is non-trivial and relies on one key observation: if there exists an execution that satisfies all four session guarantees, there must exist a (possibly different) execution with the same set of transactions and read values that is causally consistent.

This result allows us to, for the first time, precisely characterize the guarantees that result from combining multiple session gurantees with the increasing support for transactions that modern cloud storage systems provide.

## 4.4 Limitations

The current formalism takes an exclusively client-centric view of consistency. It defines the commit test per session, as different clients may observe a different ordering of operations. Our model presents three limitations. First, guarantees like causal consistency regulate the order in which writes are observed by different clients. They remain silent however on how conflicting, concurrent write operations are handled. Asynchronously replicated systems like COPS [118], Ficus [163], Bayou [187], Walter [178], Coda [99] or conflict replicated data types (CRDTs) [172] use different *merging* functions to incorporate conflicting writes. As we discuss in the next chapter (Chapter 5), these merging functions can be significantly impact the overall correctness of the system. Formalising these merging strategies in our framework is consequently important. Second, our model takes a strictly local view of consistency: it defines a commit test per session. Investigating alternative strategies to account for the differing executions and how they relate to each other would be an interesting avenue for future work. Finally, our model considers only read/write operations. Some merging functions, like CRDTs, inherently rely on the semantic of objects for correctness. Extending our model to abstract datatypes would consequently enhance the expressivity of our framework.

## 4.5 Conclusion

This chapter extends the state-based formulation of isolation to consistency and presents the first unified framework of consistency and isolation. It demonstrates that a state-based formalism can not only simplify the expression of traditional isolation guarantees, but also consistency guarantees.

# Chapter 5

# Simplifying weak consistency with client-centric forking and merging

The previous two chapters identified a new client-centric and state-based formalism for reasoning about consistency and isolation. We demonstrated that this formalism could bring *semantic* benefits to the complex field of weak isolation and weak consistency. The second half of this thesis focuses on the *practical benefits* that this new way of thinking can bring. Specifically, it focuses on the design of two systems, TARDiS in this chapter, and Obladi in Chapter 6. Both embrace the client, state-centric approach that we advocate to improve the functionality and performance of transactional datastores.

**The context** As we highlight in previous chapters, applications that favour weaker notions of consistency must deal with the complexity of programming on top of these systems. Many different consistency definitions have been proposed that trade-off consistency for performance. These consistency guarantees place constraints on what read operations can observe at a given point in time (potentially as a function of what they have already observed, and of where they originate). These definitions, however, remain silent on the impact of *conflicting write operations*. Without systematic coordination, geographically distinct replicas can issue conflicting operations, and the read-write and write-write conflicts that can ultimately result from these operations may cause replicas' states to diverge.

Many prior systems have attempted to insulate applications from this complexity by relying on a combination of two techniques: *causal consistency* [9, 26, 64, 119, 178, 186] to mitigate the effects of read-write conflicts, and per-object *eventual convergence* [62, 118, 123, 124, 178, 187] to address write-write conflicts. These systems strive to keep complexity in check by aggressively preserving the familiar abstraction that an application's state evolves through a linear sequence of updates. Any perturbation to this abstraction is nipped in the bud, either within the storage layer—by enforcing per-object convergence through simple deterministic resolution policies (i.e. taking a system-centric view to merging)—or by asking the application to resolve the state of objects with conflicting updates as soon as conflicts arise (i.e. taking an operational view to merging) [62, 91, 118, 187].

These techniques, however, are not sufficient to uphold the abstraction of sequential storage in the presence of concurrent updates. Worse, causal order and per-object convergence provide no support for meaningfully resolving conflicts between concurrent sequences of updates that involve multiple objects: indeed, they often destroy information that could have helped the application address these anomalies. For example, deterministic writer-wins, a common technique to achieve convergence [118], hides write-skew from applications. Similarly, exposing multivalued objects without context obscures cross-object semantic dependencies (§5.1).

**Our secret sauce** Anomalies like write-skew are intrinsic to ALPS applications. The issue is then neither how to prevent them (one can't), nor how to resolve them transparently (application-specific knowledge is often indispensable): rather, it is how to provide ALPS applications with the best possible system support when merging conflicting states. This thesis argues that geo-replicated storage should eschew the system-centric and operational centric view of merging that hides distribution from users. Instead, it should adopt a client, state-centric view of conflict resolution and let clients drive the process. To this end, this thesis proposes a new storage system, TARDiS (**T**ransactional **A**synchronously **R**eplicated **Di**vergent **S**torage). TARDiS deliberately abandons a strictly sequential view of storage, and instead gives applications flexibility. If all is well, storage at each replica appears sequential; when conflicts must be resolved, however, the intricate details of distribution become available. As in Git [80], users operate on their own branch and explicitly request (when convenient) to see concurrent modifications, using the history recorded by the underlying branching storage to help them resolve conflicts. Unlike Git, however, branching in TARDiS does not rely on specific user commands, but occurs implicitly, to preserve availability in the presence of conflicts, using three core mechanisms: (*i*) branch-on-conflict, (*ii*) inter-branch isolation, and (*iii*) application-driven cross-object merge.

1. **Branch-on-conflict** Branch-on-conflict lets TARDiS logically fork its state whenever it detects conflicting operations, and store the conflicting branches explicitly.
2. **Inter-branch isolation** Inter-branch isolation guarantees that storage will appear as sequential to any thread of execution that extends a branch, keeping application logic simple. This view aligns with how the client perceives the system as evolving.
3. **Application-driven cross-object merge** Finally, TARDiS leaves the task of deciding if, when, and how divergent branches should be merged to the application, rather than to the storage layer, which is generally unsuited to leverage relevant semantic information.

**The benefits** At first glance, TARDiS' design may appear counter-intuitive: isn't a simpler abstraction, such as sequential storage, easier to reason about? Our view is that abstractions should indeed be made as simple as possible—but no simpler: a simplistic abstraction that overlooks critical context can actually make reasoning harder. Through its richer interface, TARDiS gives applications access to context that is essential to reasoning about concurrent updates, reducing the complexity of programming ALPS applications and improving their performance. It does so via the following four key properties.

- **TARDiS knows history.** Each TARDiS site stores a DAG that records all branches generated during an execution, and uses a new algorithm, *DAG compression*, to track the minimal information needed to support branch merges. This context, which traditional storage systems hastily discard, can prove invaluable when programming ALPS applications (§5.6). We find, for example, that using TARDiS rather than BerkeleyDB [144] to implement CRDTs [171]—a library of scalable, weakly-consistent datatypes—cuts code size by half, improves performance by four to eight times, and reduces development time by a factor of three.
- **TARDiS merges branches, not objects.** Prior systems that, like TARDiS, admit parallel versions of the same object [8, 62, 124, 187] have systematically taken a strictly per-object

view of multiversions. With no support for enforcing the cross-object consistency demands expressed in many application invariants, such systems make conflict resolution more difficult and error prone. Instead, TARDiS merges branches. Conflict resolution is done at the granularity of entire states, not individual objects. This is in line with the approach outlined in the prior chapters: clients observe states, not objects. To efficiently construct and maintain branches, TARDiS introduces the notion of *conflict tracking*. By summarizing branches as a set of fork points and merge points, conflict tracking significantly reduces the metadata overhead experienced by many systems that enforce causal consistency [24, 64].

- **TARDiS is expressive.** Despite exposing a different abstraction, TARDiS supports many isolation levels (serializability, snapshot isolation, read-committed [31]) and consistency guarantees (read-my-writes [185], causal consistency [9]). It does so with minimal changes to applications' code and with performance comparable to that of BerkeleyDB, a commercially available Java database. TARDiS achieves this flexibility by reformulating isolation and consistency requirements as a set of pre- and post- conditions, derived from the commit tests defined in Chapter 3 and 4.

- **TARDiS improves performance of the local site.** A unique feature of TARDiS is that it allows ALPS applications to apply weak-consistency principles end-to-end, by triggering branch-on-conflict not only for operations issued by different sites, but also for locally conflicting operations. This is inline with our initial observation: individual components of the system do not need to enforce a guarantee stronger than what clients perceive. When so configured, TARDiS handles local conflicts not through abort/rollback and locking, but by logically forking the local datastore, which in our implementation is a very fast operation. Of course, this feature is not beneficial to all applications, as producing a large number of additional branches may increase merging complexity dramatically. However, we find that the ALPS applications that TARDiS targets, where weak consistency and merging are first-order concerns, can leverage this feature to increase their throughput significantly: applying this technique to Retwis [165], a commonly used Twitter-like ALPS application [151, 165, 178, 200], yields a three-fold improvement in throughput with negligible increase in complexity. TARDiS enables this speedup by extending the copy-on-write techniques present in multiversioned systems to support not just stale snapshots, but also branches.

In summary, this chapter makes three contributions:

1. It highlights how conflicting writes can corrupt the full database state, not simply the individual objects to which they are applied.
2. It identifies a new abstraction that is better suited to how clients perceive the diverging executions that replicas in weakly storage systems execute.
3. It presents the design of TARDiS, an asynchronously replicated, multi-master, transactional key-value store designed for applications built above weakly-consistent systems. TARDiS renounces the one-size-fits-all abstraction of sequential storage and instead exposes applications, when appropriate, to concurrency and distribution. This unconventional design is predicated on a simple notion: to help developers resolve the anomalies that arise in such applications, *each replica should faithfully store the full context necessary to understand how the anomalies arose in the first place, but only expose that context to applications when needed.*

**Roadmap** This chapter is structured as follows: we first illustrate why isolating application from conflict resolution is problematic (§5.1), and suggest alternative abstractions, better suited to large scale distributed systems (§5.2). We then describe how developers can use TARDiS (§5.4) along with the system's architecture (§5.3) and design (§5.5). Finally, we report on performance and application experiences (§5.6), summarize related work (§5.7), highlight limitations (§5.8), conclude (§5.9).[1].

## 5.1 The gap between causality and reality

Chapter 2 illustrated the virtues of causal consistency by sketching out the dangers of causally related writes arriving out of order at a replicated site (recall this dissertation's author removing their advisor from social media before attempting to cook pasta). ALPS applications, however, face another class of anomalies—write-write-conflicts—that causal consistency cannot prevent, detect, or repair.

To illustrate, consider the process of updating a Wikipedia page consisting of multiple HTML objects

[1] This work revises the previously published paper: *TARDiS: A branch-and-merge approach to weak consistency*, published at SIGMOD 2016. Youer Pu, Nancy Estrada and Trinabh Gupta contributed the evaluation of the paper as well as general discussions on paper writing

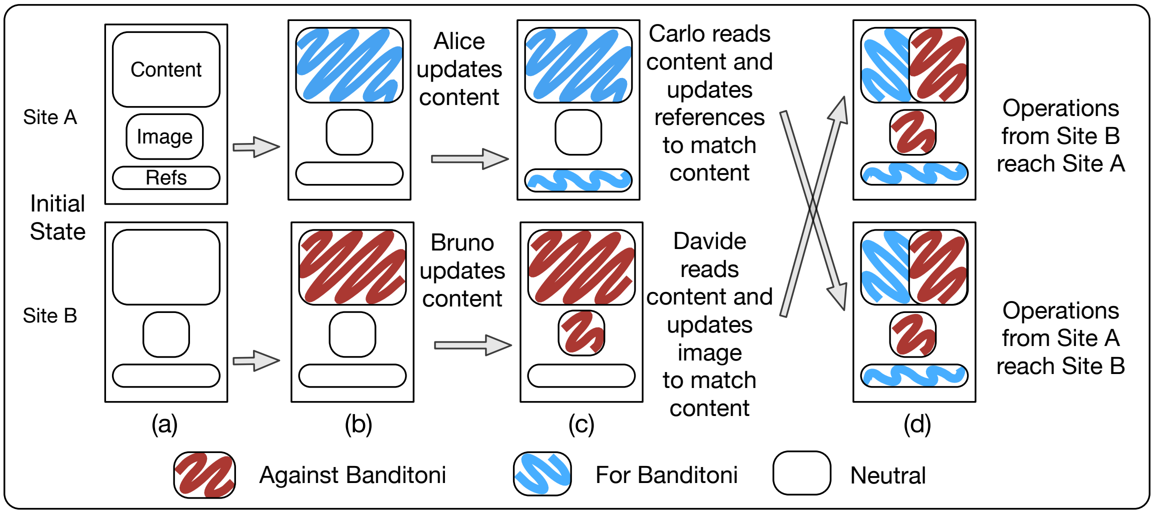

Site A
Initial State
Site B
Content
Image
Refs
Alice updates content
Bruno updates content
Carlo reads content and updates references to match content
Davide reads content and updates image to match content
Operations from Site B reach Site A
Operations from Site A reach Site B
(a)
(b)
(c)
(d)
Against Banditoni
For Banditoni
Neutral

*Figure 5.1: Weakly-consistent Wikipedia*

(Figure 5.1(a)). The page in our example, about a controversial politician, Mr. Banditoni, is frequently modified, and is thus replicated on two sites, A and B. Assume, for simplicity, that the page consists of just three objects—the content, references, and an image. Alice and Bruno, who respectively strongly support and strongly oppose Mr. Banditoni, concurrently modify the content section of the webpage on sites A and B to match their political views (Figure 5.1(b)). Carlo reads the content section on site A, which now favors Mr. Banditoni, and updates the reference section accordingly by adding links to articles that praise the politician. Similarly, Davide reads the update made by Bruno on site B and chooses to strengthen the case made by the content section by updating the image to a derogatory picture of Mr. Banditoni (Figure 5.1(c)). Eventually, the operations reach the other site and, although nothing in the preceding sequence of events violates causal consistency, produce the inconsistent state shown in Figure 5.1(d): a content section that exhibits a write-write conflict; a reference section in favor of Mr. Banditoni; and an image that is against him. Worse, there is no straightforward way for the application to detect the full extent of the inconsistency: unlike the explicit conflict in the content sections, the discrepancy between image and references is purely semantic, and would not trigger an automatic resolution procedure.

To the best of our knowledge, this scenario presents an open challenge to existing weakly-consistent systems, which exhibit at least one of the following two properties:

(*i*) *Syntactic conflict resolution.* To maintain the abstraction of sequential storage, many systems use fixed, syntactic resolution policies to reconcile write-write conflicts [118]. Deterministic writer-wins (DWW), for example, resolves write-write conflicts identically at all sites, ensuring that applications never see conflicting writes and guaranteeing eventual convergence. In our example, this policy would choose Bruno's update. However, this is not sufficient to restore consistency, as it ignores the relationship between the content, references, and images of the webpage. The datastore's greedy attempt at syntactic conflict resolution is not only inadequate to bridge this semantic gap, but leads to losing valuable information (here, Alice's update).

(*ii*) *Lack of cross-object semantics.* Some systems choose to push conflict resolution to the application [62, 187], but on a per-object basis only. Though more flexible than a purely syntactic solution, this approach, which reduces conflict resolution to the merging of explicitly conflicting writes, is still overly narrow. For example, it would not address the inconsistencies, such as the one between the references and the image, that do not produce a write-write conflict. Yet, the effects of a write-write conflict on an object do not end with that object: Carlo and Davide update references and images as they do because they have *read* the conflicting updates to the original content section. Indeed, any operation that depends on one of two conflicting updates is potentially incompatible with all the operations that depend on the other: the shockwaves from even a single write-write conflict may spread to affect the state of the entire database.

There is currently no straightforward way for applications to resolve consistently the kind of multi-object, indirect conflicts that our example illustrates. Transactions [118, 119], an obvious candidate, are powerless when the objects that directly or indirectly reflect a write-write conflict are updated, as in our example, by different users. After Bruno's update, the application has no way to know that Davide's update is forthcoming: it must therefore commit Bruno's transaction, forcing Bruno's and Davide's updates into separate transactions. Nor would it help to change the granularity of the object that defines the write-write conflict—in our example, by making that object be the entire page. It would be easy to correspondingly scale up the example, using distinct pages that link each other. Short of treating the entire database as the "object", it is futile to try to define away these inconsistencies by redrawing the objects' semantic boundaries.

## 5.2 Bridging the gap: branches

TARDiS' design is motivated by the belief that isolating ALPS applications from the harsh but inescapable reality of independent conflicting writes, and from the resolution process that they require, is a well-intentioned fallacy. This fallacy is driven both by looking at consistency in a *bottom-up* fashion, starting from the system that implements these guarantees, and by viewing objects as the unit of conflict resolution. This dissertation instead argues that one should look at consistency in a top-down fashion, starting from the clients that perceive these guarantees, and should consider *states* as the unit of conflict resolution. TARDiS thus embraces transparency. By default, applications execute on a branch, and hence perceive storage as sequential. But when anomalies arise, TARDiS provides two novel features that simplify reconciliation.

First, it exposes applications to the resulting independent branches, and to the states at which the branches are created (*fork points*) and merged (*merge points*). Second, it supports atomic merging of conflicting branches and lets applications choose when and how to reconcile them (§5.2.1). These features allow TARDiS to offer ALPS applications the opportunity to pursue, down to each site's local datastore, an intriguing notion: that of turning weak consistency, through a bit of system-design judo, from a weakness to an unlikely strength (§5.2.2).

### 5.2.1 State branching and merging

As the discussion in Section 5.1 has illustrated, even a single write-write conflict has the potential to affect the entire state of a database. In essence, conflicting operations fork the entire state of the system, creating distinct *branches*, each tracking the linear evolution of the datastore according to a separate thread of execution. The Wikipedia example hence consists of two branches: one in support of Banditoni, and one against him. Elevating branches to the datastore's fundamental abstraction has two complementary advantages. First, users that operate within a given thread of execution continue to perceive the application's state as evolving linearly. Second, when it becomes necessary to alert users to the existence of concurrent updates that conflict with that linear view, branches are the natural unit of merging.

Resolving conflicts in ALPS applications often requires semantic context. Replicas, however, only see a sequence of read/write operations and are unaware of the application-level logic and invariants that relate these operations [11]. Therefore, they should avoid deterministic quick fixes, and instead give applications the information they need to decide what is best. Branches, together with their fork and merge points, naturally encapsulate such information: they make it easy to identify all the objects to be considered during merging and pinpoint when and how the conflict developed. This context can reduce the complexity and improve the efficiency of automated merging procedures, as well as help system administrators when user involvement is required. In our example, a Wikipedia moderator presented with the two conflicting branches would be able to reconstruct the events that led to them and handle the conflicting sources according to Wikipedia's guidelines [197]. Note that merging need not simply involve deleting one branch. Indeed, branching and merging states enables merging strategies with richer semantics than aborts or rollbacks [178].

### 5.2.2 Weak consistency end-to-end

Write-write conflicts in distributed systems are not restricted to remote sites: conflicting operations can also happen locally. Unlike remote conflicts, however, they are immediately detectable, and hence they are typically handled by the datastore through locking or rollback. When considering the specific nature of ALPS applications, however, two observations bring this common-sense approach into question. First, desirable as it may be, the abstraction of a sequential store cannot be preserved end-to-end: at the distributed system level, it falls apart. Second, the design complexity of having to program against the possibility of remote conflicts is already factored into ALPS applications, whose semantics often support simple merging procedures.

These observations lead us to explore an unconventional proposition: design and implement a datastore for ALPS applications with branching as its fundamental abstraction, used to model conflicts end-to-end, from the level of the distributed system down to that of local storage. This stance does not simply have an aesthetic appeal: eliminating locks and rollbacks from the performance critical path offers the potential, through a lightweight implementation of branching, to improve throughput at local sites.

Accordingly, TARDiS gives ALPS applications the option of handling local conflicts through branch-on conflict, rather than syncronization. Naturally, one must tread carefully: out-of-control branching can turn reasoning about the state of the system into a nightmare. TARDiS thus lets applications tune the degree of local branching allowed, so they can strike the balance between performance and complexity that best meets their requirements.

### 5.2.3 System Goals

The challenge is then to develop a datastore that can keep track of independent execution branches, record fork and merge points, facilitate reasoning about branches and, as appropriate, atomically merge them—while keeping performance and resource overheads comparable to those of weakly-consistent systems. Such a system should satisfy the following three requirements:

- **Simple interface** The datastore should expose an interface that allows developers to navigate and manipulate branches; that interface should minimize any increase in complexity and need to modify legacy code.
- **Good Performance** The datastore should efficiently create, track, and merge branches; its performance should match or surpass that of a storage system that is strictly sequential and does not keep track of history.
- **Minimal Space Overhead** The datastore should have a reasonable memory footprint and manage efficiently the space overhead associated with keeping multiple executions and their fork points.

The rest of this chapter presents the design of TARDiS, focusing on how it satisfies these three requirements.

## 5.3 TARDiS Architecture

The TARDiS transactional key-value store tracks conflicting execution branches using three mechanisms: *branch-on-conflict*, *inter-branch isolation*, and *application-specific merge*. TARDiS uses multi-master asynchronous replication: transactions first execute locally at a specific site, and are then asynchronously propagated to all other replicas. Each replica consists of four components: a storage layer, a consistency layer, a garbage collector unit, and a replicator service (Figure 5.2).

| Constraint | B | E | Description |
|---|---|---|---|
| *Any* | √ | √ | Always Satisfies |
| *Serializability* | | √ | Guarantees Serializability |
| *Snapshot Iso* | | √ | Guarantees Snapshot Isolation |
| *Read Committed* | | √ | Guarantees Read Committed |
| *No Branching* | | √ | State has no children |
| *K-Branching* | | √ | State has fewer than k-1 children |
| *Parent* | √ | | State where client last committed |
| *Ancestor* | √ | | Child of client's last committed state |
| *State Identifier* | √ | | State ID matches the specified ID |

*Table 5.1: Begin (B) and end (E) constraints supported by TARDiS*

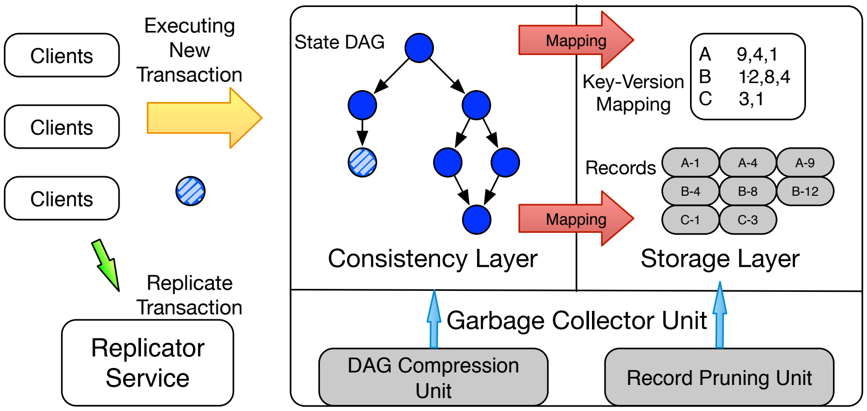


*Figure 5.2: TARDiS architecture*

The *storage layer* stores records in a disk-backed B-Tree. Currently, every site stores a full copy of the database, though TARDiS can be extended to support data partitioning (§5.5.4). TARDiS is a multiversioned system: every update operation creates a new record version. The mapping between versions and keys is stored in an in-memory cache for fast traversal.

The *consistency layer* tracks branches with the help of a directed acyclic graph, whose vertices correspond to logical *states* of the datastore: each transaction that updates a record generates a new state. TARDiS' logic is geared towards efficiently mapping the consistency layer to the storage layer: when a transaction creates a new state, it extends the State DAG by appending the state to its chosen branch of execution. Any newly created record version is marked by this state's identifier. When a transaction executes a get operation, it uses information contained in the State DAG to determine which object versions are visible to its branch.

| S | M | return type | method |
|---|---|---|---|
| √ |  | transaction | begin(*beginConstraint*) |
|  | √ | transaction | beginMerge(*beginConstraint*) |
| √ | √ | void | put(*key, value*) |
| √ |  | value | get(*key*) |
|  | √ | value | getForID(*key, StateID[ ]*) |
|  | √ | key[] | findConflictWrites(*StateID[ ]*) |
|  | √ | forkPoints[] | findForkPoints(*StateID[ ]*) |
| √ | √ | abort\|commit | commit(*endConstraint*) |

*Table 5.2: TARDiS API - S:single mode, M:merge mode*

TARDiS' *garbage collector unit* comprises a DAG compression submodule and a record pruning submodule. DAG compression periodically discards intermediate states that are no longer needed, and record pruning then removes the associated object versions. Garbage collection allows TARDiS' to maintain memory and storage overheads that are comparable to traditional weakly-consistent systems that do not track history.

Finally, the *replicator service* propagates committed transactions and applies remote transactions as appropriate.

## 5.4 Using TARDiS

To help weakly-consistent applications deal with the complexity of resolving the conflicts they encounter, TARDiS' API (Table 5.2) addresses three competing concerns: (*i*) minimizing programming complexity, (*ii*) simplifying reasoning about concurrent branches, and (*iii*) controlling the degree of local branching.

### 5.4.1 Interface

To ease programming, TARDiS can operate in either *single mode* or *merge mode*. In the default *single mode*, the programmer is allowed to (transactionally) read from and write to a single branch. Programming proceeds exactly as in traditional transactional systems, except that programmers must now select a branch to operate on. Thus, porting an application to TARDiS requires only adding a branch-selection call. Figure 5.3 illustrates this point by sketching the implementation of a simple counter. The two single mode operators, `increment` and `decrement`, now take as parameters predicates that specify the properties of the desired branch (more on these predicates below), but these operators are otherwise implemented exactly as one would on sequential storage.

In *merge mode*, programmers can instead explicitly reconcile conflicting branches via *merge transactions*, that read from multiple states and write back to a single, merged state. TARDiS' merge mode allows users to perform cross-object resolution atomically: this simplifies conflict resolution significantly, much like transactions simplify application logic by allowing users to modify multiple objects atomically.

When reconciling branches, applications will typically (*i*) detect conflicting objects; (*ii*) identify where branches forked; and (*iii*) determine the values of conflicting keys at this fork point (§5.2). To help applications obtain the information that they need for merging, TARDiS adds three new API calls: `findConflictWrites`, `findForkPoints`, and `getForID`. `findConflictWrites` returns the list of objects with conflicting values across all selected branches, freeing programmers from the need to implement application-level mechanisms for tracking what has to be resolved. `findForkPoints` returns, for a given set of states, the structured set of fork points that reveals the branching structure of the corresponding State DAG. For simplicity of exposition, we restrict ourselves to the case where `findForkPoints` returns a single fork point. Finally, `getForID` allows the application to request any object version, freeing application programmers from the need to track how the datastore evolves. We expect applications to call this function primarily to obtain object values at the fork point(s), and to use this information to resolve conflicts in an application-specific way before writing the merged value back. Though TARDiS supports concurrent merges for full flexibility, we expect that, for simplicity, most applications will restrict merging to a single site.

TARDiS helps applications reason about concurrent execution branches and control the degree of local branching through *begin* and *end constraints*—predicates associated with begin and commit commands that specify which branch a transaction can execute from. Intuitively, begin constraints select what states the transaction can read, while end constraints specify what conditions must hold upon commit. TARDiS supports the constraints listed in Table 5.1: along with their union and intersection, they are sufficiently flexible to express traditional database isolation levels, such as

```
func increment(counter)
 Tx t = begin(AncestorConstraint)
 int value = t.get(counter)
 t.put(counter, value + 1)
 t.commit(SerializabilityConstraint)

func decrement(counter)
 Tx t = begin(AncestorConstraint)
 int value = t.get(counter)
 t.put(counter, value - 1)
 t.commit(SerializabilityConstraint)

func merge()
  Tx t = beginMerge(AnyConstraint)
  forkPoint forkPt =
          t.findForkPoints(t.parents).first
  int forkVal = t.getForID(counter,forkPt)
  list<int> currentVals =
          t.getForID(counter, t.parents)
  int result = forkVal
  foreach c in currentVals
    result += (c - forkVal)
  t.put(counter,result)
  t.commit(SerializabilityConstraint)
```

*Figure 5.3: TARDiS' counter implementation*

serializability and snapshot isolation [31], as well as distributed-system guarantees such as read-my-writes [185]. For example, an application could use the *Ancestor* begin constraint and the union of the *Serializability* and *No Branching* end constraint to mimic the local behavior of a traditional sequential storage and achieve causal consistency globally. Similarly, applications using the *Ancestor* begin constraint and the *Snapshot Isolation* end constraint would always see their own writes and maintain snapshot isolation within a branch. Alternatively, the *K-Branching* constraint explicitly bounds the degree of branching in the system, giving developers the ability to balance the performance benefit of allowing local branching with the degree of divergence that this entails.

By default, TARDiS uses the *Ancestor* begin constraint and *Serializability* end constraint. Though some applications may benefit from multiple consistency levels [186], in most cases this default will let programmers write almost unmodified code, without explicitly specifying constraints.

### 5.4.2 Coding with TARDiS

TARDiS' greater fidelity in capturing the context that leads to conflicting operations is key to reducing the complexity of developing ALPS applications and improving their performance. This feature is evident even in simple programs, such as the counter presented in Figure 5.3. In a traditional, non-branching causally consistent system, counters are often implemented as two separate vector clocks (one for increment operations, the other for decrement operations) with an entry for each replica [171]. Reading the value of a counter requires adding the values in the increment vector and substracting those in the decrement vector. Similarly, applying a remote operation requires merging the local increment and decrement vectors with those of the incoming remote operation by taking the maximum of each corresponding vector element. Thus, all operations, including non-conflicting reads, in effect involve a merge: the system must reconstruct the global view from each replica's local view, at a cost linear in the number of replicas.

In TARDiS, instead, single mode and merge mode are cleanly separated. In single mode (see Figure 5.3), increment and decrement operations access a single field, just as they would in a non-distributed scenario. Access to fork points makes merge operations both simpler and more flexible. In TARDiS, merging distinct counter branches is easy: one can simply compute the merged value by summing, for all branches, the difference between the value of the counter at the fork point and the current value for the branch. The application can then choose to merge branches periodically, during periods of low load, or to do so more frequently, if the counter nears boundary values.

The benefits of the TARDiS API become especially notable in examples that involve multiple objects with richer semantics. Consider the case of an online game store that sells both board games and extension packs that are only playable after buying the corresponding board game. The store tracks inventory by keeping a counter object per each item it sells, and associates each customer with a shopping cart. Suppose Alice and Bruno have, on different sites, both bought the last copy of a boardgame. Bruno has additionally bought an extension pack. Figure 5.4 gives the simplified pseudocode of the merging process.[2] On a merge, the application iterates over the keys in conflict and detects which items have been bought on different sites (lines 19-44), reconciling counter values through the merging process discussed above (lines 22-24). When the counter for a particular item falls below zero, as in our scenario, the merging logic must select the shopping cart from which the oversold items should be removed, while maintaining the invariant that no user should buy the extension pack without the game. The application has several options: it can choose, as does the

[2] For clarity of explanation, we assume two branches only.

```
func buy(customer, item, cart)
  Tx t = client.begin(AncestorConstraint)
  list<itemId> items = t.get(cart.items)
  items += item.itemId
  t.put(cart.items, items)
  int stock = t.get(item.stock)
  t.put(item.stock, stock-1)
  list<cartId> carts = t.get(item.carts)
  carts += cartId
  t.put(item.carts, carts)
  t.commit(SerializabilityConstraint)

func merge()
  Tx t = client.beginMerge(AnyConstraint)
  list<item> conflictItems =
           t.findConflictWrites(t.parents)
  forkPoint forkP =
           t.findForkPoints(t.parents).first
  foreach item in conflictItems
    list<int> stockVals = new list
    int forkPtStock = t.getForID(item.stock,forkP)
    foreach branch in t.parents:
      stockVals.add(t.getForID(item.stock,branch))
      int newStock = mergeCounter(stockVals, forkPtStock)
    if(newStock > 0)
      t.put(item.stock,newStock)
      confirmItem(item.itemId, item.carts)
    else
      // get orders since fork point
      set<cartId> carts = new set
      foreach branch in t.parents:
        carts+= t.get(item.carts)
      carts = carts - t.getForID(item.carts,forkP)
      carts.sortBy(valueOfCart)
      foreach cart in carts
        if (forkPtStock > 0)
          // confirm item until run out
          --forkPtStock
          confirmItem(item.itemId, cart.cartId)
        else
          // apologize to other users
          removeRelatedItems(item,cart)
          sendApology(cart.clientId)
      t.put(item.stock,0)
  t.commit(SerializabilityConstraint)
```

*Figure 5.4: TARDiS' shopping cart implementation*

pseudocode in Figure 5.4 (lines 30-44), to leave Bruno with both the game and the expansion pack, and send an apology to Alice, maximizing its overall profit. Alternatively, it can observe that Alice is a better customer than Bruno, and choose to privilege customer loyalty.

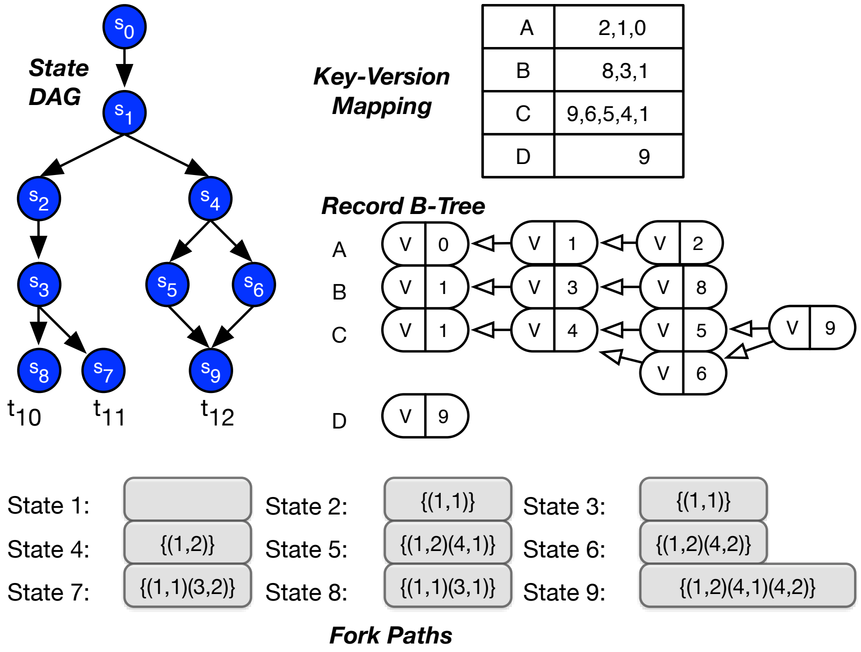
In current systems, merging that spans conflicts across multiple objects and requires application involvement is not achievable without significant engineering effort. Through the combination of branch-on-conflict, inter-branch isolation, and application-specific merge, TARDiS makes it easy for applications to acquire the context that led to such conflicts and empowers them with the flexibility and expressiveness necessary to meaningfully reconcile them.

## 5.5 Design and Implementation

To ensure that branches are cheaply created, maintained, and merged, TARDiS proceeds as follows. A transaction starts by identifying a most recent state that satisfies its begin constraint: this is the state from which the transaction can read (its *read state*). Likewise, upon commit, the transaction identifies a most recent state since its read state that satisfies the end constraint (the *commit state*). Since this process is identical for all transactions and independent of concurrently executing transactions, it naturally leads to state forking and to transactions aborting. If two concurrent transactions select the same state from which to commit, a new branch is created; alternatively, if no state satisfies a transaction's end constraint, the transaction aborts. There is thus no conceptual difference between sequential execution and forking, and TARDiS' design ensures that the implementation is similarly uniform. We describe this process in further detail below, focusing on the life of a particular transaction and relying on Figure 5.5 to illustrate the TARDiS' main datastructures.

### 5.5.1 Basic Operation

**Begin Transaction: Read State Selection** A TARDiS transaction begins by selecting a branch. To choose one among the most recent suitable states, the transaction conducts a breadth-first search through the State DAG from its leaves up, looking for a state that satisfies the transaction's begin constraint. For example, given the State DAG in Figure 5.6(a), a newly executing transaction ($t_{13}$) would visit, in order, $s_8$, $s_7$, and $s_4$ and select the latter—the first state to satisfy the transaction's begin constraint—as its read state. Some constraints may require states to store additional information



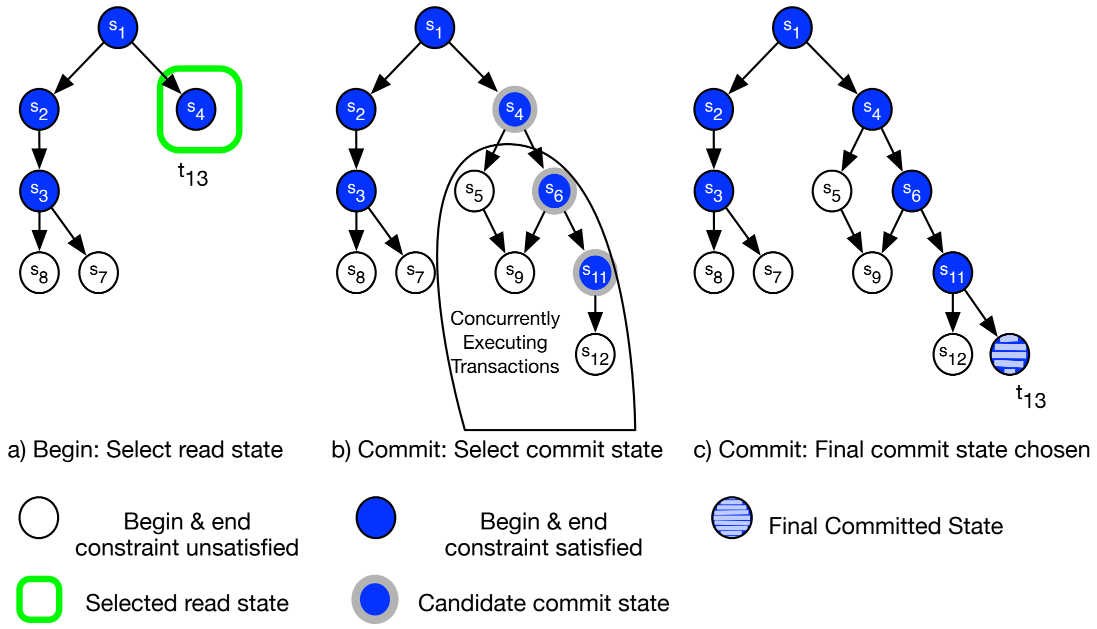
| | |
|---|---|
| A | 2,1,0 |
| B | 8,3,1 |
| C | 9,6,5,4,1 |
| D | 9 |

*Figure 5.5: Main system datastructures*

beyond pointers to their parents or children: the *Parent* and *StateID* constraints, for example, require all states in the DAG to be uniquely identifiable. Similarly, the *Serializability* and *Snapshot Isolation* (end) constraints demand to store, with each state, the read and write sets of the transaction responsible for creating it. In practice, there is often a unique state that satisfies the begin constraint; if instead multiple states are suitable, TARDiS simply selects one of them at random.

**Commit Transaction: Commit State Selection** The process for committing a transaction is similar: it requires identifying the most recent state, since the read state, that satisfies the end constraint. At commit, the transaction first checks whether its read state satisfies the end constraint. If it does not, the transaction aborts, as it read from an invalid state. Otherwise, the transaction checks whether more recent states also satisfy the end constraint. In effect, starting from its read state, the transaction "ripples" down the DAG, stopping before the first state that no longer satisfies the end constraint. Figure 5.6(b) illustrates the process. Transaction $t_{13}$ first checks that the read state $s_4$ satisfies the end constraint, and then ripples down through states $s_6$ and $s_{11}$ until it encounters $s_{12}$. As $s_{12}$ does not satisfy $t_{13}$'s end constraint, $t_{13}$ commits after $s_{11}$, creating a new branch (Figure 5.6(c)).

**Reading records** Logically, non-read-only transactions create a new database state every time they



s1
s2
s4
t13
s3
s8
s7
a) Begin: Select read state
s5
s6
s9
s11
s12
Concurrently Executing Transactions
b) Commit: Select commit state
c) Commit: Final commit state chosen
Begin & end constraint unsatisfied
Begin & end constraint satisfied
Final Committed State
Selected read state
Candidate commit state

*Figure 5.6: Transaction commit logic*

execute. Storing each of these states as physically distinct instances is unsustainable. Write operations therefore simply create new record versions, and TARDiS relies on system logic to reconstruct the appropriate state for reads. This copy-on-write approach is similar to that of most multiversioned concurrency control (MVCC) systems [32], but with one key difference: TARDiS must not only provide support for stale snapshots, but also for divergent snapshots. As in traditional MVCC systems, to read a record a transaction must determine the most recent record version in the key-version map. Unlike MVCC systems, however, TARDiS must also ensure that the record version belongs to the branch it has selected. The selected version is then read from the record B-tree and returned.

To quickly determine whether a record version belongs to the selected branch, TARDiS abandons traditional dependency checking [64, 118, 124], which quickly becomes a bottleneck in causally consistent systems [24, 64], and instead relies on *fork point checking*. In TARDiS, a branch is summarized only by its fork points (§3). A fork point is a tuple $(i, b)$, indicating that the current state is a descendant of the $b^{th}$ child of state $i$. Together, the set of fork points for a branch denotes its *fork path*. A record version belongs to the selected branch if the fork path of the state associated with this record version is a subset of the fork path of the transaction's read state (see pseudocode in

```
descendantCheck($x$, $y$):
  if $x.id = y.id$ then return true
  else if $x.id > y.id$ then return false
  else if $x.path \not\subseteq y.path$ then return false
  else return true
```

*Figure 5.7: Check if state $y$ can see records associated with state $x$*

Figure 5.7). Figure 5.5 shows the fork path associated with each state: one can quickly determine that $s_7$ is on the same branch as $s_3$, as the fork path of $s_3$ is a subset of that of $s_7$. Similarly, $s_9$ is on the same branch as both $s_5$ and $s_6$.

By capturing *conflicts* (fork points) instead of dependencies, fork paths allow TARDiS to track concurrent branches efficiently. The small size of fork paths (conflicts are a small percentage of the total number of operations) not only limits memory overhead, but makes it possible to check quickly whether two states are on the same branch.

To guarantee that transactions select the most recent version, TARDiS keeps a topological sort of versions in each key-version mapping entry. Consider, for example, key C in Figure 5.5. The list stored in the key-version mapping is a topological order of the true structure of C's record versions, as saved in the record B-tree. As transactions iterate through the list, the first record version identified as belonging to the selected branch will necessarily be that branch's most recent version.

Putting this together, consider Figure 5.5. It shows three transactions $t_{10}$, $t_{11}$, and $t_{12}$, with respective read states $s_8$, $s_7$, and $s_9$. $t_{10}$ would read *v2* for key A, *v8* for B, *v1* for C, and *empty* for D. Similarly, $t_{11}$ would read *v2*, *v3*, *v1*, and *empty* respectively for keys A to D. Finally, $t_{12}$ would read *v1*, *v1*, *v9*, and *v9*.

**Writing values** To handle uniformly branching and non-branching scenarios, TARDiS's write logic ensures (*i*) that writes, whether conflicting or not, never block, and (*ii*) that updating the appropriate record on the correct branch is cheap.

Both aims can be very simply achieved by pushing most of the work to reads. As long as writes preserve the topological order of versions in the key-version mapping, the read logic ensures that the appropriate version is returned. Hence, a write operation in TARDiS simply creates a new record version storing the transaction's state identifier and the pertinent data, inserts it into the record B-tree, and appropriately updates the corresponding key-version mapping. Since state identifiers (and thus record identifiers) are monotonically increasing along a branch, TARDiS can cheaply maintain a topological order as a sorted list (more precisely, as a lock-free skip list). Thus, independently of whether conflicting writes occurred, all a transaction needs to do to complete a write is to insert the new version into the skip list.

**Read-only Transactions** Since read-only transactions cannot induce conflicts, TARDiS does not add them to the State DAG. This optimization limits unnecessary DAG growth, easing pressure on the garbage collector.

### 5.5.2 Merge Transactions

Merge transactions in TARDiS function similarly to single mode, but with a key difference: they select multiple read states, and hence operate on multiple branches. In merge mode, the application is thus directly exposed to any conflicting writes that forked the state of the datastore.

Merge transactions must atomically reconcile all conflicting objects, writing back a single merged state. As stated previously (§5.2), merging often requires providing the application with detailed information about the structure of the State DAG, including how branches diverged and the values of conflicting objects at the branches' fork points. This information must be made available efficiently, as a slow merge will stop applications from seeing up-to-date values. TARDiS thus provides an API to aid applications understand branch divergence. It consists of three operations: `findForkPoints`, to identify the fork point(s) of a set of branches; `findConflictWrites`, to list conflicting keys across branches; and `getForID`, to obtain, at the specified state, the record corresponding to a given key. TARDiS leverages the properties of the existing storage and consistency layers to make these operations fast. It implements `findForkPoints` by identifying the fork point(s) of the merge transaction's read states. For `findConflictWrites`, TARDiS similarly identifies the fork points of the branches and computes the conflicting key list from the write set of each intermediate state. Finally, for `getForID`, it uses the key-version mapping to select the appropriate record for a given state.

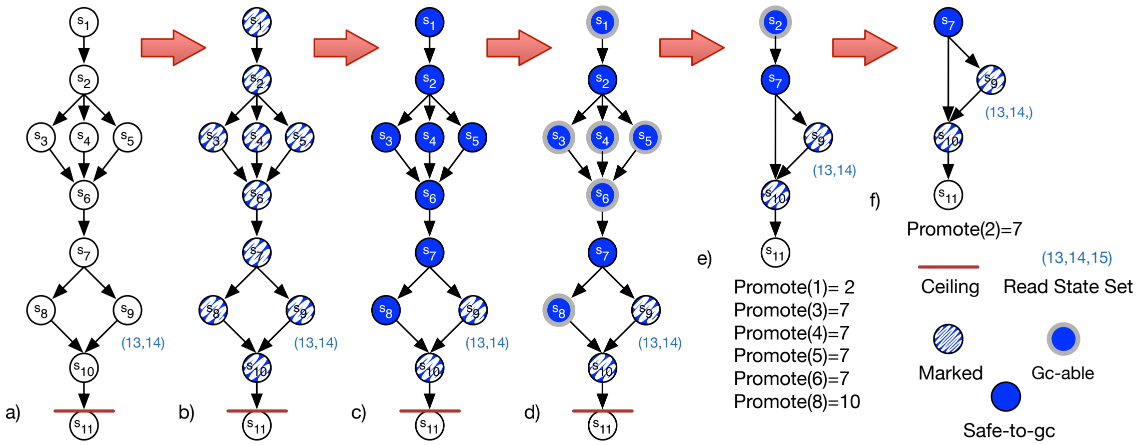


*Figure 5.8: Path Compression Algorithm - Ceiling placed above $s_{11}$. $s_8$ and ancestors are marked as safe-to-gc; since $s_{10}$ is the read state for several pending transactions, it cannot be marked as safe-to-gc. Non fork points safe-to-gc states are marked as gc-able and deleted.*

### 5.5.3 Garbage Collection

Most traditional databases store only the active frontier of records, keeping space overhead manageable as old transactions commit. TARDiS, on the contrary, stores by default all stale and parallel versions or states. For performance and efficiency under finite storage, TARDiS implements an aggressive, three-pronged garbage collection policy that runs concurrently with regular operations: (*i*) users place *ceilings* on specific states, promising never to use any state that precedes them as a read state; (*ii*) a path-compression algorithm compresses the State DAG to remove all states that are neither fork points nor leaf states; and (*iii*) a record-promotion algorithm removes record versions that are no longer visible because of ceilings or path compression.

**Path Compression** Path compression relies on one core heuristic: since most merging policies only require the fork points and the leaf states of a given execution, all intermediate states can be safely removed from the State DAG. In effect, this reduces the DAG to explicitly tracking the *nearest conflict dependency* using a three-staged process (Figure 5.8).

First, a *ceiling marking* bottom-up pass marks all the states above a recently placed ceiling. Marked states can no longer be selected as read states, ensuring that no new transaction starts above a ceiling (Figure 5.8(b)). Second, a *safe to garbage-collect* top-down pass labels as *safe-to-gc* all marked states (*i*) that are not currently selected as read states by some executing transaction and (*ii*) whose ancestors are also safe to garbage collect. This pass prevents committing transactions from rippling down deleted states and ensures that a state will only be deleted after all its ancestors have been deleted (Figure 5.8(c)). Finally, a *garbage collecting pass* marks safe-to-gc states that are not fork points as *garbage collectable* (*gc-able*). Garbage-collectable states are then "promoted" by mapping their state identifier to that of their most recent non-deleted child. All accesses looking up a deleted state are henceforth forwarded to the promoted state. In effect, the child node takes over the identity of its parent as well as its own, allowing for the parent to be garbage collected (Figure 5.8(d.e.f)). Consider in Figure 5.8 an object D that is last modified in $s_1$. Its state identifier is therefore 1. All transactions whose read states are a descendant of $s_1$ see D. But, when $s_1$ is garbage collected, any transaction that tries to read D and thus looks up the fork path of $s_1$ would fail. Promoting $s_1$ to $s_7$ ensures that any such transaction is redirected to $s_7$. Once promoted, garbage collectable states can safely be removed.

**Record Promotion** Next, TARDiS deletes record versions that are no longer needed. Record versions associated with previously deleted states are promoted to their first non-garbage-collected descendants and updated to reflect their new identifier. This promotion creates long chains of records with the same state identifier. Since TARDiS' get/put algorithm returns only the most recent visible record, none but the first of the promoted records that share an identifier will ever be accessed: the rest can therefore be safely discarded. A full record-promotion pass, as in path compression, hence ensures that the only record versions maintained are those that are either current or at a fork point.

### 5.5.4 Replication

The Replicator (§5.3) uses a gossip protocol [2] to asynchronously propagate locally executing transactions. These transactions carry a *StateID* constraint that specifies the state to which they should be applied. The Replicator applies a newly received transaction if the required parent state is present. If not, the transaction is cached to be appended later. The *StateID* constraint removes the need to track expensive dependency meta-data: it reduces dependency checking to looking up whether the state with the corresponding id is present in the remote DAG, which can be done in constant time.

Garbage collection is triggered by each local Replicator and can operate either optimistically or pessimistically. When pessimistic, TARDiS garbage collects states only after receiving unanimous consent from all Replicators [124]. As this can cause garbage collection to trail significantly during partitions, optimistic mode lets sites garbage collect states independently. If a replica later needs a state it garbage-collected, the replica simply retrieves the missing information from the appropriate replica. This may cause some transactions to block. If an application erroneously places a ceiling that causes states to be prematurely garbage collected, TARDiS simply aborts the transactions that try accessing the missing states.

The current prototype of TARDiS does not support data partitioning, but it can be extended to support this feature by using an approach similar to the one taken by COPS [118]—in essence, by executing distributed transactions within a datacenter (with the State DAG collocated with the transaction manager) and replicating transactions asynchronously across datacenters.

### 5.5.5 Fault Tolerance and Recovery

TARDiS guarantees consistency, atomicity, and (optionally) durability if, at all times, the write operations of all transactions present in the State DAG are observable by future transactions. TARDiS maintains this invariant across failures by logging, at transaction commit time, the id of the corresponding commit state, its parent state(s) ids, and the transaction's write set keys.

**Recovery** During recovery, TARDiS reconstructs the State DAG and key-version mapping by iterating chronologically over the log. For each log entry, the recovery process (*i*) inserts a new state in the DAG with matching id and adds it as a child of the states with matching parent ids; and (*ii*) adds a new entry in the key-version mapping for each key in the recorded write set. Iterating over the log chronologically guarantees that no child is recovered before its parents; the skip-list implementation of the key-version mapping guarantees that the order of entries in the key-version mapping is preserved across failures.

**Asynchrous Flush** To mitigate the overheads of writing to disk, TARDiS offers the option of asynchronously flushing both record updates and the commit log (at the cost of durability). To preserve atomicity, TARDiS ensures that the commit log is flushed sequentially and further checks, on recovery, whether each entry in the write set has been made persistent to stable storage. If only part of a transaction's effect has been made persistent, the corresponding state and all subsequent states are discarded. Orphaned records (resulting from operations that belong to, or depend on, partially applied transactions) have no effect on correctness, as it is the DAG and key-version mapping that determine what can be read. These records are simply eventually garbage collected.

**Checkpointing** To reduce log size, TARDiS periodically takes non-blocking checkpoints by (*i*) selecting a state id $s_c$, (*ii*) flushing all outstanding writes, and (*iii*) saving every DAG state with id smaller than $s_c$.

**Replication** To recover transactions that have committed elsewhere but are lost locally, the recovery process broadcasts a vector with the id of the latest surving state received from each Replicator. Replicas respond by sending states and records more recent than their corresponding entry in the vector. If these states have already been garbage collected, it may be necessary to send the full DAG.

### 5.5.6 Implementation notes

Our prototype consists of 15K lines of Java in two configurations. In TARDiS-BDB, record persistence to disk is via BerkeleyDB with concurrency control turned off, while TARDiS-MDB depends on MapDB [125]. We rely on Google Protobuffers v2.4.5 for message serialization and on the Netty networking library for inter-site communication.

## 5.6 Evaluation

TARDiS proposes branching as the fundamental abstraction to model conflicts end-to-end. To quantify the cost and benefits of this new abstraction with respect to both performance and complexity, we first benchmark the performance of the system itself. We then use the abstraction to program two representative ALPS applications: CRDTs [171], a library of scalable, weakly consistent datatypes (counters, sets, maps, and more); and Retwis [165], a Twitter clone. We find that using state branching/merging cuts the number of lines of code in half, while judicious branch-on-conflict yields a speedup of between two and eight times.

### 5.6.1 Microbenchmarks

Our microbenchmarks answer the following questions:

1. What impact has expressiveness on performance?
2. What are the overheads of tracking state?
3. When does local branching improve performance?
4. What impact has expressiveness on performance?

**Evaluation Setup** Unless otherwise stated, we run our experiments on a shared local cluster of machines equipped with a 2.67GHz Intel Xeon CPU X5650 with 48GB of memory and connected by a 2Gbps network. Inter-machine ping latencies average 0.15 ms. Each experiment is run with three dedicated server machines, three dedicated Replicators and with clients spread equally among separate machines. To the best of our ability, all runs were performed in the absence of interfering workloads.

