Seeing is Believing:
A Unified Model for Consistency and Isolation via States

Lorenzo Alvisi‡  Allen Clement§  Natacha Crooks‡  Youer Pu‡  
‡The University of Texas at Austin  §Google, inc.

Abstract
This paper introduces a unified model of consistency and isolation that minimizes the gap between how these guarantees are defined and how they are perceived. Our approach is premised on a simple observation: applications view storage systems as black-boxes that transition through a series of states, a subset of which are observed by applications. For maximum clarity, isolation and consistency guarantees should be expressed as constraints on those states. Instead, these properties are currently expressed as constraints on operation histories that are not visible to the application. We show that adopting a state-based approach to expressing these guarantees brings forth several benefits. First, it makes it easier to focus on the anomalies that a given isolation or consistency level allows (and that applications must deal with), rather than those that it proscribes. Second, it unifies the often disparate theories of isolation and consistency and provides a structure for composing these guarantees. We leverage this modularity to apply to transactions (independently of the isolation level under which they execute) the equivalence between causal consistency and session guarantees that Chockler et al. had proved for single operations. Third, it brings clarity to the increasingly crowded field of proposed consistency and isolation properties by winnowing spurious distinctions: we find that the recently proposed parallel snapshot isolation introduced by Sovran et al. is in fact a specific implementation of an older guarantee, lazy consistency (or PL-2+), introduced by Adya et al.

This is a submission to the regular track. This paper is eligible for the Best Student Paper Award and should be considered as a Brief Announcement.
1 Introduction

Large-scale applications such as Facebook, Amadeus, or Twitter offload the managing of data at scale to 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 increase scalability within a site, databases harness the power of multicore computing [13, 26, 27, 56, 57] by deploying increasingly complex concurrency control algorithms. Faced with the latent scalability bottleneck of serializability, commercial systems often privilege instead weaker but more scalable notions of isolation, such as snapshot isolation or read committed [1, 12, 47, 51]. Likewise, several recent research efforts focus on improving the scalability of strong consistency guarantees in distributed storage systems [35, 50, 52, 59]. Modern large-scale distributed systems, however, to increase scalability across sites largely renounce strong consistency in favour of weaker guarantees, from causal consistency to per-session-only guarantees [11, 19, 23, 25, 36, 39, 40, 44, 45, 58].

This trend poses an additional burden on the application programmer, as weaker isolation and consistency 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. Consider a bank account with a $50 balance and no overdraft allowed. Read-committed allows two transactions to concurrently withdraw $45, leaving the account with a negative balance [12]. Likewise, causal consistency ensures that write-read dependencies are enforced, but provides no meaningful way to handle write-write conflicts [24].

To mitigate programming complexity, many commercial databases and distributed storage systems [5, 6, 11, 29, 30, 43–45, 47, 51] interact with applications through a front-end that, like a valve, is meant to shield applications from the complex concurrency and replication protocols at play. This valve, however, is leaky at best: a careful understanding of the system that implements a given isolation or weak consistency level is oftentimes necessary to determine which anomalies the system will admit.

Indeed, isolation and consistency levels often assume features specific to the systems for which they were first defined—from the properties of storage (e.g., whether it is single or multiversioned [14]); to the chosen concurrency control (e.g., whether it is based on locking or timestamps [12]); or to other system features (e.g., the existence of a centralized timestamp [28]). These assumptions, furthermore, are not always explicit: the claim, in the original ANSI SQL specification, that serializability is equivalent to preventing four phenomena [12] only holds for lock-based, single version databases. Clarity on such matters is important: to this day, multiversioned commercial databases claiming to implement serializability in fact implement the weaker notion of snapshot isolation [9, 28, 48].

We believe that at the root of this complexity is the current practice of defining consistency and isolation guarantees in terms of the ordering of low-level operations such as reads and writes, or sends and receives. This approach has several drawbacks for application programmers. First, it requires them 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.

To bridge the semantic gap between how isolation and consistency guarantees are specified and how they are being used, we introduce a new, unified framework for expressing both consistency and isolation guarantees that relies exclusively on application-observable states rather than on low-level operations. The framework is general: we use it to express most modern consistency and isolation definitions, and prove that the definitions we obtain are equivalent to their existing counterparts.

In addition to cleanly separating consistency and isolation guarantees from the implementation of the system to which they apply, we find that the new framework yields three advantages:

1. It brings clarity to the increasingly crowded field of proposed consistency and isolation properties, winnowing out spurious distinctions. In particular, we show that the distinction between PSI [20, 52] and lazy consistency [1, 2] is in fact an artefact of the replication model assumed.

2. It provides a simple framework for composing consistency and isolation guarantees. We leverage its expressiveness to prove that causal consistency is equivalent to jointly guaranteeing all four session guarantees
3. It simplifies reasoning end-to-end about a system’s design, opening up new opportunities for optimizing its implementation. In particular, we show that PSI can be enforced without totally ordering the transactions executed at each site (as instead the original PSI definition requires), thus making the system less prone to have its performance dictated by the rate at which its slowest shard can enforce dependencies.

We provide an extended motivation in Section 2. We introduce the model in Section 3 and use it to define isolation in Section 4 and consistency in Section 5. We highlight practical benefits of our approach in Section 6. Finally, we summarize and conclude in Section 7.

2 A system’s case for a new formalism

Much of the complexity associated with weakly consistent systems stems from an intricate three-way semantic gap between how applications are encouraged to use these systems, how the guarantees these systems provide are expressed, and how they are implemented.

On the one hand, applications are invited to think of these systems as black boxes that benevolently hide the complexity involved in achieving the availability, integrity, and performance guarantees that applications care about. For example, PaaS (Platform as a Service) cloud-based storage systems, databases or webservers free applications from having to configure hardware, and let them simply pay for reserved storage or throughput. Essentially, it is as if applications were querying or writing to a logically centralized, failure-free node that will scale as much as one’s wallet will allow. On the other hand, the precise guarantees that these black boxes provide are generally difficult to pin down, as different