We compare TARDiS with both BerkeleyDB v6.2.31 Java Edition (**BDB**) and a custom OCC implementation (**OCC**) that uses BDB as its backend. BerkeleyDB, as a widely-used, pure-Java ACID datastore, provides a sound basis for comparison against our TARDiS prototype. We configure BerkeleyDB so that (*i*) read-write transactions are flushed to disk asynchronously; and (*ii*) all requests hit the cache. Our OCC implementation is based on a modified version of Kung et al.'s algorithm [107] that does not require read-write transactions to be verified against read-only transactions.

Each client issues transactions consisting of six operations in a closed loop, running for two minutes. We use a set of clients large enough to saturate the system. Read-only transactions contain only reads; read-write transactions contain three reads and three writes. We consider four transaction mixes differentiated by the ratio of read-only transactions to read-write transactions: Read-only (R-O, 100/0), Read-heavy (R-H, 75/25), Mixed (M, 25/75), and Write-Heavy (W-H, 0/100). We report only on the read-heavy and write-heavy workloads. We further consider two access patterns from the YCSB benchmark [55]: uniform and Zipfian (p=0.99). We report on TARDiS-BDB only: the performance of TARDiS-MDB is similar (approximately 10% better).

**Baseline TARDiS performance** We begin by establishing a baseline for the performance of TARDiS: we want to determine how it compares, when local branch-on-conflict is not enabled, against both our simple implementation of OCC and a commercial system like BDB. Transactions use *Ancestor* as begin constraint and the union of *Serializability* and *No Branching* as end constraint. We expect this setup to be common: it guarantees that each application sees its own writes and that the execution is serializable.

The throughput-latency graph in Figures 5.9 and 5.10 shows that, despite tracking the full system's history and paying the overhead of selecting a read and commit state, TARDiS-BDB performs similarly to BDB for both read-heavy and write-heavy workloads. TARDiS incurs a 10% slowdown as its begin and commit phase are more costly than in BDB. As contention increases, however, TARDiS-BDB's lock-free writes reduce the performance gap. In both cases, our OCC implementation lags behind. For the read-heavy workload, OCC must verify read-only transactions, which increases latency and reduces throughput. For the write-heavy workload, OCC suffers from a long validation phase. Though also optimistic, TARDiS's validation phase (commit state identification) is less costly: it requires checking only the write set of those transactions that already committed as children of the selected read state. In contrast, OCC requires checking against all concurrently committing transactions.

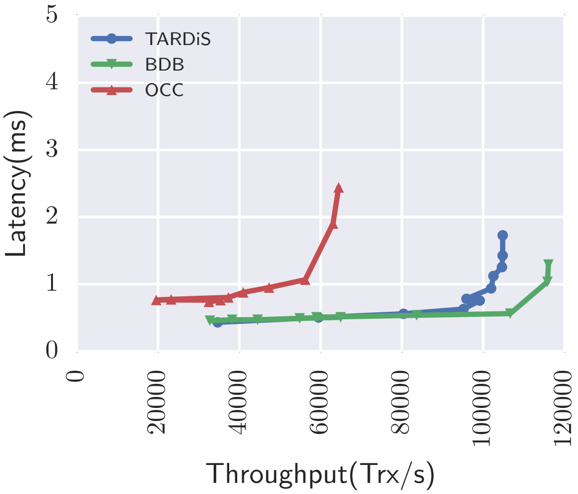


*Figure 5.9: TARDiS-BDB vs BerkeleyDB vs OCC - Read-Heavy*

**Impact of branching** TARDiS lets ALPS applications choose to branch on conflict rather than abort.

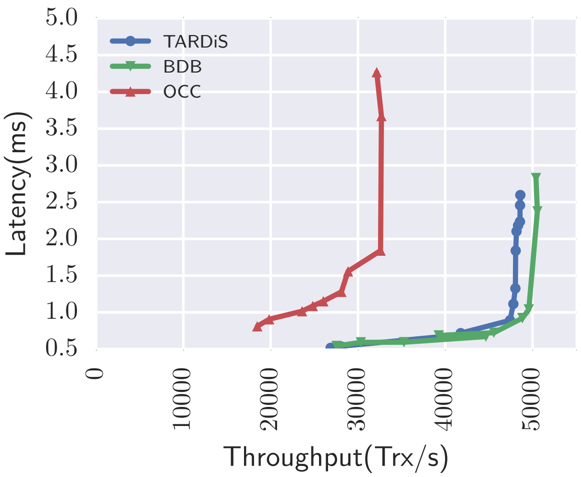


*Figure 5.10: TARDiS-BDB vs BerkeleyDB vs OCC - Write-Heavy*

| Workload | | Begin | Get | Put | Commit |
|---|---|---|---|---|---|
| **RH-Uniform** | TARDiS | 0.6 | 0.6 | 1 | 0.2 |
| | BDB | 0.3 | 0.4 | 1.2 | 0.1 |
| | OCC | 0.5 | 0.6 | 1.1 | 3.5 |
| **WH-Uniform** | TARDiS | 1 | 1.2 | 1.1 | 1.8 |
| | BDB | 0.6 | 0.8 | 2.7 | 0.1 |
| | OCC | 0.9 | 1.2 | 1.1 | 6.7 |
| **WH-Zipfian** | TARDiS | 1 | 1.4 | 1.2 | 0.8 |
| | BDB | 0.6 | 8 | 23 | 0.1 |
| | OCC | 0.4 | 0.9 | 1.2 | 9 |

*Table 5.3: Per-operation latency breakdown ($\times 10^{-2} ms$)*

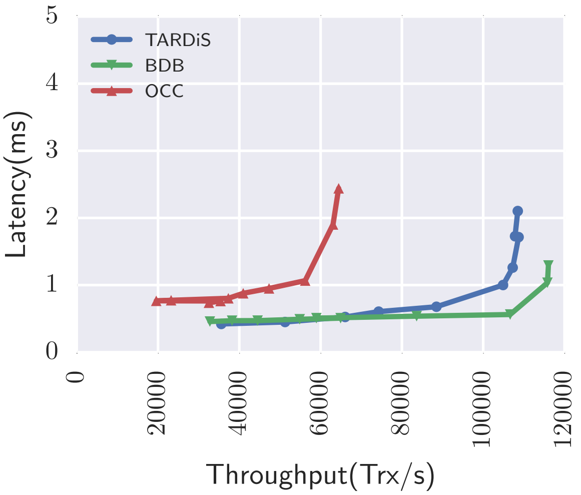


*Figure 5.11: Uniform Read-Heavy*

Unsurprisingly, we find that the relative benefits of branching increase with contention. Results are shown in Figures 5.11, 5.12, and 5.13 (all transactions run with branch-on-conflict enabled and with *Ancestor* and *Serializability* as, respectively, their begin and end constraint). Table 5.3 provides a per-operation breakdown of the same experiments, excluding network latency and retries. Figure 5.11 shows that, when contention is low, branching indeed does not help: TARDiS's performance is slightly lower than BDB's. By contrast, with higher contention (Figure 5.12), TARDiS outperforms BDB by 35%. The performance of BDB drops by half, as transactions wait longer for locks to become available, increasing the cost of gets and puts by a factor of almost two (Table 5.3 shows that reads take 0.004ms in the R-H workload, and 0.008ms in the W-H workload). The 37% throughput drop in TARDiS is due to several factors. First, the lack of read-only transactions: since all transactions now have to identify a commit state, the commit cost of the transaction increases (from Table 5.3: from 0.002ms to 0.018ms). Second, reads become more expensive, as more record versions need to be checked (from 0.006ms to 0.012ms). With both low and high contention, the throughput of OCC is bottlenecked by the verification phase. The relative performance increase of branching over sequential storage is most marked when requests follow a Zipfian workload in which a small number of objects, accessed very frequently, experience a high degree of contention (Figure 5.13). In this scenario, TARDiS outperforms BDB by a factor of eight. Whereas moving from a uniform to a Zipfian distribution causes BDB's performance to collapse (7x throughput decrease), TARDiS'

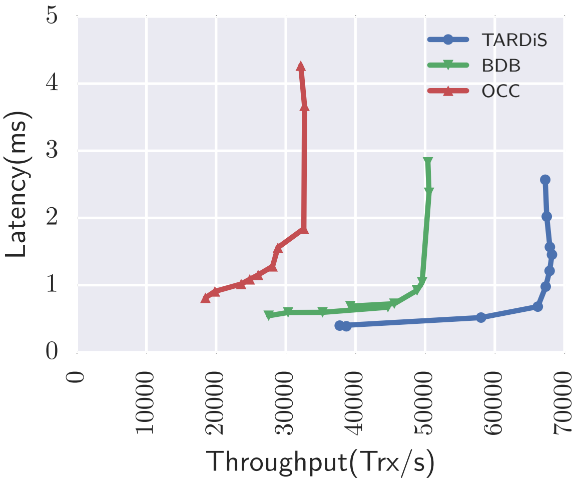


*Figure 5.12: Uniform Write-Heavy*

throughput is much less affected: the lock-free implementation of the write skip-list ensures that writes never block, even when conflicting. Table 5.3 confirms this: the cost of writes increases only moderately in TARDiS (from 0.011ms to 0.012ms). Similarly, the use of fork points to efficiently summarize the DAG means that, despite an eight-fold increase in the branching factor in the Zipfian workload, the cost of reads only increases by 16%. In contrast, locking causes the read and write time for BDB to increase by a factor of ten. OCC performs comparatively better, as it ensures that (*i*) at least one transaction will always commit, and (*ii*) readers will not block writers. Nonetheless, its high abort rate and expensive validation phase limit its throughput to a fifth of TARDiS'.

Branching, however, is not always beneficial. In the workload of Figure 5.14 transactions consist of a single write accessing the database uniformly, conflicts are rare, and locks are held for a very short period of time: here, branching does not help, but still incurs the cost of tracking past states. Since the current implementation of TARDiS' garbage collector is unable to keep up with the speed at which new states are generated, the increased memory pressure grows the number of stalls induced by garbage collection, causing TARDiS to perform 10% worse than BDB.

### Impact of constraint choice

The choice of begin and end constraints involves a complex trade-off between consistency and performance. To shed some light on this trade-off for our current implementation, we plot in Figure 5.15 the throughput of TARDiS for several different constraint choices for the same configuration (15

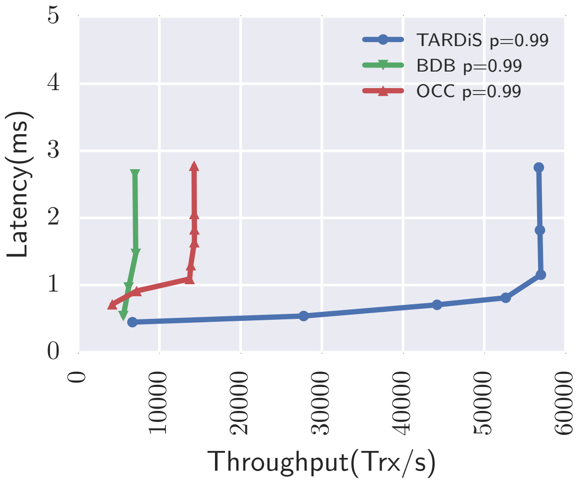


*Figure 5.13: Zipfian Write-Heavy*

machines, each with seven clients) that led TARDiS to reach the elbow in the write-heavy throughput/latency graph of Figure 5.10. We focus on the *Ancestor* and *Parent* begin constraints because we expect them to be the most popular. *Ancestor* ensures that clients will see their own writes, along with those of any non-conflicting clients. In the Wikipedia example, this would allow Alice to read back Carlo's operations, but not Davide's or Bruno's. *Parent* results in a behavior very much akin to that of a local Git branch, as clients will see their own operations only. A detailed per-operation performance breakdown (omitted for lack of space) gives us clues for why, despite commit selection being 30% cheaper in *Parent* (as only the read state can satisfy it), *Ancestor* still outperforms *Parent* by 21%. First, read state selection is 40% more expensive in *Parent*, as it requires a look-up over the full DAG rather than one that only involves its leaves. Second, since *Parent* results in more branches, fork path checking becomes more expensive, increasing the cost of reads by 15%. Finally, *Parent* prevents states from being quickly garbage collected.

Unlike begin constraints, we find that end constraints, as long as they do not cause transactions to abort, do not significantly affect throughput or latency: throughput results for the (branching) *Serializability* and *Snapshot Isolation* constraints are within 5% of each other, mostly because the former, by creating twice as many branches, increases the cost of reads by 10%. Non-branching serializability and snapshot isolation both perform poorly in comparison. Though each individual operation is cheap, they see repeated aborts due to write-write or read-write conflicts.

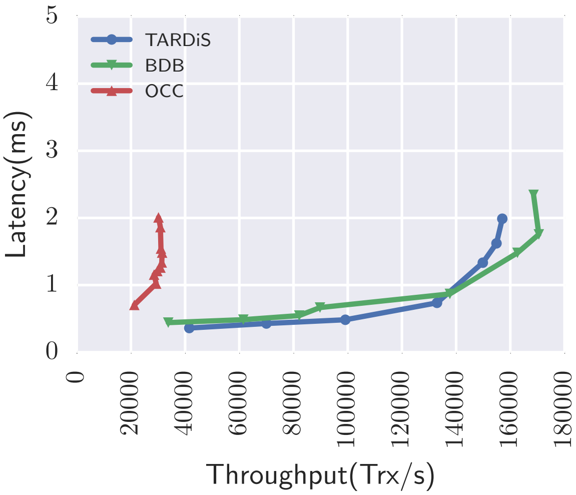


*Figure 5.14: Uniform Blind Writes*

**Garbage Collection** Figures 5.17 and 5.18 report, respectively, the throughput and number of DAG states and records generated by a single server running TARDiS, with and without garbage collection, over a ten minute run with *Ancestor* begin constraint, *Serializability* end constraint, and clients placing ceilings (§5.5.3) every 1000 transactions. Without DAG compression, throughput drops dramatically after three minutes, since old and new generation Java garbage collection pause TARDiS constantly. With DAG compression, throughput instead remains constant, as (*i*) states are removed from the DAG and their underlying datastructures recycled for incoming transactions, and (*ii*) record promotion/deletion keeps the key-version mapping structure small. Compression should ideally bound the DAG size to the product of the number of clients times the ceiling placing interval (i.e., $30 \times 1000 = 30,000$ states) but, as Figure 5.18 shows, the DAG is $55,000$—$98\%$ fewer states than without gargage collection, but still two times higher than the ideal. This mismatch is due to record compression. Though, in principle, states can be removed from the DAG as soon as they become garbage-collectible, in practice the promotion table must first be flushed, which can only happen after a full record promotion pass. Unfortunately, these records are often not present in the cache and must be read from disk, causing the record promotion threads to trail behind. Likewise, the storage size in TARDiS is 15x larger than in OCC or BDB. We are currently investigating alternative schemes that do not have this drawback.

**Replication** We measure the scalability of TARDiS when running in geographically distinct replicas.

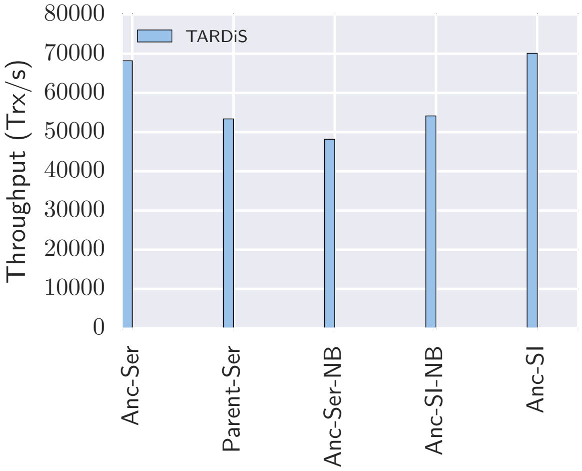


*Figure 5.15: Constraint Choice*

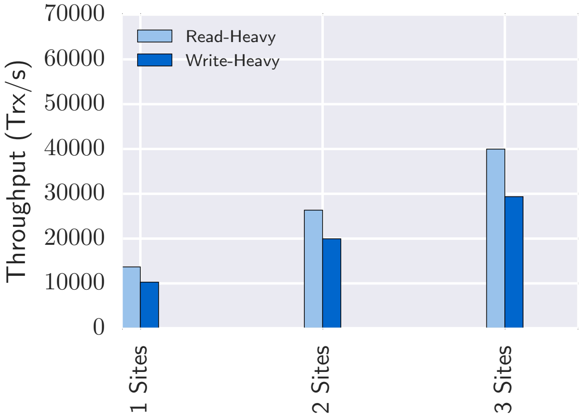


*Figure 5.16: TARDiS Scalability*

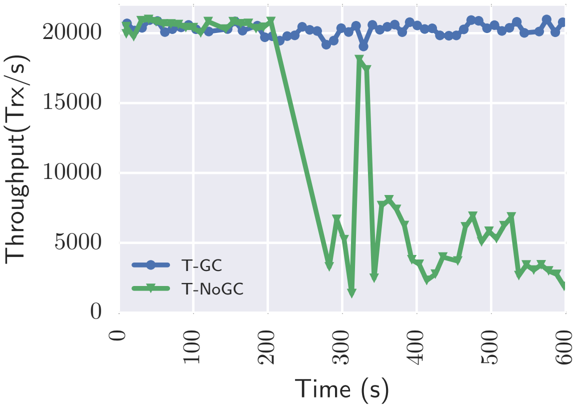


*Figure 5.17: Throughput over time*

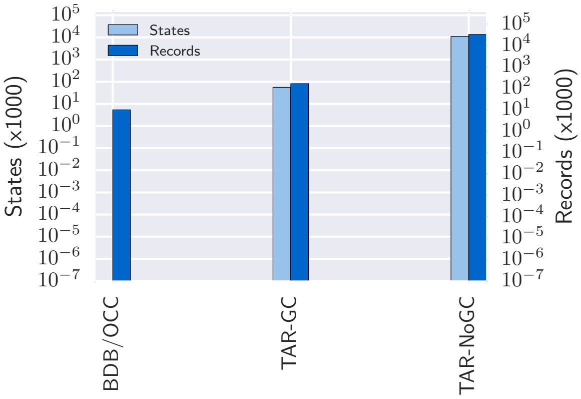


*Figure 5.18: Number of records/states*

We use a Google Cloud Services cluster of three machines (2.5GHz Intel Xeon E5 v2 machines with 60GB of memory), running in three geographical zones (us-central1-f, europe-west1-b, asia-east-1). Figure 5.16 shows how the aggregated throughput scales with the number of sites. As transactions are asynchronously replicated, latency is unchanged from the single site experiments. TARDiS scales linearly with the number of sites, as its design ensures that transactions, when applied to remote sites, do not contend with local transactions.

### 5.6.2 Applications

In porting ALPS applications to TARDiS, we answer two questions: (*i*) Do explicit state branching and merging simplify conflict resolution? (*ii*) Can ALPS applications use local branch-on-conflict to improve their performance?

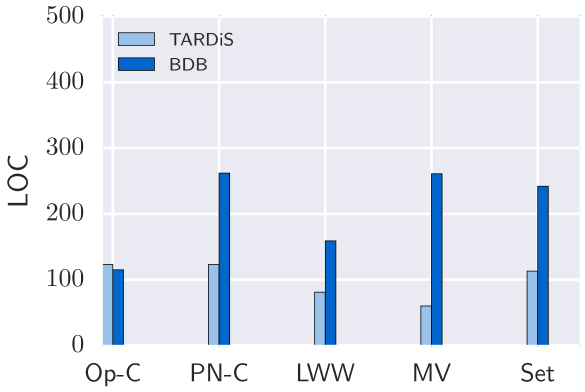


*Figure 5.19: CRDT lines of Code on BerkeleyDB and TARDiS. Op-C:Operation Based Counter, PN-C: State Based Counter, LWW: Last-Writer-Wins Register, MV: Multivalued Register, Set: Or-Set*

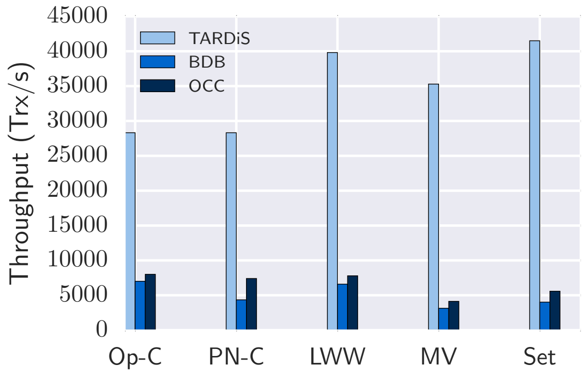


*Figure 5.20: CRDT Throughput*

*Figure 5.21: Retwis Throughput*

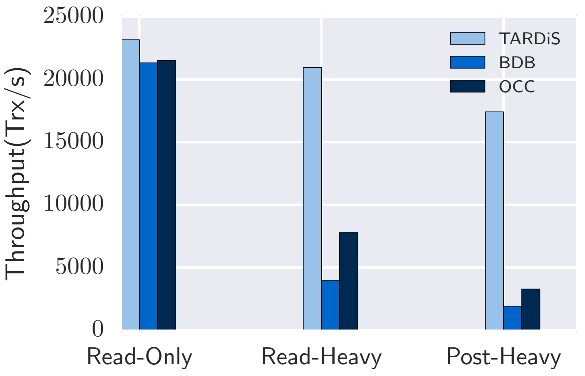
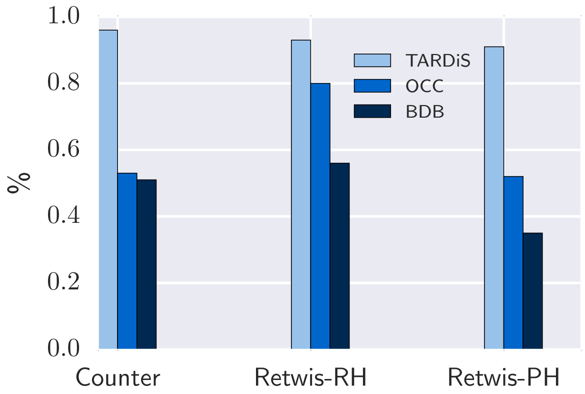


*Figure 5.22: Application Goodput*



**Simplifying Merging: CRDTs** Convergent Replicated Data Types (CRDTs) [171] are a family of data types that support lazy replication by including a set of semantically meaningful conflict resolution functions. A variety of CRDTs have been developed [159, 171], including counters, sets, registers, and trees. To date, they have been incorporated into cooperative text editing tools [159] and in the Riak key-value store [101].

We followed the algorithms developed Shapiro et al. [171] to implement, on top of both TARDiS and BDB, a subset of CRDTs sufficient to design realistic applications. The TARDiS implementation proved easier to write and required less code, often by a factor of two or more (Figure 5.19): the entire effort took less than a day with TARDiS, compared to over three days with BDB. Two features of TARDiS account for this: *StateID* replication and conflict tracking. *StateID* replication guarantees that operations that execute locally on a given state will execute on the same state at all remote sites, and therefore eliminates the need to capture and replicate side-effects. In turn, conflict tracking makes it easy, for each CRDT replica implemented in TARDiS, to access, through the State DAG, a consistent view of the updates that need to be merged. Implementations on flat storage systems like BDB, in contrast, need to explicitly track the updates applied at each replica and make sure that the state of the CRDTs is replicated consistently everywhere. Consider, for example, the counter CRDT, which is modeled as two separate increment and decrement vectors, containing an entry per replica. On BDB, it is up to the CRDT developer to ensure that, as new operations are applied, the global state is tracked correctly at each replica. TARDiS instead tracks the necessary information by design. With access to the fork point, merging becomes as easy as adding, for each branch, the difference between the counter's value at the fork point and at the current state.

To guarantee eventual convergence, CRDTs implemented on sequential storage must mutate local state atomically and sequentially (e.g., new vector clocks must be created atomically to guarantee causal delivery, and updating counters requires read-modify-write operations). Each replica must consequently be serializable, thus limiting the throughput at each site. Branch-on-conflict removes this limitation without sacrificing consistency. For a workload consisting of 90% reads and 10% writes, with transactions configured to use the *Ancestor* and *Serializability* constraint set and periodic merging, TARDiS' CRDT implementations achieve a four to eight times speedup over their sequential counterparts (Figure 5.20). Three factors contribute to this speedup. First, each individual operation is simpler: for instance, reading or updating a counter no longer involves manipulating a vector, but simply requires reading or writing an integer. Second, operations are no longer serialized, but fork on conflict, and are later merged back. Finally, merges can be batched. Traditional CRDTs require a merge for every remote operation received; in TARDiS, merges need only take place periodically, as operations are consistently recorded in separate branches. These effects are displayed in Figure 5.22. It shows, for a CRDT counter implemented in TARDiS, BDB, and OCC, the percentage of *useful work*, measured as the time spent executing committed transactions (i.e., excluding time spent waiting on locks, aborted transactions, and merge transactions): useful work in TARDiS is at 0.96, while for BDB and OCC almost half the time is wasted.

**Pushing weak consistency down: Retwis** To understand the performance implications of building ALPS applications in TARDiS, we implemented Retwis [165], a popular Twitter clone [151, 178, 200]. Retwis users can create accounts (*createAccount*), follow users (*followUser*), post new content (*post*), and read their own timeline (*readOwnTimeline*), which includes their own tweets and those of the users they follow. In our implementation, *readOwnTimeline* returns the 50 most recent posts. Contention primarily arises when a user posts new content, as the Retwis implementation must ensure that the tweet becomes visible to all the user's followers. Retwis, like many ALPS applications, has relatively weak consistency requirements: as long as posts are not incorrectly attributed and are presented in causal order, users can tolerate small delays in seeing posts. Porting Retwis on TARDiS was straightforward. We simply extended each transactional call to take the *Ancestor* begin constraint and *Serializability* end constraint, and implemented a separate conflict resolver that periodically merges conflicting branches by resolving duplicate user ids and merging timelines (preserving the order of posts).

Figure 5.21 shows the throughput of Retwis on TARDiS, BDB, and OCC, for three workloads: read-only (100% reads), read-heavy (85% reads, 5% follows, and 10% posts), and post-heavy (65% reads, 5% follows, and 30% posts). Branching does not benefit the read-only workload, but it significantly softens the performance blow caused by contention in the remaining two workloads. An analysis of the per-transaction behavior (omitted for lack of space) reveals that the throughput of *readOwnTimeline* operations drops by 70% for OCC in the read-heavy workload, as *posts* cause these transactions to abort, and by 80% in BDB, as write operations block both reads and other writes (causing in turn read operations to block for longer). These effects are amplified in the post-heavy workload. Moving to Figure 5.22, we see that TARDiS, by branching and merging asychronously, is able to maintain a much higher fraction of useful work than OCC and BDB for both of these worklods because, unlike waiting on locks, merging does not prevent the system from making progress. The small drop in TARDiS' throughput is due to the need to identify commit states for *posts*. Thus, for ALPS applications, executing conflicting transactions optimistically and reconciling them later can improve scalability within a site, much like weak consistency improves scalability across sites.

## 5.7 Related Work

**Avoiding Conflicts** The core challenge in geo-replication is handling conflicts at different sites. One option is to preemptively ensure that conflicts do not happen, either through strong synchronization [7, 32, 56] or through scheduling transactions to avoid conflicts (using static analysis [205] or upon admission [188]). TARDiS explicitly targets applications whose availability or latency constraints preclude this option.

**Weakening Consistency** Systems designed for ALPS applications trade-off strong consistency for performance and provide weaker guarantees such as causal consistency [9, 62], timeline consistency [54], parallel snapshot isolation [178], and non-monotonic snapshot isolation [19]. Others have sought to give applications the flexibility of adapting the consistency level required as a function of the operation (Pileus [186], Red-Blue Consistency [114], MDCC [105]) or the object (Continuous

Consistency Model [203], CRDTs [171], Escrow Transactions [176]). TARDiS instead allows for general conflict resolution strategies defined by the application.

**Resolving Conflicts** When conflicts are allowed, most systems resolve conflicting executions by projecting them onto sequential storage. COPS [118] adopts a first-writer-wins policy; Ficus [91], Dynamo [62], and Bayou all leave resolution up to the users; and operational transforms [184] leverage specific textual properties. Unlike in TARDiS, these resolution functions are per-object and do not allow programmers to see and resolve the entire state atomically.

**Branching** Conflicts in a distributed system introduce implicit branching that must be reconciled when conflicts are detected. A number of systems expose this branching: version control systems (Git) allow users to operate on different branches; the Olive [8] file-system allows users to create/and fork snapshots efficiently; and ORI [126] tracks possibly divergent histories across multiple devices. They contrast with TARDiS as their branching is explicit (git branch) rather than implicit: explicit branching requires synchronization, which is precisely what ALPS applications want to avoid. Some causally consistent systems also allow for concurrent writes to be exposed to the users (Ficus [91], Dynamo [62]). This is a limited notion of branching, which forks individual objects rather than a state. These systems, as a result, provide neither conflict tracking nor branches, and increase complexity for the application by systematically exposing it to multiple values. By contrast, in TARDiS programmers only deal with multivalued objects if they explicitly request it in merge mode. In the Byzantine context, SUNDR [115] and FAUST [42] develop fork consistency/linearizability to isolate clients that see different values in a faulty server, and Depot [124] extends SUNDR's model to support fork-joining. Depot does not, however, expose the abstraction of branches, and provides no support for cross-object atomic merges, unlike TARDiS. Finally, Sporc [74] does provide atomic joining of forks, but only for the restricted case of collaborative text applications, and uses operational transformation to resolve conflicts.

## 5.8 Limitations

TARDiS's DAG, begin constraints and end constraints are inspired by our state-based approach to isolation and consistency. Begin constraints (, for instance, select candidate read states for a given transaction, while end constraints define a given isolation level's commit test. There is however, currently no equivalent to the notions of single mode and merge mode. Likewise, our model does not explicitly incorporate the notion of branching nor does it provide support for defining merging functions. Instead, it projects all isolation guarantees onto a totally ordered execution. Our consistency model allows for different sessions to observe different executions, but continues to require these executions to be totally ordered. Yet, some of these isolation and consistency guarantees implicitly create (limited) forms of branching. Consider snapshot isolation and parallel snapshot isolation for instance. These guarantees allow, respectively, an anomaly called a short fork and a long fork. Short forks allows concurrent transactions to make concurrent updates to different objects, causing the system to temporarily fork. Once transactions commit, these forks are merged back. As the concurrent writes cannot be conflicting, there is no need for an explicit merging procedure. Long forks extend this notion to non-concurrent transactions. The state can remain forked for an arbitrary time. As there can be no conflicting writes, the merging is once again trivial. Expressing parallel snapshot isolation in our model requires reconstructing these observed forks through read states. We suspect that these definitions could be simplified if the existence of branches became explicit in the model. Extending our formalism to support TARDiS's branch and merging model is thus an interesting avenue for future work that we hope to explore.

## 5.9 Conclusion

This chapter introduced TARDiS, a novel transactional key-value store designed to support weakly-consistent systems. By explicitly tracking concurrent branches and exposing them, when needed, to applications, TARDiS simplifies conflict resolution. By giving applications the option of applying weak consistency principles end-to-end, TARDiS can significantly improve the performance of the local site.

# Chapter 6

# Oblivious transactions through client-centric serializability

The previous chapters have focused on improving semantic and system support for programming atop the weak consistency and isolation guarantees that large scale cloud storage systems provide. By taking a client-centric approach to expressing these guarantees, we were able to simplify how these guarantees were expressed and understood, while also improving system support for handling the write-write conflicts that inherent in weak consistency.

**The context** Applications offload data to cloud storage for convenience: cloud services [13, 14, 134, 135, 155] offer clients scalable, reliable IT solutions and present application developers with feature-rich environments (transactional support, flexible consistency [66, 138], etc.). Medical practices, for instance, increasingly prefer to use cloud-based software to manage electronic health records (EHR) [53, 108]. Cloud storage thus provides performance and scalability with relatively low management overheads.

Offloading data to a third party cloud service, however, raises significant privacy concerns. Applications that could benefit from cloud services store personal data. This data can reveal sensitive information even when encrypted or anonymized [140, 141, 177, 194]. For example, charts accessed by oncologists can reveal not only whether a patient has cancer, but also, depending on the frequency of accesses (e.g., the frequency of chemotherapy appointments), indicate the cancer's type and severity. Similarly, travel websites have been suspected of increasing the price of frequently searched flights [194]. Hiding *access patterns*—that is, hiding not only the content of an object, but also when and how frequently it is accessed, is thus often desirable.

This chapter targets precisely this scenario: it seeks to mitigate the tension that exists between the privacy concerns that doing so creates. To this effect, we introduce the notion of oblivious transactions, and presents the design of Obladi, the first cloud-based key value store that supports transactions, while also hiding access patterns from cloud providers.

**The problem** To mitigate the privacy concerns associated with offloading data to the cloud, the systems community has recently taken a fresh look at private data access. Recent solutions, whether based on private information retrieval [6, 90], Oblivious RAM [45, 120, 167], function sharing [194], or trusted hardware [17, 27, 70, 120, 190], show that it is possible to support complex SQL queries without revealing access patterns.

This chapter addresses a complementary issue: supporting ACID transactions while guaranteeing data access privacy. This combination raises unique challenges as concurrency control mechanisms used to enforce isolation, and techniques used to enforce atomicity and durability make hiding access patterns more problematic (§6.2).

We take as our starting point Oblivious RAM, which provably hides all access patterns. Existing ORAM implementations, however, cannot support transactions. First, they are not fault-tolerant. For security and performance, they often store data in a client-side *stash*; durability requires the stash content to be recoverable after a failure, and preserving privacy demands hiding the stash's size and contents, even during failure recovery. Second, ORAM provides limited or no support for concurrency [38, 167, 179, 199], while transactional systems are expected to sustain highly concurrent loads.

**Our secret sauce** This chapter demonstrates that the demands of supporting transactions can be met at a reasonable cost. Its key insight is that transactions actually afford more flexibility than the single-value operations supported by previous ORAMs. The traditional system-centric definition of serializability  [153] requires that the effects of transactions be reflected consistently in the state of the database *only after they commit*. Consistent with the approach taken in prior chapters, we reformulate this definition in a client-centric way as follows: only transactions that are *observed by clients* as committed need be serializable. Using this notion of client-centric serializability, we design a system, Obladi, which is the first system to support oblivious ACID transactions.

Obladi achieves serializable transactions, guarantees durability with moderate overhead by relying on one key principle: that of *delayed visibility*. Delayed visibility embraces the client-centric approach that this dissertation has been advocating. It recognises that transactions should only be serializable when a transaction is observed by the client as committed, but makes the additional observation that commit notfications can be delayed. Obladi leverages this flexibility to delay committing transactions until the end of fixed-size epochs, buffering their execution at a trusted proxy and enforcing consistency and durability only at epoch boundaries.

**The benefits** The ethos of *delayed visibility* is the core that drives Obladi's design. First, it allows Obladi to implement a multiversioned database atop a single-versioned ORAM, so that read operations proceed without blocking, as with other multiversioned databases [32], and intermediate writes are buffered locally: only the *last* value of any key modified during an epoch is written back to the ORAM. Delaying writes reduces the number of ORAM operations needed to commit a transaction, lowering amortized CPU and bandwidth costs without increasing contention: Obladi's concurrency control ensures that delaying commits does not affect the set of values that transactions executing within the same epoch can observe.

Second, it allows Obladi to securely parallelize Ring ORAM [164], the ORAM construction on which it builds. Obladi pipelines conflicting ORAM operations rather than processing them sequentially, as existing ORAM implementations do. This parallelization, however, is only secure if the write-back phase of the ORAM algorithm is delayed until pre-determined times, namely, epoch boundaries.

Finally, delaying visibility gives Obladi the ability to abort entire epochs in case of failure. Obladi leverages this flexibility, along with the near-deterministic write-back algorithm used by Ring ORAM, to drastically reduce the information that must be logged to guarantee durability and privacy-preserving crash recovery.

The results of a prototype implementation of Obladi are promising. On three applications (TPC-

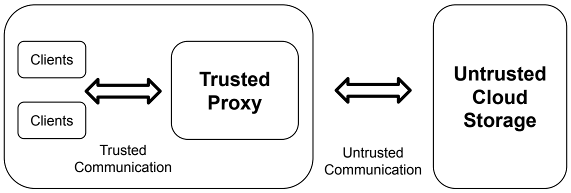


*Figure 6.1: Trusted Proxy Model*

C [189], SmallBank [63], and FreeHealth [116], a real medical application) Obladi is within 5×-12× of the throughput of non-private baselines. Latency is higher (70×), but remains reasonable (in the hundreds of milliseconds).

To summarize, this chapter makes three contributions:

1. It presents the design, implementation, and evaluation of the first ACID transactional system that also hides access patterns.
2. It introduces an epoch-based design that leverages the flexibility of transactional workloads to increase overall system throughput and efficiently recover from failures.
3. It provides the first formal security definition of a transactional, crash-prone, and private database. Obladi uses the UC-security framework [43], ensuring that security guarantees hold under concurrency and composition.

**Roadmap** The rest of this chapter is organised as follows: we first discuss our threat model (§6.1) and summarise the necessary background (§6.3). We then overview the design of Obladi (§6.4) focusing on the proxy (§6.5) and the ORAM (§6.6). We then discuss how to support durability (§6.7) and integrity (§6.9), before proving the security of our system (§6.8) and evaluating it (§6.11). We summarise related work in Section 6.12 and conclude in Section 6.14[1].

## 6.1 Threat and Failure Model

Obladi's threat and failure assumptions aim to model deployments similar to those of medical practices, where doctors and nurses access medical records through an on-site server, but choose to

[1] This chapter revises the previously published paper: *Obladi: Oblivious Serializable Transactions*, previously published at OSDI 2018. Ethan Cecchetti, Sitar Harel and Matthew Burke contributed to the implementation of parallel Ring ORAM, to the evaluation of the paper and to the formal proof of the system.

outsource the integrity and availability of those records to a cloud storage service [53, 108].

**Threat Model**. Obladi adopts a *trusted proxy* threat model [167, 179, 199]: it assumes multiple mutually-trusting client applications interacting with a single trusted proxy in a single shared administrative domain. The applications issue transactions and the proxy manages their execution, sending read and write requests on their behalf over an asynchronous and unreliable network to an *untrusted storage server* (Figure 6.1). This server is controlled by an honest-but-curious adversary that can observe and control the timing of communication to and from the proxy, but not the on-site communication between application clients and the proxy. We extend our threat model to a fully malicious adversary in Section 6.9. We consider attacks that leak information by exploiting timing channel vulnerabilities in modern processors [41, 102, 117] to be out of scope. Obladi guarantees that the adversary will learn no information about: (*i*) the decision (commit/abort) of any ongoing transaction; (*ii*) the number of operations in an ongoing transaction; (*iii*) the type of requests issued to the server; and (*iv*) the actual data they access. We does not seek to hide the type of application that is currently executing (ex: OLTP vs OLAP).

**Failure Model**. Obladi assumes cloud storage is reliable, but, unlike previous ORAMs, explicitly considers that both application clients and the proxy may fail. These failures should invalidate neither Obladi's privacy guarantees nor the Durability and Atomicity of transactions.

## 6.2 Towards Private Transactions

Many distributed, disk-based commercial database systems [28, 56, 147] separate concurrency control logic from storage management: SQL queries and transactional requests are regulated in a concurrency control unit and are subsequently converted to simple read-write accesses to key-value/file system storage. As ORAMs expose a read-write address space to users, a logical first attempt at implementing oblivious transactions would simply replace the database storage with an arbitrary ORAM. This black-box approach, however, raises both security concerns (§6.2.1) and performance/functionality issues (§6.2.2)

Security guarantees can be compromised by simply enforcing the ACID properties. Ensuring Atomicity, Isolation, and Durability imposes additional structure on the order of individual reads and writes, introducing sources of information leakage [17, 173] that do not exist in non-transactional ORAMs (§6.2.1). Performance and functionality, on the other hand, are hampered by the inability of current ORAMs to efficiently support highly concurrent loads and guarantee Durability.

### 6.2.1 Security for Isolation and Durability

The mechanisms used to guarantee Isolation, Atomicity, and Durability introduce timing correlations that directly leak information about the data accessed by ongoing transactions.

**Concurrency Control**. Pessimistic concurrency controls like two-phase locking [71] delay operations that would violate serializability: a write operation from transaction $T_1$ cannot execute concurrently with any operation to the same object from transaction $T_2$. Such blocking can potentially reveal sensitive information about the data, even when executing on top of a construction that hides access patterns: a sudden drop in throughput could reveal the presence of a deadlock, of a write-heavy transaction blocking the progress of read transactions, or of highly contended items accessed by many concurrent transactions. More aggressive concurrency control schemes like timestamp ordering or multiversioned concurrency control [32, 97, 113, 160, 161, 201] allow transactions to observe the result of the writes of other ongoing transactions. These schemes improve performance in contended workloads, but introduce the potential for *cascading aborts*: if a transaction aborts, all transactions that observed its write must also abort. If a write-heavy transaction $T_{heavy}$ aborts, it may cause a large number of transactions to rollback, again revealing information about $T_{heavy}$ and, perhaps more problematically, about the set of objects that $T_{heavy}$ accessed.

**Failure Recovery**. When recovering from failure, Durability requires preserving the effects of committed transactions, while Atomicity demands removing any changes caused by partially-executed transactions. Most commercial systems [136, 147, 148] preserve these properties through variants of *undo* and *redo* logging. To guarantee Durability, write and commit operations are written to a redo log that is replayed after a failure. To guarantee Atomicity, writes performed by partially-executed transactions are *undone* via an *undo log*, restoring objects to their last committed state. Unfortunately, this undo process can leak information: the number of undo operations reveals the existence of ongoing transactions, their length, and the number of operations that they performed.

### 6.2.2 Performance/functionality limitations

Current ORAMs align poorly with the need of modern OLTP workloads, which must support large numbers of concurrent requests; in contrast, most ORAMs admit little to no concurrency [38, 167, 179, 199] (we benchmark the performance of sequential Ring ORAM in Figure 6.13).

More problematically, ORAMs provide no support for fault-tolerance. Adding support for Durability presents two main challenges. First, most ORAMs require the use of a *stash* that temporarily buffers objects at the client and requires that these objects be written out to server storage in very specific ways (as we describe further in §6.3). This process aligns poorly with guaranteeing Durability for transactions. Consider for example a transaction $T_1$ that reads the version of object $x$ written by $T_2$ and then writes object $y$. To recover the database to a consistent state, the update to $x$ should be flushed to cloud storage before the update to $y$. It may however not be possible to *securely* flush $x$ from the stash before $y$. Second, ORAMs store metadata at the client to ensure that cloud storage observes a request pattern that is independent of past and currently executing operations. As we show in §6.7, recovering this metadata after a failure can lead to duplicate accesses that leak information.

### 6.2.3 Introducing Obladi

These challenges motivate the need to co-design the transactional and recovery logic with the underlying ORAM data structure. The design should satisfy three goals: (*i*) security—the system should not leak access patterns; (*ii*) correctness—Obladi should guarantee that transactions are serializable; and (*iii*) performance—Obladi should scale with the number of clients. The principle of *workload independence* underpins Obladi's security: the sequence of requests sent to cloud storage shoud remain independent of the type, number, and access set of the transactions being executed. Intuitively, we want Obladi's sequence of accesses to cloud storage to be statistically indistinguishable from a sequence that can be generated by an Obladi *simulator* with no knowledge of the actual transactions being run by Obladi. If this condition holds, then observing Obladi's accesses cannot reveal to the adversary any information about Obladi's workload. We formalize this intuition in our security definition in §6.8.

Much of Obladi's novelty lies not in developing new concurrency control or recovery mechanisms,

|  |  |
|---|---|
| Stash | Temporary client-side storage<br>for recently accessed or evicted objects |
| Position Map | Current assignment of real objects to paths |
| Bucket | Every bucket contains Z + S slots |
| S | S slots reserved in each bucket<br>for **dummy objects** (dummy slots) |
| Z | Z slots reserved in each bucket for **real objects**<br>(real slots). Max # of real objects that can be stored. |
| z | Current number of real objects in a bucket (z<Z) |
| (In)valid | A slot is marked as invalid after it is accessed |
| A | Evict path is run every A accesses |

*Table 6.1: Ring ORAM Terminology*

but in identifying what standard database techniques can be leveraged to lower the costs of ORAM while retaining security, and what techniques instead subtly break obliviousness.

To preserve workload independence while guaranteeing good performance in the presence of concurrent requests, Obladi centers its design around the notion of *delayed visibility*. Delayed visibility leverages the observation that, on the one hand, ACID consistency and Durability apply only when transactions commit, and, on the other, commit operations can be delayed. Obladi leverages this flexibility to delay commit operations until the end of *fixed-size epochs*. This approach allows Obladi to (*i*) amortize the cost of accessing an ORAM over many concurrently executing requests; (*ii*) recover efficiently from failures; and (*iii*) preserve workload independence: the epochs' deterministic structure allows Obladi to decouple its externally observable behavior from the specifics of the transactions being executed.

## 6.3 Background

**O**blivious **R**emote **A**ccess **M**emory is a cryptographic protocol that allows clients to access data outsourced to an untrusted server without revealing what is being accessed [82]; it generates a sequence of accesses to the server that is completely *independent* of the operations issued by the client. Most ORAM constructions follow a similar pattern: data objects are stored encrypted on untrusted storage in a fixed-size recursive datastructure. The exact position of these objects is obfuscated through the use of encryption and empty "filler" objects. Clients store a small mount of metadata to track objects' position within the datastructure use it to access individual objects without the server learning their exact position., This is usually achieved by *remapping* objects to a new random location on every access, and periodically reshuffling parts of the datastructure. ORAM constructions can be split into two types:

- **Hierarchical ORAMS** are split into $O(log(N))$ fixed-size levels, where each level is exactly twice the size of its children. Each data object is stored in exactly one level of the ORAM; any empty slot is encrypted so as to appear indistinguishable from slots containing real objects. Each level is periodically merged into its parent level and shuffled. This shuffling operation is the primary source of overhead in hierarchical ORAMS: most recent research efforts focus on reducing its cost.

- **Tree-based ORAMS** were recently introduced by Shi et al. [174] (and subsequently refined in [164, 182]). These constructions remove the traditional expensive sort or shuffling phase that made previous ORAM constructions expensive and difficult to implement [174]. The conceptual simplicity of these tree-based schemes makes them susceptible to efficient implementations in real systems: to date, they have been implemented in hardware [75, 122] and as the basis for blockchain ledgers [45] with reasonable overheads. Most tree-based ORAMs follow a similar structure: objects (usually key-value pairs) are mapped to a random leaf (or *path*) in a binary tree and physically reside (encrypted) in some tree node (or *bucket*) along that path. Objects are logically removed from the tree and remapped to a new random path when accessed. These objects are eventually flushed back to storage (according to their new path) as part of an *eviction* phase. Through careful scheduling, this write-back phase does not reveal the new location of the objects; objects that cannot be flushed are kept in a small client-side *stash*.

**Ring ORAM**. Obladi builds upon Ring ORAM [164], a tree-based ORAM with two appealing properties: a constant stash size and a fully deterministic eviction phase. Obladi leverages these features for efficient failure recovery. We summarize Ring ORAM terminology in Table 6.1.

As shown in Figure 6.2, server storage in Ring ORAM consists of a binary tree of *buckets*, each with a fixed number $Z + S$ of *slots*. Of these, $Z$ are reserved for storing actual encrypted data

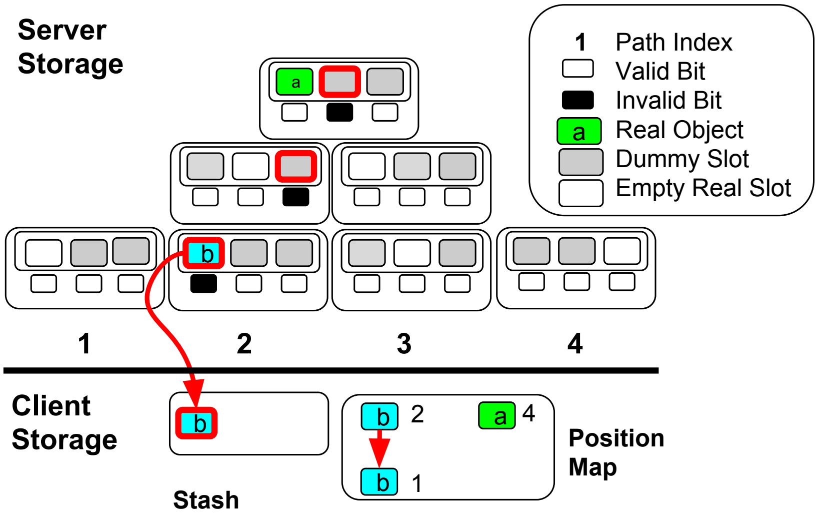


*Figure 6.2: Ring ORAM - Read (Z=1, S=2)*

(*real objects*); the remaining $S$ exclusively store *dummy objects*. Dummy objects are blocks of encrypted but meaningless data that appear indistinguishable from real objects; their presence in each bucket prevent the server from learning how many real objects the bucket contains and which slots contains them. A random permutation (stored at the client) determines the location of dummy slots. In Figure 6.2, the root bucket contains a real slot followed by two dummy slots; the real slot contains the data object $a$; its left child bucket instead contains dummy slots in positions one and three, and an empty real slot in second position.

Client storage, on the other hand, is limited to (*i*) a constant sized *stash*, which temporarily buffers objects that have yet to be replaced into the tree and, unlike a simple cache, is essential to Ring ORAM's security guarantees; (*ii*) the set of current *permutations*, which identify the role of each slot in each bucket and record which slot have already been accessed (and marked *invalid)*; and (*iii*) a *position map*, which records the random leaf (or *path*) associated with every data object. In Ring ORAM, objects are mapped to individual leaves of the tree but can be placed in any one of the buckets along the path from the root to that leaf. For instance, object $a$ in Figure 6.2 is mapped to *path 4* but stored in the root bucket, while object $b$ is mapped to *path 2* and stored in the leaf bucket of this path.

Ring ORAM maintains two core invariants. First, each data object is mapped to a new leaf chosen uniformly at random after every access, and is stored either in the stash, or in a bucket on the path from the tree's root to that leaf (**path invariant**). Second, the physical positions of the $Z + S$ dummy and real objects in each bucket are randomly permuted with respect to all past and future writes to that bucket (i.e., no slot can be accessed more than once between permutations) (**bucket invariant**). The server never learns whether the client accesses a real or a dummy object in the bucket, so the exact position of the object along that path is never revealed.

Intuitively, the path invariant removes any correlation between two accesses to the same object (each access will access independent random paths), while the bucket invariant prevents the server from learning when an object was last accessed (the server cannot distinguish an access to a real slot from a dummy slot). Together, these invariants ensure that, regardless of the data or type of operation, all access patterns will look indistinguishable from a random set of leaves and slots in buckets.

**Access Phase**. The procedures for read and write requests is identical. To access an object $o$, the client first looks up $o$'s path in the position map, and then reads one object from each bucket along that path. It reads $o$ from the bucket in which it resides and a valid dummy object from each other bucket, identified using its local permutation map. Finally, $o$ is remapped to a new path, updated to a new value (if the request was a write), and added to the stash; importantly, $o$ is not immediately written back out to cloud storage.

Figure 6.2 illustrates the steps involved in reading an object $b$, initially mapped to path 2. The client reads a dummy object from the first two buckets in the path (at slots two and three respectively), and reads $b$ from the first slot of the bottom bucket. The three slots accessed by the client are then marked as invalid in their respective buckets, and $b$ is remapped to path 1. To write a new object $c$, the client would have to read three valid dummy objects from a random path, place $c$ in the stash, and remap it to a new path.

**Access Security**. Remapping objects to independent random paths prevents the server from detecting repeated accesses to data, while placing objects in the stash prevents the server from learning the new path. Marking read slots as invalid forces every bucket access to read from a distinct slot (each selected according to the random permutation). The server consequently observes uniformly

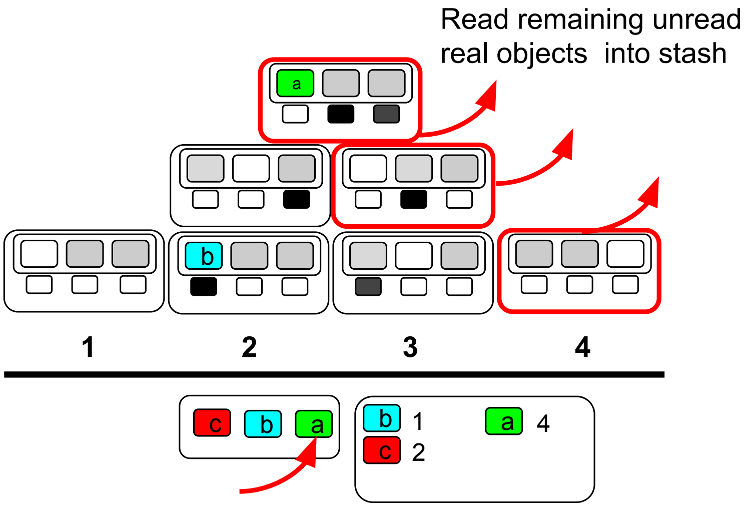


*Figure 6.3: Eviction - Read Phase*

distributed accesses (without repetition) independently of the contents of the bucket. This lack of correlation, combined with the inability to distinguish real slots from dummy slots, ensures that the server does not learn if or when a real object is accessed. Accessing dummy slots from buckets not containing the target object (rather than real slots), on the other hand, is necessary for efficiency: in combination with Ring ORAM's *eviction phase* (discussed next) it lets the stash size remain constant by preventing multiple real objects from being addded to the stash on a single access.

**Eviction Phase and Reshuffling**. The aforementioned protocol falls short in two ways. First, if objects are placed in the stash after each access, the stash will grow unbounded. Second, all slots will eventually be marked as invalid. Ring ORAM sidesteps these issues through two complementary processes: *eviction* and *bucket reshuffling*. Every $A$ accesses, the *evict path* operation evicts objects from the client stash to cloud storage. It deterministically selects a target path, flushes as much data as possible, and permutes each bucket in the path, revalidating any invalid slots. Evict path consists of a read and write phase. In the read phase, it retrieves $Z$ objects from each bucket in the path: all remaining valid real objects, plus enough valid dummies to reach a total of $Z$ objects read. In the write phase, it places each stashed object—including those read by the read phase—to the deepest bucket on the target path that intersects with the object's assigned path. Evict path then permutes the real and dummy values in each bucket along the target path, marking their slots as *valid*, and

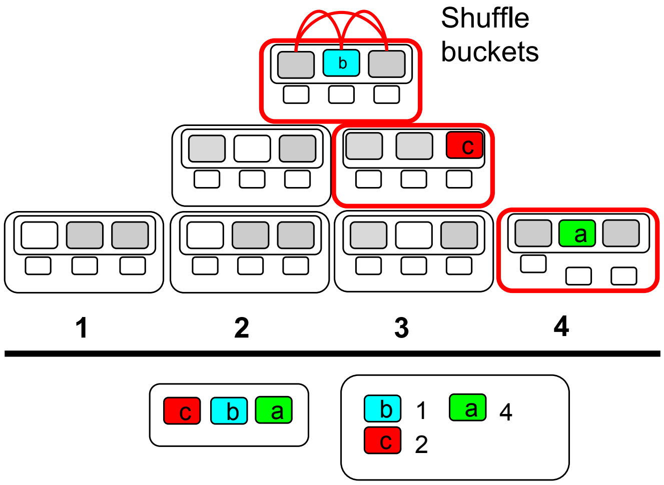


*Figure 6.4: Eviction - Write Phase*

writes their contents to server storage. Figure 6.3 and 6.4 show the evict path procedure applied to path 4. In the read phase, evict path reads the unread object $a$ from the root node and dummies from other buckets on the path. In the write phase (Fig. 6.4), $a$ is flushed to leaf 4, as its path intersects completely with the target path. Finally, we note that randomness may cause a bucket to contain only invalid slots before its path is evicted, rendering it effectively unaccessible. When this happens, Ring ORAM restores access to the bucket by performing an *early reshuffle* operation that executes the read phase and write phase of evict path only for the target bucket.

**Eviction Security**. The read phase leaks no information about the contents of a given bucket. It systematically reads exactly $Z$ valid objects from the bucket, selecting the valid real objects from the $z$ real objects in the bucket, padding the remaining $Z - z$ required reads with a random subset of the $S$ dummy blocks. The random permutation and randomized encryption ensure that the server learns no information about how many real objects exist, and how many have been accessed. Similarly, the write phase hides the values and locations of objects written. At every bucket, the storage server observes only a newly encrypted and permuted set of objects, eliminating any correlation between past and future accesses to that bucket. Together, the read and write phases ensure that no slot is accessed more than once between reshuffles, guaranteeing the bucket invariant.

Similarly, the eviction process leaks no information about the paths of the newly evicted objects: since all paths intersect at the root and the server cannot infer the contents of any individual bucket, any object in the stash may be flushed during *any* evict path. Moreover, since all paths intersect at the root, any object in the stash may be flushed during *any* evict path.

## 6.4 System Architecture

Obladi, like most privacy-preserving systems [167, 180, 199] consists of a centralized trusted component, the *proxy*, that communicates with a fault-tolerant but untrusted entity, *cloud storage* (Figure 6.5). The proxy handles concurrency control, while the untrusted cloud storage stores the private data. Obladi ensures that requests made by the proxy to the cloud storage over the untrusted network do not leak information. We assume that the proxy can crash and that when it does so, its state is lost. This two-tier design allows applications to run a lightweight proxy locally and delegate the complexity of fault-tolerance to cloud storage.

The proxy has two components: (*i*) a *concurrency control unit* and (*ii*) a *data manager* comprised of a *batch manager* and an *ORAM executor*. The batch manager periodically schedules fixed-size batches of client operations that the ORAM executor then executes on a parallel version of Ring ORAM's algorithm. The executor accesses one of two units located on server storage: *the ORAM tree*, which stores the actual data blocks of the ORAM; and *the recovery unit*, which logs all non-deterministic accesses to the ORAM to a write-ahead log [137] to enable secure failure recovery (§6.7).

## 6.5 Proxy Design

The proxy in Obladi has three goals: guarantee good performance, preserve correctness, and guarantee security. To meet these goals, Obladi designs the proxy around the concept of epochs. The proxy partitions time into a set of fixed-length, non-overlapping epochs. Epochs are the granularity at which Obladi guarantees durability and consistency. Each transaction, upon arriving at the proxy, is assigned to an epoch and clients are notified of whether a transaction has committed only when the epoch ends. Until then, Obladi buffers all updates at the proxy.

This flexibility boosts *performance* in two ways. First, it allows Obladi to implement a multiversioned

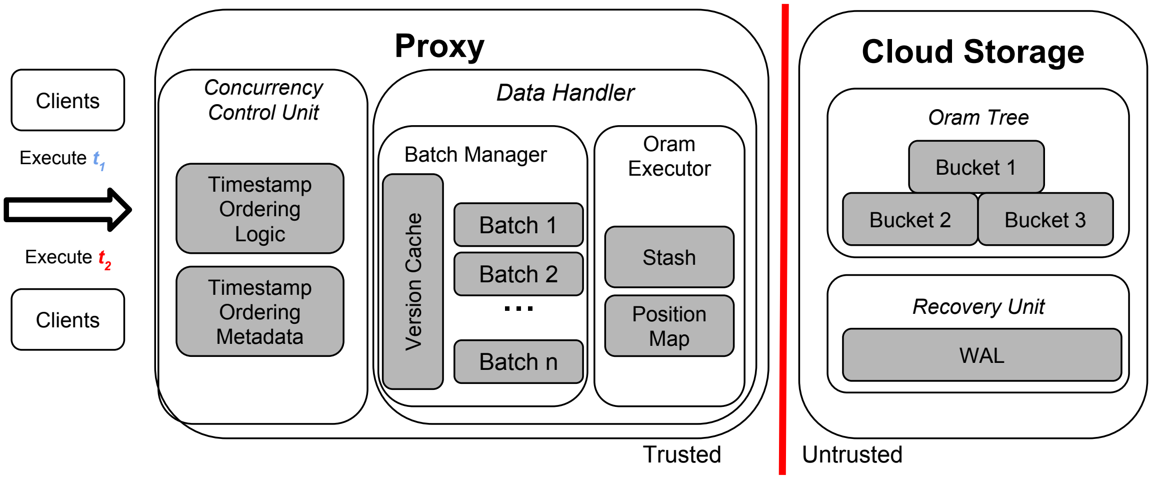


*Figure 6.5: System Architecture*

concurrency control (MVCC) algorithm on top of a single versioned Ring ORAM. MVCC algorithms can significantly improve throughput by allowing read operations to proceed with limited blocking. These performance gains are especially significant in the presence of long-running transactions or high storage access latency, as is often the case for cloud storage systems. Second, it reduces traffic to the ORAM, as only the database state at the end of the epoch needs to be written out to cloud storage.

Importantly, Obladi's choice to enforce consistency and durability only at epoch boundaries does not affect *correctness*; transactions continue to observe a serializable and recoverable schedule (i.e., committed transactions do not see writes from aborted transactions).

For transactions executing concurrently within the same epoch, serializability is guaranteed by concurrency control; transactions from different epochs are naturally serialized by the order in which the proxy executes their epochs. No transaction can span multiple epochs; unfinished transactions at epoch boundaries are aborted, so that no transaction is ongoing during epoch changes.

Durability is instead achieved by enforcing epoch fate-sharing [191] during proxy or client crashes: Obladi guarantees that either all *completed* transactions (i.e., transactions for which a commit request has been received) in the epoch are made durable or all transactions abort. This way, no *committed* transaction can ever observe non-durable writes.

Finally, the deterministic pattern of execution that epochs impose drastistically simplifies the task of guaranteeing workload independence: as we describe further below, the frequency and timing at which requests are sent to untrusted storage are fixed and consequently independent of the workload.

The proxy processes epochs with two modules: the concurrency control unit (CCU) ensures that execution remains serializable, while the data handler (DH) accesses the actual data objects. We describe each in turn.

### 6.5.1 Concurrency Control

Obladi, like many existing commercial databases [146, 158], uses multiversioned concurrency control [32]. Obladi specifically chooses multiversioned timestamp ordering (MVTSO) [32, 162] because it allows uncommitted writes to be immediately visible to concurrently executing transactions. To ensure serializability, transactions log the set of transactions whose uncommitted values they have observed (their write-read dependencies) and abort if any of their dependencies fail to commit. This optimistic approach is critical to Obladi's performance: it allows transactions within the same epoch to see each other's effects even as Obladi delays commits until the epoch ends. In contrast, a pessimistic protocol like two-phase locking [71], which precludes transactions from observing uncommitted writes, would artificially increase contention by holding exclusive write-locks for the duration of an epoch. When a transaction starts, MVTSO assigns it a unique timestamp that determines its serialization order. A write operation creates a new object version marked with its transaction's timestamp and inserts it in the *version chain* associated with that object. A read operation returns the object's latest version with a timestamp smaller than its transaction's timestamp. Read operations further update a *read marker* on the object's version chain with their transaction's timestamp. Any write operation with a smaller timestamp that subsequently tries to write to this object is aborted, ensuring that no read operation ever fails to observe a write from a transaction that should have preceded it in the serialization order.

Consider for example the set of transactions executing in Figure 6.6. Transaction $t_1$'s update to object $a$ ($w(a_1)$) is immediately observed by transaction $t_3$ ($r_3(a_1)$). $t_3$ becomes dependent on $t_1$ and can only commit once $t_1$ also commits. In contrast, $t_2$'s write to object $d$ causes $t_2$ to abort: a transaction with a higher timestamp ($t_3$) had already read version $d_0$, setting the version's read marker to 3.

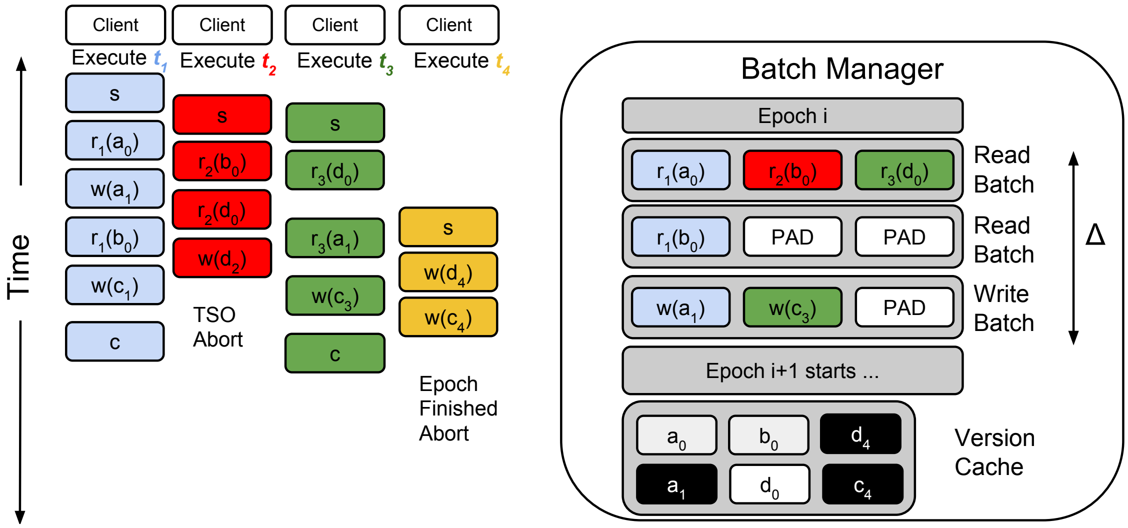


*Figure 6.6: Batching Logic - $r_x(a_y)$ denotes that transaction $t_x$ reads the version of object $a$ written by transaction $t_y$*

### 6.5.2 Data Handler

Once a version is selected for reading or writing, the DH becomes responsible for accessing or modifying the actual object. Whereas it suffices to guarantee durability and consistency only at epoch boundaries, security must hold at all times, posing two key challenges. First, the number of requests executed in parallel can leak information, e.g., data dependencies within the same transaction [37, 167]. Second, transactions may abort (§6.5.1), requiring their effects to be rolled back without revealing the existence of contended objects [17, 173]. To decouple the demands of these workloads from the timing and set of requests that it forwards to cloud storage, Obladi leverages the following observation: transactions can always be re-organized so that all reads from cloud storage execute before all writes [56, 107, 129, 204]. Indeed, while operations within a transaction may depend on the data returned by a read from cloud storage, no operation depends on the execution of a write. Accordingly, Obladi organizes the DH into a read phase and a write phase: it first reads all necessary objects from cloud storage, before applying all writes.

**Read Phase**. Obladi splits each epoch's read phase into a *fixed* set of $R$ *fixed-sized* read batches ($b_{read}$) that are forwarded to the ORAM executor at *fixed* intervals ($\Delta_{epoch}$). This deterministic structure allows Obladi to execute dependent read operations without revealing the internal control flow of the epoch's transactions. Read operations are assigned to the epoch's next unfilled read batch. If no such batch exists, the transaction is aborted. Conversely, before a batch is forwarded to the ORAM executor, all remaining empty slots are padded with dummy requests. Obladi further *deduplicates* read operations that access the same key. As we describe in §6.6, this step is necessary for security since parallelized batches may leak information unless requests all access distinct keys [38, 199]. Deduplicating requests also benefits performance by increasing the number of operations that can be served within a fixed-size batch.

**Write Phase**. While transactions execute, Obladi buffers their write operations into a *version cache* that maintains all object versions created by transactions in the epoch. At the end of an epoch, transactions that have yet to finish executing (recall that epochs terminate at fixed intervals) are aborted and their operations are removed. The latest versions of each object in the version cache according to the version chain are then aggregated in a fixed-size *write batch* ($b_{write}$) that is forwarded to the ORAM executor, with additional padding if necessary.

This entire process, including write buffering and deduplication, must not violate serializability. The DH guarantees that write buffering respects serializability by directly serving reads from the version cache for objects modified in the current epoch. It guarantees serializability in the presence of duplicate requests by only including the last write of the version chain in a write batch. Since Obladi's epoch-based design guarantees that transactions from a later epoch are serialized after all transactions from an earlier epoch, intermediate object versions can be safely discarded. In this context, MVTSO's requirement that transactions observe the *latest* committed write in the serialization order reduces to transactions reading the tail of the previous epoch's version chain.

In the presence of failures, Obladi guarantees serializability and recoverability by enforcing epoch fate sharing: either all transactions in an epoch are made durable or none are. If a failure arises during epoch $e_i$, the system simply recovers to epoch $e_{i-1}$, aborting all transactions in epoch $e_i$. Once again, this flexibility arises from Obladi delaying commit notifications until epoch boundaries.

**Example Execution**. We illustrate the batching logic once again with the help of Figure 6.6. Transactions $t_1$, $t_2$, $t_3$ first execute read operations. These operations are aggregated into the first read batch of epoch $i$. The values returned by these reads are then *cached* into the version cache. $t_2$ then executes a write operation, which Obladi also buffers into the version cache. When executing $r_2(d_0)$), $t_3$ reads object $d$ directly from the version cache (we discuss the security of this step in the next section). Similarly, $r_1(a_1)$ reads the buffered uncommitted version of $a$. In contrast, Obladi schedules $r_1(b_0)$ to execute as part of the next read batch as $b_0$ is not present in the version cache. The read batch is then padded to its fixed $b_{read}$ size and executed. $t_4$ contains no read operations: its write operations are simply executed and buffered at the version cache. Obladi then finalizes the epoch by aborting all transactions (and their dependencies) that have not yet finished executing: $t_4$ is consequently aborted. Finally, Obladi aggregates the last version of every update into the write batch (skipping version $c_1$ of object $c$ for instance, instead only writing $c_2$), before notifying clients of the commit decision.

### 6.5.3 Reducing Work

Obladi reduces work in two additional ways: it caches reads within an epoch and allows Ring ORAM to execute write operations without also executing dummy queries. While these optimizations may appear straightforward, ensuring that they maintain workload independence requires care.

**Caching Reads**. Ring ORAM maintains a client-side stash (§6.3) that stores ORAM blocks until their eviction to cloud storage. Importantly, a request for a block present in the stash still triggers a dummy request: a dummy object is still retrieved from each bucket along its path. While this access may appear redundant at first, it is in fact necessary to preserve *workload independence*: removing it removes the guarantee that the set of paths that Obladi requests from cloud storage is uniformly distributed. In particular, blocks present in the stash are more likely to be mapped to paths farther away from the one visited by the last evict path, as they correspond to paths that could not be flushed: buckets have limited space for real blocks and blocks mapped to paths that only intersect near the top of the tree are less likely to find a free slot to which they can be flushed. The degree to which this effect skews the distribution leaks information about the stash size, and, consequently, about the workload. To illustrate, consider the execution in Figure 6.7. Objects mapped to paths 1 and 2 ($a$, $b$, and $f$) were not flushed from the stash in the previous eviction of path 4. When these objects are subsequently accessed, naively reading them from the stash without performing dummy reads skews the set of paths accessed toward the right subtree (paths 3 and 4)

Obladi securely mitigates some of this work by drawing a novel distinction between objects that are

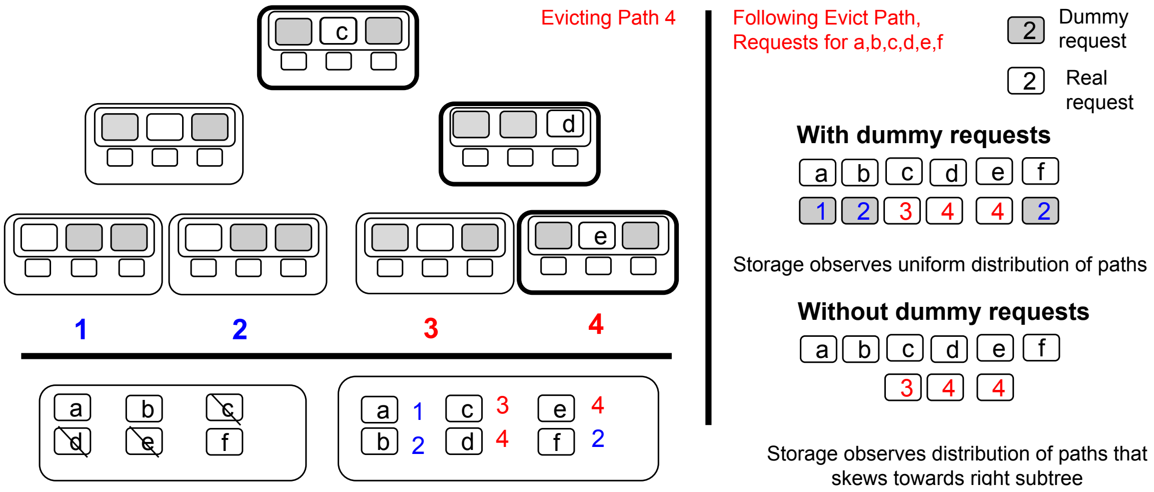


*Figure 6.7: Skew introduced by caching arbitrary objects*

in the stash as a result of a logical access and those present because they could not be evicted. The former can be safely accessed without performing a dummy read, while the latter cannot. Objects present in the stash following a logical access are mapped to independently uniformly distributed paths. Ring ORAM's path invariant ensures that, without caching, the set of accessed paths is uniformly distributed. Removing an independent uniform subset of those paths (namely, the dummy requests) will consequently not change the distribution. Thus, caching these objects, and filling out a read batch with other real or dummy requests, preserves the uniform distribution of paths and leaks no information. Obladi consequently allows all read objects to be placed in the version cache for the duration of the epoch. Objects $a$, $b$, $d$ are, for instance, placed in the version cache in Figure 6.6, allowing read $r_2(d_0)$ to read $d$ directly from the cache. In contrast, objects present in the stash because they could not be evicted are mapped to paths that skew away from the latest evict path. Caching these objects would consequently skew the distribution of requests sent to the storage away from a uniform distribution, as illustrated in Figure 6.7.

**Dummiless Writes**. Ring ORAM must hide whether requests correspond to read or write operations, as the specific pattern in which these operations are interleaved can leak information [206]; that is why Ring ORAM executes a read operation on the ORAM for every access. In contrast, since transactions can always perform all reads before all writes, no information is leaked by informing the storage server that each epoch consists of a fixed-size sequence of potentially dummy reads followed

| | |
|---|---|
| $N$ | Number of real objects |
| $Z$ | Number of real slots |
| $S$ | Number of dummy slots |
| $A$ | Frequency of evict path |
| $L$ | Number of levels in the ORAM tree |
| $R$ | Number of read batches |
| $b_{read}$ | Size of a read batch |
| $b_{write}$ | Size of a write batch |
| $\Delta$ | Batch frequency |

*Table 6.2: Obladi's configuration parameters*

by a fixed-size sequence of potentially dummy writes. Obladi thus modifies Ring ORAM's algorithm to directly place the new version of an object in the stash, without executing the corresponding read. Note, though, that Obladi continues to increment the evict path count on write operations, a necessary step to preserve the bounds on the stash size, which is important for durability (§6.7).

### 6.5.4 Configuring Obladi

Obladi's good performance hinges on appropriately configuring the size/frequency of batches and ORAM tree for a target application. Table 6.2 summarizes the parameter space.

**Ring ORAM**. Configuring Ring ORAM first requires choosing an appropriate $Z$ parameter. Larger values of $Z$ reduce the total size of the ORAM on cloud-storage by decreasing the required height of the ORAM tree and decrease eviction frequency (reducing network/CPU overhead). In contrast, this increase the maximum stash size. Traditional ORAMs thus choose the largest value of Z for which the stash size fits on the proxy. Obladi adds an additional consideration: for durability (as we describe in §6.7), the stash must be synchronously written out every epoch. One must thus take into account the throughput loss associated with the stash writeback time. Given an appropriate value of Z, Obladi then chooses L, S, and A according to the analytical model proposed in [164].

**Epochs and batching**. Identifiying the appropriate size and number of batches hinges on several considerations. First, Obladi must provision sufficiently many read batches ($R$) to handle control flow dependencies within transactions. A transaction that executes in sequence five dependent read operations, will for instance require five read batches to execute (it will otherwise repeatedly abort).

Second, the ratio of reads ($R * b_{read}$) to writes ($w_{write}$) must closely approximate the application's read/write ratio. An overly large write batch will waste resources as it will be padded with many dummy requests. A write batch that is too small will lead to frequent aborts caused by the batch filling up. Third, the size of a read or write batch (respectively $b_{read}$ and $b_{write}$) defines the degree of parallelism that can be extracted. The desired batch size is thus a function of the concurrent load of the system, but also of hardware considerations, as increasing parallelism beyond an I/O or CPU bottleneck serves no purpose. Finally, the number and frequency of read batches within an epoch increases overall latency, but reduces amortized resource costs through caching and operation pipelining (introduced in §6.6). Latency-sensitive applications may favor smaller batch sizes, while others may prefer longer epochs, but lower overheads.

**Security Considerations**. Obladi does not attempt to hide the size and frequency of batches from the storage server (we formalize this leakage in §6.8). Carefully tuning the size and frequency of batches to best match a given application may thus leak information about the application itself. An OLTP application, for instance, will likely have larger batch sizes ($b_{read}$), but fewer read batches ($R$), as OLTP applications sustain a high concurrent load of fairly short transactions. OLAP applications will prefer small or non-existent write batches ($b_{write}$), as they are predominantly read-only, but require many read batches to support the complex joins/aggregates that they implement. Obladi does not attempt to hide the type of application that is being run. It does, however, continue to hide what data is being accessed and what transactions are currently being run at any given point in time. While Obladi's configuration parameters may, for instance, suggest that a medical application like FreeHealth is being run, they do not in any way leak information about how, when, or which patient records are being accessed.

## 6.6 Parallelizing the ORAM

Existing ORAM constructions make limited use of parallelism. Some allow requests to execute concurrently between eviction or shuffle phases [38, 167, 199], while others target intra-request parallelism to speed up execution of a single request [120]. Obladi explicitly targets both forms of parallelism. Parallelizing Ring ORAM presents three challenges: (*i*) preserving the correct abstraction of a sequential datastore, (*ii*) enforcing security by concealing the position of real blocks in the

ORAM (thereby maintaining workload independence), and (*iii*) preserving existing bounds on the stash size. While these issues also arise in prior work [167], the idiosyncrasies of Ring ORAM add new dimensions to these challenges.

**Correctness**. Obladi makes two observations. First, while all operations conflict at the Ring ORAM tree's root, they can be split into suboperations that access mostly disjoint *buckets* (§6.3). Second, conflicting bucket operations can be further parallelized by distinguishing accesses to the bucket's metadata from those to its physical data blocks.

Obladi draws from the theory of multilevel serializability [196], which guarantees that an execution is serializable if the system enforces level-by-level serializability: if operation $o$ is ordered before $o'$ at level $i$, all suboperations of $o$ must precede conflicting suboperations of $o'$. Thus, if Obladi orders conflicting operations at a level $i$, it enforces the same order at level $i + 1$ for all their conflicting suboperations; conversely, if two operations do not conflict at level $i$, Obladi executes their suboperations in parallel. To this end, Obladi simply tracks dependencies across operations and orders conflicting suboperations accordingly. Obladi extracts further parallelism in two ways. First, since in Ring ORAM reads to the same bucket between consecutive eviction or reshuffling operations always target different physical data blocks (even when bucket operations conflict on metadata access), Obladi executes them in parallel. Second, Obladi's own batching logic ensures that requests within a batch touch different objects, preventing read and write methods from ever conflicting. Together, these techniques allow Obladi to execute most requests and evictions in parallel.

We illustrate the dependency tracking logic in Figure 6.8. The read operation to path 1 conflicts with the evict path for path 2, but only at the root (bucket 1). Thus, reads to buckets 2 and 3 can proceed concurrently, even though accesses to the root's metadata must be serialized, as both operations update the bucket access counter and valid/invalid map (§6.3).

**Security**. For security, Obladi's parallel evict path operation must flush the same blocks flushed by a sequential implementation. Reproducing this behavior without sacrificing parallelism is challenging. It requires that all real objects brought in during the last $A$ accesses be present in the stash when data is flushed, which may introduce *data dependencies*. Unlike dependencies that arise between operations that access the same physical location in cloud storage, these dependencies are not a

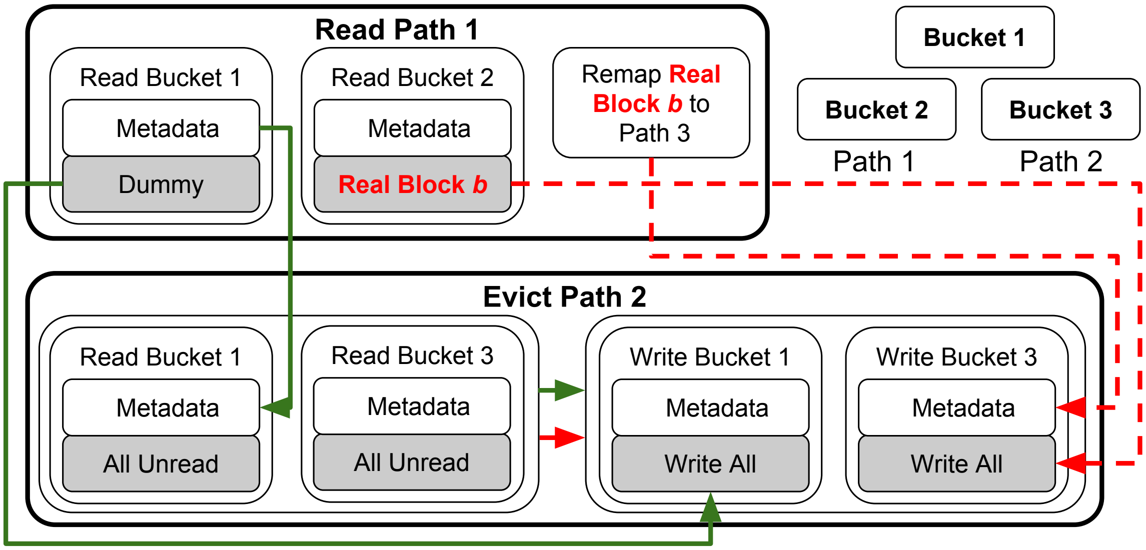


*Figure 6.8: Multilevel Pipelining for a read of path 1 and an evict path of path 2 executing in parallel. Solid green lines represent physical dependencies and dashed red lines represent data dependencies. Inner blocks represent nested operations*

deterministic function of an epoch's operations already known to the adversary.

Consider, for instance, block $b$ in Figure 6.8. In a sequential implementation, $b$ would enter the stash as a result of reading path 1 and be flushed to bucket 3 by the following evict path. Thus, evict path would have to *wait* until $b$ is placed in the stash. Honoring these dependencies opens a timing channel: delay in flushing certain blocks can reveal object placement. As blocks holding real objects can exist anywhere in the tree and be remapped to any path, it follows that it is never secure to execute an eviction operation until all previous access operations have terminated.

Obladi mitigates this restriction by again leveraging delayed visibility and the idea to separate read and write operations within an epoch—but with an important difference. In §6.5.2 the proxy created separate batches for *logical* read and write operations; to improve parallelism, Obladi, expanding on an idea used by Shroud [120], assigns to separate phases within an epoch the *physical* read and write operations that underlie each of those logical operations. The read phase computes all necessary metadata and executes the set of physical read operations for all logical read path, early reshuffle, and evict path operations. This set is workload independent, so its operations need not be delayed. Physical writes, however, are only flushed at the end of an epoch. The proxy can again apply write deduplication: if a bucket is repeatedly modified during an epoch, only the last version must be written back. Reads that should have read an intermediate write are served locally from the buffered buckets.

The adversary thus always observes a set of reads to random paths followed by a deterministic set of writes independent of the contents of the ORAM and, consequently, of the workload. Data dependencies between read and evict operations no longer create a timing channel. Meanwhile parallelism remains high, as the physical blocks accessed in each phase are guaranteed to be distinct—Ring ORAM directly guarantees this for reads, while bucket deduplication does it for writes.

## 6.7 Durability

Obladi guarantees durability at the granularity of epochs: after a crash, it recovers to the state of the last failure-free epoch. Obladi adds two demands to the need of recovering to a consistent state: recovery should leak no information about past or future transactions, and it should be efficient, accessing minimal data from cloud storage. Obladi guarantees the former by ensuring that recovery logic and data logged for recovery maintain workload independence (§6.1). It strives towards the latter by leveraging the determinism of Ring ORAM.

**Consistency**. Obladi recovery logic relies on two well-known techniques: write-ahead logging [137] and shadow paging [87]. Obladi mandates that transactions be durable only at the end of an epoch; thus, on a proxy failure, all ongoing transactions can be aborted, and the system reverted to the previous epoch. To make this possible, Obladi must (*i*) recover the proxy metadata lost during the proxy crash, and (*ii*) ensure that the ORAM does not contain any of the aborted transactions' updates. To recover the metadata, Obladi logs three data structures before declaring the epoch committed: the position map, the permutation map, and the stash. The position map and the permutation map identify the position of real objects in the ORAM tree (respectively, in a path and in a bucket); logging them prevents the recovery logic from having to scan the full ORAM to recover the position of buckets. Logging the stash is necessary for correctness. As eviction may be unable to flush the entire stash, some newly written buckets may be present only in the stash, even at epoch boundaries. Failing to log the stash could thus lead to data loss.

To undo partially executed transactions, Obladi adapts the traditional copy-on-write technique of shadow paging [87]: rather than updating buckets in place, it creates new versions of each bucket on every write. Obladi then leverages the inherent determinism of Ring ORAM to reconstruct a consistent snapshot of the ORAM at a given epoch. In Ring ORAM, the current version of a bucket (i.e. the number of times a bucket has been written) is a deterministic function of the number of prior evict paths. The number of evict paths per epoch is similarly fixed (evict paths happen every $A$ accesses, and epochs are of fixed size). Obladi can then trivially revert the ORAM on failures by setting the evict path counter to its value at the end of the last committed epoch. This counter determines the number of evict paths that have occurred, and consequently the object versions of the corresponding epoch.

**Security**. Obladi ensures that (*i*) the information logged for durability remains independent of data accesses, and (*ii*) that the interactions between the failed epoch, the recovery logic, and the next epoch preserve workload independence.

Obladi addresses the first issue by encrypting the position map and the contents of the permutations table. It similarly encrypts the stash, but also *pads* it to its maximum size, as determined in canonical Ring ORAM [164], to prevent it from indicating skew (if a small number of objects are accessed frequently, the stash will tend to be smaller).

The second concern requires more care: workload independence must hold before, during, and after failures. Ring ORAM guarantees workload independence through two invariants: the bucket invariant and the path invariant (§6.3). Preserving bucket slots from being read twice between evictions is straightforward. Obladi simply logs the invalid/valid map to track which slots have already been read and recovers it during recovery; there is no need for encryption, as the set of slots read is public information. Ensuring that the ORAM continues to observe a uniformly distributed set of paths is instead more challenging. Specifically, read requests from partially executed transactions can potentially leak information, even when recovering to the previous epoch. Traditionally, databases simply *undo* partially executed transactions, mark them as aborted, and proceed as if they had never existed. From a security standpoint, however, these transactions were still observed by the adversary, and thus may leak information. Consider a transaction accessing object $a$ (mapped to path 1) that aborts because of a proxy failure. Upon recovery, it is likely that a client will attempt to access $a$ again. As the recovery logic restores the position map of the previous epoch, that new operation on $a$ will result in another access to path $1$, revealing that the initial access to path $1$ was likely real (rather than padded), as the probability of collisions between two uniformly chosen paths is low. To mitigate this concern while allowing clients to request the same objects after failure, Obladi durably logs the list of paths and slot indices that it accesses, before executing the actual requests, and replays those paths during recovery (remapping any real blocks). While this process is similar to traditional database redo logging [137], the goal is different. Obladi does not try to reapply transactions (they have all aborted), but instead forces the recovery logic to be deterministic: the adversary always sees the paths from the aborted epoch repeated after a failure.

**Optimizations**. To minimize the overhead of checkpointing, Obladi checkpoints deltas of the position, permutation, and valid/invalid map, and only periodically checkpoints the full data structures. While the number of changes to the permutation and valid/invalid maps directly follows from the set of physical requests made to cloud storage, the size of the delta for the position map reveals how many real requests were included in an epoch—padded requests do not lead to position map updates. Obladi thus pads the map delta to the maximum number of entries that could have changed in an epoch (i.e., the read batch size times the number of read batches, plus the size of the single write batch).

## 6.8 System Security

We now outline Obladi's security guarantees, deferring a formal treatment to Appendix H. To the best of our knowledge, we are the first to formalize the notion of crashes in the context of oblivious RAM.

**Model** We express our security proof within the Universal Composability (UC) framework [43], as it aligns well with the needs of modern distributed systems: a UC-secure system remains UC-secure under concurrency or if composed with other UC-secure systems. Intuitively, proving security in the UC model proceeds as follows. First, we specify an *ideal functionality* $\mathcal{F}$ that defines the expected functionality of the protocol for both correctness and security. For instance, Obladi requires that

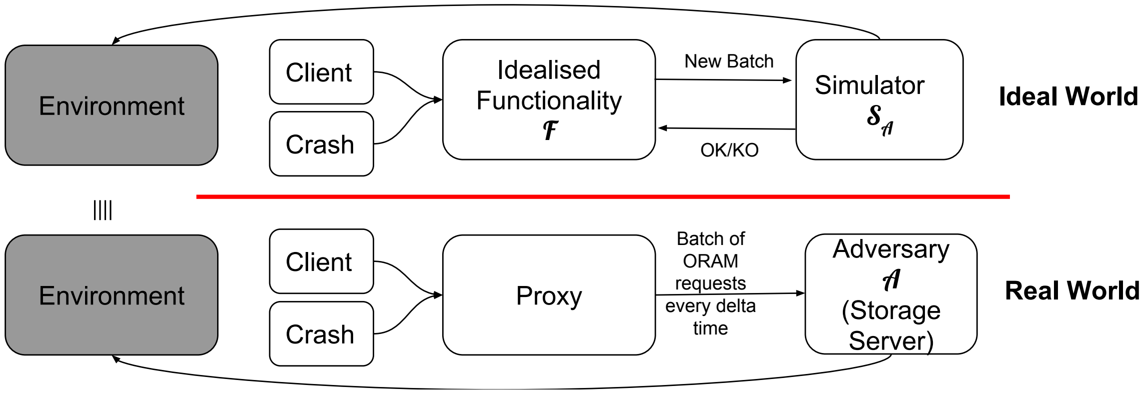


*Figure 6.9: UC Framework*

the execution be serializable, and that only the frequency of read and write batches be learned. We must ensure that the real protocol provides the same functionality to honest parties while leaking no more information than $\mathcal{F}$ would. To establish this, we consider two different worlds: one where the real protocol interacts with an adversary $\mathcal{A}$, and one where $\mathcal{F}$ interacts with $\mathcal{S}_{\mathcal{A}}$, our best attempt at simulating $\mathcal{A}$. $\mathcal{A}$'s transcript—including its inputs, outputs, and randomness—and $\mathcal{S}_{\mathcal{A}}$'s output are given to an environment $\mathcal{E}$, which can also observe all communications within each world. $\mathcal{E}$'s goal is to determine which world contains the real protocol. To prompt the worlds to diverge, $\mathcal{E}$ can delay and reorder messages, and even control external inputs (potentially causing failures). Intuitively, $\mathcal{E}$ represents anything external to the protocol, such as concurrently executing systems. We say that the real protocol is secure if, for any adversary $\mathcal{A}$, we can construct $\mathcal{S}_{\mathcal{A}}$ such that $\mathcal{E}$ can never distinguish between the worlds. We represent the situation pictoraly in Figure 6.9.

**Assumptions** The security of Obladi relies on three assumptions. (*i*) Canonical Ring ORAM is linearizable (*ii*) MVTSO generates serializable executions. (*iii*) The network will retransmit dropped packets. The adversary learns of the retransmissions, but nothing more.

**Ideal Functionality** To define the ideal functionality $\mathcal{F}_{Ob}$, recall that the proxy is considered trusted while interactions with the cloud storage are not. This allows $\mathcal{F}_{Ob}$ to replace the proxy and intermediate between clients and the storage server, performing the same functions as the proxy (we do not try to hide the concurrency/batching logic). We must, however, define $\mathcal{F}_{Ob}$ to obliviously hide data values and access patterns. To this end, when the proxy logic finalizes a batch, $\mathcal{F}_{Ob}$ simply informs the storage server that it is executing a read or write batch. Since $\mathcal{F}_{Ob}$ is a theoretical ideal, we allow it to manage all storage internally, so it then updates its local storage and furnishes the appropriate response to each client.

In this setup, modeling proxy crashes is straightforward. Crashes can occur at any time and cause the proxy to lose all state. So, on an external input to crash, $\mathcal{F}_{Ob}$ simply clears its state. Since we accept that $\mathcal{A}$ may learn of proxy crashes, $\mathcal{F}_{Ob}$ also sends a message to the storage server that it has crashed.

**Proof Sketch** The correctness of the system is straightforward, as $\mathcal{F}_{Ob}$ behaves much the same as the proxy.

To prove security, we must demonstrate that, for any algorithm $\mathcal{A}$ defining the behavior of the storage server, we can accurately simulate $\mathcal{A}$'s behavior using only the information provided by $\mathcal{F}_{Ob}$. Note that the simulator $\mathcal{S}_{\mathcal{A}}$ can run $\mathcal{A}$ internally, as $\mathcal{A}$ is simply an algorithm. Thus we can define $\mathcal{S}_{\mathcal{A}}$ to operate as follows. When $\mathcal{S}_{\mathcal{A}}$ receives notification of a batch, it constructs a parallel ORAM batch from uniformly random accesses of the correct type. It provides these accesses to $\mathcal{A}$ and produces $\mathcal{A}$'s response.

The security of this simulation hinges on two key properties: (*i*) the caching and deduplication logic do not affect the distribution of physical accesses, and (*ii*) the physical access pattern of a parallelized batch is entirely determined by the physical accesses proscribed by sequential Ring ORAM for the same batch. The first follows from Ring ORAM's guarantee that each access will be an independent uniformly random path—removing an independently-sampled element does not change the distribution of the remaining set. The second follows from the parallelization procedure simply aggregating all accesses and performing all reads followed by all writes.

These properties ensure that the random access pattern produced by $\mathcal{S}_{\mathcal{A}}$ is identical to the access pattern produced by the proxy when operating on real data. Thus the simulated $\mathcal{A}$ must behave exactly as it would when provided with real data, and produce indistinguishable output.

## 6.9 Ensuring Data Integrity in Obladi

As we described in §6.1, we assume the untrusted storage server is *honest-but-curious*. In many cases this is a very strong assumption that system operators may not be happy to make. We can remove this requirement and add integrity to Obladi with the use of Message Authentication Codes (MACs) and a trusted counter—used to ensure freshness—that persists across crashes. We describe this technique here.

When we assumed the server was honest-but-curious, we assumed it could deny service, but would otherwise correctly respond to all queries. In order to remove this assumption while maintaining security, we must detect whether the storage server returns incorrect data. This reduces such attacks to DoS attacks. To detect misbehaviour, the proxy must verify that the returned value is the value (*i*) most recently written (*ii*) by the proxy (*iii*) to the specified location.

We can guarantee (*ii*) using MACs. At initialization, the proxy generates a secret MAC key (in addition to its secret encryption key) and attaches a MAC to every piece of data it stores on the cloud server. This allows the proxy to verify that the cloud server did not modify the data or manufacture its own.

The use of MACs alone is insufficient to guarantee (*i*) and (*iii*), as the cloud server can provide an old copy of the data or valid data from a different location, both of which will have valid MACs. We additionally need to include a unique identifier that the proxy can easily recompute. For data that is written at most once per epoch, this unique identifier can be the pair of epoch, ORAM location. Due to Ring ORAM's deterministic eviction algorithm, the proxy can compute the epoch during which any given block was most recently written knowing only the current epoch counter and the early reshuffle table.

There is exactly one value which is written multiple times per epoch: each read batch, of which there may be many per epoch, logs the accessed locations. This means the counter associated with those writes must uniquely identify the read batch, not just the epoch. In fact, since every epoch has the same number of read batches, a read batch counter is sufficient for all values.

**Handling Crashes** The above modifications are sufficient to guarantee integrity if the proxy never crashes. When the proxy crashes, however, it needs information from the cloud storage to recover. To guarantee integrity—in particular freshness—of the recovery data, the epoch/read batch counter we describe above must persist in a trustworthy fashion across failures. Perhaps the easiest way to implement this requirement is to store the counter on a small amount of nonvolatile storage locally on the proxy, but any trustworthy and persistent storage mechanism is sufficient.

This, of course, raises the question of when to update this trustworthy persistent counter. Once the update occurs, a recovering proxy will expect the cloud storage to provide data associated with that counter value. This means that the counter must be updated after writing to cloud storage. Because a recovering proxy will be unaware of the newly-written data until the counter is updated, we do not consider the write complete until the counter is properly updated. As usual, if the proxy crashes while a write is in-progress, the write is simply rolled back.

As long as the storage server cannot learn anything from incomplete writes, this new strategy is entirely secure. Because the timing of Obladi's writes is completely deterministic and their locations are determined entirely by the locations of prior reads, the fact that a write has aborted does not inherently leak any information. The contents of the write, however, can still leak information. Most data in the system is already encrypted, but one value is not: the read logs written during read batches. The proxy can crash after sending data to the cloud server but before updating its trusted counter. The storage server could withhold that data on recovery without detection and learn whether the proxy accessed the same locations after recovery is complete. To fix this leak, we encrypt the read batch logs in the cloud and update the counter after writing the log but before reading any values. In this way, the cloud storage gains no information about what data will be read until after the write is complete, at which point the proxy will always replay the read if a crash occurs. This removes the leakage.

As we will see in Appendix H, these modifications are sufficient to guarantee both confidentiality and integrity (though naturally not availability) even against an arbitrarily malicious cloud storage server.

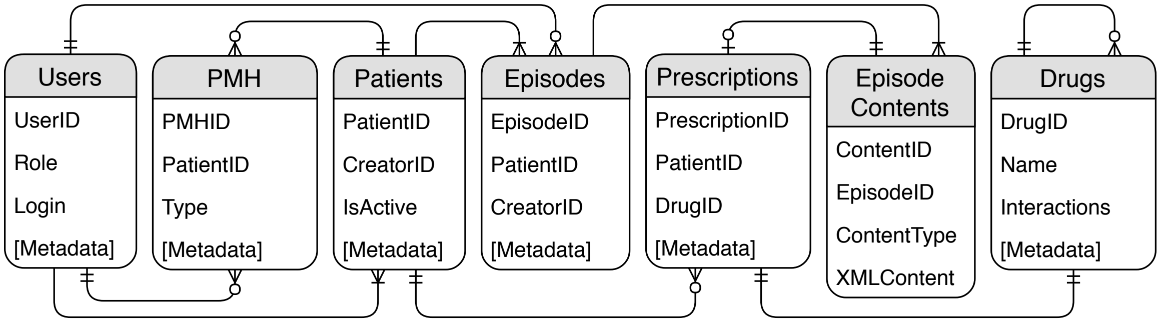


*Figure 6.10: FreeHealth Database Architecture*

## 6.10 Implementation

Our prototype consists of 41,000 lines of Java code. We use the Netty library for network communication (v4.1.20), Google protobuffers for serialization (v3.5.1), the Bouncy Castle library (v1.59) for encryption, and the Java MapDB library (v3) for persistence. We additionally implement a non-private baseline (NoPriv). NoPriv shares the same concurrency control logic (TSO), but replaces the proxy data handler with non-private remote storage. NoPriv neither batches nor delays operations; it buffers writes at the local proxy until commit, and serves writes locally when possible.

## 6.11 Evaluation

Obladi leverages the flexibility of transactional commits to mitigate the overheads of ORAM. To quantify the benefits and limitations of this approach, we ask:

1. How much does Obladi pay for privacy? (§6.11.1)
2. How do epochs affect these overheads? (§6.11.2)
3. Can Obladi recover efficiently from failures? (§6.11.3)

**Experimental Setup** The proxy runs on a c5.xlarge Amazon EC2 instance (16 vCPUs, 32GB RAM), and the storage on an m5.4xlarge instance (16 vCPUs, 64GB RAM). The ORAM tree is configured with $Z = 100$ and optimal values of $S$ and $A$ (respectively, 196 and 168) [164]. We report the average of three 90 seconds runs (30 seconds ramp-up/down).

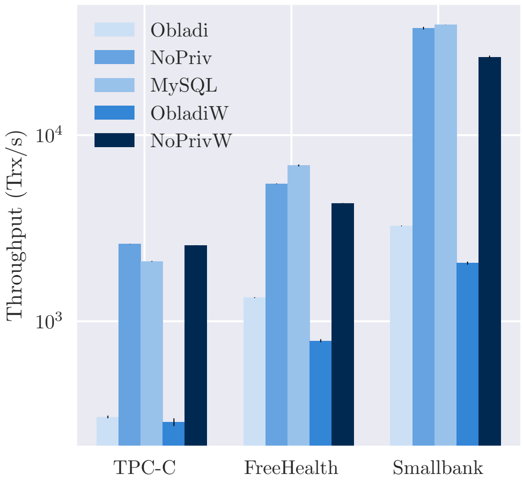


*Figure 6.11: Application Throughput*

**Benchmarks** We evaluate the performance of our system using three applications: TPC-C [63, 189], SmallBank [63], and FreeHealth [77, 116]. Our microbenchmarks use the YCSB [55] workload generator. **TPC-C**, the defacto standard for OLTP workloads, simulates the business logic of e-commerce suppliers. We configure TPC-C to run with 10 warehouses [201]. In line with prior transactional key-value stores [183], we use a separate table as a secondary index on the `order` table to locate a customer's latest order in the `order status` transaction, and on the `customer` table to look up customers by their last names (`order status` and `payment`). **Smallbank** [63] models a simple banking application supporting money transfers, withdrawals, and deposits. We configure it to run with one million accounts. Finally, we port **FreeHealth** [77, 116], an actively-used cloud EHR system (Figure 6.10). FreeHealth supports the business logic of medical practices and hospitals. It consists of 21 transaction types that doctors use to create patients and look up medical history, prescriptions, and drug interactions.

### 6.11.1 End-to-end Performance

Figures 6.14 and 6.12 summarize the results from running the three end-to-end applications in two setups: a local setup in which the latency between proxy and server is low (0.3ms) (**Obladi, NoPriv**), and a more realistic WAN setup with 10ms latency (**ObladiW**, **NoPrivW**). We additionally compare those results with a local MySQL setup. MySQL, unlike NoPriv, cannot buffer writes. We consequently do not evaluate MySQL in the WAN setting.

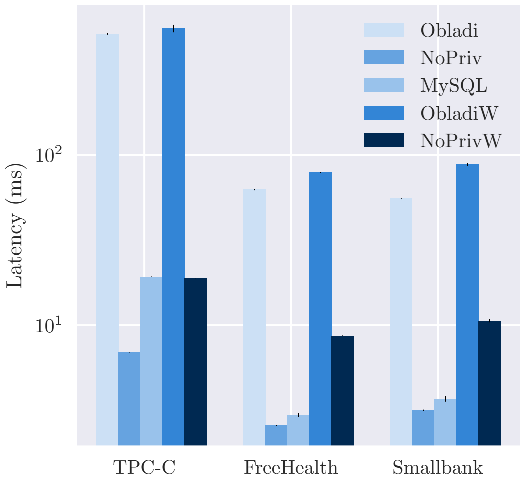


*Figure 6.12: Application Latency*

**TPC-C** Obladi comes within 8× of NoPriv's throughput, as NoPriv is contention-bottlenecked on the high rate of conflicts between the `new-order` and `payment` transactions on the `district` table. NoPriv's performance is itself slightly higher than MySQL as the use of MVTSO allows for the `new-order` and `payment` transactions to be pipelined. In contrast, MySQL acquires exclusive locks for the duration of the transactions. Latency, however, spikes to 70× over NoPriv because of the inflexible execution pattern Obladi needs for security. Transactions in TPC-C vary heavily in size. Epochs must be large enough to accommodate all transactions, and hence artificially increase the latency of short instances. Moreover, write operations must be applied atomically during epoch changes. For a write batch size of 2,000, this process takes on average 340ms, further increasing latency for individual transactions. The write-back process also limits throughput, even preventing non-conflicting operations from making progress (in contrast, NoPriv can benefit from writes never blocking reads in MVTSO). Epoch changes also introduce additional aborts for transactions that straddle epochs. The additional 10ms latency of the WAN setting has comparatively little effect, as the large write batch size of TPC-C is the primary bottleneck: throughput remains within 9x of NoPrivW. Also NoPrivW's performance does not degrade: since MVTSO exposes uncommitted writes immediately, increasing commit latency does not increase contention.

**Smallbank** Transactions in Smallbank are more homogeneous (between three and six operations); thus, the length of an epoch can be set to more closely approximate most transactions, reducing latency overheads (17× NoPriv). NoPriv is CPU bottlenecked for Smallbank; the relative throughput drop for Obladi is higher (12×) because of the overhead of changing epochs and the blocking that it introduces. Transaction dependency tracking becomes a bottleneck in NoPriv, resulting in a 15% throughput loss over MySQL. Increasing latency between proxy and storage causes both systems' throughput to drop. ObladiW's 35% drop is due to the increased duration of epoch changes (during which no other transactions can execute) while NoPrivW's 30% drop stems from the larger dependency chains that arise from the relatively long commit phase.

**FreeHealth** Like SmallBank, FreeHealth consists of fairly short transactions and can thus choose a fairly small epoch (five read batches), reducing the impact on latency (20× NoPriv). Unlike Smallbank, however, FreeHealth consists primarily of read operations, and so it can choose a much smaller write batch (200), minimizing the cost of epoch changes and maximizing throughput (only a 4× drop over NoPriv and a 5.5× over NoPrivW for ObladiW). Both NoPriv and Obladi are contention-bottlenecked on the creation of *episodes*, the core units of EHR systems that encapsulate prescriptions, medical history, and patient interaction.

### 6.11.2 Impact of Epochs

Though epochs create blocking and cause aborts, they are key to reducing the cost of accessing ORAM, as they allow to (*i*) securely parallelize the ORAM and (*ii*) delay and buffer bucket writes. To quantify epochs' impact on performance as a function of their size and the underlying storage properties, we instantiate an ORAM with 100K objects and choose three different storage backends: a local dummy (storing no real data) that responds to all reads with a static value and ignores writes (`dummy`); a remote server backend with an in-memory hashmap (`server`, ping time 0.3ms) and a remote WAN server backend with an in-memory hashmap (`server WAN`, ping time 10ms); and DynamoDB (`dynamo`, provisioned for 80K req/s, read ping 1ms, write 3ms).

**Parallelization** We first focus on the performance impact of parallelizing Ring ORAM (ignoring other optimizations). Graph 6.13 shows that, unsurprisingly, the benefits of parallelism increase with the latency of individual requests. Parallelizing the ORAM for `dummy`, for instance, yields no performance gain; in fact, it results in a 3× slowdown (from 72K req/s to 24K req/s). Sequential Ring ORAM on `dummy` is CPU-bound on metadata computation (remapping paths, shuffling buckets,

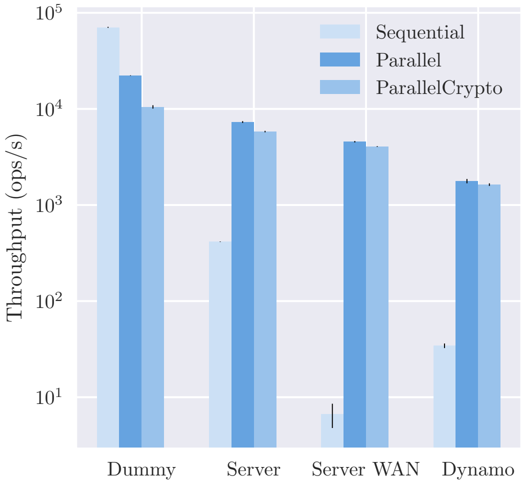


*Figure 6.13: Parallelism (Batch Size 500)*

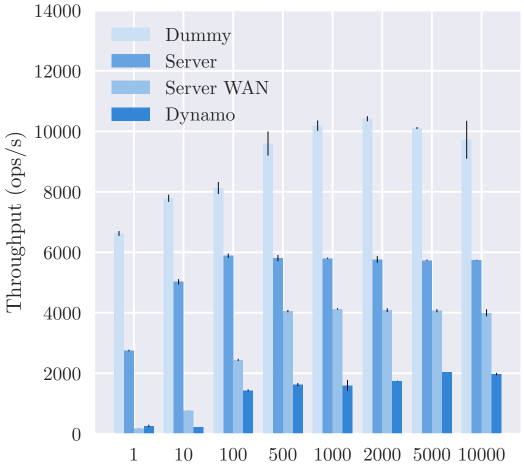


*Figure 6.14: Batch Size Throughput*

etc.), so adding coordination mechanisms to guarantee multi-level serializability only increases the cost of accessing a bucket. As storage access latency increases and the ORAM becomes I/O-bound, the benefits of parallelism become more salient. For a batch size of 500, throughput increases by 12× for `server`, as much as 51× for `dynamo`, and 510× for `WAN server`. The available parallelism is a function of both the size/fan-out of the tree and the underlying resource bottlenecks of the proxy. Graph 6.14 captures the parallelization speedup for both intra- and inter-request parallelism, while Graph 6.14 quantifies the latency impact of batching. The parallelization speedup achieved for a batch size of one captures intra-request parallelism: the eleven levels of the ORAM can be accessed

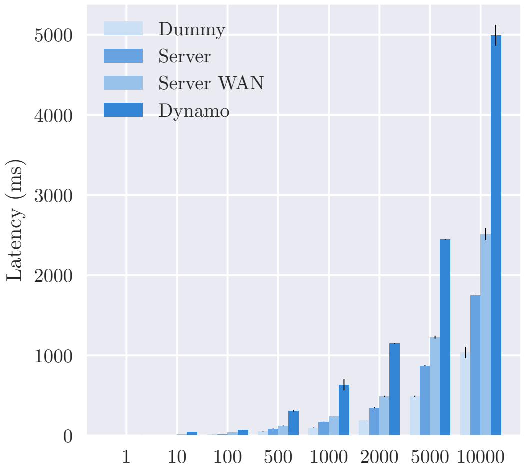


*Figure 6.15: Batch Size Latency*

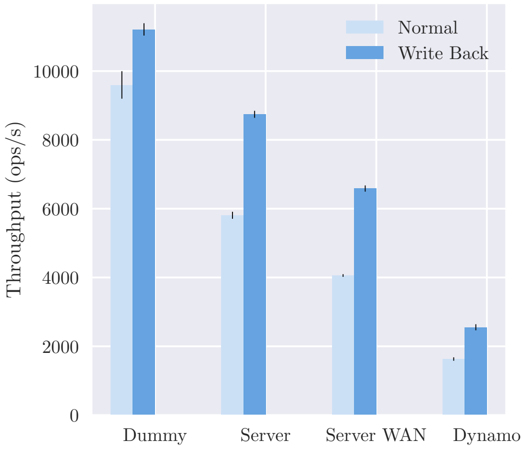


*Figure 6.16: Delayed Visibility*

concurrently, yielding an 11× speedup. As batch sizes increase, Obladi can leverage inter-request parallelism to process non-conflicting physical operations in parallel, with little to no impact on latency. `Dynamo` peaks early (at 1750 req/s) because its client API uses blocking HTTP calls, and `dummy`'s storage eventually bottlenecks on encryption, but `server` and `WAN server` are more interesting. Their throughput is limited by the physical and data dependencies on the upper levels of the tree (recall that paths always conflict at the root (§6.6)).

**Work Reduction** To amortize ORAM overheads across a large number of operations, Obladi relies on delayed visibility to buffer bucket writes until the end of an epoch, when they can be executed in parallel, discarding intermediate writes. Reads to those buckets are directly served from the proxy,

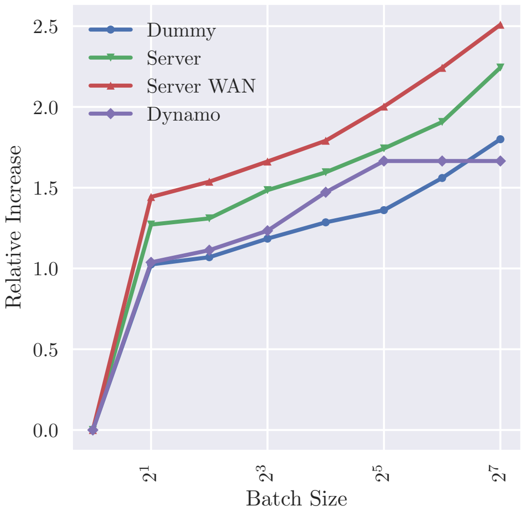


*Figure 6.17: Epoch Size Impact - ORAM*

reducing network communication and CPU work (as encryption is not needed). Graph 6.16 shows that enabling this optimization for an epoch of eight batches (a setup suitable for FreeHealth and TPC-C) yields a 1.5× speedup on both `dynamo` and the server, a 1.6× speedup on the WAN server, but only minimal gains for `dummy` (1.1×). When using a small number of batches, throughput gains come primarily from combining duplicate operations in buckets near the top of the tree. For example, the root bucket is written 27 times in an epoch of size eight (once per eviction, every 168 requests). As these operations conflict, they must be executed sequentially and quickly become the bottleneck (other buckets have fewer operations to execute). Our optimization lets Obladi write the root bucket only once, significantly reducing latency and thus increasing throughput. As epochs grow in size, increasingly many buckets are buffered locally until the end of the epoch (§6.6), allowing reads to be served locally and further reducing I/O with the storage. Consider Graph 6.17: throughput increases almost logarithmically; metadata computation eventually becomes a bottleneck for `dummy`, while `server` and `server WAN` eventually run out of memory from storing most of the tree (our AWS account did not allow us to provision `dynamo` adequately for larger batches). Larger epochs reduce the raw amount of work per operation: with one batch, Obladi requires 41 physical requests per logical operation, but only requires 24 operations with eight batches. For real transactional workloads, however, epochs are not a silver bullet. Graph 6.18 suggests that applications are very sensitive to identifying the right epoch duration: too short and transactions cannot make progress, repeatedly

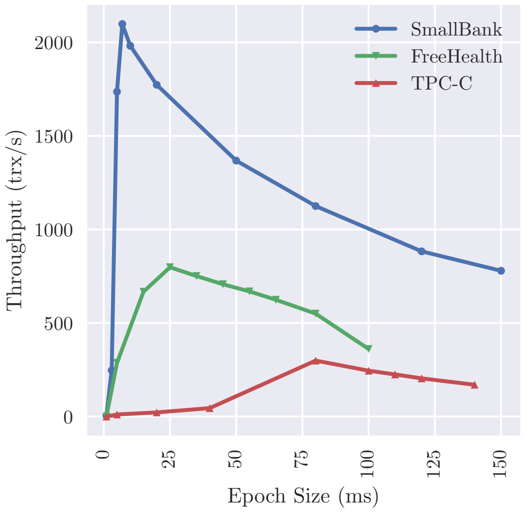


*Figure 6.18: Epoch Size Impact - Proxy*

aborting; too long and the system will remain unnecessarily idle.

### 6.11.3 Durability

Table 6.20 quantifies the efficiency of failure recovery and the cost it imposes on normal execution for ORAMS of different sizes (we show space results for only the WAN server as Dynamo follows a similar trend). During normal execution, durability imposes a moderate throughput drop (from 0.83× for 10K to 0.89× for 1M). This slowdown is due to the need to checkpoint client metadata and to synchronously log read paths to durable storage before reading. As seen in Graph 6.19, computing diffs mitigates the impact of checkpointing. Recovery time similarly increases as the ORAM grows, from 1.5s to 6.1s (Table 6.20, *RecTime*). The costs of decrypting the position and permutation maps (*Pos* and *Perm*) are low for small datasets, but grow linearly with the number of keys. Read path logging (*Paths*) instead starts much larger, but grows only with the depth of the tree.

## 6.12 Related Work

**Batching** Obladi amortizes ORAM costs by grouping operations into epochs and committing at epoch boundaries. Batching can mitigate expensive security primitives, e.g., it reduces server-side computation in private information retrieval (PIR) schemes [30, 90, 95, 121], amortizes the cost of shuffling networks in Atom [110] and the cost of verifying integrity in Concerto [18].

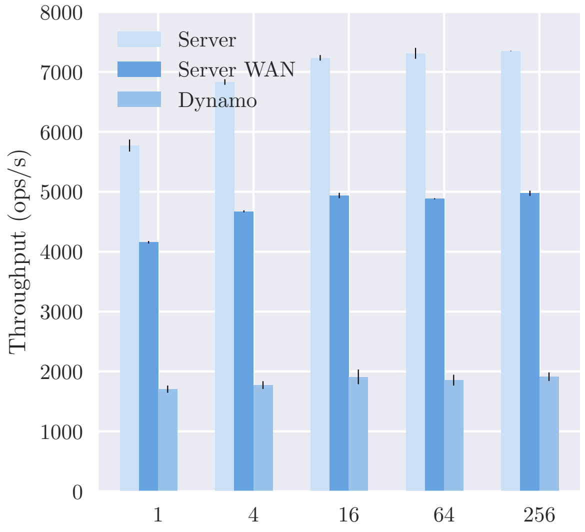


*Figure 6.19: Checkpoint Frequency (100K)*

| | 10K | 100K | 1M |
|---|---|---|---|
| Levels | 7 | 11 | 14 |
| Slowdown | 0.83 | 0.88 | 0.89 |
| RecTime | 1452 | 2604 | 6080 |
| Network | 182 | 681 | 848 |
| Pos | 8 | 74 | 1610 |
| Perm | 15 | 218 | 1424 |
| Paths | 864 | 1104 | 1341 |

*Figure 6.20: Server Wan Recovery Time (ms)*

Changing when operations output commit is a popular performance-boosting technique: it yields significant gains for state-machine replication [98, 104, 157], file systems [142], and transactional databases [58, 129, 191].

**ORAM parallelism** Obladi extends recent work on parallel ORAM constructions [37, 120, 199] to extract parallelism both *within* and *across* requests. Shroud [120] targets intra-request parallelism by concurrently accessing different levels of tree-based ORAMs. Chung et al [38] and PrivateFS [199] instead target inter-request parallelism, respectively in tree-based [175] and hierarchical [198] ORAMs. Both works execute requests to distinct logical keys concurrently between reshuffles or evictions and deduplicate concurrent requests for the same key to increase parallelism. Obladi leverages delayed visibility to separate batches into read and write phases, extracting concurrency both within requests and across evictions. Furthermore, Obladi parallelizes across requests by deduplicating requests at the trusted proxy.

ObliviStore [181] and Taostore [167] instead approach parallelization by focusing on asynchrony. ObliviStore [181] formalizes the security challenges of scheduling requests asynchronously; the oblivious scheduling mechanism that it presents for that model however is computationally expensive and requires a large stash, making ObliviStore unsuitable for implementing ACID transactions. Like ObliviStore, Taostore leverages asynchrony to parallelize Path ORAM [182], a tree-based construction from which Ring ORAM descends. Taostore, however, targets a different threat model: it assumes both that requests must be processed immediately, and that the timing of responses is visible to the adversary. Request latencies thus necessarily increase linearly with the number of clients [199].

**Hiding access patterns for non-transactional systems** Many systems seek to provide access pattern protections for analytical queries: Opaque [206] and Cipherbase [17] support oblivious operators for queries that scan or shuffle full tables. Both rely on hardware enclaves for efficiency: Opaque runs a query optimizer in SGX [94], while Cipherbase leverages secure co-processors to evaluate predicates more efficiently. Others seek to hide the parameters of the query rather than the query itself: Olumofin et al. [145] do it via multiple rounds of keyword-based PIR operations [51]; Splinter [194] reduces the number of round-trips necessary by mapping these database queries to function secret sharing primitives. Finally, ObliDB [70] adds support for point queries and efficient updates by designing an oblivious B-tree for indexing. The concurrency control and recovery mechanisms of all these approaches introduce timing channels and structure writes in ways that leak access patterns [17].

**Encryption** Many commercial systems offer the possibility to store encrypted data [67, 169]. Efficiently executing data-dependent queries like joins, filters, or aggregations without knowledge of the plaintext is challenging: systems like CryptDB [156], Monomi [190], and Seabed [152] tailor encryption schemes to allow executing certain queries directly on encrypted data. Others leverage trusted hardware [27]. In contrast, executing transactions on encrypted data is straightforward: neither concurrency control nor recovery requires knowledge of the plaintext data.

## 6.13 Limitations

Obladi also has several limitations. First, like most ORAMs that regulate the interactions of multiple clients, it relies on a local centralized proxy, which introduces issues of fault-tolerance and scalability. Second, Obladi does not currently support range or complex SQL queries. Addressing the consistency challenge of maintaining oblivious indices [17, 70, 206] in the presence of transactions is a promising avenue for future work.

## 6.14 Conclusion

This chapter presents Obladi, a system that, for the first time, considers the security challenges of providing ACID transactions without revealing access patterns. Obladi guarantees security and durability at moderate cost through two simple observations: transactional guarantees are only required to hold for transactions that clients observe as committed, and commit notifications can be delayed. By delaying commits until the end of epochs, Obladi inches closer to providing practical oblivious ACID transactions.

# Chapter 7

# Conclusion

This dissertation argued that accessing cloud storage as a black-box requires revisiting how correctness guarantees are expressed and proposes taking a *client-centric approach to system development*. Such an approach has both semantic and practical benefits. From a semantic standpoint, we showed that it makes it easier to understand, compare and relate *database isolation guarantees* [58]. Second, we demonstrated that it can simplify the handling of *write-write conflicts* in causal consistency [59]. The performance benefits are also salient: taking a client-centric approach to enforcing serializability allowed us to design transactional cloud storage systems that provably hide accesses to user data [57].

## 7.1 Other Work

This section briefly summarises work to which I made significant contributions to but will not be included in the PhD dissertation.

**Occult** Occult [129] In existing causal systems, such as COPS [118] or Eiger [119], a datacenter performs a write operation only after applying all writes that causally precede it. This approach guarantees that reads never block, as all replicas are always in a causally consistent state, but, in the presence of slow or failed shards, may cause writes to be buffered for arbitrarily long periods of time, delaying the visibility of updates across all shards. These *slowdown cascades* thus violate a basic commandment for scalability: do not let your performance be determined by the slowest component in your system. To resolve this issue, we propose to revisit what implementing causal consistency actually requires: for clients's reads to reflect a causal snapshot of the system. The underlying system itself need never store a causally consistent state, as long as it appears indistinguishable to clients from a system that does! We suggest shifting the burden of enforcing causal consistency from the datastore to those actually perceiving consistency anomalies, the clients. This approach- allows Occult to make its updates available as soon as it receives them, without having to first apply all causally preceding writes. Causal consistency is then enforced by clients on reads, but only for those updates that they are actually interested in observing, removing all prior artificial delays. By leveraging intelligent timestamp compression techniques to implement this client-centric causal consistency, Occult becomes impervious to slowdown cascades at a moderate cost of 10% throughput reduction.

**Popcorn [90]** Popcorn implements a Netflix-like delivery system that provably hides which movies users access. Popcorn does this with moderate cost, while respecting the current legal framework for media dissemination. Popcorn takes as its starting point Private Information Retrieval (PIR) and makes three observations. First, Popcorn combines the two types of PIR: large media objects are retrieved using the lighter weight ITPIR [52] for performance, while the smaller decryption keys are stored at a centralised Netflix-like server and retrieved using the heavier-weight CPIR [109]. Second, Popcorn amortises the high linear cost of PIR by batching requests from large number of concurrent clients accessing the service. Popcorn further leverages the specificities of media streaming to create large batches without introducing playback delays. Third, Popcorn side-steps the fixed size object requirement of PIR with minimal overhead by observing that the size of media objects can be modulated by changing their bitrate. We find that, while Popcorn has high overheads (1080x CPU and 14x I/O bandwidth), it can scale to Netflix-sized libraries and successfully stream movies at a reasonable dollar-cost.

**Tebaldi [183]** Tebaldi, a transactional key-value store, harnesses the performance opportunities offered by federating optimised concurrency controls. Tebaldi partitions conflicts between transactions *hierarchically* and matches them to specialised concurrency controls (CCs). Tebaldi's secret sauce lies in reasoning about concurrency control mechanisms in terms of the *ordering decisions* that they make. Ordering decisions provide a common language to (*i*) correctly compose concurrency controls' *guarantees* and (*ii*) efficiently compose their *implementation*. To achieve this, Tebaldi's framework for CC coordination observes that, despite their diversity, the steps taken by most CC protocols to order a transaction can be decomposed into four phases. In each phase, Tebaldi allows concurrency controls to constrain, delegate, or follow the ordering decisions of the other concurrency controls. These well-defined and narrow communication channels allow for the implementation of each concurrency control to remain independent of other CCs in the system and for new CCs to be added when necessary. Tebaldi achieves over 3.7× the throughput of other federated systems.

**Musketeer [81]** Musketeer, a framework for large-scale distributed processing, decouples how data processing workflows are defined from how they are executed. To do so, it identifies a common intermediate abstraction that captures the semantics of most data processing computations: a direct acyclic graph (DAG) of data-flow operators, based on the relational algebra. Musketeer then takes jobs written in existing workflow specification languages (Hive, Lindi, GraphLinq, etc.), maps them to this intermediate abstraction, and compiles them dynamically to the best data-processing system(s). Our prototype speeds up realistic workflows by up to 9x by targeting different execution engines, without requiring any manual effort. The overheads of federation remain moderate: Musketeer's automatically generated back-end code comes within 5%–30% of the performance of hand-optimized implementations.

## 7.2 Acknowledgements

I would like to acknowledge the work of my co-authors in this dissertation: Youer Pu was essential in proving equivalences to existing isolation guarantees in Chapter 3 and to proving the causal consistency result in Chapter 4. Matthew Burke implemented the parallel version of RingORAM in Chapter 5 and Ethan Cecchetti helped prove the security of the system. I would also like to acknowledge Sitar Harel, Rachit Agarwal, Lorenzo Alvisi, Allen Clement, Nancy Estrada and Trinabh Gupta for their role in various parts of the papers from which this thesis derives.

I am grateful for the support I received from Google through the Google Doctoral Fellowship, from the University of Texas at Austin through the Harrington Fellowship, from Microsoft Research through the Microsoft Diversity Fellowship, and to the National Science Foundation through grants

1762015 (CSR: Small: Client-Centric Consistency) and 1758043 (CSR: Medium: Salt: combining ACID and BASE in a distributed database).

# Appendix

# A Equivalence to Adya et al.

In this section, we prove the following theorems:

**Theorem 1** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{SER}(T, e) \equiv \neg\mathrm{G1} \wedge \neg\mathrm{G2}$ (§A.2).

**Theorem 2** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{SI}(T, e) \equiv \neg\mathrm{G1} \wedge \neg\text{G-SI}$ (§A.3).

**Theorem 3** $\exists e : \forall t \in \mathcal{T} : \mathrm{CT}_{RC}(T, e) \equiv \neg\mathrm{G1}$ (§A.4).

**Theorem 4** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{RU}(T, e) \equiv \neg\mathrm{G0}$ (§A.5).

## A.1 Adya et al. model [4] summary

Adya et al. [4] introduces a cycle-based framework for specifying weak isolation levels. We summarize its main definitions and theorems here.

To capture a given system run, Adya uses the notion of *history*.

**Definition 3** *A history H over a set of transactions consists of two parts: i) a partial order of events E that reflects the operations (e.g., read, write, abort, commit) of those transactions, and ii) a version order, $<<$, that is a total order on committed object versions.*

We note that the version-order associated with a history is implementation specific. As stated in Bernstein et al [32]: as long as there exists a version order such that the corresponding direct serialization graph satisfies a given isolation level, the history satisfies that isolation level. The model introduces several types of direct read/write conflicts, used to specify the *direct serialization graph*.

**Definition 4** *Direct conflicts:*

**Directly write-depends** $T_i$ *writes a version of x, and* $T_j$ *writes the next version of x, denoted as* $T_i \xrightarrow{ww} T_j$

**Directly read-depends** $T_i$ *writes a version of x, and* $T_j$ *reads the version of x* $T_i$ *writes, denoted as* $T_i \xrightarrow{wr} T_j$

**Directly anti-depends** $T_i$ *reads a version of x, and* $T_j$ *writes the next version of x, denoted as* $T_i \xrightarrow{rw} T_j$

**Definition 5** *Time-Precedes Order. The time-precedes order, $\prec_t$ , is a partial order specified for history H such that:*

1. *$b_i \prec_t c_i$, i.e., the start point of a transaction precedes its commit point.*
2. *for all i and j, if the scheduler chooses $T_j$'s start point after $T_i$ 's commit point, we have $c_i \prec_t b_j$; otherwise, we have $b_j \prec_t c_i$.*

**Definition 6** *Direct Serialization Graph. We define the direct serialization graph arising from a history H, denoted DSG(H), as follows. Each node in DSG(H) corresponds to a committed transaction in H and directed edges correspond to different types of direct conflicts. There is a read/write/anti-dependency edge from transaction $T_i$ to transaction $T_j$ if $T_j$ directly read/write/antidepends on $T_i$.*

The model is augmented with a logical notion of time, used to define the *start-ordered serialization graph*.

**Definition 7** *Start-Depends. $T_j$ start-depends on $T_i$ if $c_i \prec_t b_j$ i.e., if it starts after $T_i$ commits. We write $T_i \xrightarrow{sd} T_j$*

**Definition 8** *Start-ordered Serialization Graph or SSG. For a history H, SSG(H) contains the same nodes and edges as DSG(H) along with start-dependency edges.*

The model introduces several *phenomema*, of which isolation levels proscribe a subset.

**Definition 9** *Phenomena:*

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

**G1a: Dirty Reads** *A history H exhibits phenomenon G1a if it contains an aborted transaction $T_i$ and a committed transaction $T_j$ such that $T_j$ has read an object (maybe via a predicate) modified by $T_i$.*

**G1b: Intermediate Reads** *A history H exhibits phenomenon G1b if it contains a committed transaction $T_j$ that has read a version of object x written by transaction $T_i$ that was not $T_i$'s final modification of x.*

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

**G2: Anti-dependency Cycles** *A history H exhibits phenomenon G2 if DSG(H) contains a directed cycle having one or more anti-dependency edges.*

**G-Single: Single Anti-dependency Cycles** DSG(H) contains a directed cycle with exactly one anti-dependency edge.

**G-SIa: Interference** A history H exhibits phenomenon G-SIa if SSG(H) contains a read/write-dependency edge from $T_i$ to $T_j$ without there also being a start-dependency edge from $T_i$ to $T_j$.

**G-SIb: Missed Effects** A history H exhibits phenomenon G-SIb if SSG(H) contains a directed cycle with exactly one anti-dependency edge.

**Definition 10** *Each isolation level is defined as proscribing one or more of these phenomena*

**Serializability (PL-3)** $\neg$*G1* $\wedge$ $\neg$*G2*

**Read Committed (PL-2)** $\neg$*G1*

**Read Uncommitted (PL-1)** $\neg$*G0*

**Snapshot Isolation** $\neg$*G1* $\wedge$ $\neg$*G-SI*

## A.2 Serializability

**Theorem 1**. $\exists e : \forall T \in \mathcal{T}.\text{CT}_{SER}(T, e) \equiv \neg\text{G1} \wedge \neg\text{G2}$.

*Proof.* ($\Rightarrow$) **We first prove** $\neg\text{G1} \wedge \neg\text{G2} \Rightarrow \exists e : \forall T \in \mathcal{T} : \text{CT}_{SER}(T, e)$.

Let $H$ define a history over $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ and let $DSG(H)$ be the corresponding direct serialization graph. Together $\neg$G1c and $\neg$G2 state that the $DSG(H)$ must not contain anti-dependency or dependency cycles: $DSG(H)$ must therefore be acyclic. Let $i_1, ...i_n$ be a permutation of $1, 2, ..., n$ such that $T_{i_1}, ..., T_{i_n}$ is a topological sort of $DSG(H)$ ($DSG(H)$ is acyclic and can thus be topologically sorted). We construct an execution $e$ according to the topological order defined above:

$e : s_0 \rightarrow s_{T_{i_1}} \rightarrow s_{T_{i_2}} \rightarrow ... \rightarrow s_{T_{i_n}}$ and show that $\forall t \in \mathcal{T}.\text{CT}_{SER}(T, e)$. Specifically, we show that for all $T = T_{i_j}$, $\text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_{j-1}}})$ where $s_{T_{i_{j-1}}}$ is the parent state of $T_{i_j}$.

Consider the three possible types of operations in $T_{i_j}$:

1. *External Reads*: an operation reads an object version that was created by another transaction.

2. *Internal Reads*: an operation reads an object version that it itself created.

3. *Writes*: an operation creates a new object version.

We show that the parent state of $T_{i_j}$ is included in the read set of each of those operation types:

1. *External Reads*. Let $r_{i_j}(x_{i_k})$ read the version for $x$ created by $T_{i_k}$, where $k \neq j$.

   We first show that $s_{T_{i_k}} \xrightarrow{*} s_{T_{i_{j-1}}}$. As $T_{i_j}$ directly read-depends on $T_{i_k}$, there must exist an edge $T_{i_k} \xrightarrow{wr} T_{i_j}$ in $DSG(H)$, and $T_{i_k}$ must therefore be ordered before $T_{i_j}$ in the topological sort of $DSG(H)$ ($k < j$). Given $e$ was constructed by applying every transaction in $\mathcal{T}$ in topological order, it follows that $s_{T_{i_k}} \xrightarrow{*} s_{T_{i_{j-1}}}$.

   Next, we argue that the state $s_{T_{i_{j-1}}}$ contains the object-value pair $(x, x_{i_k})$. Specifically, we show that there does not exists a $s_{T_{i_l}}$, where $k < l < j$, such that $T_{i_l}$ writes a different version of $x$. We prove this by contradiction. Consider the smallest such $l$: $T_{i_j}$ reads the version of $x$ written by $T_{i_k}$ and $T_{i_l}$ writes a different version of $x$. $T_{i_l}$, in fact, writes the next version of $x$ as $e$ is constructed according to $ww$ dependencies: if there existed an intermediate version of $x$, then either $T_{i_l}$ was not the smallest transaction, or $e$ does not respect $ww$ dependencies. Note that $T_{i_j}$ thus directly anti-depends on $T_{i_l}$, i.e. $T_{i_j} \xrightarrow{rw} T_{i_l}$. As the topological sort of $DSG(H)$ from which we constructed $e$ respects anti-dependencies, we finally have $s_{i_j} \xrightarrow{*} s_{T_{i_l}}$, i.e. $j \leq l$, a contradiction. We conclude: $(x, x_{i_k}) \in s_{T_{i_{j-1}}}$, and therefore $s_{T_{i_{j-1}}} \in \mathcal{RS}_e(r_{i_j}(x_{i_k}))$.

2. *Internal Reads*. Let $r_{i_j}(x_{i_j})$ read $x_{i_j}$ such that $w(x_{i_j}) \xrightarrow{to} r(x_{i_j})$. By definition, the read state set of such an operation consists of $\forall s \in \mathcal{S}_e : s \xrightarrow{*} s_p$. Since $s_{T_{i_{j-1}}}$ is $T_{i_j}$'s parent state, it trivially follows that $s_{T_{i_{j-1}}} \in \mathcal{RS}_e(r_{i_j}(x_{i_j}))$.

3. *Writes*. Let $w_{i_j}(x_{i_j})$ be a write operation. By definition, its read state set consists of all the states before $s_{T_{i_j}}$ in the execution. Hence it also trivially follows that $s_{T_{i_{j-1}}} \in \mathcal{RS}_e(w_{i_j}(x_{i_j}))$.

Thus $s_{T_{i_{j-1}}} \in \bigcap\limits_{o \in \Sigma_{T_{i_j}}} \mathcal{RS}_e(o)$. We have $\textsc{complete}_{e,T_{i_j}}(s_{T_{i_{j-1}}})$ for any $T_{i_j}$ : $\forall T \in \mathcal{T} : \text{CT}_{SER}(T, e)$.

(⇐) **We next prove** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{SER}(T, e) \Rightarrow \neg\text{G1} \wedge \neg\text{G2}$.

To do so, we prove the contrapositive $\text{G1} \vee \text{G2} \Rightarrow \forall e\ \exists T \in \mathcal{T} : \neg\text{CT}_{SER}(T, e)$. Let $H$ be a history that displays phenomena G1 or G2. We generate a contradiction. Consider any execution $e$ such that $\forall T \in \mathcal{T} : \text{CT}_{SER}(T, e)$. We first instantiate the version order for $H$, denoted as $<<$, as follows: given an execution $e$ and an object $x$, $x_i << x_j$ if and only if $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \wedge s_{T_i} \xrightarrow{*} s_{T_j}$. First, we show that:

**Claim 1** *$T_i \to T_j$ in $DSG(H) \Rightarrow s_{T_i} \xrightarrow{*} s_{T_j}$ in the execution e ($i \neq j$).*

*Proof.* Consider the three edge types in $DSG(H)$:

$T_i \xrightarrow{ww} T_j$ There exists an object $x$ s.t. $x_i << x_j$ (version order). By construction, we have $s_{T_i} \xrightarrow{*} s_{T_j}$.

$T_i \xrightarrow{wr} T_j$ There exists an object $x$ s.t. $T_j$ reads version $x_i$ written by $T_i$. Let $s_{T_k}$ be the parent state of $s_{T_j}$, i.e. $s_{T_k} \to s_{T_j}$. By assumption $\text{CT}_{SER}(e, T)$ ($T = T_j$), i.e. $\textsc{complete}_{e,T_j}(s_{T_k})$, hence we have $(x, x_i) \in s_{T_k}$. For the effects of $T_i$ to be visible in $s_{T_k}$, $T_i$ must have been applied at an earlier point in the execution. Hence we have: $s_{T_i} \xrightarrow{*} s_{T_k} \to s_{T_j}$.

$T_i \xrightarrow{rw} T_j$ There exist an object $x$ s.t. $T_i$ reads version $x_m$ written by $T_m$, $T_j$ writes $x_j$ and $x_m << x_j$. By construction, $x_m << x_j$ implies $s_{T_m} \xrightarrow{*} s_{T_j}$. Let $s_{T_k}$ be the parent state of $s_{T_j}$, i.e. $s_{T_k} \to s_{T_i}$. As $\text{CT}_{SER}(e, T)$, where $t = T_j$, holds by assumption, i.e. $\textsc{complete}_{e,T_i}(s_{T_k})$, the key-value pair $(x, x_m) \in s_{T_k}$, hence $s_{T_m} \xrightarrow{*} s_{T_k}$ as before. In contrast, $s_{T_i} \xrightarrow{*} s_{T_j}$: indeed,$(x, x_m) \in s_{T_k}$ and $x_m << x_j$. Hence, $T_j$ has not yet been applied. We thus have $s_{T_k} \to s_{T_i} \xrightarrow{*} s_{T_j}$.

□

We now derive a contradiction in all cases of the disjunction $\text{G1} \vee \text{G2}$:

- Let us assume that $H$ exhibits phenomenon G1a (aborted reads). There must exists events $w_i(x_i), r_j(x_i)$ in $H$ such that $T_i$ subsequently aborted. $\mathcal{T}$ and any corresponding execution $e$, however, consists only of committed transactions. Hence $\forall e : \nexists s \in \mathcal{S}_e$, s.t. $s \in \mathcal{RS}_e(r_j(x_i))$: no complete state can exists for $T_j$. There thus exists a transaction for which the commit test cannot be satisfied, for any $e$. We have a contradiction.

- Let us assume that $H$ exhibits phenomenon G1b (intermediate reads). In an execution $e$, only the final writes of a transaction are applied. Hence, $\nexists s \in \mathcal{S}_e$, s.t. $s \in \mathcal{RS}_e(r(x_{intermediate}))$. There thus exists a transaction, which for all $e$, will not satisfy the commit test. We once again have a contradiction.
- Finally, let us assume that the history $H$ displays one or both phenomena G1c or G2. Any history that displays G1c or G2 will contain a cycle in the $DSG$. Hence, there must exist a chain of transactions $T_i \rightarrow T_{i+1} \rightarrow ... \rightarrow T_j$ such that $i = j$ in $DSG(H)$. By Claim 1, we thus have $s_{T_i} \xrightarrow{*} s_{T_{i+1}} \xrightarrow{*} \ldots \xrightarrow{*} s_{T_j}$ for any $e$. By definition however, a valid execution must be totally ordered. We have our final contradiction.

All cases generate a contradiction. We have G1 $\vee$ G2 $\Rightarrow \forall e : \exists T \in \mathcal{T} : \neg\text{CT}_{SER}(e, T)$. This completes the proof. □

## A.3 Snapshot Isolation

**Theorem 2.** $\exists e : \forall T \in \mathcal{T}.\text{CT}_{SI}(T, e) \equiv \neg\text{G1} \wedge \neg\text{G-SI}$

*Proof.* ($\Rightarrow$) **We first prove** $\neg$G1 $\wedge$ $\neg$G-SI $\Rightarrow \exists e : \forall T \in \mathcal{T} : \text{CT}_{SI}(T, e)$.

**Commit Test** We can construct an execution $e$ such that every committed transaction satisfies the commit test $\text{CT}_{SI}(e, T)$. Let $i_0, ...i_n$ be a permutation of $1, 2, ..., n$ such that $T_{i_1}, ..., T_{i_n}$ are sorted according to their commit point. We construct an execution $e$ according to the topological order defined above: $e : s_0 \rightarrow s_{T_{i_1}} \rightarrow s_{T_{i_2}} \rightarrow ... \rightarrow s_{T_{i_n}}$ and show that $\forall T \in \mathcal{T}.\text{CT}_{SI}(T, e)$. Specifically, we prove the following: consider the largest *k* such that $T_{i_k} \xrightarrow{sd} T_{i_j}$, i.e. $c_{T_{i_k}} \prec_t b_{T_{i_j}}$ then $\text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_k}}) \wedge (\Delta(s_{T_{i_k}}, s_{T_{i_{j-1}}}) \cap \mathcal{W}_{s_{T_{i_j}}} = \emptyset)$.

**Complete State** We first prove that $\text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_k}})$. Consider the three possible types of operations in $T_{i_j}$:

1. *External Reads*: an operation reads an object version that was created by another transaction.

2. *Internal Reads*: an operation reads an object version that itself created.

3. *Writes*: an operation creates a new object version.

We show that the $s_{T_{i_k}}$ is included in the read set of each of those operation types:

1. *External Reads*. Let $r_{i_j}(x_{i_q})$ read the version for $x$ created by $T_{i_q}$, where $q \neq j$.

   We first show that $s_{T_{i_q}} \xrightarrow{*} s_{T_{i_k}}$. As $T_{i_j}$ directly read-depends on $T_{i_q}$, there must exist an edge $T_{i_q} \xrightarrow{wr} T_{i_j}$ in $SSG(H)$. Given that $H$ disallows phenomenon G-SIa by assumption, there must therefore exist a start-dependency edge $T_{i_q} \xrightarrow{sd} T_{i_j}$ in $SSG(H)$. Therefore we have $c_{T_{i_q}} \prec_t b_{T_{i_j}}$. By definition of time-precedes order, $b_{T_{i_j}} \prec_t c_{T_{i_j}}$. By transitivity of the partial order $c_{T_{i_q}} \prec_t c_{T_{i_j}}$. Given $e$ was constructed by applying every transaction $\mathcal{T}$ in topological order of $c$, and that we select the largest $k$ such that $T_{i_k} \xrightarrow{sd} T_{i_j}$, it follows that $q \leq k < j$ and $s_{T_{i_q}} \xrightarrow{*} s_{T_{i_k}} \xrightarrow{+} s_{T_{i_j}}$.

   Next, we argue that the state $s_{T_{i_k}}$ contains the object value pair $(x, x_{i_q})$. Specifically, we argue that there does not exist a $s_{T_{i_m}}$, where $q < m \leq k$, such that $T_{i_m}$ writes a new version of $x$. We prove this by contradiction. Consider the smallest such $m$: $T_{i_k}$ reads the version of $x$ written by $T_{i_q}$ and $T_{i_m}$ writes the next version of $x$. $T_{i_j}$ thus directly anti-depends on $T_{i_m}$— i.e., $T_{i_j} \xrightarrow{rw} T_{i_m}$. Given that in time-precedes order, for any two transactions, the start point of one is always comparable to the commit point of the other, we necessarily have $b_{T_{i_j}} \prec_t c_{T_{i_m}}$. Otherwise we would have $c_{T_{i_j}} \prec_t b_{T_{i_m}} \prec_t c_{T_{i_m}}$, i.e. $c_{T_{i_j}} \prec_t c_{T_{i_m}}$, which is inconsistent with the order defined by the execution. In addition, it holds by assumption that $T_{i_k} \xrightarrow{sd} T_{i_j}$. We can conclude that $c_{T_{i_k}} \prec_t b_{T_{i_j}}$. Combined with $b_{T_{i_j}} \prec_t c_{T_{i_m}}$, we will have $c_{T_{i_k}} \prec_t c_{T_{i_m}}$. However, we constructed the execution respecting the time-precedes order of commit point. We have a contradiction. Hence we conclude: $(x, x_{i_q}) \in s_{T_{i_k}}$ and therefore $s_{T_{i_k}} \in \mathcal{RS}_e(r_{i_j}(x_{i_q}))$.

2. *Internal Reads*. Let $r_{i_j}(x_{i_j})$ read $x_{i_j}$ such that $w_{i_j}(x_{i_j}) \xrightarrow{to} r_{i_j}(x_{i_j})$. By definition, the read state set of such an operation consists of $\forall s \in \mathcal{S}_e : s \xrightarrow{*} s_p$. Since $s_{T_{i_k}}$ precedes $s_{T_{i_j}}$ in the topological order ($T_{i_k} \xrightarrow{sd} T_{i_j}$, therefore $c_{T_{i_k}} \prec_t b_{T_{i_j}}$. Combined with $b_{T_{i_j}} \prec_t c_{T_{i_j}}$, we have $c_{T_{i_k}} \prec_t c_{T_{i_j}}$. and $e$ respects time-precedes order) , it trivially follows that $s_{T_{i_k}} \in \mathcal{RS}_e(r_{i_j}(x_{i_j}))$.

3. *Writes*. Let $w_{i_j}(x_{i_j})$ be a write operation. By definition, its read state set consists of all the states before $s_{T_{i_j}}$ in the execution. Hence it also trivially follows that $s_{T_{i_k}} \in \mathcal{RS}_e(w_{i_j}(x_{i_j}))$.

Thus $s_{T_{i_k}} \in \bigcap_{o \in \Sigma_{T_{i_j}}} \mathcal{RS}_e(o)$.

**Distinct Write Sets** We now prove the second half of the commit test: $(\Delta(s_{T_{i_k}}, s_{T_{i_{j-1}}}) \cap \mathcal{W}_{s_{T_{i_j}}} = \emptyset)$ We prove this by contradiction. Consider the largest $m$, where $k < m < j$ such that $\mathcal{W}_{s_{T_{i_m}}} \cap \mathcal{W}_{s_{T_{i_j}}} \neq \emptyset$. $T_{i_m}$ thus directly write-depends on $T_{i_j}$, i.e. $T_{i_m} \xrightarrow{ww} T_{i_j}$. By assumption, $H$ proscribes phenomenon G-SIa. Hence, there must exist an edge $T_{i_m} \xrightarrow{sd} T_{i_j}$ in $SSG(H)$. Similarly, we have $c_{T_{i_k}} \prec_t b_{T_{i_j}} \prec_t c_{T_{i_j}}$, i.e. $c_{T_{i_k}} \prec_t c_{T_{i_j}}$ As $e$ respects time-precedes order of commit points, it follows that $s_{i_m} \xrightarrow{+} s_{i_j}$ $(m < j)$. By assumption however, $T_{i_k}$ is the latest transaction in $e$ such that $T_{i_k} \xrightarrow{sd} T_{i_j}$, so $m \leq k$. Since we had assumed that $k < m < j$, we have a contradiction. Thus, $\forall m, k < m < j, \mathcal{W}_{s_{T_{i_m}}} \cap \mathcal{W}_{s_{T_{i_j}}} = \emptyset$. We conclude that $\Delta(s_{T_{i_k}}, s_{T_{i_{j-1}}}) \cap \mathcal{W}_{s_{T_{i_j}}} = \emptyset$ We have $\text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_k}}) \wedge (\Delta(s_{T_{i_k}}, s_{T_{i_{j-1}}}) \cap \mathcal{W}_{s_{T_{i_j}}} = \emptyset)$ for any $T_{i_j}$: $\forall T \in \mathcal{T} : \text{CT}_{SI}(T, e)$.

($\Leftarrow$) **We next prove** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{SI}(T, e) \Rightarrow \neg\text{G1} \wedge \neg\text{G-SI}$.

Let $e$ be an execution such that $\forall T \in \mathcal{T} : \text{CT}_{SI}(T, e)$, and $H$ be a history for committed transactions $\mathcal{T}$. We first instantiate the version order for $H$, denoted as $<<$, as follows: given an execution $e$ and an object $x$, $x_i << x_j$ if and only if $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \wedge s_{T_i} \xrightarrow{*} s_{T_j}$. It follows that, for any two states such that $(x, x_i) \in T_{i_m} \wedge (x, x_j) \in T_{i_n} \Rightarrow s_{T_m} \xrightarrow{+} s_{T_n}$. We next assign the start and commit points of each transaction. We assume the existence of a monotonically increasing timestamp counter: if a transaction $T_i$ requests a timestamp $ts$, and a transaction $T_j$ subsequently requests a timestamp $ts'$, then $ts < ts'$. Writing $e$ as $s_0 \rightarrow s_{T_1} \rightarrow s_{T_2} \rightarrow \cdots \rightarrow s_{T_n}$, our timestamp assignment logic is then the following:

1. Let $i = 0$.

2. Set $s = s_{T_i}$; if $i = 0$, $s = s_0$.

3. Assign a commit timestamp to $T_{s_i}$ if $i \neq 0$.

4. Assign a start timestamp to all transactions $T_k$ such that $T_k$ satisfies $\text{COMPLETE}_{e,T_k}(s) \wedge (\Delta(s, s_p(T_k)) \cap \mathcal{W}_{s_{T_k}} = \emptyset)$ and $T_k$ does not already have a start timestamp.

5. Let $i = i + 1$. Repeat 1-4 until the final state in $e$ is reached.

We can relate the history's start-dependency order and execution order as follows:

**Claim 2** $\forall T_i, T_j \in \mathcal{T} : s_{T_j} \xrightarrow{*} s_{T_i} \Rightarrow \neg T_i \xrightarrow{sd} T_j$

*Proof.* We have $T_i \xrightarrow{sd} T_j \Rightarrow c_i \prec_t b_j$ by definition. Moreover, the start point of a transaction $T_i$ is always assigned before its commit point. Hence: $c_i \prec_t b_j \prec_t c_j$. It follows from our timestamp assignment logic that $s_{T_i} \xrightarrow{+} s_{T_j}$. We conclude: $T_i \xrightarrow{sd} T_j \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$. Taking the contrapositive of this implication completes the proof. □

**G1** We first prove that: $\forall T \in \mathcal{T} : \mathrm{CT}_{SI}(T, e) \Rightarrow \neg$G1. We do so by contradiction for each of G1a, G1b, G1c.

**G1a** Let us assume that $H$ exhibits phenomenon G1a (aborted reads). There must exist events $w_i(x_i), r_j(x_i)$ in $H$ such that $T_i$ subsequently aborted. $\mathcal{T}$ and any corresponding execution $e$, however, consists only of committed transactions. Hence $\forall e : \nexists s \in \mathcal{S}_e : s \in \mathcal{RS}_e(r_j(x_i))$: no complete state can exists for $T_j$. There thus exists a transaction for which the commit test cannot be satisfied, for any $e$. We have a contradiction.

**G1b** Let us assume that $H$ exhibits phenomenon G1b (intermediate reads). In an execution $e$, only the final writes of a transaction are applied. Hence, $\nexists s \in \mathcal{S}_e : s \in \mathcal{RS}_e(r(x_{intermediate}))$. There thus exists a transaction, which for all $e$, will not satisfy the commit test. We once again have a contradiction.

**G1c** Finally, let us assume that $H$ exhibits phenomenon G1c: SSG(H) must contain a cycle of read/write dependencies. We consider each possible edge in the cycle in turn:

- $T_i \xrightarrow{ww} T_j$ There must exist an object $x$ such that $x_i << x_j$ (version order). By construction, version in $H$ is consistent with the execution order $e$: we have $s_{T_i} \xrightarrow{*} s_{T_j}$.
- $T_i \xrightarrow{wr} T_j$ There must exist a read $r_j(x_i) \in \Sigma_{T_j}$ such that $T_j$ reads version $x_i$ written by $T_i$. By assumption, $\mathrm{CT}_{SI}(e, T_j)$ holds. There must therefore exists a state $s_{T_k} \in \mathcal{S}_e$ such that $\textsc{complete}_{e,T_j}(s_{T_k})$. If $s_{T_k}$ is a complete state for $T_j$, $s_{T_k} \in \mathcal{RS}_e(r_j(x_i))$ and $(x, x_i) \in s_{T_k}$. For the effects of $T_i$ to be visible in $s_{T_k}$, $T_i$ must have been applied at an earlier point in the execution. Hence we have: $s_{T_i} \xrightarrow{*} s_{T_k}$. Moreover, by definition of the candidate read states, $s_{T_k} \xrightarrow{*} s_p(T_j) \rightarrow s_{T_j}$(Definition 2). It follows that $s_{T_i} \xrightarrow{*} s_{T_j}$.

If a history $H$ displays phenomenon G1c, there must exist a chain of transactions $T_i \to T_{i+1} \to ... \to T_j$ such that $i = j$. A corresponding cycle must thus exist in the execution $e$ $s_{T_i} \xrightarrow{*} s_{T_{i+1}} \xrightarrow{*} \ldots \xrightarrow{*} s_{T_j}$. By definition however, a valid execution must be totally ordered. We once again have a contradiction.

We generate a contradiction in all cases of the disjunction: we conclude that the history $H$ cannot display phenomenon G1.

**G-SI** We now prove that $\forall T \in \mathcal{T} : \text{CT}_{SI}(T, e) \Rightarrow \neg$G-SI.

**G-SIa** We first show that G-SIa cannot happen for both write-write dependencies and write-read dependencies:

- $T_i \xrightarrow{wr} T_j$ There must exist an object $x$ such that $T_j$ reads version $x_i$ written by $T_i$. Let $s_{T_k}$ be the first state in $e$ such that $\text{COMPLETE}_{e,T_j}(s_{T_k}) \wedge (\Delta(s_{T_k}, s_p(T_j)) \cap \mathcal{W}_{s_{T_j}} = \emptyset)$. Such a state must exist since $\text{CT}_{SI}(e, T_j)$ holds by assumption. As $s_{T_k}$ is complete, we have $(x, x_i) \in s_{T_k}$. For the effects of $T_i$ to be visible in $s_{T_k}$, $T_i$ must have been applied at an earlier point in the execution. Hence we have: $s_{T_i} \xrightarrow{*} s_{T_k} \xrightarrow{*} s_{T_j}$. It follows from our timestamp assignment logic that $c_i \preceq_t c_k$. Similarly, the start point of $T_j$ must have been assigned after $T_k$'s commit point (as $s_{T_k}$ is $T_j$'s earliest complete state), hence $c_k \prec_t s_j$. Combining the two inequalities results in $c_i \prec_t s_j$: there will exist a start-dependency edge $T_i \xrightarrow{sd} T_j$. $H$ will not display G-SIa for write-read dependencies.
- $T_i \xrightarrow{ww} T_j$ There must exist an object $x$ such that $T_j$ writes the version $x_j$ that follows $x_i$. By construction, it follows that $s_{T_i} \xrightarrow{*} s_{T_j}$. Let $s_{T_k}$ be the first state in the execution such that $\text{COMPLETE}_{e,T_j}(s_{T_k}) \wedge (\Delta(s_{T_k}, s_p(T_j)) \cap \mathcal{W}_{T_j} = \emptyset)$. We first show that: $s_{T_i} \xrightarrow{*} s_{T_k}$. Assume by way of contradiction that $s_{T_k} \xrightarrow{+} s_{T_i}$. The existence of a write-write dependency between $T_i$ and $T_j$ implies that $\mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \neq \emptyset$, and consequently, that $\Delta(s_{T_k}, s_p(T_j)) \cap \mathcal{W}_{T_j} \neq \emptyset$, contradicting our assumption that $\text{CT}_{SI}(e, T_j)$. We conclude that: $s_{T_i} \xrightarrow{*} s_{T_k}$. It follows from our timestamp assignment logic that $c_i \preceq_t c_k$. Similarly, the start point of $T_j$ must have been assigned after $T_k$'s commit point (as $s_{T_k}$ is $T_j$'s earliest complete state), hence $c_k \prec_t s_j$. Combining the two inequalities results in $c_i \prec_t s_j$: there will exist a start-dependency edge $T_i \xrightarrow{sd} T_j$. $H$ will not display G-SIa for write-write dependencies.

The history $H$ will thus not display phenomenon G-SIa.

**G-SIb** We next prove that $H$ will not display phenomenon G-SIb. Our previous result states that $H$ proscribes G-SIa: all read-write dependency edges between two transactions implies the existence of a start dependency edge between those same transactions. We prove by contradiction that $H$ proscribes G-SIb. Assume that SSG(H) consists of a directed cycle $cyc_1$ with exactly one anti-dependency edge (it displays G-SIb) but proscribes G-SIa. All other dependencies will therefore be write/write dependencies, write/read dependencies, or start-depend edges. By G-SIa, there must exist an equivalent cycle $cyc_2$ consisting of a directed cycle with exactly one anti-dependency edge and start-depend edges only. Start-edges are transitive (consider three transactions $T_1, T_2$ and $T_3$: if $c_1 \prec_t b_2$ and $c_2 \prec_t b_3$ then $c_1 \prec_t b_3$ as $b_2 \prec_t c_2$ by definition), hence there must exist a cycle $cyc_3$ with exactly one anti-dependency edge and one start-depend edge. We write $T_i \xrightarrow{rw} T_j \xrightarrow{sd} T_i$. Given $T_i \xrightarrow{rw} T_j$, there must exist an object $x$ and transaction $T_m$ such that $T_m$ writes $x_m$, $T_i$ reads $x_m$ and $T_j$ writes the next version of $x$, $x_j$ ($x_m << x_j$). Let $s_{T_k}$ be the earliest complete state of $T_i$. Such a state must exist as $\text{CT}_{SI}(e, T_i)$ by assumption. Hence, by definition of read state $(x, x_m) \in s_{T_k}$. Similarly, $(x, x_j) \in s_{T_j}$ by the definition of state transition (Definition 1). By construction, we have $s_{T_k} \xrightarrow{+} s_{T_j}$. Our timestamp assignment logic maintains the following invariant: given a state $s_T$, $\forall T_k : \text{COMPLETE}_{e,T_k}(s_T) : \forall s_{T_n} : s_T \xrightarrow{+} s_{T_n} \Rightarrow b_k \prec_t c_n$. Intuitively, the start timestamp of all transactions associated with a particular complete state $s_T$ is smaller than the commit timestamp of any transaction that follows $s_T$ in the execution. We previously showed that $s_{T_k} \xrightarrow{+} s_{T_j}$. Given $s_{T_k}$ is a complete state for $T_i$, we conclude $b_i \prec_t c_j$. However, the edge $T_j \xrightarrow{sd} T_i$ implies that $c_j \prec_t b_i$. We have a contradiction: no such cycle can exist and $H$ will not display phenomenon G-SI. We generate a contradiction in all cases of the conjunction, hence $\forall T \in \mathcal{T} : \text{CT}_{SI}(T, e) \Rightarrow \neg$G-SI holds. We conclude $\forall T \in \mathcal{T} : \text{CT}_{SI}(T, e) \Rightarrow \neg\text{G-SI} \wedge \neg\text{G1}$. This completes the proof. □

## A.4 Read Committed

**Theorem 3**. $\exists e : \forall T \in \mathcal{T}.\text{CT}_{RC}(T, e) \equiv \neg\text{G1}$.

*Proof.* **We first prove** $\neg\text{G1} \Rightarrow \exists e : \forall T \in \mathcal{T} : \text{CT}_{RC}(T, e)$.

Let $H$ define a history over $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ and let $DSG(H)$ be the corresponding direct serialization graph. ¬G1c states that the $DSG(H)$ must not contain dependency cycles: the subgraph of $DSG(H)$, $SDSG(H)$ containing the same nodes but including only dependency edges, must be acyclic. Let $i_1, ...i_n$ be a permutation of $1, 2, ..., n$ such that $T_{i_1}, ..., T_{i_n}$ is a topological sort of $SDSG(H)$ ($SDSG(H)$ is acyclic and can thus be topologically sorted). We construct an execution $e$ according to the topological order defined above: $e : s_0 \to s_{T_{i_1}} \to s_{T_{i_2}} \to ... \to s_{T_{i_n}}$ and show that $\forall T \in \mathcal{T}.\text{CT}_{RC}(T, e)$. Specifically, we show that for all $T = T_{i_j}$, $\text{PREREAD}_e(T)$. Consider the three possible types of operations in $T_{i_j}$:

1. *External Reads*: an operation reads an object version that was created by another transaction.

2. *Internal Reads*: an operation reads an object version that itself created.

3. *Writes*: an operation creates a new object version.

We show that the read set for each of operation type is not empty:

1. *External Reads*. Let $r_{i_j}(x_{i_k})$ read the version for $x$ created by $T_{i_k}$, where $k \neq j$. We first show that $s_{T_{i_k}} \xrightarrow{*} s_{T_{i_j}}$. As $T_{i_j}$ directly read-depends on $T_{i_k}$, there must exist an edge $T_{i_k} \xrightarrow{wr} T_{i_j}$ in $SDSG(H)$, and $T_{i_k}$ must therefore be ordered before $T_{i_j}$ in the topological sort of $SDSG(H)$ ($k < j$), it follows that $s_{T_{i_k}} \xrightarrow{+} s_{T_{i_j}}$. As $(x, x_{i_k}) \in s_{T_{i_k}}$, we have $s_{T_{i_k}} \in \mathcal{RS}_e(r_{i_j}(x_{i_k}))$, and consequently $\mathcal{RS}_e(r_{i_j}(x_{i_k})) \neq \emptyset$.

2. *Internal Reads*. Let $r_{i_j}(x_{i_j})$ read $x_{i_j}$ such that $w(x_{i_j}) \xrightarrow{to} r(x_{i_j})$. By definition, the read state set of such an operation consists of $\forall s \in \mathcal{S}_e : s \xrightarrow{*} s_p$. $s_0 \xrightarrow{*} s$ trivially holds. We conclude $s_0 \in \mathcal{RS}_e(r_{i_j}(x_{i_j}))$, i.e. $\mathcal{RS}_e(r_{i_j}(x_{i_j})) \neq \emptyset$.

3. *Writes*. Let $w_{i_j}(x_{i_j})$ be a write operation. By definition, its read state set consists of all the states before $s_{T_{i_j}}$ in the execution. Hence $s_0 \in \mathcal{RS}_e(r_{i_j}(x_{i_j}))$, i.e. $\mathcal{RS}_e(r_{i_j}(x_{i_j})) \neq \emptyset$.

Thus $\forall o \in \Sigma_T : \mathcal{RS}_e(o) \neq \emptyset$. We have $\text{PREREAD}_e(T_{i_j})$ for any $T_{i_j} : \forall T \in \mathcal{T} : \text{CT}_{RC}(T, e)$.

($\Leftarrow$) **We next prove** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{RC}(T, e) \Rightarrow \neg$G1.

To do so, we prove the contrapositive G1 $\Rightarrow \forall e\ \exists T \in \mathcal{T} : \neg\text{CT}_{RC}(T, e)$. Let $H$ be a history that displays phenomena G1. We generate a contradiction. Assume that there exists an execution $e$ such that $\forall T \in \mathcal{T} : \text{CT}_{RC}(T, e)$. We first instantiate the version order for $H$, denoted as $<<$, as follows:

given an execution $e$ and an object $x$, $x_i << x_j$ if and only if $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \wedge s_{T_i} \xrightarrow{+} s_{T_j}$. First, we show that:

**Claim 3** *$T_i \rightarrow T_j$ in $SDSG(H) \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$ in the execution e ($i \neq j$).*

*Proof.* Consider the two edge types in $SDSG(H)$:

$T_i \xrightarrow{ww} T_j$ There exists an object $x$ s.t. $x_i << x_j$ (version order). By construction, we have $s_{T_i} \xrightarrow{+} s_{T_j}$.

$T_i \xrightarrow{wr} T_j$ There exists an object $x$ s.t. $T_j$ reads version $x_i$ written by $T_i$, i.e. $r_j(x, x_i) \in \Sigma_{T_j}$. By assumption CT$_{RC}(e, T)$ ($T = T_j$), i.e. PREREAD$_e(T_j)$, $\mathcal{RS}_e(o) \neq \emptyset$. Let $s \in \mathcal{RS}_e(o)$, by definition of $\mathcal{RS}_e(o)$, we have $(x, x_i) \in s \wedge s \xrightarrow{+} s_{T_j}$, therefore $T_i$ must be applied before or on state $s$, hence we have $s_{T_i} \xrightarrow{*} s \xrightarrow{+} s_{T_j}$, i.e. $s_{T_i} \xrightarrow{+} s_{T_j}$.

□

We now derive a contradiction in all cases of G1:

- Let us assume that $H$ exhibits phenomenon G1a (aborted reads). There must exists events $w_i(x_i), r_j(x_i)$ in H such that $T_i$ subsequently aborted. $\mathcal{T}$ and any corresponding execution $e$, however, consists only of committed transactions. Hence $\forall e : \nexists s \in \mathcal{S}_e, s.t.\ s \in \mathcal{RS}_e(r_j(x_i))$: no complete state can exists for $T_j$. There thus exists a transaction for which the commit test cannot be satisfied, for any e. We have a contradiction.

- Let us assume that $H$ exhibits phenomenon G1b (intermediate reads). In an execution $e$, only the final writes of a transaction are applied. Hence, $\nexists s \in \mathcal{S}_e, s.t.\ s \in \mathcal{RS}_e(r(x_{intermediate}))$. There thus exists a transaction, which for all e, will not satisfy the commit test. We once again have a contradiction.

- Finally, let us assume that the history $H$ displays G1c. Any history that displays G1c will contain a cycle in the SDSG(H). Hence, there must exist a chain of transactions $T_i \rightarrow T_k \rightarrow$ ... $\rightarrow T_j$ such that $i = j$. By Claim 3, we thus have $s_{T_i} \xrightarrow{+} s_{T_k} \xrightarrow{+} \ldots \xrightarrow{+} s_{T_j}$, $i = j$ for any $e$. By definition however, a valid execution must be totally ordered. We have our final contradiction.

All cases generate a contradiction. We have G1 $\Rightarrow \forall e : \exists T \in \mathcal{T} : \neg\text{CT}_{RC}(e, T)$. This completes the proof. □

## A.5 Read Uncommitted

**Theorem 4.**$\exists e : \forall t \in \mathcal{T}.\text{CT}_{RU}(t, e) \equiv \neg$G0.

*Proof.* **We first prove** $\neg$G0 $\Rightarrow \exists e : \forall T \in \mathcal{T} : \text{CT}_{RU}(T, e)$.

Let $H$ define a history over $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ and let $DSG(H)$ be the corresponding direct serialization graph. $\neg$G0 implies that the DSG(H) must not contain write-write dependency cycles. Let $i_1, ...i_n$ be a permutation of $1, 2, ..., n$ such that $T_{i_1}, ..., T_{i_n}$ is a topological sort of the DSG(H) according to the write-write edges (the projection of DSG(H) that considers write-write edges only is acyclic and can thus be topologically sorted). We construct an execution $e$ according to the topological order defined above: $e : s_0 \rightarrow s_{T_{i_1}} \rightarrow s_{T_{i_2}} \rightarrow ... \rightarrow s_{T_{i_n}}$. As $\text{CT}_{RU}(t, e) = True$, every transaction $T$ in $e$ trivially satisfies the commit test. This completes the proof.

($\Leftarrow$) **We next prove** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{RU}(T, e) \Rightarrow \neg$G0 To do so, we prove the contrapositive G0 $\Rightarrow \forall e\ \exists T \in \mathcal{T} : \neg\text{CT}_{RU}(T, e)$. Let $H$ be a history that displays phenomena G0. We generate a contradiction. Consider any execution $e$ such that $\forall T \in \mathcal{T} : \text{CT}_{RU}(T, e)$. We first instantiate the version order for $H$, denoted as $<<$, as follows: given an execution $e$ and an object $x$, $x_i << x_j$ if and only if $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \wedge s_{T_i} \xrightarrow{*} s_{T_j}$. First, we show that $T_i \xrightarrow{ww} T_j$ in $DSG(H) \Rightarrow s_{T_i} \xrightarrow{*} s_{T_j}$ in the execution e ($i \neq j$). The presence of a $ww$ edge implies the existence of an object $x$ s.t. $x_i << x_j$ (version order). It follows by construction that $s_{T_i} \xrightarrow{*} s_{T_j}$. Any history that displays G0 will contain a cycle consisting of $ww$ edges in the $DSG(H)$. Hence, there must exist a chain of transactions $T_i \xrightarrow{ww} T_{i+1} \xrightarrow{ww} ... \xrightarrow{ww} T_j$ such that $i = j$ in $DSG(H)$. As shown above, this sequence of $ww$ edges implies that $s_{T_i} \xrightarrow{*} s_{T_{i+1}} \xrightarrow{*} \ldots \xrightarrow{*} s_{T_j}$ for any $e$. By definition however, a valid execution must be totally ordered. We have a contradiction. We have G0 $\Rightarrow \forall e : \exists T \in \mathcal{T} : \neg\text{CT}_{RU}(e, T)$. This completes the proof.

□

# B Equivalence to read-atomic

Read atomic [25], like PSI, was introduced a scalable alternative to snapshot isolation. Read atomic preserves atomic visibility (transactions observe either all or none of a committed transaction's effects) but does not preclude write-write conflicts nor guarantee that transactions will read from a causally consistent prefix of the execution. These weaker guarantees allow for efficient implementations in which one client's transactions cannot cause another client's transactions to fail (synchronization independence). We can express read atomic in our state-based model as follows:

**Definition 11** $\mathrm{CT}_{RA}(T, e) \equiv \textit{PREREAD}_e(T) \wedge \forall r_1(k_1, v_1), r_2(k_2, v_2) \in \Sigma_T \wedge k_2 \in \mathcal{W}_{T_{sf_{r_1}}} \Rightarrow sf_{r_1} \xrightarrow{*} sf_{r_2}$

Intuitively, this definition states that, if an operation $o_1$ observes a transaction $T_i$'s writes, all subsequent operations that read a key included in $T_i$'s write-set must read from a state that includes $T_i$'s effects. In this section, we prove the following theorem:

**Theorem 2.** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{RA}(T, e) \equiv$ *Read Atomic (§B.2).*

## B.1 Bailis et al. [25] model summary

We summarize the key definitions of the model here (an alternative formalization was given by Cerone et al. [46]).

A history $H$ consists of a set of reads/writes where each write creates a version of an item $x$, $x_i$, where $i$ is a unique timestamp taken from a totally ordered set such that timestamps induce a total order on versions of each item (and a partial order across versions of different items). We write $x_i <<_H x_j$ if $i < j$.

A history is read-atomic iff it prevents the following anomalies:

- uncommitted, aborted, or intermediate reads (G0, G1 anomalies)
- fractured reads. A transaction $T_j$ exhibits fractured reads if transaction $T_i$ writes versions $x_m$ and $y_n$ (where x and y can be equal), $T_j$ reads version $x_m$ and version $y_k$ and $k < n$.

## B.2 Read Atomic

We now prove the following theorem: $\exists e : \forall T \in \mathcal{T} : \text{CT}_{RA}(T, e) \equiv \mathit{Read\ Atomic}$.

($\Leftarrow$) **First, we prove:** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{RA}(T, e) \Leftarrow$ Read Atomic. We show that a history $H$ consisting of the set of transactions $\mathcal{T}$ exhibiting no fractured reads and intermediate/aborted/uncommitted reads, implies the existence of an execution $e$ that contains $\mathcal{T}$ such that every $T \in \mathcal{T}$ satisfies the commit test: $\text{CT}_{RA}(T, e) \equiv \text{PREREAD}_e(T) \wedge \forall r_1(k_1, v_1), r_2(k_2, v_2) \in \Sigma_T \wedge k_2 \in \mathcal{W}_{T_{sf_{r_1}}} \Rightarrow sf_{r_1} \xrightarrow{*} sf_{r_2}$

The set of transactions $\mathcal{T}$ defines a partial order $<_T$ such that: if $r_j(x_i) \in \Sigma_{T_j}$ then $T_i <_T T_j$ (read dependency edges) and if $x_i <<_H x_j$ then $T_i <_T T_j$, where $T_i$ writes $x_i$ and $T_j$ writes $x_j$ (write dependency edges).. Such a partial order must exist as read atomic proscribes cycles consisting exclusively of read dependency edges and write dependency edges (it precludes G1). We construct this execution $e$ to be a linearization of this partial order. Consider an arbitrary transaction $T$ in $e$ and consider two operations $r_1(x, x_i)$ and $r_2(y, y_j)$ such that $y \in \mathcal{W}_{T_{sf_{r_1}}}$ For simplicity, let us refer to $T_i$ for $T_{sf_{r_1}}$, to $T_j$ for $T_{sf_{r_2}}$. These are the transactions that created the versions $x_i$ and $y_j$. Finally, let us refer to $T_i$'s write of $y$ as $y_i$.

Considering an arbitrary transaction $T$, we prove that $\text{PREREAD}_e(T)$ holds. Let us assume by contradiction that there exists an operation $o$ executed by an operation $T$ for which $\mathcal{RS}_e(o) = \emptyset$. There are two possibilities: either $o = r(k, v)$ read from a state that succeeds $T$ in the execution (let that state be $s_{T_v}$), or $\nexists s \in \mathcal{S}_e : (k, v) \in s$. In the first case, we have that $o$ reads from $T_v$ and hence that $T_v$ precedes $T$ in the partial order. But our execution $e$ is a linearization of that partial order, hence we cannot have $s \xrightarrow{+} s_{T_v}$. We have a contradiction. In the second case, $\nexists s \in \mathcal{S}_e : (k, v) \in s$. There are two sub-scenarios: either the transaction that wrote $v$ does not exist in the execution, in which case $T_v$ aborted (a contradiction as read atomic disallows aborted reads), or the state that $T_v$ created has $(k, v') \in s_{T_v}$ where $v \neq v'$, in which case $v$ was not the final write of $T_v$ (a contradiction as read atomic disallows intermediate reads). In all cases, we have a contradiction, hence $\mathcal{RS}_e(o) \neq \emptyset$ and $\text{PREREAD}_e(T)$ holds for all $T$. Moreover, as $\text{PREREAD}_e(T)$ holds, $\forall o \in \Sigma_T . \mathcal{RS}_o \neq \emptyset$ so $sf_{r_1}$ and $sf_{r_2}$ exist.

Assume by contradiction that $s_{T_j} \xrightarrow{+} s_{T_i}$. $T_j$ and $T_i$ both write to object $y$ and are therefore ordered according to $<_T$ ( $T_j <_T T_i$ and therefore $y_j <<_H y_i$). By assumption, $H$ does not exhibit fractured reads: if $T_i$ writes $x_i$ and $y_i$ and $T$ reads version $x_i$ and version $y_j$ , then $y_i <<_H y_j$ or $y_i = y_j$. But we just argued that $y_j <<_H y_i$. We have a contradiction: $s_{T_i} \xrightarrow{*} s_{T_j}$.

($\Rightarrow$) **Next, we prove:** ($\exists e : \forall T \in \mathcal{T} : \text{CT}_{RA}(T, e)$) We show that, given an execution $e$ and associated set of transactions $\mathcal{T}$ such that every $T \in \mathcal{T}$ satisfies the commit test, the history $H$ does not exhibit fractured reads, and intermediate/aborted/uncommitted reads.

By $\text{PREREAD}_e(T)$, the object versions observed by all transactions stem from a state that existed in the execution: that is, were generated by the final write of a committed transaction. It follows that the corresponding history does not exhibit uncommitted, aborted or intermediate reads anomaly.

Next, we show that the history H does not exhibit fractured reads. We assign a monotonically increasing timestamp to each transaction in $e$ (we write $T_i$ for a transaction with timestamp $i$) such that $s_{T_i} \xrightarrow{+} s_{T_j} \equiv i \leq j$ and the version order on each object is consistent with timestamps and execution order. Let us assume by way of contradiction that $H$ exhibits fractured reads: there exists a transaction $T_i$ that writes versions $x_i$ and $y_i$ of objects $x$ and $y$ such that a transaction $T_j$ reads version $x_i$ and version $y_k$ and $y_k << y_i$. $T_j$ satisfies the commit test by definition. That is, $\forall r_1(k_1, v_1), r_2(k_2, v_2) \in \Sigma_T \wedge k_2 \in \mathcal{W}_{t_{sf_{r_1}}} \Rightarrow sf_{r_1} \xrightarrow{*} sf_{r_2}$ Letting $r_1$ be $T_j$'s read of $x$ and $r_2$ $T_j$'s read of $y$, we have that $sf_{r_1} \xrightarrow{*} sf_{r_2}$ or equivalently that $s_i \xrightarrow{*} s_k$. By construction, it follows that $y_i << y_k$. However, we have just argued that $y_k << y_i$ or $i = k$. We have a contradiction, so H does not exhibit fractured reads.

# C Equivalence to ANSI, Strong and Session SI

In this section, we prove the following theorems:

**Theorem 8** (**a**) $\exists e : \forall t \in \mathcal{T} : \mathrm{CT}_{ANSI\ SI}(T, e) \equiv ANSI\ SI$ (§C.2).

**Theorem 9** (**b**) $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{Session\ SI}(T, e) \equiv SSessSI$ (§C.3).

**Theorem 7** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{Strong\ SI}(T, e) \equiv Strong\ SI$ (§C.4).

## C.1 Berenson/Daudjee et al. [31, 61] model summary

Every transaction $T$ in this model has a logical *start* timestamp, written start$(T)$ and a logical *commit* timestamp, written commit$(T)$. We write $\delta$ to be the smallest unit by which two timestamps differ.

**Definition 12** ***ANSI SI*** *A history $H$, consisting of the set of transactions $\mathcal{T}$, satisfies ANSI Snapshot Isolation (or weak snapshot isolation) iff, for every $T$:*

- *$T$'s start timestamp is less than or equal to the actual start time of $T$: $\forall T, T'$ : start$(T) \leq T.start$* ***(R1)****.*
- *$T$'s commit timestamp is more recent than any start or commit timestamp previously assigned: $\forall T, T'$ : $T'.commit < T.commit \equiv$ commit$(T') <$ commit$(T)$* ***(R2a)****, $\forall T, T'$ : $T.commit > T'.start \Rightarrow$ commit$(T) >$ start$(T')$* ***(R2b)*** *and start$(T) <$ commit$(T)$* ***(R2c)***
- *$T$ observes the effects of all transactions $T'$ with commit$(T') \leq$ start$(T)$ and does not observe the writes of transactions $T'$ with commit$(T) \geq$ start$(T)$* ***(R3)****.*
- *$T$ commits only if no other committed transaction $T'$ with lifespan $[$start$(T')$, commit$(T')]$ that overlaps with $T$'s lifespan of $[$start$(T)$, commit$(T)]$* ***(R4)*** *has an intersecting writeset.*

**Definition 13** ***Strong Session SI (R5)*** *A transaction execution history H is strong session SI under labeling $L_H$ iff it is ANSI SI, and if, for every pair of committed transactions $T_i$ and $T_j$ in H such that $L_H(T_i) = L_H(T_j)$ and $T_i$'s commit precedes the first operation of $T_j$, $T_i <_s T_j \Rightarrow$ commit$(T_i) \leq$ start$(T_j)$.*

**Definition 14** ***Strong SI (R6)*** *A transaction execute history H is strong SI iff it is weak SI and if, for every pair of committed transaction $T_i$ and $T_j$ in H such that $T_i$'s commit precedes the first operation* *of* $T_j$, $T_i <_s T_j \Rightarrow$ *commit*$(T_i) \leq$ *start*$(T_j)$

## C.2 ANSI SI

We now prove the following theorem:

**Theorem 8 (a)** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{ANSI\ SI}(T, e) \equiv ANSI\ SI.$

*Proof.* ($\Leftarrow$) **First, we prove:** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{ANSI\ SI}(T, e) \Leftarrow ANSI\ SI$. We show that a history $H$ consisting of the set of transactions $\mathcal{T}$ implies the existence of an execution $e$ that contains $\mathcal{T}$ such that every $T \in \mathcal{T}$ satisfies the commit test: $\text{CT}_{ANSI\ SI}(T, e) = \text{C-ORD}(T_{s_p}, T) \wedge \exists s \in S_e : \text{COMPLETE}_{e,T}(s) \wedge \text{NO-CONF}_T(s) \wedge T_s <_s T$

To do so, we consider the execution resulting from applying every transaction in $\mathcal{T}$ in the order of their commit timestamps. More precisely, we have that $\forall T, T' \in \mathcal{T}$ : commit$(T) <$ commit$(T') \equiv s_T \xrightarrow{+} s_{T'}$.

**C-ORD**$(T_{s_p}, T)$ First, we show that C-ORD$(T_{s_p}, T)$ holds. As any parent state $s_p$ must precede $T$ in the execution, we have that $s_p \rightarrow s_T$, ie: commit$(T_{s_p}) <$ commit$(T)$ By (R2a), $\forall T, T' \in \mathcal{T}$ : $T.commit < T'.commit \equiv$ commit$(T) <$ commit$(T')$. It follows that $T_{s_p}.commit < T.commit$ or C-ORD$(T_{s_p}, T)$ holds.

**Complete State** Second, we show that there exists a complete state $s$, where $s$ is the the state resulting from applying the transaction $T_s$ with the highest commit timestamp that is smaller than start$(T)$. There exists a single such transactions as commit timestamps are unique (R2). We show that $s$ is a candidate read state for every operation $o \in \Sigma_T$. If $o$ is a write, then $s \in \mathcal{RS}_e(o)$ trivially. If $o$ is a read and returns a value $v$ for object $k$ (written by a transaction $T_v$), we show by contradiction that $(k, v) \in s$. Assume that $(k, v') \in s$ with $v \neq v'$, consider the last transaction $T_{v'} \neq T_v$ that writes $v'$, where either $s_{T_v} \xrightarrow{+} s_{T_{v'}} \xrightarrow{*} s$ (1) or $s_{T_{v'}} \xrightarrow{*} s \xrightarrow{+} s_{T_v}$ in $e$ (2). In the first case, commit$(T_v) \leq$ commit$(T_{v'})$ by construction and commit$(T_v') \leq$ commit$(T_s) \leq$ start$(T)$ by definition of $T_s$. By (R3), $T$ should therefore observe the effects of $T_{v'}$, but it does not as it reads $v$, a contradiction so $(k, v) \in s$. In the second case, commit$(T_{v'}) \leq$ commit$(T_s) \leq$ commit$(T_v)$. By (R3), commit$(T_v) \leq$ start$(T)$ as $T$ observes the effect of $T_v$. $T_v$'s commit timetamp is greater than $T_s$'s but smaller than $T$'s start timestamp. Yet, we defined $T_s$ to be the transaction with the highest commit timestamp that is smaller than the start timestamp of $T$. We have a contradiction, so $(k, v) \in s$. It follows that $s$ is a candidate read state for every $o \in \Sigma_T$: it is a complete state.

**Time Order** Third, we show that $T_s <_s T$ holds. By construction commit$(T_s) \leq$ start$(T)$. By (R2b) we have that $T_s.commit > T.start \Rightarrow$ commit$(T_s) >$ start$(T)$. Taking the contrapositive, commit$(T_s) \leq$ start$(T) \Rightarrow T_s.commit \leq T.start$. By assumption, real-time values of *start* and *commit* are distinct, so we can strengthen the inequality to $T_s.commit \leq T.start$, and therefore: $T_s <_s T$. **NO-CONF**$_T(s)$ Finally, we show that NO-CONF$_T(s)$. First, we show that any transaction corresponding to states between $s$ and $s_p$ (included) must overlap with $T$. Let that set be $\mathcal{T}_c$. By construction of $e$, $s$ and $T_s$, every transaction $T_c \in \mathcal{T}_c$ has a commit timestamp greater than commit$(T_s)$ and smaller than commit$(T)$ so commit$(T_s) \leq$ commit$(T_c) \leq$ commit$(T)$. By construction of $T_s$, we know that it is the transaction with the highest commit timestamp that is smaller or equal to start$(T)$. Any higher commit timestamp must be greater than start$(T)$. It follows that start$(T) \leq$ commit$(T_c) \leq$ commit$(T)$ and that $T_c$ necessarily overlaps with $T$. By R4, its write-set cannot intersect $T$'s. As such, no transaction in $\mathcal{T}_c$ has a write-set that overlaps with $T$'s. Hence NO-CONF$_T(s)$.

This concludes the proof.

($\Rightarrow$) **Next, we prove:** ($\exists e : \forall T \in \mathcal{T} :$ CT$_{ANSI\ SI}(T, e) \Rightarrow ANSI\ SI$) We show that, given an execution $e$ and associated set of transactions $\mathcal{T}$ such that every $T \in \mathcal{T}$ satisfies the commit test, we can assign to every transaction a start and commit timestamp such that (R1),(R2),(R3) and (R4) hold.

First, we denote the latest complete state that satisfies NO-CONF$_T(s) \wedge T_s <_s T$ for transaction $T$ as the *selected read state* (formally $\exists s$ : (COMPLETE$_{e,T}(s) \wedge s \xrightarrow{*} s_T) \wedge (\forall s'.$COMPLETE$_{e,T}(s') \Rightarrow s' \xrightarrow{*} s)$). That state must exist as every $T \in \mathcal{T}$ satisfies the commit test (by assumption).

We then assign commit timestamps using the following algorithm: let $s_{latest}$ the last state in $e$ (such that $\nexists s : s_{latest} \rightarrow s$) , let COMMIT_MAX be the maximum assignable value of any commit$(T)$, let selected$(T)$ be $T$'s selected read state, and finally, let $ts$ be an array indexed by transaction id that stores the maximum candidate commit timestamp for every transaction. An entry in $ts[T]$ represent an upper bound on the final timestamp commit$(T)$ of a transaction $T$.

```
1. $ts[] = \{$COMMIT_MAX, ... , COMMIT_MAX$\}$
2. $s_{curr} = s_{latest}$
3. $T_{curr} = T_{latest}$ (where $T_{latest}$) the transaction that created $s_{latest}$
4. max-commit = COMMIT_MAX
5. $do\{$
    (a) commit$(T_{curr}) = min($max-commit$, ts[T_{curr}])$
    (b) $max - commit =$ commit$(T_{curr}) - \delta$
    (c) $s_c = selected(T_{curr})$
    (d) $ts[T_c] = min(T_{curr}.start, ts[T_c])$
    (e) $s_{curr} = s$ for $s \to s_{curr}$
    (f) $\}$
6. while ($s_{curr}$ exists)
```

Intuitively, this algorithm assigns a logical commit timestamp commit$(T)$ to every transaction $T$ that is 1) smaller than the real-time start timestamp of every transaction that has $s_T$ (the state that $T$ creates) as selected read state 2) smaller than all the commit timestamps of states that succede $s_T$ in the execution.

We can then assign the start timestamp of every transaction $T \in \mathcal{T}$ to be the commit timestamp of the transaction $T_c$ associated with $T$'s selected complete state $s_c = selected(T) + \epsilon$ where $\epsilon$ is a small constant that is smaller than a timestamp time unit.

We first prove the following lemma:

**Lemma 1.** $\forall T, T'.commit(T) \leq commit(T') \equiv s_T \xrightarrow{+} s_{T'}$

Assuming first that $s_T \xrightarrow{+} s_{T'}$: by construction, we assign the commit timestamp of a transaction $T_i$ to be $min($max-commit$, ts[T_i])$, where max-commit $=$ commit$(T_j) - \delta$ where $s_{T_j} \to s_{T_i}$ and hence commit$(T_i) \leq$ commit$(T_j)$. By transitivity, $\forall T, T' \in \mathcal{T}.s_T \xrightarrow{+} s_{T'} \Rightarrow$ commit$(T) \leq$

commit($T'$). Assuming instead that commit($T$) $\leq$ commit($T'$) and, assuming by contradiction that $s_{T'} \xrightarrow{*} s_T$. If $T = T'$, it directly follows that $T$ and $T'$ must have the same commit timestamp, which gives us a contradiction. Otherwise, we have by construction that, given two transactions $T_i$ and $T_j$ we assign the commit timestamp of a transaction $T_i$ to be $min$(max-commit, $ts[T_i]$), where max-commit = commit($T_j$) $- \delta$ where $s_{T_j} \rightarrow s_{T_i}$ and hence commit($T_i$) $\leq$ commit($T_j$). By transitivity, $s_T \xrightarrow{+} s_{T'} \Rightarrow$ commit($T$) $\leq$ commit($T'$), which again gives us a contradiction.

**R1** First, we prove that (R1) holds. Consider an arbitratry transaction $T$ in $e$ and let $s_c$ be its selected read state. By construction, start($T$) = commit($T_c$) where commit($T_c$) is defined to be the minimum $T'.start$ of all transactions $T'$ that have it as selected read state, including $T$. It follows trivially that start($T$) $\leq T.start$ (for $T' = T$)

**R2** Second, we prove that R2 holds. To prove R2a, consider an arbitrary transaction $T$ in $e$ and let $s_c$ be its selected complete state. By assumption, we have that C-ORD($T_{s_p}, T$). By induction, one can easily prove that $\forall T, T' \in \mathcal{T} : T.commit < T'.commit \Leftrightarrow s_T \xrightarrow{+} s_{T'}$ (1). Combined with Lemma 1, we have $\forall T, T' \in \mathcal{T} : T.commit < T'.commit \Rightarrow s_T \xrightarrow{+} s_{T'} \Rightarrow$ commit($T$) $<$ commit($T'$) so (R2a) holds. Moreover, as start($T$) = commit($T_c$) and the selected read state of $T$ necessarily precedes $T$ in $e$, it also follows that start($T$) $<$ commit($T$). Hence R2c also holds. Finally, we show that R2b holds by contradiction. Assume that commit($T'$) $\leq$ start($T$) and that $T.start < T'.commit$. As start($T$) is equal (modulo $\epsilon$) to the timestamp of the selected commit state of $T$, $s_c$ we have commit($T'$) $\leq$ commit($T_c$) By Lemma 1 and the aforementioned property (1), it follows that $T'.commit \leq T_c.commit$. By assumption, we have $T_c <_s T$, ak $T_c.commit < T.start$, and consequently $T'.commit \leq T_c.commit \leq T.start$. But we had $T.start < T'.commit$. We have a contradiction and thus R2b holds.

**R3** Third, we prove that R3 holds. We first show that a transaction $T$ observes the effects of all transactions with commit($T'$) $<$ start($T$) by contradiction. Consider this transaction $T$ which generates state $s$ when committing. Let $s_c$ and $T_c$ be the selected read state for $T$. Assume that there exists a transaction $T'$ with timestamp commit($T'$) $<$ start($T$) whose effects $T$ does not observe: there exists a key $k$ that is written by two transactions $T'$ (WRITE($k$,$v$')) and $T_v$ (WRITE($k$,$v$)) such that $s_{T_v} \xrightarrow{+} s_{T'} \xrightarrow{*} s$ in $e$ and $T$ reads value $v$. By construction, we know that start($T$) = commit($T_c$)

so commit$(T')$ $\leq$ commit$(T_c)$. It follows that $s_{T'} \xrightarrow{*} s_c$ by Lemma 1 and $s_{T'} \xrightarrow{*} s_c \xrightarrow{+} s$ (as the selected read state necessarily precedes $s$). As $T$ misses the effect of $T'$ by reading $v$, we can extend this to $s_{T_v} \xrightarrow{+} s_{T'} \xrightarrow{*} s_c \xrightarrow{+} s$ We know, however, that $(k, v) \in s_c$ as $s_c$ is a complete read state for $T$ so $k \notin \Delta(s_{T_v}, s_c)$. Yet, we had $T'$ write $k$. We have a contradiction: $T$ observes the effects of all transactions $T'$ with commit$(T')$ $<$ start$(T)$.

Next, we show that $T$ does not observe the writes of transactions $T'$ with commit$(T')$ $>$ start$(T)$. Assume that $T$ observes the effect of transaction $T'$ with commit$(T')$ $>$ start$(T)$. As before, let $s_c$ and $T_c$ be the selected read state for $T$. By construction, start$(T)$ $=$ commit$(T_c) + \epsilon$, hence commit$(T_c)$ $\leq$ commit$(T')$ and consequently $s_{T_c} \xrightarrow{+} s_{T'}$ by Lemma 1. We know by assumption that $s_{T_c}$ is the latest complete state such that NO-CONF$_T(s) \wedge T_s <_s T$. Let $o$ =WRITE($k$,$v$) be the write from $T'$ that $T$ observes. As $T'$ created version $v$ and $s_{T_c} \xrightarrow{+} s_{T'}$ holds, $(k, v) \notin s_c$, so $s_c$ cannot be a read state for $o$, and as such cannot be a complete state for $T$. We have a contradiction. R3 holds.

**R4** Fourth, we prove that R4 holds. Consider an arbitratry transaction $T$ in $e$ and let $s_c$ be its selected complete state. A transaction $T$ overlaps with committed $T'$ if start$(T)$ $\leq$ commit$(T')$ and commit$(T')$ $\leq$ commit$(T)$. By construction, start$(T)$ $=$ commit$(T_c)$, so we have commit$(T_c)$ $\leq$ commit$(T')$. By Lemma 1, commit$(T_c)$ $\leq$ commit$(T')$ $\leq$ commit$(T)$ $\Rightarrow s_{T_c} \xrightarrow{*} s_{T'} \xrightarrow{*} s_T$. By NO-CONF$_T(s)$ it follows that the writeset of $T'$ does not intersect the writeset of $T$, so R4 holds.

□

## C.3 Strong Session SI

We now prove the following theorem: **Theorem 9** (a): $\exists e : \forall T \in \mathcal{T} :$ CT$_{Session\ SI}(T, e) \equiv SSessSI$.

*Proof.* ($\Leftarrow$) **First, we prove:** $\exists e : \forall T \in \mathcal{T} :$ CT$_{Session\ SI}(T, e) \Leftarrow$ Session SI. We show that a history $H$ consisting of the set of transactions $\mathcal{T}$ implies the existence of an execution $e$ that contains $\mathcal{T}$ such that every $T \in \mathcal{T}$ satisfies the commit test: CT$_{Session\ SI}(T, e) =$ C-ORD$(T_{s_p}, T) \wedge \exists s \in S_e :$ COMPLETE$_{e,T}(s) \wedge$ NO-CONF$_T(s) \wedge (T_s <_s T) \wedge (\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s)$

C-ORD$(T_{s_p}, T) \wedge \exists s \in S_e :$ COMPLETE$_{e,T}(s) \wedge$ NO-CONF$_T(s)$ holds by an identical proof to that of

Theorem 8(a)($\Leftarrow$). We do not repeat the proof here and simply show that $(\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s)$ To do so, we consider, as in C.2, the execution resulting from applying every transaction in $\mathcal{T}$ in the order of their commit timestamps. More precisely, we have that $\forall T, T' \in \mathcal{T}$ : commit$(T) <$ commit$(T') \equiv s_T \xrightarrow{+} s_{T'}$. Let us consider a transaction $T$ and let $s$ be the state resulting from applying the transaction $T_s$ with the highest commit timestamp that is smaller or equal to start$(T)$. This state is a complete state (as shown in C.2).

Assume by contradiction that there exists a transaction $T'$ such that $T' \xrightarrow{se} T$ and $s \xrightarrow{+} s_{T'}$. By construction, we have that commit$(T_s) \leq$ commit$(T')$. As $s$ is the state associated with the transaction with highest commit timestamp that is smaller or equal to start$(T)$, it follows that commit$(T') >$ start$(T)$. But, by assumption (R5) $T' \xrightarrow{se} T \Rightarrow$ commit$(T') \leq$ start$(T)$. We have a contradiction, hence $(\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s)$ holds. This completes the proof.

($\Rightarrow$) **Next, we prove:** $(\exists e : \forall T \in \mathcal{T} :$ CT$_{Session\ SI}(T, e) \Rightarrow$ Session SI). (R1), (R2), (R3), (R4) hold by an identical proof to that of Theorem 8(a)($\Rightarrow$). We consider the same execution $e$ for which the start/commit timestamps are assigned according to the algorithm described in the proof. We do not repeat the proof here and simply show that (R5) holds. To do so, we consider two transactions $T$ and $T'$ such that $T' \xrightarrow{se} T$ and show that commit$(T') \leq$ start$(T)$. As $T$ and $T'$ both satisfy the commit test, we have that $\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s_c$ where $s_c$ is the selected read state for $T$. By Lemma 1, $s_{T'} \xrightarrow{*} s_c \Rightarrow$ commit$(T') \leq$ commit$(T_c)$. Moreover, we have by construction that start$(T) =$ commit$(T_c) + \epsilon$. Hence commit$(T') \leq$ start$(T)$. This completes the proof.

□

## C.4 Strong SI

*Proof.* We now prove the following theorem: **Theorem 7** : $\exists e : \forall T \in \mathcal{T} :$ CT$_{Strong\ SI}(T, e) \equiv$ $Strong\ SI$.

($\Leftarrow$) **First, we prove:** $\exists e : \forall T \in \mathcal{T} :$ CT$_{Strong\ SI}(T, e) \Leftarrow Strong\ SI$. We show that a history $H$ consisting of the set of transactions $\mathcal{T}$ implies the existence of an execution $e$ that contains $\mathcal{T}$ such that every $T \in \mathcal{T}$ satisfies the commit test: CT$_{Strong\ SI}(T, e) =$ C-ORD$(T_{s_p}, T) \wedge \exists s \in S_e$ : COMPLETE$_{e,T}(s) \wedge$ NO-CONF$_T(s) \wedge (T_s <_s T) \wedge (\forall T' <_s T : s_{T'} \xrightarrow{*} s)$.

$\textsc{c-ord}(T_{s_p}, T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s)$ holds by an identical proof to that of Theorem 8(a)(⇐). We do not repeat the proof here and simply show that $(\forall T' <_s T : s_{T'} \xrightarrow{*} s)$ To do so, we consider, as in § C.2, the execution resulting from applying every transaction in $\mathcal{T}$ in the order of their commit timestamps. More precisely, we have that $\forall T, T' \in \mathcal{T}$ : commit$(T) <$ commit$(T') \equiv s_T \xrightarrow{*} s_{T'}$. Let us consider a transaction $T$ and let $s$ the state resulting from applying the transaction $T_s$ with the highest commit timestamp that is smaller or equal to start$(T)$. This state is a complete state (as shown in § C.2).

Assume by contradiction that there exists a transaction $T'$ such that $T' <_s T$ and $s \xrightarrow{*} s_{T'}$. By construction, we have that commit$(T_s) \leq$ commit$(T')$. As $s$ is the state associated with the transaction with highest commit timestamp that is smaller or equal to start$(T)$, it follows that commit$(T') >$ start$(T)$. But, by assumption (R6) $T' <_s T \Rightarrow$ commit$(T') \leq$ start$(T)$. We have a contradiction, hence $(\forall T' <_s T : s_{T'} \xrightarrow{*} s)$ holds. This completes the proof.

(⇒) **Next, we prove:** $(\exists e : \forall T \in \mathcal{T} : \text{CT}_{Strong\ SI}(T, e) \Rightarrow Strong\ SI)$. (R1), (R2), (R3), (R4) hold by an identical proof to that of Theorem 8(a)(⇒). We consider the same execution $e$ for which the start/commit timestamps are assigned according to the algorithm described in the proof. We do not repeat the proof here and simply show that R6 holds. To do so, we consider two transactions $T$ and $T'$ such that $T' <_s T$ and show that commit$(T') \leq$ start$(T)$. As $T$ and $T'$ both satisfy the commit test, we have that $\forall T' <_s T : s_{T'} \xrightarrow{*} s_c$ where $s_c$ is the selected read state for $T$. By Lemma 1, $s_{T'} \xrightarrow{*} s_c \Rightarrow$ commit$(T') \leq$ commit$(T_c)$. Moreover, we have by construction that start$(T) =$ commit$(T_c) + \epsilon$. Hence commit$(T') \leq$ start$(T)$. This completes the proof.

□

# D Equivalence to PC-SI and GSI

In this section, we prove the following theorems:

**Theorem 8** (**b**) $\exists e : \forall T \in \mathcal{T} : \text{CT}_{ANSI\ SI}(T, e) \equiv GSI$ (§D.2)

**Theorem 9** (**b**) $\exists e : \forall T \in \mathcal{T} : \text{CT}_{Session\ SI}(T, e) \equiv PC\text{-}SI$ (§D.3)

## D.1 Elnikety et al. [154] model summary

Every transaction $T$ in this model has a real-time *start* timestamp, written start($T$) and a real-time *commit* timestamp, written commit($T$). All the timestamps are distinct.

- $snapshot(T_i)$: the time at which $T_i$'s snapshot is taken.
- $start(T_i)$: the time of the first operation of $T_i$.
- $commit(T_i)$: the time of $C_i$, if $T_i$ commits.
- $abort(T_i)$: the time of $A_i$, if $T_i$ aborts.
- $end(T_i)$: the time of either $C_i$ or $A_i$.
- $T_j$ impacts $T_i$: $writeset(T_i) \wedge writeset(T_j) \neq \emptyset$ and $snapshot(T_i) \leq commit(T_j) < commit(T_i)$

Note that $start(T_i)$ and $commit(T_i)$ have the same definition as $T_i.start$ and $T_i.commit$, they will be used interchangeably in the proof.

**Definition 15** ***Generalized Snapshot Isolation (GSI)*** *For any history H created by GSI, the following two properties hold (where i, j, and k are distinct)*

***D1. (GSI Read Rule)*** $\underline{\forall T_i, X_j \textit{ such that } R_i(X_j) \in h:}$

***1-*** $W_j(X_j) \in h$ ***and*** $C_j \in h$;

***2-*** $commit(T_j) < snapshot(T_i)$;

***3-*** $\forall T_k$ *such that* $W_k(X_k), C_k \in h : [commit(T_k) < commit(T_j)$ ***or*** $snapshot(T_i) < commit(T_k)]$

***D2. (GSI Commit Rule)*** $\underline{\forall T_i, T_j \textit{ such that } C_i, C_j \in h \textit{ :}}$

***4-*** $\neg(T_j \; impacts \; T_i)$.

**Definition 16** ***Prefix-consistent Snapshot Isolation (PC-SI)*** *For any history H created by PC-SI, the following two properties hold (where i,j,k are distinct)*

***P1. (PC-SI Read Rule)*** $\underline{\forall T_i, X_j \textit{ such that } R_i(X_j) \in h \textit{ :}}$

***1-*** $W_j(X_j) \in h$ ***and*** $C_j \in h$*;*

***2-*** $commit(T_j) < snapshot(T_i)$*;*

***3-*** $\forall T_k$ *such that* $W_k(X_k), C_k \in h$ *:* $[commit(T_k) < commit(T_j)$ ***or*** $snapshot(T_i) < commit(T_k)]$

***4-*** $T_i \sim T_j$ ***and*** $commit(T_j) < start(T_i)$*:* $commit(T_j) < snapshot(T_i)$

***P2. (PC-SI Commit Rule)*** $\underline{\forall T_i, T_j \textit{ such that } C_i, C_j \in h \textit{ :}}$

***5-*** $\neg(T_j \; impacts \; T_i)$.

## D.2 Generalized Snapshot Isolation

We now prove **Theorem 8** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{ANSI}\text{SI}(T, e) \equiv$ GSI i.e. $\text{C-ORD}(T_{s_p}, T) \wedge \exists s \in S_e : \text{COMPLETE}_{e,T}(s) \wedge \text{NO-CONF}_T(s) \wedge (T_s <_s T) \equiv D1 \wedge D2$.

*Proof.* **(⇐) We first prove** $D1 \wedge D2 \Rightarrow \exists e : \forall T \in \mathcal{T} : \text{CT}_{ANSI\ SI}(e, T)$.

**Commit Test** We can construct an execution $e$ such that every committed transaction satisfies the commit test $\text{CT}_{ANSI\ SI}(e, T)$. By definition, all operations are assigned distinct timestamps (including start and commit operations). Let $i_1, ...i_n$ be a permutation of $1, 2, ..., n$ such that committed transactions $T_{i_1}, ..., T_{i_n}$ are totally ordered by their commit time. We construct an execution $e$ according to the topological order defined above: $e : s_0 \rightarrow s_{T_{i_1}} \rightarrow s_{T_{i_2}} \rightarrow ... \rightarrow s_{T_{i_n}}$ and show that $\forall T_{i_j} \in \mathcal{T} : \text{CT}_{ANSI\ SI}(e\ , T_{i_j})$. Specifically, we prove the following: consider the largest *k* such that $commit(T_{i_k}) < snapshot(T_{i_j})$, $D1 \wedge D2 \Rightarrow \text{C-ORD}(T_{s_p(T_j)}, T_j) \wedge \text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_k}}) \wedge \text{NO-CONF}_{T_{i_j}}(s_{T_{i_k}}) \wedge (T_{i_k} <_s T_{i_j})$ .

**Commit Order** We first prove that $\text{C-ORD}(T_{s_p(T_j)}, T_j)$ is true. In the execution, we ordered the transactions by their commit time, it therefore directly follows that $T_{s_p(T_j)}.commit < T_j.commit$.

**Complete State** Next, we prove that $\text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_k}})$. Consider the three possible types of operations in $T_{i_j}$:

1. *External Reads*: an operation reads an object version that was created by another transaction.
2. *Internal Reads*: an operation reads an object version that itself created.
3. *Writes*: an operation creates a new object version.

We show that the $s_{T_{i_k}}$ is included in the read set of each of those operation types:

1. *External Reads*. Let $r_{i_j}(x_{i_q})$ read the version for $x$ created by $T_{i_q}$ where $q \neq j$, i.e. $R_{i_j}(X_{i_q}) \in h$ in the definition of GSI.

   We first show that $s_{T_{i_q}} \xrightarrow{*} s_{T_{i_k}}$. By rule **D1-2**, we have $\mathit{commit}(T_{i_q}) < \mathit{snapshot}(T_{i_j})$. Since $k$ is the largest number such that $\mathit{commit}(T_{i_k}) < \mathit{snapshot}(T_{i_j})$, we have $q \leq k$, and consequently $s_{T_{i_q}} \xrightarrow{*} s_{T_{i_k}}$. Next, we argue that the state $s_{T_{i_k}}$ contains the object value pair $(x, x_{i_q})$. Specifically, we argue that there does not exist a $s_{T_{i_m}}$, where $q < m \leq k$, such that $T_{i_m}$ writes a new version of $x$. We prove this by contradiction. Consider any such $m$ (and note that the execution contains only committed transactions), we have $W_{i_m}(X_{i_m})$ and $C_{i_m} \in h$. By **D1-3**, we have either (i) $\mathit{commit}(T_{i_m}) < \mathit{commit}(T_{i_q})$ or (ii) $\mathit{snapshot}(T_{i_j}) < \mathit{commit}(T_{i_m})$. By assumption, we have $q \leq m \leq k$. By construction, it follows that $\mathit{commit}(T_{i_m}) > \mathit{commit}(T_{i_q})$, a contradiction with (i). Therefore (ii) should hold. However, since $m \leq k$, we have $\mathit{commit}(T_{i_m}) \leq \mathit{commit}(T_{i_k})$. Given that $\mathit{commit}(T_{i_k}) < \mathit{snapshot}(T_{i_j})$, we have $\mathit{commit}(T_{i_m}) < \mathit{commit}(T_{i_j})$, a contradiction with $\mathit{snapshot}(T_{i_j}) < \mathit{commit}(T_{i_m})$. Neither (i) nor (ii) holds, we conclude that such $m$ does not exist. Hence we conclude: $(x, x_{i_q}) \in s_{T_{i_k}}$ and therefore $s_{T_{i_k}} \in \mathcal{RS}_e(r_{i_j}(x_{i_q}))$.

2. *Internal Reads*. Let $r_{i_j}(x_{i_j})$ read $x_{i_j}$ such that $w_{i_j}(x_{i_j}) \xrightarrow{to} r_{i_j}(x_{i_j})$. By definition, the read state set of such an operation consists of $\forall s \in \mathcal{S}_e : s \xrightarrow{+} s_{T_{i_j}}$. Since $\mathit{commit}(T_{i_k}) < \mathit{snapshot}(T_{i_j}) < \mathit{commit}(T_{i_j})$, $s_{T_{i_k}} \xrightarrow{+} s_{T_{i_j}}$ by construction. It trivially follows that $s_{T_{i_k}} \in \mathcal{RS}_e(r_{i_j}(x_{i_j}))$.

3. *Writes*. Let $w_{i_j}(x_{i_j})$ be a write operation. By definition, its read state set consists of all the states before $s_{T_{i_j}}$ in the execution. Hence it also trivially follows that $s_{T_{i_k}} \in \mathcal{RS}_e(w_{i_j}(x_{i_j}))$.

Thus $s_{T_{i_k}} \in \bigcap_{o \in \Sigma_{T_{i_j}}} \mathcal{RS}_e(o)$, i.e. $\text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_k}})$.

**Distinct Write Sets** We now prove the third part of the commit test: $\textsc{no-conf}_{T_{i_j}}(s_{T_{i_k}})$, i.e. $(\Delta(s_{T_{i_k}}, s_{T_{i_{j-1}}}) \cap \mathcal{W}_{T_{i_j}} = \emptyset)$. We prove this by contradiction. Consider any $m$, where $k < m < j$ such that $\mathcal{W}_{T_{i_m}} \cap \mathcal{W}_{T_{i_j}} \neq \emptyset$. Since $k$ is the largest index such that $commit(T_{i_k}) < snapshot(T_{i_j})$, we have $commit(T_{i_m}) \geq snapshot(T_{i_j})$. Furthermore, we have $commit(T_{i_m}) < commit(T_{i_j})$ by construction. Combining the two inequalities, we have $snapshot(T_{i_j}) \leq commit(T_{i_m}) < commit(T_{i_j})$ and consequently $writeset(T_{i_m}) \cap writeset(T_{i_j}) = \emptyset$ by D2(4). Yet, we assumed $\mathcal{W}_{T_{i_m}} \cap \mathcal{W}_{T_{i_j}} \neq \emptyset$. We have a contradiction. Thus, $\forall m, k < m < j, \mathcal{W}_{T_{i_m}} \cap \mathcal{W}_{T_{i_j}} = \emptyset$. We conclude that $\Delta(s_{T_{i_k}}, s_{T_{i_{j-1}}}) \cap \mathcal{W}_{T_{i_j}} = \emptyset$.

**Time Order** Finally, we prove that $T_{i_k} <_s T_{i_j}$. Since $commit(T_{i_k}) < snapshot(T_{i_j})$ and $snapshot(T_{i_j}) \leq start(T_{i_j})$ by definition, we have $commit(T_{i_k}) < start(T_{i_j})$, i.e. $T_{i_k} <_s T_{i_j}$. We have consequently proved that $\textsc{c-ord}(T_{s_p(T_{i_j})}, T_{i_j}) \wedge \textsc{complete}_{e,T_{i_j}}(s_{T_{i_k}}) \wedge \textsc{no-conf}_{T_{i_j}}(s_{T_{i_k}}) \wedge (T_{i_k} <_s T_{i_j})$, and consequently that $D1 \wedge D2 \Rightarrow \exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{ANSI\ SI}(e, T)$.

**(⇒) We next prove** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{ANSI\ SI}(e, T) \Rightarrow D1 \wedge D2$. Let $e$ be an execution such that $\forall T \in \mathcal{T} : \mathrm{CT}_{ANSI\ SI}(T, e)$, and $H$ be a history for committed transactions $\mathcal{T}$. Note that since $e$ satisfies $\textsc{c-ord}(T_{s_p(T)}, T)$, the order of transactions in $e$ is the same as ordering by time, i.e. $s_T \xrightarrow{*} s_{T'} \equiv T.commit < T'.commit$. Now we assign a snapshot time to each transaction. For any $T_i$, let $s_{T_k}$ be the state such that $\textsc{complete}_{e,T_i}(s_{T_k}) \wedge \textsc{no-conf}_{T_i}(s_{T_k}) \wedge (T_k <_s T_i)$ (by $\mathrm{CT}_{ANSI\ SI}(T, e)$) and set $snapshot(T_i) = commit(T_k) + \epsilon$, where $\epsilon$ is a small constant that is smaller than a time unit. The assigned snapshot time satisfies $snapshot(t) \leq start(t)$: since $T_k <_s T_i$, we have $commit(T_k) < start(T_i)$, therefore $snapshot(T_i) = commit(T_k) + \epsilon < start(T_i)$ as $\epsilon$ is smaller than a time unit.

**D1** First, we prove that **D1** is satisfied. Consider $x_j$ such that $R_i(x_j) \in h$. Since $\textsc{complete}_{e,T_i}(s_{T_k})$, we have $(x, x_j) \in s_{T_k}$, therefore the transaction executing $W_j(x_j)$ has been applied in the execution, i.e. $s_{T_j} \xrightarrow{*} s_{T_k}$. Moreover $e$ contains only committed transactions. Hence $W_j(x_j) \in h$ and $C_j \in h$ and consequently that **D1-1** holds. Moreover, since $s_{T_j} \xrightarrow{*} s_{T_k}$, by $\textsc{c-ord}(T_{s_p(T)}, T)$, we have $commit(T_j) \leq commit(T_k) < snapshot(T_i)$, hence **D1-2** is also satisfied. We now consider **D1-3**. For any $T_q$ such that $W_q(x_q), C_q \in h$, there can be only two cases: (i) $commit(T_q) < commit(T_j)$, for which **D1-3** is directly satisfied ; (ii) $commit(T_q) \geq commit(T_j)$: since $(x, x_j) \in s_{T_k}$, $T_q$ must be applied after $T_k$, it follows that $commit(T_q) > commit(T_k) + \epsilon = snapshot(T_i)$. In either case, **D1-3** is satisfied. Combining all previous results, we conclude that D1 is satisfied.

**D2** Now, we prove that **D2** is satisfied. Consider any $T_j$, there can only be two cases: $writeset(T_i) \cap writeset(T_j) = \emptyset$ (1) and $writeset(T_i) \cap writeset(T_j) \neq \emptyset$ (2). The first case trivially satisfies $\neg(T_j\ impacts\ T_i)$. In the second case, as $\text{NO-CONF}_{T_i}(s_{T_k})$ holds, we have either $s_{T_j} \xrightarrow{*} s_{T_k}$ or $s_{T_i} \xrightarrow{*} s_{T_j}$. If $s_{T_j} \xrightarrow{*} s_{T_k}$, $commit(T_j) \leq commit(T_k) < snapshot(T_i)$, and hence that $\neg(T_j\ impacts\ T_i)$ holds. If $s_{T_i} \xrightarrow{*} s_{T_j}$, we have $commit(T_i) \leq commit(T_j)$, therefore $\neg(T_j\ impacts\ T_i)$ is true. In both cases, **D2** holds.

We conclude $\forall T \in \mathcal{T} : \text{CT}_{ANSI\ SI}(T, e) \Rightarrow D1 \wedge D2$. This completes the proof. □

## D.3 Prefix-consistent Snapshot Isolation

We now prove **Theorem 9** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{Session\ SI}(T, e) \equiv PC\text{-}SI$ i.e. $\text{C-ORD}(T_{s_p}, T) \wedge \exists s \in S_e : \text{COMPLETE}_{e,T}(s) \wedge \text{NO-CONF}_T(s) \wedge (\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s) \equiv P1 \wedge P2$.

*Proof.* **(⇐)We first prove** $P1 \wedge P2 \Rightarrow \exists e : \forall T \in \mathcal{T} : \text{CT}_{Session\ SI}(e, T)$.

**Commit Test** We can construct an execution $e$ such that every committed transaction satisfies the commit test $\text{CT}_{Session\ SI}(e, T)$. By definition, we assign distinct timestamps to all operations (including start, commit operations). Let $i_1, ...i_n$ be a permutation of $1, 2, ..., n$ such that committed transactions $T_{i_1}, ..., T_{i_n}$ are totally ordered by their commit time. We construct an execution $e$ according to the topological order defined above: $e : s_0 \rightarrow s_{T_{i_1}} \rightarrow s_{T_{i_2}} \rightarrow ... \rightarrow s_{T_{i_n}}$ and show that $\forall T_{i_j} \in \mathcal{T} : \text{CT}_{Session\ SI}(e\ , T_{i_j})$. Specifically, we prove the following: consider the largest $k$ such that $commit(T_{i_k}) < snapshot(T_{i_j})$, $P1 \wedge P2 \Rightarrow \text{C-ORD}(T_{s_p(T_j)}, T_j) \wedge \text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_k}}) \wedge \text{NO-CONF}_{T_{i_j}}(s_{T_{i_k}}) \wedge (\forall T' \xrightarrow{se} t : s_{T'} \xrightarrow{*} s_{T_{i_k}})$.

Note that since PC-SI rules are strictly stronger than GSI rules and we construct the execution the same as the proof in §D.2, the proof of $\text{C-ORD}(T_{s_p(T_j)}, T_j) \wedge \text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_k}}) \wedge \text{NO-CONF}_{T_{i_j}}(s_{T_{i_k}})$ is identical to the proof in §D.2. We therefore simply prove that $\forall t' \xrightarrow{se} t : s_{t'} \xrightarrow{*} s_{T_{i_k}}$. Consider any $T_{i_m} \xrightarrow{se} T_{i_j}$, i.e. $T_{i_m} \sim T_{i_j}$ and $T_{i_m}.commit < T_{i_j}.start$. By **D4**, we have that $commit(T_{i_m}) < snapshot(T_{i_j})$. Since $T_{i_k}$ is the largest transaction whose $commit(T_{i_k}) < snapshot(T_{i_j})$, we have

$commit(T_{i_m}) \leq commit(T_{i_k})$. By the execution construction, we have $s_{T_{i_m}} \xrightarrow{*} s_{T_{i_k}}$.

**(⇒)We next prove** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{Session\ SI}(e, T) \Rightarrow P1 \wedge P2$. Let $e$ be an execution such that $\forall T \in \mathcal{T} : \text{CT}_{GSI}(T, e)$, and $H$ be a history for committed transactions $\mathcal{T}$. Since $e$ satisfies $\text{C-ORD}(T_{s_p(T)}, T)$, the order of transactions in $e$ is the same as ordering by time, i.e. $s_T \xrightarrow{*} s_{T'} \equiv T.commit < T'.commit$. Now we assign a snapshot time to transactions. For any $T_i$, let $s_{T_k}$ be the state such that $\text{COMPLETE}_{e,T_i}(s_{T_k}) \wedge \text{NO-CONF}_{T_i}(s_{T_k}) \wedge (T_k <_s T_i)$ (by $\text{CT}_{GSI}(T, e)$), then $snapshot(T_i) = commit(T_k) + \epsilon$, where $\epsilon$ is a small constant that is smaller than a time unit; otherwise, $snapshot(T_i) = 0$. The snapshot time assigned satisfies $snapshot(T) \leq start(T)$: since $T_k <_s T_i$, we have $commit(T_k) < start(T_i)$, therefore $snapshot(T_i) = commit(T_k) + \epsilon < start(T_i)$. **P1-1, P1-2, P1-3** holds by an identical proof to proving **D1-1, D1-2, D1-3** of Theorem 8(b)(⇒). Now we prove **P1-4**. If $T_i \sim T_j$ and $commit(T_j) < start(T_i)$, we have $T_j \xrightarrow{se} T_i$, therefore by $(\forall t' \xrightarrow{se} t : s_{t'} \xrightarrow{*} s_{T_{i_k}})$, we have $s_{T_j} \xrightarrow{*} s_{T_k}$. By $s_t \xrightarrow{*} s_{t'} \equiv t.commit < t'.commit$, we have $commit(T_j) < commit(T_k) < snapshot(T_i)$. Combining all previous results, we conclude that **P1** is satisfied

**P2** Now, we prove that **P2** is satisfied. Consider any $T_j$, there can only be two cases: $writeset(T_i) \cap writeset(T_j) = \emptyset$ (1) and $writeset(T_i) \cap writeset(T_j) \neq \emptyset$ (2). The first case trivially satisfies $\neg(T_j\ impacts\ T_i)$. In the second case, as $\text{NO-CONF}_{T_i}(s_{T_k})$ holds, we have either $s_{T_j} \xrightarrow{*} s_{T_k}$ or $s_{T_i} \xrightarrow{*} s_{T_j}$. If $s_{T_j} \xrightarrow{*} s_{T_k}$, $commit(T_j) \leq commit(T_k) < snapshot(T_i)$, and hence that $\neg(T_j\ impacts\ T_i)$ holds. If $s_{T_i} \xrightarrow{*} s_{T_j}$, we have $commit(T_i) \leq commit(T_j)$, therefore $\neg(T_j\ impacts\ T_i)$ is true. In both cases, **P2** holds.

We conclude $\forall T \in \mathcal{T} : \text{CT}_{Session\ SI}(T, e) \Rightarrow P1 \wedge P2$. This completes the proof. □

# E Equivalence to PL-2+ and PSI

In this section, we prove that our state-based definition of PSI is equivalent to the axiomatic formulation of PSI ($PSI_A$) by Cerone et al. [46] and to the cycle-based specification of PL-2+. Specifically, we prove the following theorems:

**Theorem 10 (a)** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T, e) \equiv \neg G1 \wedge \neg G\text{-single}$.

**Theorem 10 (b)** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T, e) \equiv PSI_A$.

Before beginning, we first prove a useful lemma: if an execution $e$, written $s_0 \rightarrow s_{T_1} \rightarrow s_{T_2} \rightarrow \cdots \rightarrow s_{T_n}$ satisfies the predicate $\textsc{preread}_e(\mathcal{T})$, then any transaction $T$ that depends on a transaction $T'$ ($T \in \textsc{prec}_e(T')$) will always commit after $T'$ and all its dependents in the execution. We do so in two steps: we first prove that $T$ will commit after the transactions that it directly reads from (Lemma 2), and then extend that result to all the transaction's transitive dependencies (Lemma 3). Formally

**Lemma 2.** $\textit{PREREAD}_e(\mathcal{T}) \Rightarrow \forall \hat{T} \in \mathcal{T} : \forall T \in \textit{D-PREC}_e(\hat{T}), s_T \xrightarrow{+} s_{\hat{T}}$

*Proof.* Consider any $\hat{T} \in \mathcal{T}$ and any $T \in \textsc{d-prec}_e(\hat{T})$. $T$ is included in $\textsc{d-prec}_e(\hat{T})$ if one of two cases hold: if $\exists o \in \Sigma_{\hat{T}}, T = T_{sf_o}$ ($\hat{T}$ reads the value created by $T$) or $s_T \xrightarrow{+} s_{\hat{T}} \wedge \mathcal{W}_{\hat{T}} \cap \mathcal{W}_T \neq \emptyset$ (t and $\hat{T}$ write the same objects and $T$ commits before $\hat{T}$).

1. $T \in \{T | \exists o \in \Sigma_{\hat{T}} : T = T_{sf_o}\}$ Let $o_i$ be the operation such that $T = T_{sf_{o_i}}$. By assumption, we have $\textsc{preread}_e(\mathcal{T})$. It follows that $\forall o, sf_o \xrightarrow{+} s_{\hat{T}}$. and consequently that $sf_{o_i} \xrightarrow{+} s_{\hat{T}}$ and $s_T \xrightarrow{+} s_{\hat{T}}$.
2. $T \in \{T | s_T \xrightarrow{+} s_{\hat{T}} \wedge \mathcal{W}_{\hat{T}} \cap \mathcal{W}_T \neq \emptyset\}$, trivially we have $s_T \xrightarrow{+} s_{\hat{T}}$.

□

We now generalize the result to hold transitively.

**Lemma 3.** $\textit{PREREAD}_e(\mathcal{T}) \Rightarrow \forall T' \in \textit{PREC}_e(T) : s_{T'} \xrightarrow{+} s_T$.

*Proof.* We prove this implication by induction.

**Base Case** Consider the first transaction $T_1$ in the execution. We want to prove that for all transactions $T$ that precede $T_1$ in the execution $s_T \xrightarrow{*} s_{T_1} : \forall T' \in \text{PREC}_e(T) : s_{T'} \xrightarrow{*} s_T$. As $T_1$ is the first transaction in the execution, $\text{D-PREC}_e(T_1) = \emptyset$ and consequently $\text{PREC}_e(T) = \emptyset$. We see this by contradiction: assume there exists a transaction
$T \in \text{D-PREC}_e(T_1)$, by implication $s_T \xrightarrow{+} s_{T_1}$ (Lemma 2), which violates our assumption that $T_1$ is the first transaction in the execution. Hence the desired result trivially holds.

**Induction Step** Consider the $i$-th transaction in the execution. We assume that $\forall T$ s.t. $s_T \xrightarrow{*} s_i$ the property $\forall T' \in \text{PREC}_e(T) : s_{T'} \xrightarrow{*} s_T$ holds. In other words, we assume that the property holds for the first $i$ transactions. We now prove that the property holds for the first $i+1$ transactions, specifically, we show that $\forall T' \in \text{PREC}_e(T_{i+1}) : s_{T'} \xrightarrow{*} s_{T_{i+1}}$. A transaction $T'$ belongs to $\text{PREC}_e(T_{i+1})$ if one of two conditions holds: either $T' \in \text{D-PREC}_e(T_{i+1})$, or $\exists t_k \in \mathcal{T} : t' \in \text{PREC}_e(T_k) \wedge t_k \in \text{D-PREC}_e(T_{i+1})$. We consider each in turn:

- If $T' \in \text{D-PREC}_e(T_{i+1})$: by Lemma 2, we have $s_{T'} \xrightarrow{+} s_{T_{i+1}}$.
- If $\exists t_k \in \text{D-PREC}_e(T_{i+1}) : T' \in \text{PREC}_e(T_k)$: As $T_k \in \text{D-PREC}_e(T_{i+1})$, by Lemma 2, we have $s_{T_k} \xrightarrow{+} s_{T_{i+1}}$, i.e. $s_{T_k} \xrightarrow{*} s_{T_i}$ ($s_{T_i}$ directly precedes $s_{T_{i+1}}$ in $e$ by construction). The induction hypothesis holds for every transaction that strictly precedes $T_{i+1}$ in $e$, hence $\forall t_{k'} \in \text{PREC}_e(T_k) : s_{T_{k'}} \xrightarrow{+} s_{T_k}$. As $T' \in \text{PREC}_e(T_k)$ by construction, it follows that $s_{T'} \xrightarrow{+} s_{T_k}$. Putting everything together, we have $s_{T'} \xrightarrow{+} s_{T_k} \xrightarrow{+} s_{T_{i+1}}$, and consequently $s_{T'} \xrightarrow{+} s_{T_{i+1}}$. This completes the induction step of the proof.

Combining the base case, and induction step, we conclude: $\text{PREREAD}_e(\mathcal{T}) \Rightarrow \forall T' \in \text{PREC}_e(T) : s_{T'} \xrightarrow{+} s_T$. □

## E.1 Cerone et al. [46]'s model summary

We note that this axiomatic specification, defined by Cerone et al. [46, 48] is proven to be equivalent to the operational specification of Sovran et al. [178], modulo an additional assumption: that each replica executes each transaction sequentially. The authors state that this is for syntactic elegance only, and does not change the essence of the proof. We provide a brief summary and explanation of the main terminology introduced in Cerone et al.'s framework. We refer the reader to [46] for the full set of definitions. The authors consider a database storing a set of objects $Obj = \{x, y, ...\}$, with operations $Op = \{read(x, n), write(x, n)|x \in Obj, n \in \mathbb{Z}\}$. For simplicity, the authors assume the value space to be $\mathbb{Z}$.

**Definition 17** *History events are tuples of the form $(\iota, op)$, where $\iota$ is an identifier from a countably infinite set EventId and $op \in Op$. Let $WEvent_x = \{(\iota, write(x, n))|\iota \in EventId, n \in \mathbb{Z}\}$, $REvent_x = \{(\iota, read(x))|\iota \in EventId, n \in \mathbb{Z}\}$, and $HEvent_x = REvent_x \cap WEvent_x$.*

**Definition 18** *A transaction $T$ is a pair $(E, po)$, where $E \subseteq HEvent$ is an non-empty set of events with distinct identifiers, and the program order $po$ is a total order over $E$. A history $\mathcal{H}$ is a set of transactions with disjoint sets of event identifiers.*

**Definition 19** *An abstract execution is a triple $A = (\mathcal{H}, VIS, AR)$ where visibility $VIS \subseteq \mathcal{H} \times \mathcal{H}$ is an acyclic relation; and arbitration $AR \subseteq \mathcal{H} \times \mathcal{H}$ is a total order such that $AR \supseteq VIS$.*

For simplicity, we summarize the model's main notation specificities:

- **_** Denotes a value that is irrelevant and implicitly existentially quantified.
- $\boldsymbol{\max_R(A)}$ Given a total order $R$ and a set $A$, $\max_R(A)$ is the element $u \in A$ such that $\forall v \in A. v = u \vee (v, u) \in R$.
- $\boldsymbol{R_{-1}(u)}$ For a relation $R \subseteq A \times A$ and an element $u \in A$, we let $R_{-1}(u) = \{v|(v, u) \in R\}$.
- $\boldsymbol{T \vdash Write\ x : n}$ $T$ writes to $x$ and the last value written is $n$: $\max_{po}(E \cap WEvent_x) = (\_, write(x, n))$.
- $\boldsymbol{T \vdash Read\ x : n}$ $T$ makes an external read from $x$, i.e., one before writing to $x$, and $n$ is the value returned by the first such read: $\min_{po}(E \cap HEvent_x) = (\_, read(x, n))$.

A consistency model specification is a set of consistency axioms $\Phi$ constraining executions. The model allow those histories for which there exists an execution that satisfies the axioms.

**Definition 20** $Hist_\Phi = \{\mathcal{H}|\exists Vis, AR.(\mathcal{H}, AR) \models \Phi\}$

The authors define the axioms in Table 1. $PSI_A$ is then defined with the following set of consistency axioms.

**Definition 21** *$PSI_A$ allows histories for which there exists an execution that satisfies INT,*

| | |
|---|---|
| INT | $\forall(E, po) \in \mathcal{H}.\forall event \in E.\forall x, n.(event = (\_, read(x, n)) \wedge (po^{-1}(event) \cap \mathit{HEvent}_x \neq \emptyset)) \Rightarrow \max_{po}(po^{-1}(event$ |
| EXT | $\forall T \in \mathcal{H}.\forall x, n.T \vdash Read\ x : n \Rightarrow ((VIS^{-1}(T) \cap \{S \mid S \vdash Write\ x : \_\} = \emptyset \wedge n = 0) \vee \max_{AR}((VIS^{-1}(T) \cap \{S \mid S \vdash$ |
| TRANSVIS | VIS is transitive |
| NOCONFLICT | $\forall T, S \in \mathcal{H}.(T \neq S \wedge T \vdash Write\ x : \_ \wedge S \vdash Write\ x : \_) \Rightarrow (T \xrightarrow{VIS} S \vee S \xrightarrow{VIS} T)$ |

*Table 1: PSI Axioms*

*EXT, TRANSVIS and NOCONFLICT:* $Hist_{PSI} = \{H \mid \exists VIS, AR.(\mathcal{H}, VIS, AR) \models$ *INT, EXT, TRANSVIS, NOCONFLICT}.*

## E.2 PL-2+

Before beginning, we first prove a useful lemma. Let us consider a history $H$ that contains the same set of transactions $\mathcal{T}$ as an execution $e$. The version order for $H$ , denoted as $<<$, is instantiated as follows: given an execution $e$ and an object $x$, $x_i << x_j$ if and only if $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \wedge s_{T_i} \xrightarrow{*} s_{T_j}$. We show that, if a transaction $T'$ is in the depend set of a transaction $T$ ($T' \in \text{PREC}_e(T)$), then there exists a path of write-read/write-write dependencies from $T'$ to $T$ in the $DSG(H)$. Formally:

**Lemma 4.** *PREREAD*$_e(\mathcal{T}) \Rightarrow \forall T' \in$ *PREC*$_e(T) : T' \xrightarrow{ww/wr}^{+} T$ *in* $DSG(H)$*.*

*Proof.* We improve this implication by induction.

**Base Case** Consider the first transaction $T_1$ in the execution. We want to prove that for all transactions $T$ that precede $T_1$ in the execution $\forall T \in \mathcal{T}$ such that $s_T \xrightarrow{*} s_{T_1}$, the following holds: $\forall T' \in \text{PREC}_e(T) : T' \xrightarrow{ww/wr}^{+} T$ in $DSG(H)$. As $T_1$ is the first transaction in the execution, $\text{D-PREC}_e(T_1) = \emptyset$ and consequently $\text{PREC}_e(T) = \emptyset$. We see this by contradiction: assume there exists a transaction $T \in \text{D-PREC}_e(T_1)$, by implication $s_T \xrightarrow{+} s_{T_1}$ (Lemma 2), violating our assumption that $T_1$ is the first transaction in the execution. ence the implication trivially holds.

**Induction Step** Consider the $i$-th transaction in the execution. We assume that $\forall T$, s.t. $s_T \xrightarrow{*} s_{T_i}$, $\forall T' \in \text{PREC}_e(T) : T' \xrightarrow{ww/wr}^{+} T$. In other words, we assume that the property holds for the first $i$ transactions. We now prove that the property holds for the first $i+1$ transactions, specifically, we show that $\forall T' \in \text{PREC}_e(T_{i+1}) : T' \xrightarrow{ww/wr}^{+} T_{i+1}$. A transaction $T'$ belongs to $\text{PREC}_e(T_{i+1})$ if one of two conditions holds: either $T' \in \text{D-PREC}_e(T_{i+1})$, or $\exists T_k \in \mathcal{T} : T' \in \text{PREC}_e(T_k) \wedge T_k \in \text{D-PREC}_e(T_{i+1})$. We consider each in turn:

- If $T' \in \text{D-PREC}_e(T_{i+1})$: There are two cases: $T' \in \{T|\exists o \in \Sigma_{T_{i+1}} : t = T_{sf_o}\}$ or, $T' \in \{T|s_T \xrightarrow{+} s_{T_{i+1}} \wedge \mathcal{W}_{T_{i+1}} \cap \mathcal{W}_T \neq \emptyset\}$. If $T' \in \{T|\exists o \in \Sigma_{T_{i+1}} : t = T_{sf_o}\}$, $T_{i+1}$ reads the version of an object that $T'$ wrote, hence $T_{i+1}$ read-depends on $T'$, i.e. $T' \xrightarrow{wr} T$.

  If $T' \in \{T|s_T \xrightarrow{+} s_{T_{i+1}} \wedge \mathcal{W}_{T_{i+1}} \cap \mathcal{W}_T \neq \emptyset\}$: trivially, $s_{T'} \xrightarrow{+} s_{T_{i+1}}$. Let $x$ be the key that is written by $T$ and $T_{i+1}$: $x \in \mathcal{W}_{T_{i+1}} \cap \mathcal{W}_T$. By construction, the history $H$'s version order for $x$ is $x_{T'} << x_{T_{i+1}}$. By definition of version order, there must therefore a chain of $ww$ edges between $T'$ and $T_{i+1}$ in $DSG(H)$, where all of the transactions in the chain write the next version of $x$. Thus: $T' \xrightarrow{ww}^{+} T_{i+1}$ holds.

- If $\exists T_k : T' \in \text{PREC}_e(T_k) \wedge T_k \in \text{D-PREC}_e(T_{i+1})$. As $T_k \in \text{D-PREC}_e(T_{i+1})$, we conclude , as above that $T_k \xrightarrow{ww/wr}^{+} T_{i+1}$. Moreover, by Lemma 2, we have $s_{T_k} \xrightarrow{+} s_{T_{i+1}}$, i.e. $s_{T_k} \xrightarrow{*} s_{T_i}$ ($s_{T_i}$ directly precedes $s_{T_{i+1}}$ in $e$ by construction). The induction hypothesis holds for every transaction that precedes $T_{i+1}$ in $e$, hence $\forall T_{k'} \in \text{PREC}_e(T_k)$: $T_{k'} \xrightarrow{ww/wr}^{+} T_k$. Noting $T' \in \text{PREC}_e(T_k)$, we see that $T' \xrightarrow{ww/wr}^{+} T_k$. Putting everything together, we obtain $T' \xrightarrow{ww/wr}^{+} T_k \xrightarrow{ww/wr}^{+} T_{i+1}$, i.e. $T' \xrightarrow{ww/wr}^{+} T_{i+1}$ by transitivity.

Combining the base case, and induction step, we conclude: $\forall t : \forall T' \in \text{PREC}_e(T) : T' \xrightarrow{ww/wr}^{+} T$. □

**Now, we prove Theorem 10 (a)** Let $\mathcal{I}$ be PSI. Then $\exists e : \forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e) \equiv \neg\text{G1} \wedge \neg\text{G-Single}$

*Proof.* Let us recall the definition of PSI's commit test: $\text{PREREAD}_e(\mathcal{T}) \wedge \forall o \in \Sigma_T : \forall T' \in \text{PREC}_e(T) : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$

($\Rightarrow$) **First we prove** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e) \Rightarrow \neg\text{G1} \wedge \neg\text{G-Single}$. Let *e* be an execution that $\forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e)$, and *H* be a history for committed transactions $\mathcal{T}$. We first instantiate the version order for *H*, denoted as $<<$, as follows: given an execution $e$ and an object $x$, $x_i << x_j$ if and only if $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \wedge s_{T_i} \xrightarrow{*} s_{T_j}$. It follows that, for any two states such that $(x, x_i) \in T_m \wedge (x, x_j) \in T_n \Rightarrow s_{T_m} \xrightarrow{+} s_{T_n}$.

**G1** We next prove that $\forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e) \Rightarrow \neg\text{G1}$:

**G1a** Let us assume that $H$ exhibits phenomenon G1a (aborted reads). There must exist events $w_i(x_i), r_j(x_i)$ in H such that $T_i$ subsequently aborted. $\mathcal{T}$ and any corresponding execution $e$, however, consists only of committed transactions. Hence $\forall e : \nexists s \in \mathcal{S}_e, s.t.\ s \in \mathcal{RS}_e(r_j(x_i))$: i.e. $\neg$PREREAD$_e(T_j)$, therefore $\neg$PREREAD$_e(\mathcal{T})$. There thus exists a transaction for which the commit test cannot be satisfied, for any $e$. We have a contradiction.

**G1b** Let us assume that $H$ exhibits phenomenon G1b (intermediate reads). In an execution $e$, only the final writes of a transaction are applied. Hence,$\forall e : \nexists s \in \mathcal{S}_e, s.t.\ s \in \mathcal{RS}_e(r(x_{intermediate}))$, i.e. $\neg$PREREAD$_e(T)$, therefore $\neg$PREREAD$_e(\mathcal{T})$. There thus exists a transaction $T$, which for all $e$, will not satisfy the commit test. We once again have a contradiction.

**G1c** Finally, let us assume that $H$ exhibits phenomenon G1c: $DSG(H)$ must contain a cycle of read/write dependencies. We consider each possible edge in the cycle in turn:

- $T_i \xrightarrow{ww} T_j$ There must exist an object $x$ such that $x_i << x_j$ (version order). By construction, version order in $H$ is consistent with the execution order $e$: we have $s_{T_i} \xrightarrow{*} s_{T_j}$.
- $T_i \xrightarrow{wr} T_j$ There must exist a read $o = r_j(x_i) \in \Sigma_{T_j}$ such that $T_j$ reads version $x_i$ written by $T_i$. By assumption, CT$_{PSI}(e, T_j)$ holds. By PREREAD$_e(\mathcal{T})$, we have $sf_o \xrightarrow{+} s_{T_j}$; and since $sf_o$ exists, $sf_o = s_{T_i}$. It follows that $s_{T_i} \xrightarrow{+} s_{T_j}$.

If a history $H$ displays phenomena G1c there must exist a chain of transactions $T_i \rightarrow T_{i+1} \rightarrow ... \rightarrow T_j$ such that $i = j$. A corresponding cycle must thus exist in the execution $e$: $s_{T_i} \xrightarrow{*} s_{T_{i+1}} \xrightarrow{*} \ldots \xrightarrow{*} s_{T_j}$. By definition however, a valid execution must be totally ordered. We once again have a contradiction.

**G-Single** We now prove that $\forall T \in \mathcal{T}$ : CT$_{PSI}(T, e) \Rightarrow \neg$G-Single

By way of contradiction, let us assume that $H$ exhibits phenomenon G-Single: $DSG(H)$ must contain a directed cycle with exactly one anti-dependency edge. Let $T_1 \xrightarrow{ww/wr} T_2 \xrightarrow{ww/wr} \ldots \xrightarrow{ww/wr} T_k \xrightarrow{rw} T_1$ be the cycle in $DSG(H)$. We first prove by induction that $T_1 \in$ PREC$_e(T_k)$, where $T_k$ denotes the $k - th$ transaction that succedes $T_1$. We then show that there exist a $T' \in$ PREC$_e(T_k)$ such that $o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$ does not hold.

**Base case** We prove that $T_1 \in$ PREC$_e(T_2)$. We distinguish between two cases $T_1 \xrightarrow{ww} T_2$, and

$T_1 \xrightarrow{wr} T_2$.

- If $T_1 \xrightarrow{ww} T_2$, there must exist an object $k$ that $T_1$ and $T_2$ both write: $k \in \mathcal{W}_{T_1}$ and $k \in \mathcal{W}_{T_2}$, therefore $\mathcal{W}_{T_1} \cap \mathcal{W}_{T_2} \neq \emptyset$. By construction, $T_i \xrightarrow{ww} T_j \Leftrightarrow s_{T_i} \xrightarrow{*} s_{T_j}$. Hence we have $s_{T_1} \xrightarrow{*} s_{T_2}$. By definition of D-PREC$_e(T)$, it follows that $T_1 \in$ D-PREC$_e(T_2)$.
- If $T_1 \xrightarrow{wr} T_2$, there must exist an object $k$ such that $T_2$ reads the version of the object created by transaction $T_1$: $o = r(k_1)$. We previously proved that $T_i \xrightarrow{wr} T_j \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$. It follows that $s_{T_1} \xrightarrow{+} s_{T_2}$ and $sf_o = s_{T_1}$, i.e. $T_1 = T_{sf_o}$. By definition, $T_1 \in$ D-PREC$_e(T_2)$.

Since D-PREC$_e(T_2) \subseteq$ PREC$_e(T_2)$, it follows that $T_1 \in$ PREC$_e(T_2)$.

**Induction step** Assume $T_1 \in$ PREC$_e(T_i)$, we prove that $T_1 \in$ PREC$_e(T_{i+1})$. To do so, we first prove that $T_i \in$ D-PREC$_e(T_{i+1})$. We distinguish between two cases: $T_i \xrightarrow{ww} T_{i+1}$, and $T_i \xrightarrow{wr} T_{i+1}$.

- If $T_i \xrightarrow{ww} T_{i+1}$, there must exist an object $k$ that $T_i$ and $T_{i+1}$ both write: $k \in \mathcal{W}_{T_i}$ and $k \in \mathcal{W}_{T_{i+1}}$, therefore $\mathcal{W}_{T_i} \cap \mathcal{W}_{T_{i+1}} \neq \emptyset$. By construction, $T_i \xrightarrow{ww} T_j \Leftrightarrow s_{T_i} \xrightarrow{*} s_{T_j}$. Hence we have $s_{T_i} \xrightarrow{*} s_{T_{i+1}}$. By definition of D-PREC$_e(T)$, it follows that $T_i \in$ D-PREC$_e(T_{i+1})$.
- If $T_i \xrightarrow{wr} T_{i+1}$, there must exist an object $k$ such that $T_{i+1}$ reads the version of the object created by transaction $T_i$: $o = r(k_i)$. We previously proved that $T_i \xrightarrow{wr} T_j \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$. It follows that $s_{T_i} \xrightarrow{+} s_{T_{i+1}}$ and $sf_o = s_{T_i}$, i.e. $T_i = T_{sf_o}$. By definition, $T_i \in$ D-PREC$_e(T_{i+1})$.

Hence, $T_i \in$ D-PREC$_e(T_{i+1})$. The depends set includes the depend set of every transaction that it directly depends on: consequently PREC$_e(T_i) \subseteq$ PREC$_e(T_{i+1})$. We conclude: $T_1 \in$ PREC$_e(T_{i+1})$. Combining the base step and the induction step, we have proved that $T_1 \in$ PREC$_e(T_k)$.

We now derive a contradiction. Consider the edge $T_k \xrightarrow{rw} T_1$ in the G-Single cycle: $T_k$ reads the version of an object $x$ that precedes the version written by $T_1$. Specifically, there exists a version $x_m$ written by transaction $T_m$ such that $r_k(x_m) \in \Sigma_{T_k}$, $w_1(x_1) \in \Sigma_{T_1}$ and $x_m << x_1$. By definition of the PSI commit test for transaction $T_k$, if $T_1 \in$ PREC$_e(T_k)$ and $T_1$'s write set intersect with $T_k$'s read set, then $s_{T_1} \xrightarrow{*} sl_{r_k(x_m)}$. However, from $x_m << x_1$, we have $\forall s, s', s.t.(x, x_m) \in s \wedge (x, x_1) \in s' \Rightarrow s \xrightarrow{+} s'$. Since $(x, x_m) \in sl_{r_k(x_m)} \wedge (x, x_1) \in s_{T_1}$, we have $sl_{r_k(x_m)} \xrightarrow{+} s_{T_1}$. But, we previously proved that T $s_{T_1} \xrightarrow{*} sl_{r_k(x_m)}$. We have a contradiction: $H$ does not exhibit phenomenon G-Single, i.e. $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T, e) \Rightarrow \neg G1 \wedge \neg$G-Single.

($\Leftarrow$) We now prove the other direction $\neg$G1 $\wedge$ $\neg$G-Single $\Rightarrow \exists e : \forall T \in \mathcal{T} :$ CT$_{PSI}(T, e)$. We construct $e$ as follows: Consider only dependency edges in the DSG(H), by $\neg G1$, there exist no cycle consisting of only dependency edges, therefore the transactions can be topologically sorted respecting only dependency edges. Let $i_1, ...i_n$ be a permutation of $1, 2, ..., n$ such that $T_{i_1}, ..., T_{i_n}$ is a topological sort of DSG(H) with only dependency edges. We construct an execution $e$ according to the topological order defined above: $e : s_0 \to s_{T_{i_1}} \to s_{T_{i_2}} \to ... \to s_{T_{i_n}}$.

First we show that PREREAD$_e(\mathcal{T})$ is true: consider any transaction $T$, for any operation $o \in \Sigma_T$. If $o$ is a internal read operation or $o$ is a write operation, by definition $s_0 \in \mathcal{RS}_e(o)$ hence $\mathcal{RS}_e(o) \neq \emptyset$ follows trivially. Consider the case now where $o$ is a read operation that reads a value written by another transaction $T'$. Since the topological order includes $wr$ edges and $e$ respects the topological order, $T' \xrightarrow{wr} T$ in $DSG(H)$ implies $s_{T'} \xrightarrow{*} s_T$, then for any $o = r(x, x_{T'}) \in \Sigma_T$, $s_{T'} \in \mathcal{RS}_e(o)$. It follows that $\mathcal{RS}_e(o) \neq \emptyset$ and PREREAD$_e(T)$ is true. In conclusion: PREREAD$_e(\mathcal{T})$ holds.

Next, we prove that $\forall o \in \Sigma_T : \forall T' \in$ PREC$_e(T) : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$ holds. For any $T' \in$ PREC$_e(T)$, by Lemma 3, $s_{T'} \xrightarrow{+} s_T$. Consider any $o \in \Sigma_T$, let $T'$ be a transaction such that $T' \in$ PREC$_e(T) \wedge o.k \in \mathcal{W}_{T'}$, we now prove that $s_{T'} \xrightarrow{*} sl_o$. Consider the three possible types of operations in $T$:

1. *External Reads*: an operation reads an object version that was created by another transaction.
2. *Internal Reads*: an operation reads an object version that itself created.
3. *Writes*: an operation creates a new object version.

We show that $s_{T'} \xrightarrow{*} sl_o$ for each of those operation types:

1. *External Reads*. Let $o = r(x, x_{\hat{T}}) \in \Sigma_T$ read the version for $x$ created by $\hat{T}$, where $\hat{T} \neq T$. Since PREREAD$_e(T)$ is true, we have $\mathcal{RS}_e(o) \neq \emptyset$, therefore $s_{\hat{T}} \xrightarrow{+} s_T$ and $\hat{T} = T_{sf_o}$. From $\hat{T} = T_{sf_o}$, we have $\hat{T} \in$ D-PREC$_e(T)$. Now consider $T'$ and $\hat{T}$, we have that $s_{T'} \xrightarrow{+} s_T$ and $s_{\hat{T}} \xrightarrow{+} s_T$. There are two cases:
   - $s_{T'} \xrightarrow{*} s_{\hat{T}}$: Consequently $s_{T'} \xrightarrow{*} s_{\hat{T}} = sf_o \xrightarrow{*} sl_o$ It follows that $s_{T'} \xrightarrow{*} sl_o$.
   - $s_{\hat{T}} \xrightarrow{+} s_{T'}$: We prove that this cannot happen by contradiction. Since $o.k \in \mathcal{W}_{T'}$, $T'$ also writes

key $x_{T'}$. By construction, , $s_{\hat{T}} \xrightarrow{+} s_{T'}$ in $e$ implies $x_{\hat{T}} << x_{T'}$. There must consequently exist a chain of $ww$ edges between $\hat{T}$ and $T'$ in $DSG(H)$, where all the transactions on the chain writes a new version of key $x$. Now consider the transaction in the chain directly after to $\hat{T}$, denoted as $\hat{T}_{+1}$, where $\hat{T} \xrightarrow{ww} \hat{T}_{+1} \xrightarrow{ww}^{*} T'$. $\hat{T}_{+1}$ overwrites the version of $x$ $T$ reads. Consequently, $T$ directly anti-depends on $\hat{T}_{+1}$, i.e. $T \xrightarrow{rw} \hat{T}_{+1}$. Moreover $T' \in \text{PREC}_e(T)$, by Lemma 4, we have $T' \xrightarrow{ww/wr}^{+} T$. There thus exists a cycle consists of only one anti dependency edges as $T \xrightarrow{rw} \hat{T}_{+1} \xrightarrow{ww}^{*} T' \xrightarrow{ww/wr}^{+} T$, in contradiction with G-Single. $s_{T'} \xrightarrow{*} s_{\hat{T}}$ holds.

$s_{T'} \xrightarrow{*} s_{\hat{T}}$ holds in all cases. Noting that $s_{\hat{T}} = sl_o$, we conclude $s_{T'} \xrightarrow{*} sl_o$.

2. *Internal Reads*. Let $o = r(x, x_T)$ read $x_T$ such that $w(x, x_T) \xrightarrow{to} r(x, x_T)$. By definition of $\mathcal{RS}_e(o)$, we have $sl_o = s_p$. Since we have proved that $s_{T'} \xrightarrow{+} s_T$, therefore we have $s_{T'} \xrightarrow{*} s_p = sl_o$ (as $s_p \rightarrow s_T$).

3. *Writes*. Let $o = w(x, x_T)$ be a write operation. By definition of $\mathcal{RS}_e(o)$, we have $sl_o = s_p$. We previously proved that $s_{T'} \xrightarrow{+} s_T$. Consequently we have $s_{T'} \xrightarrow{*} s_p = sl_o$ (as $s_p \rightarrow s_T$).

We conclude that, in all cases, $\text{CT}_{PSI}(T, e) \equiv \text{PREREAD}_e(T) \wedge \forall o \in \Sigma_T : \forall T' \in \text{PREC}_e(T) : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$. □

## E.3 PSI

We now prove the following theorem:

**Theorem 10** **(b)** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e) \equiv PSI_A$.

We first relate Cerone et al.'s notion of transactions to transactions in our model: Cerone defines transactions as a tuple $(E, po)$ where $E$ is a set of events and $po$ is a program order over $E$. Our model similalry defines transactions as a tuple $(\Sigma_T, \xrightarrow{to})$, where $\Sigma_T$ is a set of operations, and $\xrightarrow{to}$ is the total order on $\Sigma_T$. These definitions are equivalent: events defined in Cerone are extensions of operations in our model (events include a unique identifier), while the partial order in Cerone maps to the program order in our model. Finally, we relate our notion of versions to Cerone's values.

($\Rightarrow$) **We first prove** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e) \Rightarrow PSI_A$.

**Construction** Let $e$ be an execution such that $\forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e)$. We construct $AR$ and

$VIS$ as follows: $AR$ is defined as $T_i \xrightarrow{AR} T_j \Leftrightarrow s_{T_i} \rightarrow s_{T_j}$ while $VIS$ order is defined as $T_i \xrightarrow{VIS} T_j \Leftrightarrow T_i \in \textsc{prec}_e(T_j)$. By definition, our execution is a total order, hence our constructed $AR$ is also a total order. $VIS$ defines an acyclic partial order that is a subset of $AR$ ( by $\textsc{preread}_e(\mathcal{T})$ and Lemma 3). We now prove that each consistency axiom holds:

**INT** $\forall(E, po) \in \mathcal{H}.\forall event \in E.\forall x, n.(event = (\_, read(x, n)) \wedge (po^{-1}(event) \cap \mathit{HEvent}_x \neq \emptyset)) \Rightarrow \max_{po}(po^{-1}(event) \cap \mathit{HEvent}_x) = (\_, \_(x, n))$ Intuitively, the consistency axiom $INT$ ensures that the read of an object returns the value of the transaction's last write to that object (if it exists). For any $(E, po) \in \mathcal{H}$, we consider any $event$ and $x$ such that $(event = (\_, read(x, n)) \wedge (po^{-1}(event) \cap \mathit{HEvent}_x \neq \emptyset))$. We prove that $\max_{po}(po^{-1}(event) \cap \mathit{HEvent}_x) = (\_, \_(x, n))$. By assumption, $(po^{-1}(event) \cap \mathit{HEvent}_x \neq \emptyset))$ holds, there must exist an event such that $\max_{po}(po^{-1}(event) \cap \mathit{HEvent}_x)$. This event is either a read operation, or a write operation:

1. If $op = \max_{po}(po^{-1}(event) \cap \mathit{HEvent}_x)$ is a write operation: given $event = (\_, read(x, n))$ and $op \xrightarrow{po} event$, the equivalent statement in our model is $w(x, v_{op}) \xrightarrow{to} r(x, n)$. By definition, our model enforces that $w(k, v') \xrightarrow{to} r(k, v) \Rightarrow v = v'$. Hence $v_{op} = n$, i.e. $op = (\_, write(x, n))$, therefore $op = (\_, \_(x, n))$. Hence $INT$ holds.

2. If $op = \max_{po}(po^{-1}(event) \cap \mathit{HEvent}_x)$ is a read operation, We write $op = (\_, read(x, v_{op}))$. The equivalent formulation in our model is as follows. For $event = (\_, read(x, n))$, we write $o_1 = r(x, n)$, and for $op$, we write $o_2 = r(x, v_{op})$ with $o_2 \xrightarrow{to} o_1$ where $o_1, o_2 \in \Sigma_T$. Now we consider the following two cases.

   First, let us assume that there exists an operation $w(k, v)$ such that $w(k, v) \xrightarrow{to} o_2 \xrightarrow{to} o_1$ (all three operations belong to the same transaction). Given that $\xrightarrow{to}$ is a total order, we have $w(k, v) \xrightarrow{to} o_1$ and $w(k, v) \xrightarrow{to} o_2$. It follows by definition of candidate read state that $w(k, v') \xrightarrow{to} r(k, v) \Rightarrow v = v'$, where $v = n \wedge v = v_{op}$, i.e. $v_{op} = n$. Hence $op = (\_, \_(x, n))$ and $INT$ holds. Second, let us next assume that there does not exist an operation $w(k, v) \xrightarrow{to} o_2 \xrightarrow{to} o_1$. We prove by contradiction that $v_{op} = n$ nonetheless. Assume that $v_{op} \neq n$, and consider transactions $T_1$ that writes $(x, n)$, and $T_2$ that writes $(x, v_{op})$, by $\textsc{preread}_e(\mathcal{T})$ , we know that $sf_{o_1}, sf_{o_2}$ exist. We have $T_1 = T_{sf_{o_1}}$ and $T_2 = T_{sf_{o_2}}$. By definition of $\textsc{prec}_e(T)$, we have $T_1, T_2 \in \textsc{d-prec}_e(T) \subseteq \textsc{prec}_e(T)$, i.e. $T_1, T_2 \in \textsc{prec}_e(T)$. We note that the sequence of states containing $(x, n)$ is disjoint from states containing $(x, v_{op})$: in otherwords, the sequence of states bounded by $sf_{o_1}$ and $sl_{o_1}$ and $sf_{o_2}$ and $sl_{o_2}$ are disjoint. Hence, we have either $s_{T_1} \xrightarrow{*} sl_{o_1} \xrightarrow{+} s_{T_2} \xrightarrow{*} sl_{o_2}$, or $s_{T_2} \xrightarrow{*} sl_{o_2} \xrightarrow{+} s_{T_1} \xrightarrow{*} sl_{o_1}$. Equivalently either $T_2 \in \textsc{prec}_e(T) \wedge o_1.k \in \mathcal{W}_{T_2} \wedge sl_{o_1} \xrightarrow{+} s_{T_2}$, or $T_1 \in \textsc{prec}_e(T) \wedge o_2.k \in \mathcal{W}_{T_1} \wedge sl_{o_2} \xrightarrow{+} s_{T_1}$. In both cases, this violates $\mathrm{CT}_{PSI}(T, e)$, a contradiction. We conclude $v_{op} = n$, i.e. $op = (\_, read(x, n))$, therefore $op = (\_, \_(x, n))$.

We proved that $\max_{po}(po^{-1}(event) \cap \mathit{HEvent}_x) = (\_, \_(x, n))$, hence $INT$ holds.

**EXT** We now prove that $EXT$ holds for $\mathcal{H}$. Specifically, $\forall T \in \mathcal{H}.\forall x, n.T \vdash Read\ x : n \Rightarrow ((VIS^{-1}(T) \cap \{S | S \vdash Write\ x : \_\} = \emptyset \wedge n = 0) \vee \max_{AR}((VIS^{-1}(T) \cap \{S | S \vdash Write\ x : \_\}) \vdash Write\ x : n)$

We proceed in two steps, we first show that there exist a transaction $T$ that wrote $(x, n)$, and next we show that $T$ is the most recent such transaction. Consider any $T \in \mathcal{H}.\forall x, n.T \vdash Read\ x : n$ (a external read). Equivalently, we consider a transaction $T$ in our model such that $r(x, n) \in \Sigma_T$. Let $T_n$ be the transaction that writes $(x, n)$. By assumption, $\textsc{preread}_e(\mathcal{T})$ holds hence $sf_o$ exists and $sf_o = s_{T_n}$, i.e. $T_n = T_{sf_o}$, as $T_n$ created the first state from which $o$ could read from. By definition of $\textsc{prec}_e(T)$, we have $T_n \in \textsc{d-prec}_e(T) \subseteq \textsc{prec}_e(T)$, i.e. $T_n \in \textsc{prec}_e(T)$. Moreover, we defined $VIS$ as $T_i \xrightarrow{VIS} T_j \Leftrightarrow T_i \in \textsc{prec}_e(T_j)$. Hence, we have $T_n \xrightarrow{VIS} T$, and consequently $T_n \in VIS^{-1}(T)$. Since $write(x, n) \in \Sigma_{T_n}$, $T_n \vdash Write\ x : n$.

Next, we show that $T_n$ is larger than any other transaction $T'$ in $AR$: $T' \xrightarrow{VIS} T \wedge T' \vdash Write\ x : \_$. Consider the equivalent transaction $T'$ in our model, we know that $T' \in \textsc{prec}_e(T)$ ($T' \xrightarrow{VIS} T$) and $x \in \mathcal{W}_{T'}$. As $o = r(x, n) \in \Sigma_T$ and $T' \in \textsc{prec}_e(T) \wedge o.k \in \mathcal{W}_{T'}$, $\mathrm{CT}_{PSI}(T, e)$ implies that $s_{T'} \xrightarrow{*} sl_o$. We note that the sequence of states containing $(x, n)$ is disjoint from states containing $(x, x_{T'})$. It follows that $s_{T'} \xrightarrow{*} sf_o = s_{T_n}$. We can strengthen this to say $s_{T'} \xrightarrow{+} sf_o = s_{T_n}$ as $T' \neq T_n$. By construction, we have $T' \xrightarrow{AR} T_n$, i.e. $T_n = \max_{AR}((VIS^{-1}(T) \cap \{S | S \vdash Write\ x : \_\})$. We conclude, $EXT$ holds.

**TRANSVIS** If $T_i \xrightarrow{VIS} T_j \wedge T_j \xrightarrow{VIS} T_k$, by construction we have $T_i \in \textsc{prec}_e(T_j) \wedge T_j \in \textsc{prec}_e(T_k)$. From $T_j \in \textsc{prec}_e(T_k)$, we know, since precede-set is maintained transitively, that $\textsc{prec}_e(T_j) \subseteq \textsc{prec}_e(T_k)$, and consequently that $T_i \in \textsc{prec}_e(T_k)$. By construction, we have $T_i \xrightarrow{VIS} T_k$., hence we conclude: $VIS$ is transitive.

**NOCONFLICT** Recall that this axiom is defined as: $\forall T, S \in \mathcal{H}.(T \neq S \wedge T \vdash \mathit{Write}\ x : \_ \wedge S \vdash \mathit{Write}\, x : \_) \Rightarrow (T \xrightarrow{VIS} S \vee S \xrightarrow{VIS} T)$ Consider any $T, S \in \mathcal{H}.(T \neq S \wedge T \vdash \mathit{Write}\ x : \_ \wedge S \vdash \mathit{Writex} : \_)$ and let $T_i, T_j$ be the equivalent transactions in our model such that $w(x, x_i) \in \Sigma_{T_i}$ and $w(x, x_j) \in \Sigma_{T_j}$ and consequently $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j}$. Since $e$ totally orders all the committed transactions, we have either $s_{T_i} \xrightarrow{+} s_{T_j}$ or $s_{T_j} \xrightarrow{+} s_{T_i}$. If $s_{T_i} \xrightarrow{+} s_{T_j}$, it follows from $s_{T_i} \xrightarrow{+} s_{T_j} \wedge \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \neq \emptyset$ that $T_i \in \text{D-PREC}_e(T_j) \subseteq \text{PREC}_e(T_j)$, i.e. $T_i \in \text{PREC}_e(T_j)$, and consequently $T \xrightarrow{VIS} S$. Similarly, if $s_{T_j} \xrightarrow{+} s_{T_i}$, it follows from $s_{T_j} \xrightarrow{+} s_{T_i} \wedge \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \neq \emptyset$ that $T_j \in \text{D-PREC}_e(T_i) \subseteq \text{PREC}_e(T_i)$, i.e. $T_j \in \text{PREC}_e(T_i)$, and consequently $S \xrightarrow{VIS} T$. We conclude: $T \xrightarrow{VIS} S \vee S \xrightarrow{VIS} T$, NOCONFLICT is true.

($\Leftarrow$) Now we prove that $PSI_A \Rightarrow \exists e : \forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e)$.

By assumption, AR is a total order over $\mathcal{T}$. We construct an execution $e$ by applying transactions in the same order as AR, i.e. $s_{T_i} \xrightarrow{+} s_{T_j} \Leftrightarrow T_i \xrightarrow{AR} T_j$ and subsequently prove that $e$ satisfies $\forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e)$.

**Preread** First we show that $\text{PREREAD}_e(\mathcal{T})$ is true: consider any transaction $T$, for any operation $o \in \Sigma_T$. If $o$ is a internal read operation or $o$ is a write operation, by definition $sf_o = s_0$ hence $sf_o \xrightarrow{*} s_T$ follows trivially. On the other hand, consider the case where $o$ is a read operation that reads a value written by another transaction $T'$: let $T$ and $T'$ be the corresponding transactions in Cerone's model. We have $T \vdash \mathit{Read}\, x : n$ and $T' \vdash \mathit{Write}\ x : n$. Assuming that values are uniquely identifiable, we have $T' = \max_{AR}(VIS^{-1}(T) \cap \{S | S \vdash \mathit{Write}\ x : \_\})$ by EXT, and consequently $T' \in VIS^{-1}(T)$. As $VIS \subseteq AR$, $T' \xrightarrow{VIS} T$ and consequently $T' \xrightarrow{AR} T$. Recall that we apply transactions in the same order as $AR$, hence we have $s_{T'} \xrightarrow{+} s_T$. Since we have $(x, n) \in s_{T'}$ and $s_{T'} \xrightarrow{+} s_T$, it follows that $s_{T'} \in \mathcal{RS}_e(o)$, hence $\mathcal{RS}_e(o) \neq \emptyset$. We conclude: for any transaction $T$, for any operation $o \in \Sigma_T$, $\mathcal{RS}_e(o) \neq \emptyset$, hence $\text{PREREAD}_e(\mathcal{T})$ is true. Now consider any $T \in \mathcal{T}$, we want to prove that $\forall o \in \Sigma_T : \forall T' \in \text{PREC}_e(T) : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$. First we prove that $\forall T' \in \text{PREC}_e(T) \Rightarrow T' \xrightarrow{VIS} T$. We previously proved that $\text{PREREAD}_e(\mathcal{T})$ is true. Hence, by Lemma 4 we know that there is a chain $T' \xrightarrow{wr/ww}{}^{+} T$. Consider any edge on the chain: $T_i \xrightarrow{ww/wr} T_j$:

1. $T_i \xrightarrow{ww} T_j$: We have $T_i, T_j \in \mathcal{H}$ and $(T_i \neq T_j \wedge T_i \vdash \mathit{Write}\, x : \_ \wedge T_j \vdash \mathit{Write}\, x : \_)$, therefore by NOCONFLICT, we have $T_i \xrightarrow{VIS} T_j \vee T_j \xrightarrow{VIS} T_i$. Note that $s_{T_i} \xrightarrow{*} s_{T_j}$, we know that

$T_i \xrightarrow{AR} T_j$, and since $VIS \subseteq AR$, we have $T_i \xrightarrow{VIS} T_j$.

2. $T_i \xrightarrow{wr} T_j$. We map the initial values in Cerone et al from $0$ to $\perp$. Let $n$ be the value that $T_i$ writes and $T_j$ reads. A transaction cannot write the empty value, i.e. $\perp$, to a key. It follows that $T_j \vdash \mathit{Read}\ x : n$ and $n \neq 0$. By EXT, $\max_{AR}(VIS^{-1}(T_j) \cap \{S | S \vdash \mathit{Write}\ x : \_\}) \vdash \mathit{Write}\ x : n$. Since $T_i \vdash \mathit{Write}\ x : n$, $T_i = \max_{AR}(VIS^{-1}(T_j) \cap \{S | S \vdash \mathit{Write}\ x : \_\})$ hold, and consequently $T_i \in VIS^{-1}(T_j)$, i.e. $T_i \xrightarrow{VIS} T_j$.

Now we consider the chain $T' \xrightarrow{wr/ww}^{+} t$, and we have that $T' \xrightarrow{VIS}^{+} T$, by TRANSVIS, we have $T' \xrightarrow{VIS} T$. Now, consider any $o \in \Sigma_T$ such that $o.k \in \mathcal{W}_{T'}$, let $o.k = x$, therefore $T' \vdash \mathit{Write}\ x : \_$. We previously proved that $T' \xrightarrow{VIS} T$. Hence we have $T' \in VIS^{-1}(T) \cap \{S | S \vdash \mathit{Write}\ x : \_\}$. Now we consider the following two cases. If $o$ is an external read, and reads the value $(x, \hat{x})$ written by $\hat{T}$. As transactions cannot write an empty value, i.e. $\perp$, to a key, we have $T \vdash \mathit{Read}\ x : \hat{x}$ and $\hat{x} \neq 0$. By EXT, $\max_{AR}(VIS^{-1}(T_j) \cap \{S | S \vdash \mathit{Write}\ x : \_\}) \vdash \mathit{Write}\ x : \hat{x}$. Since $\hat{T} \vdash \mathit{Write}\ x : n$, we have $\hat{T} = \max_{AR}(VIS^{-1}(T_j) \cap \{S | S \vdash \mathit{Write}\ x : \_\})$ , therefore $T' \xrightarrow{AR} \hat{T}$ or $T' = \hat{T}$. Note that we apply transactions in the same order as $AR$, therefore we have $s_{T'} \xrightarrow{+} s_{\hat{T}}$ or $s_{T'} = s_{\hat{T}}$, i.e. $s_{T'} \xrightarrow{*} s_{\hat{T}}$. Since we proved that $\text{PREREAD}_e(T)$ is true, we have $sf_o$ exists and $s_{\hat{T}} = sf_o$, note that by definition $sf_o \xrightarrow{*} sl_o$. Now we have $s_{T'} \xrightarrow{*} s_{\hat{T}} = sf_o \xrightarrow{*} sl_o$, therefore $s_{T'} \xrightarrow{*} sl_o$ . If $o$ is an internal read operation or write operation, then $sl_o = s_p(T)$. Since $T' \in \text{PREC}_e(T)$, by Lemma 3, we have $s_{T'} \xrightarrow{+} s_T$, therefore $s_{T'} \xrightarrow{*} s_p(T) = sl_o$, i.e. $s_{T'} \xrightarrow{*} sl_o$.

# F Hierarchy

In this section, we prove the existence of the strict hierarchy described in Figure 1. Specifically, we prove:

**Theorem 3.** *Adya SI $\subset$ PSI.*

**Theorem 4.** *ANSI SI $\subset$ Adya SI.*

**Theorem 5.** *Strong Session SI $\subset$ ANSI SI.*

**Theorem 6.** *Strong SI $\subset$ Strong Session SI.*

The equivalence results derived from previous appendices complete the proof.

## F.1 Adya SI $\subset$ PSI

**Theorem 3** Adya SI $\subset$ PSI.

*Proof.* **Adya SI $\subseteq$ PSI** First we prove that Adya SI $\subseteq$ PSI. Specifically, we prove that, if there exists an $e$ such that $\forall T \in \mathcal{T} : \text{CT}_{AdyaSI}(T, e)$, that same $e$ also satisfies $\forall T \in \mathcal{T} : \text{CT}_{PSI}(T, e)$ where $\text{CT}_{PSI}(T, e) = \textsc{preread}_e(T) \wedge \forall T' \, \triangleright T : \forall o \in \Sigma_T : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$ Consider any $T$ that satisfies the commit test $\text{CT}_{AdyaSI}(T, e) = \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s)$ and let $s_c$ be the state that satisfies $\textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s)$. Since $\textsc{complete}_{e,t}(s_c)$, we have $s_c \in \bigcap\limits_{o \in \Sigma_T} \mathcal{RS}_e(o)$. It follows that $\forall o \in \Sigma_T : \mathcal{RS}_e(o) \neq \emptyset$ and consequently that $\textsc{preread}_e(T)$ is satisfied. Now we consider any $T'$ such that $T' \, \blacktriangleright T$ , or equivalently, $T' \in \textsc{d-prec}_e(T)$ where $\textsc{d-prec}_e(\hat{T}) = \{T | \exists o \in \Sigma_{\hat{T}} : T = T_{sf_o}\} \cup \{T | s_T \xrightarrow{+} s_{\hat{T}} \wedge \mathcal{W}_{\hat{T}} \cap \mathcal{W}_T \neq \emptyset\}$. Let us first assume that $T' \in \{\hat{T} | \exists o \in \Sigma_T : \hat{T} = T_{sf_o}\}$ such that $\exists o \in \Sigma_T : T' = sf_o$. Since $T' = sf_o$, we have that $s_{T'} \in \mathcal{RS}_e(o)$ and consequently that $s_{T'} \xrightarrow{*} s_c \xrightarrow{*} sl_o$ Assuming instead that $T' \in \{\hat{T} | s_{\hat{T}} \xrightarrow{+} s_T \wedge \mathcal{W}_T \cap \mathcal{W}_{\hat{T}} \neq \emptyset\}$. As $\textsc{no-conf}_T(s_c)$, i.e. $\Delta(s_c, sp_T) \cap \mathcal{W}_T = \emptyset$ implies that either $s_{T'} \xrightarrow{*} s$ or $s_T \xrightarrow{*} s_{T'}$ is true, we can conclude that $s_{T'} \xrightarrow{*} s_c$ holds, and consequently that $s_{T'} \xrightarrow{*} sl_o$. Combining these two results, we can conclude that if $T' \, \blacktriangleright T$, $s_{T'} \xrightarrow{*} s_c$. Strengthening this result using the definition of read state, we have $T' \, \blacktriangleright T \Rightarrow s_{T'} \xrightarrow{+} s_T$. Taking the transitive closure, we have that $T' \, \triangleright T \Rightarrow s_{T'} \xrightarrow{+} s_T$. Now, considering the definition of $\triangleright$: for any $T' \, \triangleright T$, either $T' \, \blacktriangleright T$, or $\exists \hat{T} : T' \, \triangleright \hat{T} \wedge \hat{T} \, \blacktriangleright T$. If $T' \, \blacktriangleright T$, we already proved that $s_{T'} \xrightarrow{*} s_c$. Now considering

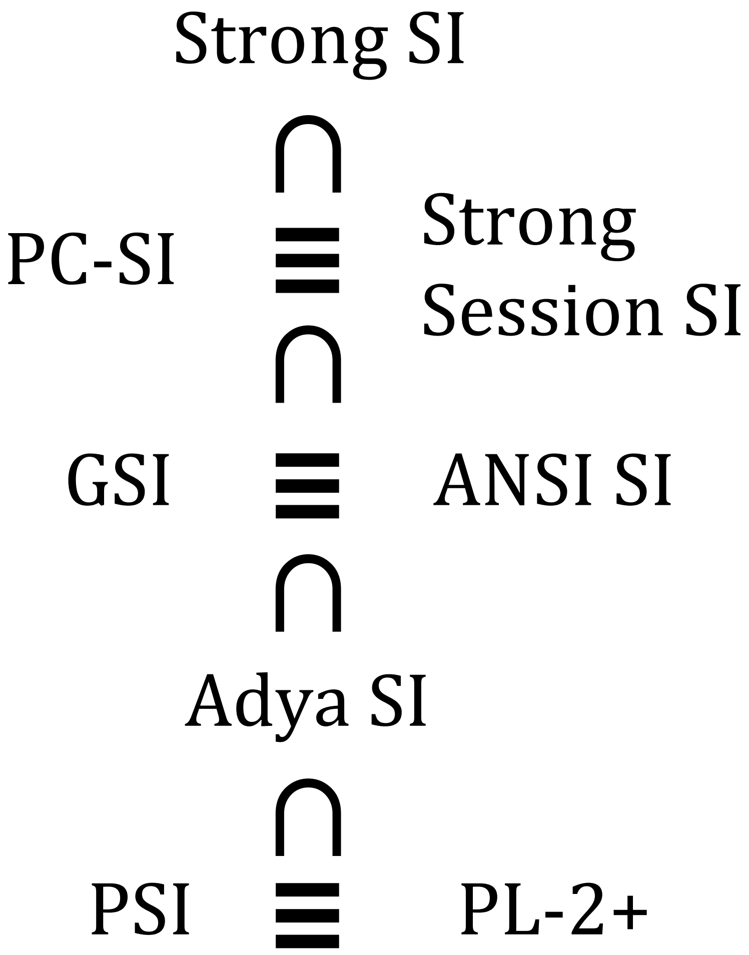


*Figure 1: Snapshot-based isolation guarantees hierarchy. Equivalences are new results (ANSI SI [31], Adya SI [4], Weak SI [61], Strong SI [61], generalized snapshot isolation (GSI) [154], parallel snapshot isolation (PSI) [178], Strong Session SI [61], PL-2+ (Lazy Consistency) [5], prefix-consistent SI (PC-SI) [61])*

$\exists \hat{T} : T' \triangleright \hat{T} \wedge \hat{T} \blacktriangleright T$. We previously proved that $T' \triangleright \hat{T} \Rightarrow s_{T'} \xrightarrow{+} s_{\hat{T}}$ and that $\hat{T} \blacktriangleright T \Rightarrow s_{\hat{T}} \xrightarrow{*} s_c$. Combining these two implications, we have $\exists \hat{T} : T' \triangleright \hat{T} \wedge \hat{T} \blacktriangleright T \Rightarrow s_{T'} \xrightarrow{*} s_c$, and consequently that $T' \triangleright T \Rightarrow s_{T'} \xrightarrow{*} s_c$. Since $s_c \in \bigcap\limits_{o \in \Sigma_T} \mathcal{RS}_e(o)$, we have $\forall o \in \Sigma_T : s_c \xrightarrow{*} sl_o$. We have proved $\forall T' \triangleright T : \forall o \in \Sigma_t : s_{T'} \xrightarrow{*} sl_o$, which trivially implies $\forall T' \triangleright T : \forall o \in \Sigma_t : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$, i.e. CAUS-VIS$(e, T)$ is satisfied. Combining all previous results, we have Adya SI $\subseteq$ PSI.

**Adya SI $\neq$ PSI** Second, we prove Adya SI $\neq$ PSI by describing a set of transactions that satisfy PSI but not Adya SI Consider the five following transactions $T_1, T_2, T_3, T_4, T_5$, where $T_1 : w(x, x_1)w(y, y_1)$, $T_2 : r(x, x_1)w(x, x_2)$, $T_3 : r(y, y_1)w(y, y_2)$, $T_4 : r(x, x_2)r(y, y_1)$, $T_5 : r(x, x_1)r(y, y_2)$. This set of transactions satisfies PSI as it admits the following execution $e$ such that all transactions satisfy the commit test: $s_0 \xrightarrow{T_1} s_1 \xrightarrow{T_2} s_2 \xrightarrow{T_3} s_3 \xrightarrow{T_4} s_4 \xrightarrow{T_5} s_5$. In contrast, the aforementioned transactions do not satisfy Adya SI as there does not exist an execution such that their commit tests are satisfied. Indeed, to satisfy the commit test of all these transactions, there should exist complete states $s$ and $s'$ for $T_4$ and $T_5$ respectively, where $s$ should contain values $(x, x_2)$ and $(y, y_1)$ , and $s'$ values $(x, x_1)$ and $(y, y_2)$. Generating $s$ requires applying transactions $T_1$ and $T_2$ before applying transaction $T_3$, while generating $s'$ requires applying $T_1$ and $T_3$ before applying $T_2$. As the execution $e$ is totally ordered, satisfying both these constraints is impossible, hence there cannot exist complete states for both $T_4$ and $T_5$. This set of transactions thus satisfies PSI but not Adya SI. We conclude: Adya SI $\subset$ PSI □

## F.2 ANSI SI $\subset$ Adya SI

**Theorem 4** ANSI SI $\subset$ Adya SI.

**ANSI SI $\subseteq$ Adya SI** First we prove that ANSI SI $\subseteq$ Adya SI. The result follows trivially from the definition of the definitions' commit tests: $\exists s \in S_e$ : COMPLETE$_{e,T}(s) \wedge$ NO-CONF$_T(s) \implies$ C-ORD$(T_{s_p}, T) \wedge \exists s \in S_e$ : COMPLETE$_{e,T}(s) \wedge$ NO-CONF$_T(s) \wedge (T_s <_s T)$.

**ANSI SI $\neq$ Adya SI** Now we prove that ANSI SI $\neq$ Adya SI. Consider the set of transactions: $T_1 : w(x, x_1)$, $T_1$ starts at real time 1 and commits at real time 4; $T_2 : r(x, x_1)$, $T_2$ starts at real time 2 and commits at real time 3. The set of transactions satisfy Adya SI as there exists an execution for which the commit test of all transactions is satisfied: $s_0 \xrightarrow{T_1} s_{T_1} \xrightarrow{T_2} s_{T_2}$, with $s_0$ being the selected complete state for $T_1$ and $s_1$ being the selected complete state for $T_2$. However, it does not satisfy ANSI SI. Given that $\textsc{c-ord}(T_{s_p}, T)$ must hold, the only possible execution is $s_0 \xrightarrow{T_2} s_{T_2} \xrightarrow{T_1} s_{T_1}$. But $s_0$ is not a valid complete state for $T_2$. As this is the only possible execution, the aforementioned set of transactions does not satisfy ANSI SI, i.e. ANSI SI $\neq$ Adya SI. Therefore, we conclude that ANSI SI $\subset$ Adya SI.

## F.3 Strong Session SI $\subset$ ANSI SI

**Theorem 5** Strong Session SI $\subset$ ANSI SI.

First, we prove that Strong Session SI $\subseteq$ ANSI SI. The result follows trivially from the definition of the definitions' commit tests: $\textsc{c-ord}(T_{s_p}, T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (T_s <_s T) \implies \textsc{c-ord}(T_{s_p}, T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (T_s <_s T) \wedge (\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s)$ . Next, we prove that Strong Session SI $\neq$ ANSI SI. Consider the set of transactions: $T_1 : w(x, x_1)$, $T_1$ starts at real time 1 and commits at real time 2; $T_2 : r(x, x_1)w(x, x_2)$, $T_2$ starts at real time 3 and commits at real time 4; $T_3 : r(x, x_1)$, $T_3$ starts at real time 5 and commits at real time 6. $T_2$ and $T_3$ are in the same session. These transactions satisfy ANSI SI as there exists an execution $e$ such that the commit test of all transactions is satisfied: $s_0 \xrightarrow{T_1} s_{T_1} \xrightarrow{T_2} s_{T_2} \xrightarrow{T_3} s_{T_3}$. The execution satisfies $\textsc{c-ord}(T_{s_p}, T)$ with $s_0$ being the satisfying state for $T_1$, $s_1$ being the satisfying state for $T_2$, $s_1$ being the satisfying state for $T_3$. Due to $\textsc{c-ord}(T_{s_p}, T)$, this is the only possible execution. This execution, however, does not satisfy Strong Session SI. In this execution, the only possible complete state for $T_3$ is $s_{T_1}$. Since $T_2 \xrightarrow{se} T_3$, satisfying Strong Session SI would require that $s_{T_2} \xrightarrow{*} s_{T_1}$, contradicting the order of state transitions in the execution. Since no other execution satisfies the commit test for all transactions, the aforementioned set of transactions does not satisfy Strong Session SI, i.e. Strong Session SI $\neq$ ANSI SI. We conclude: Strong Session SI $\subset$ ANSI SI.

## F.4 Strong SI $\subset$ Strong Session SI

**Theorem 6** Strong SI $\subset$ Strong Session SI.

First we prove that Strong SI $\subseteq$ Strong Session SI. Specifically, we prove that if there exists an execution $e$ such that $\textsc{c-ord}(T_{s_p}, T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (T_s <_s T) \wedge (\forall T' <_s T : s_{T'} \xrightarrow{*} s)$ , that same $e$ also satisfies $\textsc{c-ord}(T_{s_p}, T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge$

$\text{NO-CONF}_T(s) \wedge (T_s <_s T) \wedge (\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s)$. Let $\mathcal{T}$ denote a set of transactions satisfying Strong SI. Consider an execution $e$ that satisfy $\text{CT}_{StrongSI}(\mathcal{T}, e)$ (such $e$ must exist by definition). For each transaction $T$, consider the state $s$ that satisfies $\text{COMPLETE}_{e,T}(s) \wedge \text{NO-CONF}_T(s) \wedge (T_s <_s T) \wedge (\forall T' <_s T : s_{T'} \xrightarrow{*} s)$, we prove that $s$ also satisfies $\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s$. We know by assumption that $\forall T' <_s T : s_{T'} \xrightarrow{*} s$ for every $T$ in $e$ as it is Strong SI. Moreover, by definition, $T' \xrightarrow{se} T \Rightarrow T'.commit < T.start$, i.e. $T' <_s T$. It thus trivially follows that $\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s$ and consequently that every transaction in $e$ satisfies $\text{CT}_{Session\ SI}(T, e)$, so Strong SI $\subseteq$ Strong Session SI. Now, we prove that Strong SI $\neq$ Strong Session SI. Consider the set of transactions: $T_1 : w(x, x_1)$, $T_1$ starts at real time 1 and commits at real time 2; $T_2 : r(x, x_1)w(x, x_2)$, $T_2$ starts at real time 3 and commits at real time 4; $T_3 : r(x, x_1)$, $T_3$ starts at real time 5 and commits at real time 6. No two transactions belong to the same session. These transactions satisfy Strong Session SI as there exist an execution $e$ such that the commit test of all transactions is satisfied: $s_0 \xrightarrow{T_1} s_{T_1} \xrightarrow{T_2} s_{T_2} \xrightarrow{T_3} s_{T_3}$. The execution satisfies $\text{C-ORD}(T_{s_p}, T)$, with $s_0$ being the selected complete state for $T_1$, $s_1$ the selected complete state for $T_2$ and $T_3$. However, the set of transactions does not satisfy Strong SI: as $\text{C-ORD}(T_{s_p}, T)$ must hold, the only possible execution is $s_0 \xrightarrow{T_1} s_{T_1} \xrightarrow{T_2} s_{T_2} \xrightarrow{T_3} s_{T_3}$. The only possible complete state for $T_3$ is $s_{T_1}$. Since $T_2 <_s T_3$, satisfying Strong SI would require $s_{T_2} \xrightarrow{*} s_{T_1}$, contradicting the order of state transitions in the execution. Since no other execution satisfies the commit test for all transactions, the aforementioned set of transactions does not satisfy Strong SI, i.e. Strong Session SI $\neq$ ANSI SI. We conclude: Strong SI $\subset$ Strong Session SI.

# G Causality and Session Guarantees

We prove the following theorems:

**Theorem 1** Let $\mathcal{G} = \{RMW, MR, MW, WFR\}$, then $\forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{\mathcal{G}}(se, T, e) \equiv \forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{CC}(se, T, e)$

We first state a number of useful lemmas about the $\text{PREREAD}_e(\mathcal{T})$ predicate (Definition 3): if $\text{PREREAD}_e(\mathcal{T})$ holds, then the candidate read set of all operations in all transactions in $\mathcal{T}$ is not empty. The first lemma states that an operation's read state must reflect writes that took place before the transaction committed, while the second lemma simply argues that the predicate is closed under subset.

**Lemma 5.** *For any $\mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$, $\text{PREREAD}_e(\mathcal{T}') \Leftrightarrow \forall T \in \mathcal{T}' : \forall o \in \Sigma_T : sf_o \xrightarrow{+} s_T$.*

*Proof.* ($\Rightarrow$) We first prove $\text{PREREAD}_e(\mathcal{T}') \Rightarrow \forall T \in \mathcal{T}' : \forall o \in \Sigma_T : sf_o \xrightarrow{+} s_T$.

By the definition of $\text{PREREAD}_e(\mathcal{T}')$, we have $\forall T \in \mathcal{T}', \forall o \in \Sigma_T$, $\mathcal{RS}_e(o) \neq \emptyset$. We consider the two types of operations: reads and writes. **Reads** The set of candidate read states of a read operation $o = r(k,v)$ is defined as $\mathcal{RS}_e(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p \wedge \big((k,v) \in s \vee (\exists w(k,v) \in \Sigma_T : w(k,v) \xrightarrow{to} r(k,v))\big)\}$. The disjunction considers two cases:

1. *Internal Reads* if $\exists w(k,v) \in \Sigma_T : w(k,v) \xrightarrow{to} r(k,v)$, $\mathcal{RS}_e(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p\}$. Hence $s_0 \in \mathcal{RS}_e(o)$. It follows that $sf_o = s_0 \xrightarrow{+} s_T$
2. *External Reads* By $\text{PREREAD}_e(\mathcal{T}')$, we have that $\mathcal{RS}_e(o) \neq \emptyset$. There must therefore exist a state $s \in \mathcal{S}_e$ such that $s \xrightarrow{*} s_p \wedge (k,v) \in s$. Since $sf_o$ is, by definition, the first such $s$, we have that $sf_o \xrightarrow{*} s_p \rightarrow s_T$. We conclude: $sf_o \xrightarrow{+} s_T$.

**Writes** The candidate read states set for write operations $o = w(k,v)$, is defined as $\mathcal{RS}_e(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p\}$. Hence, $s_0 \in \mathcal{RS}_e(o)$. It trivially follows that $(sf_o = s_0) \xrightarrow{+} s_T$. We conclude: $\text{PREREAD}_e(\mathcal{T}') \Rightarrow \forall T \in \mathcal{T}' : \forall o \in \Sigma_T : sf_o \xrightarrow{+} s_T$.

($\Leftarrow$) Next, we prove that $\text{PREREAD}_e(\mathcal{T}') \Leftarrow \forall T \in \mathcal{T}' : \forall o \in \Sigma_T : sf_o \xrightarrow{+} s_T$.

By assumption, $\forall T \in \mathcal{T}' : \forall o \in \Sigma_T : sf_o \xrightarrow{+} s_T$. By definition, $sf_o \in \mathcal{RS}_e(o)$. It trivially follows that $\forall T \in \mathcal{T}' : \forall o \in \Sigma_T : \mathcal{RS}_e(o) \neq \emptyset$, i.e. $\textsc{preread}_e(\mathcal{T}')$ holds. □

**Lemma 6.** *For any $\mathcal{T}'$ that $\mathcal{T}' \subseteq \mathcal{T}$, $\textsc{preread}_e(\mathcal{T}) \Rightarrow \textsc{preread}_e(\mathcal{T}')$.*

*Proof.* Given $\textsc{preread}_e(\mathcal{T})$, by definition we have $\forall T \in \mathcal{T}, \forall o \in \Sigma_T, \mathcal{RS}_e(o) \neq \emptyset$. Since $\mathcal{T}' \subseteq \mathcal{T}$, $\forall T \in \mathcal{T}' \Rightarrow \forall T \in \mathcal{T}$, it follows that $\forall T \in \mathcal{T}', \forall o \in \Sigma_T, \mathcal{RS}_e(o) \neq \emptyset$, i.e. $\textsc{preread}_e(\mathcal{T}')$. □

We now begin in earnest our proof of Theorem 1.

($\Leftarrow$) **We first prove that** $\forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{CC}(se, T, e) \Rightarrow \forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{\mathcal{G}}(se, T, e)$.

For any $se \in SE$, consider the execution $e$, such that $\forall T \in \mathcal{T}_{se} : \text{SESSION}_{CC}(se, T, e)$. We show that this same execution satisfies the session test of all four session guarantees.

- **CC $\Rightarrow$ RMW**: By assumption, $\text{SESSION}_{CC}(se, T, e)$ for all $\mathcal{T}_{se}$. Hence: $\forall T \in \mathcal{T}_{se} : \forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} sl_o$. Weakening this statement gives the following implication: $\forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : \mathcal{W}_{T'} \neq \emptyset \Rightarrow s_{T'} \xrightarrow{*} sl_o$. Additionally, $e$ satisfies $\textsc{preread}_e(\mathcal{T})$ (by assumption) and therefore $\textsc{preread}_e(\mathcal{T}_{se})$ as $\mathcal{T}_{se} \subseteq \mathcal{T}$ (by Lemma 6). Putting it all together: $e$ satisfies $\textsc{preread}_e(\mathcal{T}_{se}) \wedge \forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : \mathcal{W}_{T'} \neq \emptyset \Rightarrow s_{T'} \xrightarrow{*} sl_o$.

  We conclude that $\forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{RMW}(se, T, e)$.

- **CC $\Rightarrow$ MW**: By assumption, $\text{SESSION}_{CC}(se, T, e)$ for all $t \in \mathcal{T}_{se}$. Hence, it holds that $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$. Weakening this statement gives the following implication: $\forall se' \in SE : \forall T_i' \xrightarrow{se'} T_j : (\mathcal{W}_{T_i} \neq \emptyset \wedge \mathcal{W}_{T_j} \neq \emptyset) \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$. Additionally, $e$ satisfies $\textsc{preread}_e(\mathcal{T})$ (by assumption) and therefore $\textsc{preread}_e(\mathcal{T}_{se})$ as $\mathcal{T}_{se} \subseteq \mathcal{T}$ (by Lemma 6). Putting it all together: $\textsc{preread}_e(\mathcal{T}_{se}) \wedge \forall se' \in SE : \forall T_i' \xrightarrow{se'} T_j : (\mathcal{W}_{T_i} \neq \emptyset \wedge \mathcal{W}_{T_j} \neq \emptyset) \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$.

  We conclude that $\forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{MW}(se, T, e)$.

- **CC $\Rightarrow$ MR**: By assumption, $\text{SESSION}_{CC}(se, T, e)$, hence $e$ ensures that $\forall T \in \mathcal{T}_{se} : \forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} sl_o$. Moreover, by assumption, we have that $\textsc{preread}_e(\mathcal{T})$. It follows that $\forall T' \in \mathcal{T} : \forall o' \in \Sigma_{T'} : sf_{o'} \xrightarrow{+} s_{T'}$ (Lemma 5). Combining the two statements, we have

$\forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : \forall o' \in \Sigma_{T'} : sf_{o'} \xrightarrow{*} s_{T'} \xrightarrow{*} sl_o$, i.e. $sf_{o'} \xrightarrow{*} sl_o$. Finally, we have that $e$ satisfies $\textsc{irc}_e(T)$ by assumption, and $\textsc{preread}_e(\mathcal{T}_{se})$ by Lemma 6: we have $\textsc{preread}_e(\mathcal{T})$ and $\mathcal{T}_{se} \subseteq \mathcal{T}$. Putting it all together, $e$ satisfies $\textsc{preread}_e(\mathcal{T}_{se}) \wedge \textsc{irc}_e(T) \wedge \forall o \in \Sigma_T : \forall T' \xrightarrow{se} t : \forall o' \in \Sigma_{T'} : sf_{o'} \xrightarrow{*} sl_o$.

We conclude that $\forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{MR}(se, T, e)$.

- **CC ⇒ WFR**: By assumption, $\text{SESSION}_{CC}(se, T, e)$. Hence $e$ satisfies $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$. By assumption, $e$ respects $\textsc{preread}_e(\mathcal{T})$. It follows from lemma 5 that $\forall T_i \in \mathcal{T} : \forall o_i \in \Sigma_{T_i} : sf_{o_i} \xrightarrow{+} s_{T_i}$. We have, by combining these two statements, that: $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : \forall o_i \in \Sigma_{T_i} : sf_{o_i} \xrightarrow{+} s_{T_i} \xrightarrow{+} s_{T_j}$, i.e. $sf_{o_i} \xrightarrow{+} s_{T_j}$. Weakening this statement results in the following implication: $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : \forall o_i \in \Sigma_{ti} : \mathcal{W}_{T_j} \neq \emptyset \Rightarrow sf_{o_i} \xrightarrow{+} s_{T_j}$. Putting it all together, $e$ satisfies $\textsc{preread}_e(\mathcal{T}) \wedge \forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : \forall o_i \in \Sigma_{ti} : \mathcal{W}_{T_j} \neq \emptyset \Rightarrow sf_{o_i} \xrightarrow{+} s_{T_j}$.

  We conclude that $\forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{WFR}(se, T, e)$.

(⇒) **We now prove that**, given $\mathcal{G} = \{RMW, MR, MW, WFR\}$, the following implication holds: $\forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{\mathcal{G}}(se, T, e) \Rightarrow \forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se} : \text{SESSION}_{CC}(se, T, e)$.

To this end, we prove that for any session $se$, given the execution $e$ such that $\forall T \in \mathcal{T}_{se} : \text{SESSION}_{\mathcal{G}}(se, T, e)$, we can construct an alternative execution $e'$ that contains the same set of transactions and satisfies all four session guarantees, such that $\forall T \in \mathcal{T}_{se} : \text{SESSION}_{CC}(se, T, e\text{'})$.

The need for constructing an alternative execution $e'$ may be counter-intuitive at first. We motivate it informally here: session guarantees place no requirements on the commit order of read-only transactions. In contrast, causal consistency requires all transactions to commit in session order. As read-only transactions have no effect on the candidate read states of other read-only or update transactions, it is therefore always possible to generate an alternative execution $e'$ such that update transactions commit in the same order as in $e$, and read-only transactions commit in session order. Our proof shows that, if $e$ satisfies all four session guarantees, $e'$ will satisfy all four session guarantees and causal consistency.

**Equivalent Execution** We now describe more formally how to construct this execution $e'$: first, we apply in $e'$ all transactions $t \in \mathcal{T}$ such that $\mathcal{W}_T \neq \emptyset$, respecting their commit order in $e$. We denote the states created by applying $t$ in $e$ and $e'$ as $s_{e,t}$ and $s_{e',t}$ respectively. Our construction enforces the following relationship between $e$ and $e'$: $\forall T \in \mathcal{T} \wedge \mathcal{W}_T \neq \emptyset : (k,v) \in s_{e,t} \Leftrightarrow (k,v) \in s_{e',t}$. All update transactions are applied in the same order, and read-only transactions have no effect on the state. Next, we consider the read-only transactions $T_i \in \mathcal{T}$ in session order: we select the parent state for $T_i$ to be $\max\{\max_{o \in \Sigma_{T_i}}\{sf_o\}, s_{T_{i-1}}\}$, where $T_{i-1}$ denotes the transaction that directly precedes $t$ in a session. A session defines a total order of transactions: $T_{i-1}$ is unique. If $T_i$ is the first transaction in the session, we simply set $s_{T_{i-1}}$ to $s_0$. As transactions do not change the value of states, this process maintains the previously stated invariant: $\forall T \in \mathcal{T} \wedge \mathcal{W}_T \neq \emptyset : s_{e,t} \equiv s_{e',t}$.

We now proceed to prove that $e'$ satisfies $\text{PREREAD}_{e'}(\mathcal{T})$ and the session test for all session guarantees.

**Preread** First, we show that $\text{PREREAD}_{e'}(\mathcal{T})$ holds. We distinguish between update and read-only transactions:

- *Read-Only Transactions.* By construction, the parent state of a read-only transaction $T_i$ is $s_p(T_i) = \max\{\max_{o \in \Sigma_T}\{sf_o\}, s_{T_{i-1}}\}$. It follows that $\forall o \in \Sigma_T, sf_o \xrightarrow{*} s_p(T)$ and consequently $sf_o \xrightarrow{*} s_p(T) \rightarrow s_T$. We have $\forall o \in \Sigma_T : sf_o \xrightarrow{+} s_T$ in $e'$.
- *Update Transactions.* Update transactions $t$ consist of both read and write operations. A write operation $o = w(k,v) \in \Sigma_T$ has for candidate read set the set of all states $s \in \mathcal{S}_{e'}$ such that $s \xrightarrow{*} s_p(T)$. Hence $sf_o = s_0$ and $sf_o \xrightarrow{+} s_T$ in $e'$ trivially holds. The state corresponding to $sf_o$ for read operations $o = r(x_i)$ is the state created by the transaction $T_i$ that wrote version $x_i$ of object $x$: $sf_o = s_{e',T_i}$. By assumption, $e$ satisfies $\text{PREREAD}_e(\mathcal{T})$, hence by Lemma 5, we have $s_{T_i} \xrightarrow{+} s_T$ in $e$. By construction (update transactions are applied in $e'$ in the same order as $e$), it follows that $s_{T_i} \xrightarrow{+} s_T$ in $e'$, i.e. $sf_o \xrightarrow{+} s_T$ in $e'$.

By Lemma 5, we conclude that $\text{PREREAD}_{e'}(\mathcal{T})$ holds.

**MW** We next show that $e'$ satisfies $\text{SESSION}_{MW}(se, T, e')$ for all sessions $se \in SE$ and $\forall T \in \mathcal{T}_{se}$. Consider any session $se'$ and two transactions $T_i, T_j \in \mathcal{T}_{se'}$ such that $T_i \xrightarrow{se'} T_j$, and $\mathcal{W}_{T_i} \neq \emptyset \wedge \mathcal{W}_{T_j} \neq \emptyset$. As $e$, by assumption, satisfies $\forall T \in \mathcal{T}_{se} : \text{SESSION}_{MW}(se, T, e)$, we have $s_{e,T_i} \xrightarrow{+} s_{e,T_j}$.

Since $T_i$ and $T_j$ are update transactions, they are applied in the same order in $e'$ as in $e$: hence, $s_{e',T_i} \xrightarrow{+} s_{e',T_j}$. Further, recall that we previously showed that $\text{PREREAD}_{e'}(\mathcal{T})$ (and consequently $\text{PREREAD}_{e'}(\mathcal{T}_{se})$ by Lemma 6).

Putting it all together, we conclude that $\text{PREREAD}_{e'}(\mathcal{T}_{se}) \wedge \forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : (\mathcal{W}_{T_i} \neq \emptyset \wedge \mathcal{W}_{T_j} \neq \emptyset) \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$, i.e., $\forall T \in \mathcal{T}_{se}$ : SESSION$_{MW}$(*se*, *T*, *e'*).

**WFR** Consider any update transaction $T_j$ such that $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : \forall o_i \in \Sigma_{T_i} : \mathcal{W}_{T_j} \neq \emptyset$. We prove that $sf_{o_i} \xrightarrow{+} s_{T_j}$. Consider the two types of operations that arise in an update transaction:

- *Reads.* The state corresponding to $sf_{o_i}$ for read operations $o = r_i(x, x_k)$ is the state created by an update transaction $T_k$ that wrote version $x_k$ of object $x$: $sf_{o_i} = s_{T_k}$. By assumption, $e$ satisfies $\forall T \in \mathcal{T}_{se}$ : SESSION$_{WFR}$(*se*, *T*, *e*), we have $sf_{o_i} \xrightarrow{+} s_{T_j}$ in $e$, i.e. $s_{T_k} \xrightarrow{+} s_{T_j}$ in $e$. Since we apply update transactions in $e'$ in the same order as in $e$, it follows that $s_{T_k} \xrightarrow{+} s_{T_j}$ in $e'$, i.e., $sf_{o_i} \xrightarrow{+} s_{T_j}$ in $e'$.
- *Writes.* The candidate read states set for write operations $o = w(x_i)$, is defined as $\mathcal{RS}_e(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p\}$. It trivially follows that $sf_o = s_0 \xrightarrow{+} s_T$.

We conclude: $\forall se \in SE : \forall T \in \mathcal{T}_{se}$ : SESSION$_{WFR}$(*se*, *T*, *e'*).

Before proving that the remaining session guarantees hold, we prove an intermediate result:

**Claim 4** $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$. *Intuitively, all transactions commit in session order.*

*Proof.* We first prove this result for update transactions, then generalise it to all transactions.

**Update Transactions** For a given session $se'$, let $\mathcal{T}_u$ be the set of all update transactions in $\mathcal{T}_{se'}$, and let $T_j$ be an arbitrary transaction in $\mathcal{T}_u$. It thus holds that $T_j \in \mathcal{T}_{se'} \wedge \mathcal{W}_{T_j} \neq \emptyset$. We associate with each such $T_j$ two further sets: $\mathcal{T}_{pre_u}$ and $\mathcal{T}_{pre_r}$. $\mathcal{T}_{pre_u}$ contains all update transactions $T_i$ such that $T_i \xrightarrow{se'} T_j$. Similarly, $\mathcal{T}_{pre_r}$ contains all read-only transactions $T_i$ such that $T_i \xrightarrow{se'} T_j$.$\mathcal{T}_{pre}$ is the union of those two sets. We prove that $\forall T_i \in \mathcal{T}_{pre} : s_{T_i} \xrightarrow{+} s_{T_j}$. If $T_i \in \mathcal{T}_{pre_u}$, hence $\mathcal{W}_i \neq \emptyset \wedge \mathcal{W}_j \neq \emptyset$, the result trivially follows from monotonic writes. We previously proved that $\forall T \in \mathcal{T}_{se'}$ : SESSION$_{MW}$(*se'*, *T*, *e'*). As such, the conjunction $\mathcal{W}_i \neq \emptyset \wedge \mathcal{W}_j \neq \emptyset$ implies

$s_{T_i} \xrightarrow{+} s_{T_j}$ in $e'$.

The proof is more complex if $T_i \in \mathcal{T}_{pre_r}$ (read-only transaction). We proceed by induction:

**Base Case** Consider the first read-only transaction $T_i \in \mathcal{T}_{pre_r}$ according to the session order $se'$. This transaction is unique (sessions totally order transactions). Recall that we choose the parent state of a read-only transaction as $s_p(T_i) = \max\{\max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}, s_{T_{i-1}}\}$, where $T_{i-1}$ denotes the transaction that directly precedes $T_i$ in session $se'$ ($s_{T_{i-1}} = s_0$ if $T_i$ is the first transaction in the session). Hence, $T_i$'s parent state is either $s_p(T_i) = \max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}$, or $s_p(T_i) = s_{T_{i-1}}$

1. If $s_p(T_i) = \max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}$: We previously proved that $\forall T \in \mathcal{T}_{se}$ : SESSION$_{WFR}$(*se*, *T*, *e'*). It follows that $\forall o_i \in \Sigma_{T_i} : sf_{o_i} \xrightarrow{+} s_{T_j}$ in $e'$ and consequently, $\max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\} \xrightarrow{+} s_{T_j}$ in $e'$. Given that $s_p(T_i) = \max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}$, the following then holds $s_p(T_i) \xrightarrow{+} s_{T_j}$ in $e'$. Finally, we note that, by definition (Definition 1), the parent state of a transaction directly precede its commit state. We can thus rephrase the aforementioned relationship as $s_p(T_i) \rightarrow s_{T_i} \xrightarrow{*} s_{T_j}$ in $e'$, concluding the proof for this subcase.
2. If $s_p(T_i) = s_{T_{i-1}}$: We defined $T_i$ to be the first read-only transaction in the session. Given that, by construction $T_{i-1} \xrightarrow{se'} T_i$, $T_{i-1}$ is necessarily an update transaction, where $\mathcal{W}_{T_{i-1}} \neq \emptyset$. Consider the pair of transactions $(T_{i-1}, T_j)$. The session order is transitive, hence $T_{i-1} \xrightarrow{se'} T_j$ given that $T_{i-1} \xrightarrow{se'} T_i$ and $T_i \xrightarrow{se'} T_j$ both hold. By construction, we have $\mathcal{W}_{T_{i-1}} \neq \emptyset \wedge \mathcal{W}_{T_j} \neq \emptyset$. We previously proved that $\forall T \in \mathcal{T}_{se}$ : SESSION$_{MW}$(*se*, *T*, *e'*). Hence, if $\mathcal{W}_{T_{i-1}} \neq \emptyset \wedge \mathcal{W}_{T_j} \neq \emptyset$, it follows that $s_{T_{i-1}} \xrightarrow{+} s_{T_j}$. As above, we conclude that: $s_p(T_i) \xrightarrow{+} s_{T_j}$ in $e'$, and finally $s_p(T_i) \rightarrow s_{T_i} \xrightarrow{*} s_{T_j}$.

To complete the base case, we note that $T_i \neq T_j$ as $T_i \xrightarrow{se'} T_j$. We conclude: $s_{T_i} \xrightarrow{+} s_{T_j}$.

**Induction Step** Consider the k-th read-only transaction $T_i$ in $se'$ such that $T_i \xrightarrow{se'} T_j$. We assume that it satisfies the induction hypothesis $s_{T_i} \xrightarrow{+} s_{T_j}$. Now consider the (k+1)-th read-only transaction $T_{i'}$ in $se'$, such that $T_{i'} \xrightarrow{se'} T_j$. By construction, we once again distinguish two cases: $T_{i'}$'s parent state is either $s_p(T_{i'}) = \max_{o_{i'} \in \Sigma_{T'_i}}\{sf_{o'_i}\}$, or $s_p(T_{i'}) = s_{T_{i'-1}}$, where $T_{i'-1}$ denotes the transaction directly preceding $T_{i'}$ in a session.

1. If $s_p(T_{i'}) = \max_{o_{i'} \in \Sigma_{T_{i'}}}\{sf_{o'_i}\}$: We previously proved that $\forall T \in \mathcal{T}_{se}$ :

$\text{SESSION}_{WFR}(se, T, e')$. It follows that $\forall o_{i'} \in \Sigma_{T_{i'}} : sf_{o_{i'}} \xrightarrow{+} s_{T_j}$ in $e'$ and consequently, $\max_{o_{i'}} \in \Sigma_{T_{i'}}\{sf_{o_{i'}}\} \xrightarrow{+} s_{T_j}$ in $e'$. Given that $s_p(T_{i'}) = \max_{o_{i'} \in \Sigma_{T_{i'}}}\{sf_{o_{i'}}\}$, the following then holds $s_p(T_{i'}) \xrightarrow{+} s_{T_j}$ in $e'$. Finally, we note that, by definition (Definition 1), the parent state of a transaction must directly precede its commit state. We can thus rephrase the aforementioned relationship as $s_p(T_{i'}) \rightarrow s_{T_{i'}} \xrightarrow{*} s_{T_j}$ in $e'$, concluding the proof for this subcase.

2. If $s_p(T_{i'}) = s_{T_{i'-1}}$: First, we note that $T_{i'-1} \xrightarrow{se'} T_j$ holds, as the session order is transitive and we have both $T_{i'-1} \xrightarrow{se'} T_{i'}$ and $T_{i'} \xrightarrow{se'} T_j$. We then distinguish between two cases: $T_{i'-1}$ is an update transaction, and $T_{i'-1}$ is a read only transaction. If $T_{i'-1}$ is an update transaction, the following conjunction holds: $\mathcal{W}_{T_{i'-1}} \neq \emptyset \wedge \mathcal{W}_{T_j} \neq \emptyset$. Given that we previously proved $\forall T \in \mathcal{T}_{se} : \text{SESSION}_{MW}(se, T, e')$ , we can infer that $s_{T_{i'-1}} \xrightarrow{+} s_{T_j}$, i.e. $s_p(T_{i'}) \xrightarrow{+} s_{T_j}$ in $e'$. We again note that by definition (Definition 1), the parent state of a transaction must directly precede its commit state. We can thus rephrase the aforementioned relationship as $s_p(T_{i'}) \rightarrow s_{T_{i'}} \xrightarrow{*} s_{T_j}$ in $e'$. We now consider the case where $T_{i'-1}$ is a read-only transaction. If $T_{i'}$ is the k+1-th read-only transaction, then, by construction $T_{i'-1}$ is the k-th read-only transaction. Hence $T_i = T_{i'-1} = s_p(T_{i'})$. Our induction hypothesis states that $s_{T_i} \xrightarrow{+} s_{T_j}$. It thus follows that $s_p(T_{i'}) \xrightarrow{+} s_{T_j}$ in $e'$. As previously, we conclude that: $s_p(T_i) \rightarrow s_{T_i} \xrightarrow{*} s_{T_j}$ in $e'$.

To complete the induction step, we note that $T_i' \neq T_j$ as $T_i' \xrightarrow{se'} T_j$. We conclude: $s_{T_i'} \xrightarrow{+} s_{T_j}$.

We proved the desired result for both the base case and induction step. By induction, we conclude that: given any $T_j$ such that $\mathcal{W}_{T_j} \neq \emptyset$,and any read-only transactions $T_i$ such that $T_i \xrightarrow{se'} T_j$, $s_{T_i} \xrightarrow{+} s_{T_j}$ holds.

We conclude that given any $T_j$ such that $\mathcal{W}_{T_j} \neq \emptyset$, and any transaction $T_i$ such that $T_i \xrightarrow{se'} T_j$, $s_{T_i} \xrightarrow{+} s_{T_j}$ holds.

**Read-Only Transactions** We now generalise the result to both update and read-only transactions. Specifically, we prove that in $e'$, $\forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$. We first prove this statement for any two consecutive transactions in a session, and then extend it to all transactions in a session. Consider any pair of transactions $T_i$,$T_{i-1}$ in $\mathcal{T}_{se'}$ such that $T_{i-1}$ directly precede $T_i$ in $se'$ ($T_{i-1} \xrightarrow{se'} T_i$). If $\mathcal{W}_{T_i} \neq$

$\emptyset$, $s_{T_{i-1}} \xrightarrow{+} s_{T_i}$ as proven above. If $T_i$ is a read-only transaction, its parent state, by construction, is equal to $s_p(T_i) = \max\{\max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}, s_{T_{i-1}}\}$. Since $\max\{\max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}, s_{T_{i-1}}\} \geq s_{T_{i-1}}$ by definition, it follows that $s_{T_{i-1}} \xrightarrow{*} s_p(T_i)$. As $s_p(T_i) \rightarrow s_{T_i}$, it follows that $s_{T_{i-1}} \xrightarrow{+} s_{T_i}$. Together each such pair of consecutive transactions $T_{i-1}, T_i$ defines a sequence: $T_1 \xrightarrow{se'} T_2 \xrightarrow{se'} \ldots \xrightarrow{se'} T_k$, where $\mathcal{T}_{se'} = \{T_1, \ldots, T_k\}$. From the implication derived above, it follows that $s_{T_1} \xrightarrow{+} s_{T_2} \xrightarrow{+} \ldots \xrightarrow{+} s_{T_k}$. We conclude: $\forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$.

This completes the proof of Claim 4. $\square$

**RMW** We now return to session guarantees and prove that $\forall T \in \mathcal{T}_{se}$ : SESSION$_{RMW}(se, T, e')$. Specifically, we show that PREREAD$_{e'}(\mathcal{T}_{se}) \wedge \forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : \mathcal{W}_{T'} \neq \emptyset \Rightarrow s_{T'} \xrightarrow{*} sl_o$.

We proceed to prove that each of the two clauses holds true.

By assumption, $e$ guarantees Read-My-Writes: $\forall T \in \mathcal{T}_{se}$ : SESSION$_{RMW}(se, T, e)$. Consider an arbitrary transaction $t$, and all update transactions $T'$ that precede $t$ in the session: $\forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : \mathcal{W}_{T'} \neq \emptyset$. We distinguish between read operations and write operations:

- Let $o$ be a read operation $o = r(k, v)$. Its candidate read set is $\mathcal{RS}_{e'}(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p(T) \wedge \big((k, v) \in s \vee (\exists w(k, v) \in \Sigma_T : w(k, v) \xrightarrow{to} r(k, v))\big)\}$. $sl_o$, the last state in $\mathcal{RS}_{e'}(o)$ can have one of two values: $sl_o = s_p(T)$, disallowing states created after $t$'s commit point, or $sl_o = s_p(\hat{T})$, where $\hat{T}$ is the update transaction that writes the next version of $k$.
  - $sl_o = s_p(T)$. We previously proved that $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$. By construction, $T' \xrightarrow{se} t$. It follows that $s_{T'} \xrightarrow{+} s_T$ and consequently that $s_{T'} \xrightarrow{*} s_p(T)$. Given $s_p(T) = sl_o$, we conclude: $s_{T'} \xrightarrow{*} sl_o$.
  - $sl_o = s_p(\hat{T})$. Consider first the relationships between read states and commit states in $e$. By assumption, $e$ satisfies $\forall T \in \mathcal{T}_{se}$ : SESSION$_{RMW}(se, T, e)$, i.e. $s_{T'} \xrightarrow{*} sl_o$ in $e$. Since $\hat{T}$ wrote the next version of the object that $t$ read, we have that $sl_o \xrightarrow{*} s_p(\hat{T}) \xrightarrow{+} s_{\hat{T}}$ in $e$. Combining the guarantee given by Read-My-Writes $s_{T'} \xrightarrow{*} sl_o$ and $sl_o \xrightarrow{+} s_{\hat{T}}$, we obtain $s_{T'} \xrightarrow{+} s_{\hat{T}}$ in $e$. Returning to the execution $e'$, since $T'$ and $\hat{T}$ are both update transactions, $\mathcal{W}_{T'} \neq \emptyset \wedge \mathcal{W}_{\hat{T}} \neq \emptyset$, if $s_{T'} \xrightarrow{+} s_{\hat{T}}$ in $e$, then $s_{T'} \xrightarrow{+} s_{\hat{T}}$ in $e'$. Given $e'$ is a total order and $s_p(\hat{T}) \rightarrow s_{\hat{T}}$, we conclude $s_{T'} \xrightarrow{*} s_p(\hat{T})$, and $s_{T'} \xrightarrow{*} sl_o$.

- Let $o = w(k, v)$ be a write operation. By Claim 4, it holds that $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$. As $T' \xrightarrow{se} t$, it follows that $s_{T'} \xrightarrow{+} s_T$ and consequently that $s_{T'} \xrightarrow{*} s_p(T)$. We conclude: $s_{T'} \xrightarrow{*} sl_o$.

Finally, as $\text{PREREAD}_{e'}(\mathcal{T})$ and $\mathcal{T}_{se} \subseteq \mathcal{T}$, by Lemma 6, $\text{PREREAD}_{e'}(\mathcal{T}_{se})$ holds.

We conclude that $\text{PREREAD}_{e'}(\mathcal{T}_{se}) \wedge \forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : \mathcal{W}_{T'} \neq \emptyset \Rightarrow s_{T'} \xrightarrow{*} sf_o$, i.e. $\forall T \in \mathcal{T}_{se}$ : SESSION$_{RMW}$(*se*, *T*, *e'*).

**MR** Finally, we prove that $e'$ satisfies the final session guarantee: SESSION$_{MR}$(*se*, *T*, *e'*). Specifically, we show that:
$\text{PREREAD}_{e'}(\mathcal{T}_{se}) \wedge \text{IRC}_{e'}(T) \wedge \forall o \in \Sigma_T : \forall T' \xrightarrow{se} t : \forall o' \in \Sigma_{T'} : sf_{o'} \xrightarrow{*} sl_o$. Intuitively, this states that the read state of $o$ must include any write seen by $o'$.

We proceed to prove that each of the three clauses holds true.

$\text{PREREAD}_{e'}(\mathcal{T}_{se})$ follows directly from Lemma 6 and the fact that $\mathcal{T}_{se} \subseteq \mathcal{T}$. We can then conclude that $sf_o, sf_{o'}, sl_o, sl_{o'}$ must exist.

We can now proceed to prove that $\forall o \in \Sigma_T : \forall T' \xrightarrow{se} t : \forall o' \in \Sigma_{T'} : sf_{o'} \xrightarrow{*} sl_o$ holds. We consider two cases, depending on whether $o'$ is a read or a write operation.

**Read** The read operation $o' = r(k', v')$ entails the existence of an update transaction $\hat{T}' \in \mathcal{T}$ that writes version $v'$ of object $k'$, i.e $sf_{o'} = s_{\hat{T}'}, k \in \mathcal{W}_{\hat{T}'}$. Now, $o$ can be either a read or a write operation.

- Let us first assume that $o$ is a read operation $o = r(k, v)$. Its candidate read set is $\mathcal{RS}_{e'}(o) = \{s \in \mathcal{S}_{e'} | s \xrightarrow{*} s_p(T) \wedge \big((k, v) \in s \vee (\exists w(k, v) \in \Sigma_T : w(k, v) \xrightarrow{to} r(k, v))\big)\}$. $sl_o$, the last state in $\mathcal{RS}_{e'}(o)$ can be one of two cases $sl_o = s_p(T)$, disallowing states created after $t$'s commit point, or $sl_o = s_p(\hat{T})$, where $\hat{T}$ is the update transaction that writes the next version of $k$.
  - $sl_o = s_p(T)$. We previously proved that $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$. By construction, $T' \xrightarrow{se} t$. It follows that $s_{T'} \xrightarrow{+} s_T$ and consequently that $s_{T'} \xrightarrow{*} s_p(T)$. Given $s_p(T) = sl_o$, we conclude: $s_{T'} \xrightarrow{*} sl_o$. Moreover, Lemma 5 states that given

$\textsc{preread}_{e'}(\mathcal{T}_{se})$, we have $sf_{o'} \xrightarrow{+} s_{T'}$ in $e'$, hence: $sf_{o'} \xrightarrow{*} sl_o$ in $e'$.

- $sl_o = s_p(\hat{T})$. Consider first the relationships between read states and commit states in $e$. By assumption, $e$ satisfies $\forall T \in \mathcal{T}_{se}$ : $\text{SESSION}_{MR}(se, T, e)$, i.e. $sf_{o'} \xrightarrow{*} sl_o$ in $e$. Since $\hat{T}$ wrote the next version of the object that $t$ read, we have that in $sl_o \xrightarrow{+} s_{\hat{T}}$ in $e$. Combining the guarantee given by monotonic reads $sf_{o'} \xrightarrow{*} sl_o$ and $sl_o \xrightarrow{+} s_{\hat{T}}$, we obtain $sf_{o'} \xrightarrow{+} s_{\hat{T}}$ in $e$, i.e. $s_{\hat{T}'} \xrightarrow{+} s_{\hat{T}}$. Returning to the execution $e'$, since $\hat{T}'$ and $\hat{T}$ are both update transactions, $\mathcal{W}_{\hat{T}'} \neq \emptyset \wedge \mathcal{W}_{\hat{T}} \neq \emptyset$; if $s_{\hat{T}'} \xrightarrow{+} s_{\hat{T}}$ in $e$, then $s_{\hat{T}'} \xrightarrow{+} s_{\hat{T}}$ in $e'$. By definition $sf_{o'} \xrightarrow{+} s_{T'}$. Moreover, by assumption, $sl_o = s_p(\hat{T})$. Putting this together, we obtain the desired result $sf_{o'} \xrightarrow{*} sl_o$ in $e'$.

- Let $o = w(k, v)$ be a write operation. The write set of a write operation is defined as $\mathcal{RS}_{e'}(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p\}$. It follows that: $sl_o = s_p(T)$. We previously proved that in $e'$, $\forall se' \in SE$ : $\forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$: given $T' \xrightarrow{se} t$, it thus follows that $s_{T'} \xrightarrow{+} s_T$, and consequently that $s_{T'} \xrightarrow{*} s_p(T)$. Noting that $s_p(T) = sl_o$ , we write $s_{T'} \xrightarrow{*} sl_o$. Moreover, as $\textsc{preread}_{e'}(\mathcal{T}_{se})$ holds for $e'$, by Lemma 5, we have $sf_{o'} \xrightarrow{+} s_{T'}$ in $e'$. Combining the relationships, we conclude: $sf_{o'} \xrightarrow{+} sl_o$ in $e'$.

**Write** The candidate read state set for a write operation $o' = w(k, v)$ is defined as the set of all states before $T'$'s commit state. Hence $sf_{o'} = s_0$. Thus $sf_{o'} \xrightarrow{*} sl_o$ trivially holds.

We can now prove that the second clause, $\textsc{irc}_{e'}(T)$, holds—namely, that $\forall o,\ o' \in \Sigma_T : o' \xrightarrow{to} o \Rightarrow sf_{o'} \xrightarrow{*} sl_o$. Once again we consider two cases, depending on whether $o'$ is a read or a write operation.

**Read** The presumpted read operation $o' = r(k', v')$ entails the existence of an update transaction $\hat{T}' \in \mathcal{T}$ that writes version $v'$ of object $k'$, i.e $sf_{o'} = s_{\hat{T}'}$, $k \in \mathcal{W}_{\hat{T}'}$.

- Let us first assume that $o$ is a read operation $o = r(k, v)$, Its candidate read set $\mathcal{RS}_{e'}(o) = \{s \in \mathcal{S}_{e'} | s \xrightarrow{*} s_p(T) \wedge \big((k, v) \in s \vee (\exists w(k, v) \in \Sigma_T : w(k, v) \xrightarrow{to} r(k, v))\big)\}$. $sl_o$, the last state in $\mathcal{RS}_{e'}(o)$ can have one of two values: $sl_o = s_p(T)$, disallowing states created after $t$'s commit point, or $sl_o = s_p(\hat{T})$, where $\hat{T}$ is the update transaction that writes the next version of $k$.
  - $sl_o = s_p(T)$. We previously showed that $\textsc{preread}_{e'}(\mathcal{T})$. Given that $o'$, like $o$ is in $\Sigma_T$, it

follows by Lemma 5 that $sf_{o'} \xrightarrow{+} s_T$ in $e'$, and consequently that $sf_{o'} \xrightarrow{*} s_p(T)$. Setting $s_p(T)$ to $sl_o$, we conclude $sf_{o'} \xrightarrow{*} sl_o$ in $e'$.

  - $sl_o = s_p(\hat{T})$: Consider first the relationships between read states and commit states in $e$. By assumption, $e$ satisfies $\textsc{irc}_e(T)$, i.e. $sf_{o'} \xrightarrow{*} sl_o$ holds in $e$. Since $\hat{T}$ wrote the next version of the object that $o$ read, we have that $sl_o \xrightarrow{*} s_p(\hat{T}) \xrightarrow{+} s_{\hat{T}}$ in $e$. Combining the guarantee given by monotonic reads $sf_{o'} \xrightarrow{*} sl_o$ and $sl_o \xrightarrow{+} s_{\hat{T}}$, it follows that $sf_{o'} \xrightarrow{+} s_{\hat{T}}$ in $e$ i.e. $s_{\hat{T}'} \xrightarrow{+} s_{\hat{T}}$. Returning to the execution $e'$, since $\hat{T}'$ and $\hat{T}$ are both update transactions, $\mathcal{W}_{\hat{T}'} \neq \emptyset \wedge \mathcal{W}_{\hat{T}} \neq \emptyset$, if $s_{\hat{T}'} \xrightarrow{+} s_{\hat{T}}$ in $e$, then $s_{\hat{T}'} \xrightarrow{+} s_{\hat{T}}$ and consequently $sf_{o'} \xrightarrow{+} s_{\hat{T}}$ . Given $e'$ is a total order and $s_p(\hat{T}) \rightarrow s_{\hat{T}}$, we conclude $sf_{o'} \xrightarrow{*} s_p(\hat{T})$, and $sf_{o'} \xrightarrow{*} sl_o$ in $e'$, as desired.

- Let us next assume that $o$ is a write operation. The candidate read set of a write operation $o = w(k, v)$ is $\mathcal{RS}_{e'}(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p\}$, where, consequently, $sl_o = s_p(T)$. We previously showed that $\textsc{preread}_{e'}(\mathcal{T})$. Given that $o'$, like $o$ is in $\Sigma_T$, it follows by Lemma 5 that $sf_{o'} \xrightarrow{+} s_T$ in $e'$, and consequently that $sf_{o'} \xrightarrow{*} s_p(T)$. Setting $_p(T)$ to $sl_o$, we conclude $sf_{o'} \xrightarrow{*} sl_o$ in $e'$.

**Write** The candidate read state set for a write operation $o' = w(k, v)$ is defined as the set of all states before $T'$'s commit state. Hence $sf_{o'} = s_0$. Thus $sf_{o'} \xrightarrow{*} sl_o$ trivially holds.

We conclude that $\textsc{irc}_{e'}(T)$ holds.

This completes the proof that Monotonic Reads holds for execution $e'$. This was the last outstanding session guarantees: we have then proved that $e'$ satisfies all four session guarantees.

We now proceed to prove that $e'$ satisfies causal consistency: $\forall T \in \mathcal{T}_{se}$ : SESSION$_{cc}$(*se*, *T*, *e'*). More specifically, we must prove that: $\textsc{preread}_{e'}(\mathcal{T}) \wedge \textsc{irc}_{e'}(T) \wedge (\forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} sl_o) \wedge (\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j})$. We proceed by proving that each of the clauses holds.

The first two clauses are easy to establish. We previously proved that $\textsc{preread}_{e'}(\mathcal{T})$ holds. Likewise, $\textsc{irc}_{e'}(T)$ holds as $\forall T \in \mathcal{T}_{se}$ : SESSION$_{MR}$(*se*, *T*, *e'*). To establish the fourth clause, we note that we previously proved that $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$.

We are then left to prove only the third clause; namely we must prove that $\forall T_j \in \mathcal{T}_{se} : (\forall o_j \in \Sigma_{T_j} : \forall T_i \xrightarrow{se} T_j : s_{T_i} \xrightarrow{*} sl_{o_j})$.

We distinguish between two cases: $T_i$ is an update transaction and $T_i$ is a read-only transaction. If $T_i$ is an update transaction, the desired result holds as $e'$ guarantees Read-My-Writes: $\forall T_j \in \mathcal{T}_{se} : (\forall o_j \in \Sigma_{T_j} : \forall T_i \xrightarrow{se} T_j : \mathcal{W}_{T_i} \neq \emptyset \Rightarrow s_{T_i} \xrightarrow{*} sl_{o_j})$

If $T_i$ is a read-only transaction, we proceed by induction. We consider an arbitrary $T_j$, and an arbitrary $o_j \in \Sigma_{T_j}$.

**Base Case** Consider the first read-only transaction $T_i$ in $se'$ such that $T_i \xrightarrow{se'} T_j$. Recall that we choose the parent state of a read-only transaction as $s_p(T_i) = \max\{\max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}, s_{T_{i-1}}\}$, where $T_{i-1}$ denotes the transaction that directly precedes $T_i$ in session $se'$ ($s_{T_{i-1}} = s_0$ if $T_i$ is the first transaction in the session). Hence, $T_i$'s parent state is either $s_p(T_i) = \max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}$, or $s_p(T_i) = s_{T_{i-1}}$.

- If $s_p(T_i) = \max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}$: We previously proved that $\forall T \in \mathcal{T}_{se} : \text{SESSION}_{MR}(se, T, e')$, hence that $\forall T_i \xrightarrow{se} T_j : \forall o_i \in \Sigma_{T_i} : sf_{o_i} \xrightarrow{*} sl_{o_j}$, and consequently $\max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\} \xrightarrow{*} sl_{o_j}$ in $e'$. Noting that $s_p(T_i) = \max_{o_i \in \Sigma_{T_i}}\{sf_{o_i}\}$, we obtain the desired result $s_p(T_i) \xrightarrow{*} sl_{o_j}$ in $e'$.
- If $s_p(T_i) = s_{T_{i-1}}$. We defined $T_i$ to be the first read-only transaction in the session. By construction $T_{i-1} \xrightarrow{se'} T_i$, $T_{i-1}$ is necessarily an update transaction given $T_i$ is the first read-only transaction, where $\mathcal{W}_{T_{i-1}} \neq \emptyset$. Given that, by transitivity $T_{i-1} \xrightarrow{se} T_j$, and that $e'$ guarantees Read-My-Writes $\forall T \in \mathcal{T}_{se} : \text{SESSION}_{RMW}(se, T, e')$, we have $s_{T'_{i-1}} \xrightarrow{*} sl_{o_j}$ in $e'$. Noting that $s_p(T_i) = s_{T_{i-1}}$, we conclude $s_p(T_i) \xrightarrow{*} sl_{o_j}$ in $e'$.

Finally, we argue that $s_p(T_i) \neq sl_{o_j}$ (and therefore that $s_{T_i} \xrightarrow{*} sl_{o_j}$ as $s_p(T_i) \rightarrow s_{T_i}$). Read-only transactions, like $T_i$ do not change the state on which they are applied, hence $\forall (k, v) \in s_p(T_i) \Rightarrow (k, v) \in s_{T_i}$. Moreover, by Claim 4, transactions commit in session order: $\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j}$. We thus have $s_{T_i} \xrightarrow{+} s_{T_j}$ and consequently $s_p(T_i) \in \mathcal{RS}_{e'}(o_j) \Rightarrow s_{T_i} \in \mathcal{RS}_{e'}(o_j)$, i.e. $s_p(T_i) \neq sl_{o_j}$: if $s_p(T_i)$ is in $\mathcal{RS}_{e'}(o_j)$, so is $s_{T_i}$. As $s_{T_i}$ follows $s_p(T_i)$ in the execution, $s_p(T_i)$ will never be $sl_{o_j}$. The following thus holds in the base case: $s_p(T_i) \xrightarrow{+} sl_{o_j}$, i.e. $s_p(T_i) \rightarrow s_{T_i} \xrightarrow{*} sl_{o_j}$.

**Induction Step** Consider the k-th read-only transaction $T_i$ in $se'$ such that $T_i \xrightarrow{se'} T_j$. We assume that it satisfies the induction hypothesis $s_{T_i} \xrightarrow{*} sl_{o_j}$. Now consider the (k+1)-th read-only transaction $T_{i'}$ in $se'$, such that $T_i \xrightarrow{se'} T_{i'}$. By construction, we once again distinguish two cases: $T_{i'}$'s parent state is either $s_p(T_{i'}) = \max_{o_{i'} \in \Sigma_{T'_i}}\{sf_{o'_i}\}$, or $s_p(T_{i'}) = s_{T_{i'-1}}$, where $T_{i'-1}$ denotes the transaction directly preceding $T_{i'}$ in a session.

1. If $s_p(T_{i'}) = \max_{o_{i'} \in \Sigma_{T_{i'}}}\{sf_{o_{i'}}\}$. We previously proved that $\forall T \in \mathcal{T}_{se}$ : SESSION$_{MR}$(*se*, *T*, *e*’), hence that $\forall T_{i'} \xrightarrow{se} T_j : \forall o_{i'} \in \Sigma_{T_{i'}} : sf_{o_{i'}} \xrightarrow{*} sl_{o_j}$, and consequently $\max_{o_{i'} \in \Sigma_{T_{i'}}}\{sf_{o_{i'}}\} \xrightarrow{*} sl_{o_j}$ in $e'$. Noting that $s_p(T_{i'}) = \max_{o_{i'} \in \Sigma_{T_{i'}}}\{sf_{o_{i'}}\}$, we obtain the desired result $s_p(T_{i'}) \xrightarrow{*} sl_{o_j}$ in $e'$.

2. If $s_p(T_{i'}) = s_{T_{i'-1}}$: First, we note that $T_{i'-1} \xrightarrow{se'} T_j$ holds, as the session order is transitive and we have both $T_{i'-1} \xrightarrow{se'} T_{i'}$ and $T_{i'} \xrightarrow{se'} j$. We distinguish between two cases: if $T_{i'-1}$ is a read-only transaction, then it must be the k-th such transaction (as, by construction, it directly precedes $T_{i'}$ in the session). Hence $T_{i'-1} = T_i$. Our induction hypothesis states that $s_{T_i} \xrightarrow{*} sl_{o_j}$, and consequently that $s_{T_{i'-1}} \xrightarrow{*} sl_{o_j}$. Noting that $s_p(T_{i'}) = s_{T_{i'-1}}$, we obtain $s_p(T_{i'}) \xrightarrow{*} sl_{o_j}$. If $T_{i'-1}$ is an update transaction, we note that $e'$ guarantee Read-My-Writes: $\forall T \in \mathcal{T}_{se}$ : SESSION$_{RMW}$(*se*, *T*, *e*’). As $T_{i'-1} \xrightarrow{se} T_j$, we have $s_{T_{i'-1}} \xrightarrow{*} sl_{o_j}$ in $e'$. Noting that $s_p(T_{i'}) = s_{T_{i'-1}}$, we conclude: $s_p(T_{i'}) \xrightarrow{*} sl_{o_j}$ in $e'$.

Finally, we argue that $s_p(T_{i'}) \neq sl_{o_j}$ (and therefore that $s_{T_{i'}} \xrightarrow{*} sl_{o_j}$ as $s_p(T_{i'}) \rightarrow s_{T_{i'}}$). Read-only transactions, like $T_{i'}$ do not change the state on which they are applied, hence $\forall (k, v) \in s_p(T_{i'}) \Rightarrow (k, v) \in s_{T_{i'}}$. Moreover, by Claim 4, transactions commit in session order: $\forall se' \in SE : \forall T_{i'} \xrightarrow{se'} T_j : s_{T_{i'}} \xrightarrow{+} s_{T_j}$. We thus have $s_{T_{i'}} \xrightarrow{+} s_{T_j}$ and consequently $s_p(T_{i'}) \in \mathcal{RS}_{e'}(o_j) \Rightarrow s_{T_{i'}} \in \mathcal{RS}_{e'}(o_j)$, i.e. $s_p(T_{i'}) \neq sl_{o_j}$: if $s_p(T_{i'})$ is in $\mathcal{RS}_{e'}(o_j)$, so is $s_{T_{i'}}$. As $s_{T_{i'}}$ succedes $s_p(T_{i'})$ in the execution, $s_p(T_{i'})$ will never be $sl_{o_j}$. The following thus holds in the induction case: $s_p(T_{i'}) \xrightarrow{+} sl_{o_j}$, i.e. $s_p(T_{i'}) \rightarrow s_{T_{i'}} \xrightarrow{*} sl_{o_j}$. We proved the desired result for both the base case and induction step. By induction, we conclude that, for read-only transactions: $\forall T_j \in \mathcal{T}_{se}, \forall o_j \in \Sigma_{T_j} : \forall T_i \xrightarrow{se} T_j, \mathcal{W}_{T_i} = \emptyset \Rightarrow s_{T_i} \xrightarrow{*} sl_{o_j}$. Hence, the desired result holds for both read-only and update transactions $\forall T_j \in \mathcal{T}_{se} : (\forall o_j \in \Sigma_{T_j} : \forall T_i \xrightarrow{se} T_j : s_{T_i} \xrightarrow{*} sl_{o_j})$.

**Conclusion** Putting everything together, if $e$ guarantees all four session guarantees, there exists an alternative execution $e'$ such that $e'$ also satisfies the session guarantees and is causally consistent: $\text{PREREAD}_e(\mathcal{T}) \wedge \text{IRC}_e(T) \wedge (\forall o \in \Sigma_T : \forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} sl_o) \wedge (\forall se' \in SE : \forall T_i \xrightarrow{se'} T_j : s_{T_i} \xrightarrow{+} s_{T_j})$, i.e. $\forall T \in \mathcal{T}_{se}$ : SESSION$_{CC}(se, T, e')$. This completes the second part of the proof.

Consequently, $\forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se}$ : SESSION$_{\mathcal{G}}(se, T, e) \equiv \forall se \in SE : \exists e : \forall T \in \mathcal{T}_{se}$ : SESSION$_{CC}(se, T, e)$ holds.

## H Formal Security

We now provide formal security definitions and proofs for Obladi. As we discuss in §6.8, we use the Universal Composability (UC) framework [43]. The UC framework requires us to specify an ideal functionality $\mathcal{F}_{Ob}$ that defines what it means for Obladi to be secure. We must then prove that, for every possible adversarial algorithm $\mathcal{A}$ specifying the behavior of the storage server, we can simulate $\mathcal{A}$'s behavior when interacting only with $\mathcal{F}_{Ob}$.

We prove security of the scheme without assuming that the cloud storage provider is trusted for integrity. As the MACs and counters are only used to verify integrity and freshness of data, they are unnecessary if the cloud server is being honest. As we will see below, removing them—as we do in our implementation—does not impact security in this case.

We also noted in Section 6.9 that the proxy requires a trusted epoch counter that persists across crashes. This could be implemented as an integer in local non-volatile storage that the proxy updates with each epoch, it could be implemented by trusting the cloud storage for integrity and saving it there, or other means. We abstract away this detail by providing the Obladi protocol with access to $\mathcal{F}_{epc}$, an ideal functionality that provides access to this counter.

### H.1 Ideal Functionality

We begin by noting that Obladi's proxy acts as a trusted central coordinator that performs publicly-known logic on private data. As this is essentially the role played by any ideal functionality, we simply subsume the proxy into $\mathcal{F}_{Ob}$. Moreover, some of the proxy's behavior, like the fact that it deduplicates and caches accesses, pads under-full batches, is public information, meaning $\mathcal{F}_{Ob}$ can explicitly perform exactly the same operations.

In §6.4 we describe the proxy as consisting of a concurrency control unit and a data manager, which itself contains a batch manager and ORAM executor. As the concurrency control and batch management functionalities do not inherently leak any information, we define $\mathcal{F}_{Ob}$ in terms of those operations. In particular, we let $\mathcal{F}^*_{Proxy}$ represent this functionality. $\mathcal{F}^*_{Proxy}$ is defined as providing the exact functionality of the concurrency control unit and batch manager as described in §6.4 and

§6.5. $\mathcal{F}^*_{Proxy}$ has the following ways to interface with $\mathcal{F}_{Ob}$:

- $\mathcal{F}_{Ob}$ can supply $\mathcal{F}^*_{Proxy}$ with an input from a client (start, read, write, or commit).
- $\mathcal{F}^*_{Proxy}$ can produce a read batch of logical data blocks. The batch need not be full, meaning it may contain fewer than the maximum number of reads for a batch. $\mathcal{F}_{Ob}$ can then respond with the requested blocks.
- $\mathcal{F}^*_{Proxy}$ can produce a write batch of logical data blocks. The batch need not be full. $\mathcal{F}_{Ob}$ can then respond confirming the writes have completed.
- $\mathcal{F}^*_{Proxy}$ can specify an epoch has ended and transactions should commit. $\mathcal{F}_{Ob}$ can then respond with confirmation.
- $\mathcal{F}_{Ob}$ can clear $\mathcal{F}^*_{Proxy}$'s internal state, representing a crash.

$\mathcal{F}^*_{Proxy}$ can additionally send a messages directly to clients.

**Modeling Crashes** In the real system the proxy can crash at any time. As all state except the cryptographic keys (and possibly trusted counters) is considered volatile, it does not matter when during a local operation the proxy crashes, as every piece of that operation is lost regardless. We can therefore simplify the ideal functionality by allowing for crashes both between requests and immediately prior to any operation within a request that either leaves the proxy (e.g., writing to cloud storage) or persists across crashes (e.g., updating the trusted epoch counter).

To model any possible crash, we control the timing of the crashes through a Crash Client. $\mathcal{F}_{Ob}$ queries the Crash Client immediately prior to any relevant action and waits for a reply. The Crash Client then waits for a prompt from the environment, which it forwards to $\mathcal{F}_{Ob}$, telling it to proceed or crash. Additionally, the Crash Client—again at the prompting of the environment—can issue a "crash" command independently between requests.

We provide the full specification for $\mathcal{F}_{Ob}$ in Algorithm 1, which references $\mathcal{F}^*_{Proxy}$. For notational clarity, we do not explicitly specify every call to the Crash Client. Instead any operation prefixed by $^\dagger$ notifies the Crash Client before executing and crashes if instructed. Note that it is possible to crash while recovering from a crash.

**Algorithm 1:** Ideal functionality $\mathcal{F}_{Ob}$ using $\mathcal{F}^{*}_{Proxy}$.

```
Data: $D$ = DatabaseState
Data: Counters $c_e = 0$ ; $c_b = 0$

Initialize
    Initialize $\mathcal{F}^{*}_{Proxy}$
    Begin epoch
end

On receive $m$ from client $\mathcal{C}$
    Forward $(m, \mathcal{C})$ to $\mathcal{F}^{*}_{Proxy}$
end

On receive "read-batch$[blks]$" from $\mathcal{F}^{*}_{Proxy}$
    †Send "read-batch-init" to $\mathcal{A}$, wait for "OK"
    †$c_b \leftarrow c_b + 1$
    †Send "read-batch-read" to $\mathcal{A}$, wait for "OK"
    Read $blks$ from $D$
    Respond to $\mathcal{F}^{*}_{Proxy}$ with results
end

On receive "write-epoch$[data]$" from $\mathcal{F}^{*}_{Proxy}$
    †Send "write-epoch" to $\mathcal{A}$, wait for "OK"
    †$c_e \leftarrow c_e + 1$ ; $c_b \leftarrow 0$
    Write $data$ to $D$
    Confirm write/epoch completed to $\mathcal{F}^{*}_{Proxy}$
end

On receive "crash" from Crash Client
    Execute crashRecover
end

function crashRecover
    Send ("crash", $c_e, c_b$) to $\mathcal{A}$
    Clear internal state of $\mathcal{F}^{*}_{Proxy}$
    Rollback writes to $D$ since beginning of epoch $c_e$
    †$c_e \leftarrow c_e + 1$ ; $c_b \leftarrow 0$
end
```

†Before executing operation, notify Crash Client. On response of "crash," abort operation and invoke `crashRecover`, otherwise proceed.

## H.2 Security Lemmas

In order to prove the security of Obladi, we rely on two lemmas which we alluded to in §6.8.

**Lemma 7** (Caching and Deduplication). *Let D be any set of logical reads or writes selected* *independently from the current ORAM position map. Let $D^*$ be the set of accesses resulting from applying the proxy batch manager's caching and deduplication logic to $D$. The set of physical accesses needed to realize $D^*$ is identically distributed to the set of physical accesses needed to realize a uniformly random set of logical accesses of the same size.*

*Proof.* Since $D$ is selected independently from the current position map in the ORAM, Ring ORAM guarantees that the set of physical accesses needed to realize $D$ is identically distributed to that for a uniformly random set of logical reads or writes. $D^*$ is simply $D$ with some elements removed, so we claim that the elements removed form an unbiased sample. Since removing an unbiased sample from a distribution does not change the distribution, this is sufficient.

We first note that Ring ORAM guarantees that any independently-selected logical access $d$ results in physical accesses sampled independently from the following distribution. First sample a uniformly random path in the tree. Then, for each bucket in that path, sample a uniformly random block from among those not read since the bucket was last written. Finally, read all selected blocks.

In Ring ORAM, whenever a block is read or written, it is immediately remapped to an independent uniformly random path in the tree that determines what will be read next time it is accessed. The proxy batch manger's caching and deduplication logic removes access requests for any block previously accessed in this epoch. Each of those blocks was mapped to a new independently uniform random path when accessed. Moreover, when an epoch ends, the cache is completely flushed, meaning there is no (potentially-biased) caching or deduplication.

Thus the sample of physical accesses removed by pairing $D$ down to $D^*$ must be unbiased, so $D^*$ must result in a uniformly random set of physical access paths. □

**Lemma 8** (Parallel ORAM)**.** *The set of parallel physical data operations performed by the proxy ORAM executor over one epoch (as described in §6.6) is completely determined by the set of sequential physical accesses required to perform the same logical actions in Ring ORAM (plus a single write to the durability store).*

*Proof.* We note that, as described in §6.6, the proxy performs all reads within an epoch before any writes (aside from the durability store). By construction, it ensures that each physical block that would be read at least once within an epoch in a fully sequential access is read exactly once in that epoch, and no other physical blocks are ever read (excluding crash recovery).

This is enforced by holding a record of every block that has been read this epoch and then performing the reads of the sequential access, but skipping blocks that have already been read. Additionally, whenever an evict path operation would happen, the proxy reads every unread block from each bucket along that path, thus marking them as read. As the timing of evict paths is determined by how many data accesses have happened and their locations are deterministic, this enforcement mechanism is dependent only on the physical blocks accessed, not in any way on the data held in those blocks.

Similarly, each block that would be written at least once in a sequentially-processed epoch is written exactly once at the end of the epoch. This is done by buffering writes in the proxy, allowing one buffered write of a physical block to overwrite any previous unflushed writes of that block. Then when the epoch ends, the proxy flushes all buffered writes. Again, the set of blocks being written is determined entirely by the physical access pattern of the sequential operation.

Finally, a fixed amount of data is written to the durability store before each read batch, and the entire durability store is written with each write batch. This means that in normal operation, the location and timing of all reads and writes are determined by only the physical operations needed to perform the epoch operations sequentially and some extra completely deterministic operations.

On crash recovery, the proxy reads the durability store and rereads all paths in the aborted epoch. This, again, is based entirely on physical access patterns.

Hence all physical read and write operations within a parallelized epoch are determined entirely by the physical data operations needed to perform that epoch sequentially. □

## H.3 Proof of Security

We now prove that the Obladi protocol $\Pi_{Ob}$ (with access to $\mathcal{F}_{epc}$) is secure with respect to the ideal functionality described in Algorithm 1. Let $\mathsf{Real}_{\mathcal{A},\mathcal{E}}(\lambda)$ denote the full transcript of $\mathcal{A}$ (including its inputs and randomness) when interacting with $\Pi_{Ob}$. Let $\mathsf{Ideal}_{\mathcal{S},\mathcal{E}}(\lambda)$ denote the transcript produced by $\mathcal{S}$ when run in the ideal world, interacting with $\mathcal{F}_{Ob}$.

**Theorem 7.** *Assume the encryption scheme used in $\Pi_{Ob}$ is semantically secure and the MACs are existentially unforgeable. For all probabilistic polynomial time (PPT) adversaries $\mathcal{A}$ and environments $\mathcal{E}$, there is a simulator $\mathcal{S}_{\mathcal{A}}$ such that for all PPT distinguishers $\mathcal{D}$ there is some negligible function $negl$ such that*

$$\Big| \Pr\Big[\mathcal{D}\Big(\mathsf{Real}_{\mathcal{A},\mathcal{E}}(\lambda)\Big) = 1\Big] \\ - \Pr\Big[\mathcal{D}\Big(\mathsf{Ideal}_{\mathcal{S}_{\mathcal{A}},\mathcal{E}}(\lambda)\Big) = 1\Big]\Big| \leq negl(\lambda).$$

*Proof.* This proof follows from a series of hybrid simulators, each of which is indistinguishable from the previous.

We define hybrids $H_0, \ldots, H_4$. $H_0$ operates in the real world with $\mathcal{S}_0$ being a "dummy" that passes all messages through to $\mathcal{A}$ unmodified. $H_1$ has two ORAMs that are identical except for the MACs, one maintained by $\mathcal{A}$ and the other maintained by $\mathcal{S}_1$. $H_2$ replaces all data in $\mathcal{A}$'s ORAM with random dummy data, independent from the actual data. $H_3$ replaces the access pattern in $\mathcal{A}$'s ORAM with random data accesses. Finally $H_4$ uses $\mathcal{S}_{\mathcal{A}}$ in the ideal world and no longer maintains its own ORAM.

**Hybrid** $H_0$ contains a dummy simulator that passes messages between $\mathcal{A}$ and the proxy unchanged. This produces a transcript identical to the real world.

**Hybrid** $H_1$ passes all messages through to $\mathcal{A}$, but also maintains its own copy of the ORAM, simultaneously processes requests internally. On initialization $\mathcal{S}_1$ generates its own MAC key according to the same distribution as $\Pi_{Ob}$'s MAC key. It then replaces the MACs of all data sent to $\mathcal{A}$ with valid MACs on the same data using this new key. When $\mathcal{A}$ responds to a request, $\mathcal{S}_1$ checks the MACs on the data. If they are correct, it forwards the (correct) response from it's own ORAM with the original MACs. If they are incorrect, it responds with a failure message. If $\mathcal{A}$'s response is correct, so too will $\mathcal{S}_1$'s. If $\mathcal{A}$'s MACs to not verify, $\Pi_{Ob}$ fails, so a failure message produces the same result. If $\mathcal{A}$'s response is wrong but the MACs verify, $\mathcal{A}$ must have forged a MAC since they include the data, position, and epoch counter, and no two pieces of data are ever given the same position and epoch counter. Moreover, because $\Pi_{Ob}$ has access to a trusted epoch counter via $\mathcal{F}_{epc}$, it can properly verify that the data has the correct epoch counter, even after crashes. Thus, if $\mathcal{S}_1$ accepts an incorrect response with non-negligible probability, we can simulate $\mathcal{A}$ to forge a MAC with non-negligible probability. Hence $H_1$ is computationally indistinguishable from $H_0$.

Note that the MACs are only used to check that $\mathcal{A}$ provided correct data. If the storage server is assumed to be honest, this will always be the case and we can eliminate the MACs entirely (and also $H_0$ and $H_1$ become identical).

**Hybrid** $H_2$ replaces all data blocks provided to $\mathcal{A}$ with valid encryptions of random data and MACs on those encryptions. It otherwise passes on requests, including the location and timing of reads and writes. $\mathcal{S}_2$ continues to furnish responses to the proxy's queries using its internal ORAM with the original data, checking MACs according to the same scheme as in $H_1$. $\mathcal{S}_2$ then output's $\mathcal{A}$'s transcript. As all data is encrypted, the only difference between $H_1$ and $H_2$ is the contents of the ciphertexts, and by assumption the encryption scheme is semantically secure. This means $H_1$ and $H_2$ must be computationally indistinguishable.

**Hybrid** $H_3$ replaces all data requests to $\mathcal{A}$ with properly-formatted requests for randomly chosen data.

When $\mathcal{S}_3$ receives a location log for a read batch, it logs an encryption of random (unrelated) data with $\mathcal{A}$. When $\mathcal{S}_3$ receives the read instruction for a read batch, it first selects a random set of dummy paths of the batch size. It then requests $\mathcal{A}$ perform the proper parallel read operation for that dummy data. If $\mathcal{A}$ replies with the data and the MACs verify, $\mathcal{S}_3$ performs the actual reads on its separate ORAM with real data and returns the real data to the proxy.

When $\mathcal{S}_3$ is notified of the end of an epoch and given the associated write batch, it determines which physical blocks to write using Ring ORAM's deterministic write sequence based on the total number of operations (both reads and writes) in an epoch. It then performs proper parallel writes of new encryptions of dummy data to each of those locations. If $\mathcal{A}$ replied with confirmed writes, $\mathcal{S}_3$ performs the originally-specified operations on its separate ORAM and confirms success to the proxy.

Finally, if $\mathcal{S}_3$ receives a request to handle a proxy crash at epoch $c_e$ and batch $c_b$, it queries $\mathcal{A}$ as per the crash recovery protocol for that epoch and batch. When $\mathcal{A}$ provides valid (MAC-verifying) read path logs for any batches this epoch, $\mathcal{S}_3$ provides the associated logs to the proxy. When the proxy issues redo read requests, $\mathcal{S}_3$ issues the same requests it did the first time to $\mathcal{A}$ for the associated batches. Because $\mathcal{S}_3$ did not crash, it is able to retain which paths were read without having to store them explicitly. In is possible that the last read batch requested during recovery corresponds to a read that was never executed, in which case $\mathcal{S}_3$ generates a new random read batch and executes that instead. If $\mathcal{A}$ responds correctly, $\mathcal{S}_3$ responds to the proxy's requests.

By Lemma 7, the physical operations needed to process all real requests in a given epoch sequentially form an identical distribution to the sequential accesses needed to process the random requests chosen by $\mathcal{S}_3$. By Lemma 8, applying the parallelization process relies only on the sequential physical access pattern, meaning it can be applied the same way to $\mathcal{S}_3$'s random operations as to the real operations provided by the proxy. This means that the operations $\mathcal{S}_3$ requests of $\mathcal{A}$ are identically distributed to those the proxy requests of $\mathcal{S}_3$ when there are no crashes.

When a crash occurs, the recovery procedure is guaranteed to reread all previously-read data, and any future reads must have independently random paths. This is because $\mathcal{S}_3$ does not even generate random paths to read until the read request is issued, by which point the persistent batch counter $c_b$ is updated. So if a crash does occur, it will redo any previous reads and future operations are treated as regular read/write batches with the same (independent) distribution. Since these are the only difference between $H_2$ and $H_3$, the two must produce identical distributions.

**Hybrid** $H_4$ now interacts with the ideal functionality and no longer maintains its own internal ORAM copy, only the data necessary to perform actions on $\mathcal{A}$'s, including the new MAC and encryption keys. The only data $\mathcal{S}_3$ was using to compute requests for $\mathcal{A}$ was the timing of batches and crash recoveries, and the epoch and batch counters during recovery. As $\mathcal{F}_{Ob}$ explicitly provides all of that information, $\mathcal{S}_4$ is able to provide $\mathcal{A}$ with an identical view. Note that on crash recovery, this identical view requires completing a crash-recover epoch, which $\mathcal{S}_4$ can do by creating an appropriate number of read and write operations as it would in $H_3$. This means that $H_3$ and $H_4$ are identically distributed.

Thus we see that $H_0$ corresponds to the real world, $H_4$ corresponds to the ideal world, and each sequential pair of $(H_i, H_{i+1})$ produce computationally indistinguishable transcripts. Thus it must be the case that $H_0$ and $H_4$ form computationally indistinguishable transcripts, so $\Pi_{Ob}$ realizes

$\mathcal{F}_{Ob}$. $\square$

# Bibliography

[1] Mysql. `http://www.mysql.com`.

[2] Gossip-based computer networking. *ACM SIGOPS Operating Systems Review 41*, 5 (2007).

[3] 23, AND ME. 23andme. `www.23andme.com`.

[4] ADYA, A. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, Cambridge, MA, USA, Mar. 1999. Also as Technical Report MIT/LCS/TR-786.

[5] ADYA, A., AND LISKOV, B. Lazy consistency using loosely synchronized clocks. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1997), PODC '97, ACM, pp. 73–82.

[6] AGUILAR-MELCHOR, C., BARRIER, J., FOUSSE, L., AND KILLIJIAN, M.-O. XPIR: Private Information Retrieval for Everyone. Cryptology ePrint Archive, Report 2014/1025, 2014. `http://eprint.iacr.org/2014/1025`.

[7] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pp. 159–174.

[8] AGUILERA, M. K., SPENCE, S., AND VEITCH, A. Olive: Distributed point-in-time branching storage for real systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation - Volume 3*, NSDI '06.

[9] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. Causal memory:

Definitions, implementation and programming. Tech. rep., Georgia Institute of Technology, 1994.

[10] Ajoux, P., Bronson, N., Kumar, S., Lloyd, W., and Veeraraghavan, K. Challenges to adopting stronger consistency at scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (2015), HOTOS'15.

[11] Alvaro, P., Bailis, P., Conway, N., and Hellerstein, J. M. Consistency without borders. In *Proceedings of the 4th ACM Symposium on Cloud Computing*, SOCC '13, pp. 23:1–23:10.

[12] Amazon. Elastic beanstalk. `https://aws.amazon.com/elasticbeanstalk/`.

[13] Amazon. S3: Simple storage service. `https://aws.amazon.com/s3/`.

[14] Amazon. Simple db. `https://aws.amazon.com/simpledb/`.

[15] Amazon elastic compute cloud. `http://aws.amazon.com/ec2/`, Mar. 2009.

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

[17] Arasu, A., Blanas, S., Eguro, K., Kaushik, R., Kossmann, D., Ramamurthy, R., and Venkatesan, R. Orthogonal Security With Cipherbase. In *Conference on Innovative Data Systems Research (CIDR)* (2013).

[18] Arasu, A., Eguro, K., Kaushik, R., Kossmann, D., Meng, P., Pandey, V., and Ramamurthy, R. Concerto: A High Concurrency Key-Value Store with Integrity. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2017).

[19] Ardekani, M. S., Sutra, P., and Shapiro, M. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*, SRDS '13, pp. 163–172.

[20] Attiya, H., Ellen, F., and Morrison, A. Limitations of highly-available eventually-consistent data stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (2015), PODC '15, ACM, pp. 385–394.

[21] AZURE, M. Decentralized Identity. https://azure.microsoft.com/en-us/overview/decentralized-identity.

[22] BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Highly available transactions: Virtues and limitations. *PVLDB 7*, 3 (2013), 181–192.

[23] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, pp. 1327–1342.

[24] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SOCC '12, pp. 22:1–22:7.

[25] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems 41*, 3 (July 2016), 15:1–15:45.

[26] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13.

[27] BAJAJ, S., AND SION, R. TrustedDB: A Trusted Hardware Based Database with Privacy and Data Confidentiality. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2011).

[28] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)* (2011).

[29] BASHO. Riak. http://basho.com/products/.

[30] BEIMEL, A., ISHAI, Y., AND MALKIN, T. Reducing the servers' computation in private

information retrieval: PIR with preprocessing. *Journal of Cryptology (JOFC) 17*, 2 (2004), 125–151.

[31] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record* (1995), vol. 24, pp. 1–10.

[32] BERNSTEIN, P. A., AND GOODMAN, N. Multiversion concurrency control;theory and algorithms. *ACM Transactions on Database Systems 8*, 4 (1983), 465–483.

[33] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency control and recovery in database systems*. 1987.

[34] BERNSTEIN, P. A., ROTHNIE, J., GOODMAN, N., AND PAPADIMITRIOU, C. A. Some computational problems related to database concurrency control.

[35] BERNSTEIN, P. A., ROTHNIE, J., GOODMAN, N., AND PAPADIMITRIOU, C. A. The concurrency control mechanism of sdd-1: A system for distributed databases (the fully redundant case). *IEEE Transactions on Software Engineering*, 3 (1978), 154–168.

[36] BERNSTEIN, P. A., SHIPMAN, D. W., AND WONG, W. S. Formal aspects of serializability in database concurrency control. *IEEE Trans. Softw. Eng. 5*, 3 (May 1979), 203–216.

[37] BINDSCHAEDLER, V., NAVEED, M., PAN, X., WANG, X., AND HUANG, Y. Practicing Oblivious Access on Cloud Storage: The Gap, the Fallacy, and the New Way Forward. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[38] BOYLE, E., CHUNG, K.-M., AND PASS, R. Oblivious Parallel RAM and Applications. In *Theory of Cryptography Conference (TCC)* (2016).

[39] BREWER, E. A. Towards robust distributed systems (abstract). In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, PODC '00.

[40] BRZEZINSKI, B., SOBANIEC, C., AND D., W. From session causality to causal consistency. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network based Processing* (2004), PDP 2004.

[41] BULCK, J. V., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F.,

SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium (USENIX)* (2018).

[42] CACHIN, C., KEIDAR, I., AND SHRAER, A. Trusting the cloud. *SIGACT News 40*, 2 (June 2009), 81–86.

[43] CANETTI, R. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science (FOCS)* (2001).

[44] CASSANDRA. Use Cases. `http://www.planetcassandra.org/apache-cassandra-use-cases/`.

[45] CECCHETTI, E., ZHANG, F., JI, Y., KOSBA, A., JUELS, A., AND SHI, E. Solidus: Confidential Distributed Ledger Transactions via PVORM. In *ACM Conference on Computer and Communications Security (CCS)* (2017).

[46] CERONE, A., BERNARDI, G., AND GOTSMAN, A. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015,* (2015).

[47] CERONE, A., AND GOTSMAN, A. Analysing snapshot isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (2016), PODC '16, ACM, pp. 55–64.

[48] CERONE, A., GOTSMAN, A., AND YANG, H. *Transaction Chopping for Parallel Snapshot Isolation*. DISC'15. 2015, pp. 388–404.

[49] At LAX, computer glitch delays 20,000 passengers. http://travel.latimes.com/articles/la-trw-lax12aug12.

[50] CHOCKLER, G., FRIEDMAN, R., AND VITENBERG, R. Consistency conditions for a corba caching service. In *Proceedings of the 14th International Conference on Distributed Computing* (London, UK, UK, 2000), DISC '00, Springer-Verlag, pp. 374–388.

[51] CHOR, B., GILBOA, N., AND NAOR, M. Private information retrieval by keywords, 1997.

[52] Chor, B., Kushilevitz, E., Goldreich, O., and Sudan, M. Private Information Retrieval. *Journal of the ACM (JACM) 45*, 6 (1998), 965–981.

[53] Cloud, C. 5 advantages of a cloud-based EHR.

[54] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[55] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)* (2010).

[56] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pp. 251–264.

[57] Crooks, N., Burke, M., Cecchetti, E., Harel, S., Agarwal, R., and Alvisi, L. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 727–743.

[58] Crooks, N., Pu, Y., Alvisi, L., and Clement, A. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *ACM Symposium on Principles of Distributed Computing (PODC)* (2017).

[59] Crooks, N., Pu, Y., Estrada, N., Gupta, T., Alvisi, L., and Clement, A. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2016).

[60] Daudjee, K., and Salem, K. Lazy database replication with ordering guarantees. In

*Proceedings of the 20th International Conference on Data Engineering* (Washington, DC, USA, 2004), ICDE '04, IEEE Computer Society, pp. 424–.

[61] DAUDJEE, K., AND SALEM, K. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (2006), VLDB '06, VLDB Endowment, pp. 715–726.

[62] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pp. 205–220.

[63] DIFALLAH, D. E., PAVLO, A., CURINO, C., AND CUDRE-MAUROUX, P. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases.

[64] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pp. 4:1–4:13.

[65] DU, W., AND ELMAGARMID, A. Quasi serializability: A correctness criterion for global concurrency control in interbase. In *Proceedings of the 15th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1989), VLDB '89, Morgan Kaufmann Publishers Inc., pp. 347–355.

[66] DYNAMODB. DynamoDB. `https://aws.amazon.com/dynamodb/`.

[67] DYNAMODB. Encryption at rest. `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/EncryptionAtRest.html`.

[68] EDWARDS, W. K., MYNATT, E. D., PETERSEN, K., SPREITZER, M. J., TERRY, D. B., AND THEIMER, M. M. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the 10th annual ACM symposium on User interface software and technology* (New York, NY, USA, 1997), UIST '97, ACM, pp. 119–128.

[69] ESCRIVA, R., WONG, B., AND SIRER, E. G. Warp: Lightweight multi-key transactions for key-value stores. *CoRR abs/1509.07815* (2015).

[70] ESKANDARIAN, S., AND ZAHARIA, M. An Oblivious General-Purpose SQL Database for the Cloud. *CoRR abs/1710.00458* (2017).

[71] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.

[72] FALEIRO, J. M., ABADI, D., AND HELLERSTEIN, J. M. High performance transactions via early write visibility. *PVLDB 10*, 5 (2017), 613–624.

[73] FEKETE, A., LIAROKAPIS, D., O'NEIL, E., O'NEIL, P., AND SHASHA, D. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst. 30*, 2 (2005), 492–528.

[74] FELDMAN, A. J., ZELLER, W. P., FREEDMAN, M. J., AND FELTEN, E. W. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10.

[75] FLETCHER, C. W., REN, L., KWON, A., AND V. DI, M. A Low-Latency, Low-Area Hardware Oblivious RAM Controller. In *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2015).

[76] FLEXIBAKE. Flexibake. `https://www.flexibake.com/`.

[77] FREEHEALTH.IO. FreeHealth EHR. `https://freehealth.io/`. Accessed 2018-05-01.

[78] GARCIA-MOLINA, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst. 8*, 2 (June 1983), 186–213.

[79] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (June 2002), 51–59.

[80] GIT. Git: the fast version control system. `http://git-scm.com`.

[81] GOG, I., SCHWARZKOPF, M., CROOKS, N., GROSVENOR, M. P., CLEMENT, A., AND

HAND, S. Musketeer: All for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 2:1–2:16.

[82] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[83] GOOGLE. Bigtable - massively scalable nosql. https://cloud.google.com/bigtable/.

[84] GOOGLE. Cloud sql - fully managed sql service. https://cloud.google.com/sql/.

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

[86] GRAY, J., LORIE, R. A., PUTZOLU, G. R., AND TRAIGER, I. L. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems* (1976).

[87] GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys (CSUR) 13*, 2 (1981), 223–242.

[88] GRAY, J. N., LORIE, R. A., AND PUTZOLU, G. R. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases* (New York, NY, USA, 1975), VLDB '75, ACM, pp. 428–451.

[89] GUERRAOUI, R., AND KAPALKA, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP '08, ACM, pp. 175–184.

[90] GUPTA, T., CROOKS, N., MULHERN, W., SETTY, S., ALVISI, L., AND WALFISH, M. Scalable and private media consumption with popcorn. In *13th USENIX Symposium on*

*Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 91–107.

[91] GUY, R. G., HEIDEMANN, J. S., MAK, W., PAGE, JR., T. W., POPEK, G. J., AND ROTHMEIR, D. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference* (1990), pp. 63–72.

[92] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[93] IMBS, D., AND RAYNAL, M. Virtual world consistency: A condition for stm systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science 444* (July 2012), 113–127.

[94] INTEL. Intel Software Guard Extension - SGX. `https://software.intel.com/en-us/sgx`.

[95] ISHAI, Y., KUSHILEVITZ, E., OSTROVSKY, R., AND SAHAI, A. Batch Codes and Their Applications. In *ACM Symposium on Theory of Computing (STOC)* (2004).

[96] IT, H. Ehr adoption. `https://dashboard.healthit.gov/evaluations/data-briefs/non-federal-acute-care-hospital-ehr-adoption-2008-2015.php`.

[97] JONES, E. P., ABADI, D. J., AND MADDEN, S. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2010).

[98] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

[99] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar Conference Center, Pacific Grove, U.S., 1991), vol. 25, ACM Press, pp. 213–225.

[100] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.* (1992).

[101] KLOPHAUS, R. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10.

[102] KOCHER, P., HORN, J., FOGH, A., , GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (SP)* (2019).

[103] KORTH, H. K., AND SPEEGLE, G. Formal model of correctness without serializabilty. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), SIGMOD '88, ACM, pp. 379–386.

[104] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems 27*, 4 (Jan. 2010), 7:1–7:39.

[105] KRASKA, T., PANG, G., FRANKLIN, M. J., MADDEN, S., AND FEKETE, A. Mdcc: multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pp. 113–126.

[106] KRIPKE, S. A. Semantical considerations on modal logic. *Acta Philosophica Fennica 16*, 1963 (1963), 83–94.

[107] KUNG, H. T., AND ROBINSON, J. T. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst. 6*, 2 (1981), 213–226.

[108] KUO, A. M.-H. Opportunities and challenges of cloud computing to improve health care services. *Journal of Medical Internet Research (JMIR) 13*, 3 (2011).

[109] KUSHILEVITZ, E., AND OSTROVSKY, R. Replication is not needed: Single database, computationally-private information retrieval. In *IEEE Symposium on Foundations of Computer Science (FOCS)* (1997).

[110] KWON, A., CORRIGAN-GIBBS, H., DEVADAS, S., AND FORD, B. Atom: Horizontally

Scaling Strong Anonymity. In *ACM Symposium on Operating System Principles (SOSP)* (2017).

[111] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 9 (Sept. 1979), 690–691.

[112] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 9 (Sept. 1979), 690–691.

[113] Larson, P.-A., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M., and Zwilling, M. High-performance Concurrency Control Mechanisms for Main-memory Databases. In *Proceedings of the VLDB Endowment (PVLDB)* (2011).

[114] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pp. 265–278.

[115] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI '04.

[116] Libre, M. FreeHealth EHR. `https://https://freemedsoft.com/fr/`. Accessed 2018-05-01.

[117] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium (USENIX)* (2018).

[118] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 401–416.

[119] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Stronger seman-

tics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pp. 313–328.

[120] Lorch, J., Parno, B., Mickens, J., Raykova, M., and Schiffman, J. Shroud: Ensuring Private Access to Large-Scale Data in the Data Center. In *Conference on File and Storage Technologies (FAST)* (2013).

[121] Lueks, W., and Goldberg, I. Sublinear Scaling for Multi-Client Private Information Retrieval. In *Financial Cryptography and Data Security (FC)* (2015).

[122] Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J., and Song, D. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[123] Mahajan, P., Alvisi, L., and Dahlin, M. Consistency, availability, convergence. Tech. Rep. TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.

[124] Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems 29*, 4 (2011), 12.

[125] MapDB. MapDB: Embedded Database Engine. `http://www.mapdb.org/`.

[126] Mashtizadeh, A. J., Bittau, A., Huang, Y. F., and Mazières, D. Replication, History, and Grafting in the Ori File System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 151–166.

[127] MB, B. Online retail. `https://www.thebalancesmb.com/compare-brick-and-mortar-stores-vs-online-retail-sites-4571050I`.

[128] Medium. The dao hack explained. `https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562`

[129] Mehdi, S. A., Littley, C., Crooks, N., Alvisi, L., Bronson, N., and Lloyd, W. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[130] MEMSQL. Memsql. https://www.memsql.com.

[131] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., KHAN, O., AND NAREDDY, K. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation* (2014), NSDI'14.

[132] MICROSOFT. Azure sql database. https://https://azure.microsoft.com/en-us/services/sql-database/?v=16.50.

[133] MICROSOFT. Azure storage - secure cloud storage. https://azure.microsoft.com/en-us/services/storage/.

[134] MICROSOFT. Azure tables. https://azure.microsoft.com/en-us/services/storage/tables/.

[135] MICROSOFT. Documentdb - nosql service for json. https://azure.microsoft.com/en-us/services/documentdb/.

[136] MICROSOFT. SQL Server. https://www.microsoft.com/en-cy/sql-server/sql-server-2016.

[137] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst. 17*, 1 (1992), 94–162.

[138] MONGODB. Agility, Performance, Scalibility. Pick three. https://www.mongodb.org/.

[139] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 479–494.

[140] NARAYANAN, A., AND SHMATIKOV, V. Robust De-anonymization of Large Sparse Datasets. In *IEEE Symposium on Security and Privacy (SP)* (2008).

[141] NARAYANAN, A., AND SHMATIKOV, V. Myths and fallacies of “personally identifiable information”. *Commun. ACM 53*, 6 (June 2010), 24–26.

[142] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Proc. 7th OSDI* (Nov. 2006).

[143] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the Sync. *ACM Transactions on Computer Systems 26*, 3 (Sept. 2008), 6:1–6:26.

[144] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’99.

[145] OLUMOFIN, F., AND GOLDBERG, I. Privacy-preserving Queries over Relational Databases. In *Privacy Enhancing Technologies Symposium (PETS)* (2010).

[146] ORACLE. InnoDB. `https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html/`.

[147] ORACLE. MySQL. `https://www.mysql.com/`.

[148] ORACLE. MySQL Cluster. `https://www.mysql.com/products/cluster/`.

[149] ORACLE. Oracle 12c - isolation levels. `https://docs.oracle.com/database/121/CNCPT/consist.htm#CNCPT1320/`.

[150] ORACLE. Oracle database 18c. `https://docs.oracle.com/en/database/oracle/oracle-database/18/index.html`, 2018.

[151] PADILHA, R., AND PEDONE, F. Augustus: Scalable and robust storage for cloud applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys ’13, pp. 99–112.

[152] PAPADIMITRIOU, A., BHAGWAN, R., CHANDRAN, N., RAMJEE, R., HAEBERLEN, A., SINGH, H., MODI, A., AND BADRINARAYANAN, S. Big Data Analytics over Encrypted Datasets with Seabed. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[153] Papadimitriou, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.

[154] Pedone, F., Zwaenepoel, W., and Elnikety, S. Database replication using generalized snapshot isolation. *24th IEEE Symposium on Reliable Distributed Systems* (2005), 73–84.

[155] Platform, G. C. Cloud spanner. `http://cloud.google.com/spanner/`.

[156] Popa, R. A., Redfield, C. M. S., Zeldovich, N., and Balakrishnan, H. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM Symposium on Operating System Principles (SOSP)* (2011).

[157] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, pp. 43–57.

[158] PostgreSQL. `http://www.postgresql.org/`.

[159] Preguica, N., Marques, J. M., Shapiro, M., and Letia, M. A commutative replicated data type for cooperative editing. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pp. 395–403.

[160] Reddy, P. K., and Kitsuregawa, M. Speculative Locking Protocols to Improve Performance for Distributed Database Systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 16*, 2 (2004), 154–169.

[161] Reed, D. P. Implementing Atomic Actions on Decentralized Data (Extended Abstract). In *ACM Symposium on Operating System Principles (SOSP)* (1979).

[162] Reed, D. P. Implementing Atomic Actions on Decentralized Data. *ACM Trans. Comput. Syst. 1*, 1 (1983), 3–23.

[163] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. Resolving file conflicts in the ficus file system, 1994.

[164] Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., and De-

VADAS, S. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security Symposium (USENIX)* (2015).

[165] RETWIS. Twitter-like Clone. `http://retwis.redis.io/`.

[166] ROSENKRANTZ, D. J., STEARNS, R. E., AND LEWIS, II, P. M. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.

[167] SAHIN, C., ZAKHARY, V., EL ABBADI, A., LIN, H., AND TESSARO, S. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *IEEE Symposium on Security and Privacy (SP)* (2016).

[168] SAP. Hana. `https://www.sap.com/products/hana.html`.

[169] SERVER, M. S. Always Encrypted. `https://www.microsoft.com/en-us/research/project/always-encrypted/`.

[170] SHAPIRO, M., ARDEKANI, M. S., AND PETRI, G. Consistency in 3d (invited paper). In *27th International Conference on Concurrency Theory (CONCUR 2016)* (2016).

[171] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011.

[172] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. X. Défago, F. Petit, and V. Villain, Eds., vol. 6976, pp. 386–400.

[173] SHEFF, I., MAGRINO, T., LIU, J., MYERS, A. C., AND VAN RENESSE, R. Safe Serializable Secure Scheduling: Transactions and the Trade-Off Between Security and Consistency. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[174] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with O((Logn)3) Worst-case Cost. In *International Conference on The Theory and Application of Cryptology and Information Security* (2011).

[175] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$

Worst-Case Cost. In *International Conference on The Theory and Application of Cryptology and Information Security* (2011).

[176] Shrira, L., Tian, H., and Terry, D. Exo-leasing: escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pp. 42–61.

[177] Singel, R. Netflix spilled your *Brokeback Mountain* secret, lawsuit claims. *Wired* (Dec. 2009). `http://www.wired.com/images_blogs/threatlevel/2009/12/doe-v-netflix.pdf`.

[178] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 385–400.

[179] Stefanov, E., and Shi, E. ObliviStore: High Performance Oblivious Cloud Storage. In *IEEE Symposium on Security and Privacy (SP)* (2013).

[180] Stefanov, E., and Shi, E. ObliviStore: High Performance Oblivious Distributed Cloud Data Store. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[181] Stefanov, E., Shi, E., and Song, D. Towards Practical Oblivious RAM.

[182] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[183] Su, C., Crooks, N., Ding, C., Alvisi, L., and Xie, C. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, pp. 283–297.

[184] Sun, C., and Ellis, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pp. 59–68.

[185] Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M. J., Theimer, M. M., and

WELCH, B. B. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on on Parallel and Distributed Information Systems*, PDIS '94, pp. 140–150.

[186] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 309–324.

[187] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pp. 172–182.

[188] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 1–12.

[189] TRANSACTION PROCESSING PERFORMANCE COUNCIL. The TPC-C home page. http://www.tpc.org/tpcc.

[190] TU, S., KAASHOEK, M. F., MADDEN, S., AND ZELDOVICH, N. Processing Analytical Queries over Encrypted Data. In *Proceedings of the VLDB Endowment (PVLDB)* (2013).

[191] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy Transactions in Multicore In-memory Databases. In *ACM Symposium on Operating System Principles (SOSP)* (2013).

[192] VIOTTI, P., AND VUKOLIĆ, M. Consistency in non-transactional distributed storage systems. *ACM Computing Survey 49*, 1 (June 2016), 19:1–19:34.

[193] VOGELS, W. Eventually consistent. *Queue 6*, 6 (Oct. 2008), 14–19.

[194] WANG, F., YUN, C., GOLDWASSER, S., VAIKUNTANATHAN, V., AND ZAHARIA, M. Splin-

ter: Practical Private Queries on Public Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[195] WARSZAWSKI, T., AND BAILIS, P. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, pp. 5–20.

[196] WEIKUM, G. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Trans. Database Syst. 16*, 1 (1991), 132–180.

[197] WIKIPEDIA. Wikipedia: Conflicting Sources. `http://en.wikipedia.org/wiki/Wikipedia:Conflicting_sources`.

[198] WILLIAMS, P., SION, R., AND CARBUNAR, B. Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage. In *ACM Conference on Computer and Communications Security (CCS)* (2008).

[199] WILLIAMS, P., SION, R., AND TOMESCU, A. PrivateFS: A Parallel Oblivious File System. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[200] WU, Z., BUTKIEWICZ, M., PERKINS, D., KATZ-BASSETT, E., AND MADHYASTHA, H. V. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 292–308.

[201] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance ACID via Modular Concurrency Control. In *ACM Symposium on Operating System Principles (SOSP)* (2015).

[202] YANNAKAKIS, M. Serializability by locking. *J. ACM 31*, 2 (Mar. 1984), 227–244.

[203] YU, H., AND VAHDAT, A. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation - Volume 4*, OSDI '00.

[204] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th*

*Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 263–278.

[205] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 276–291.

[206] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).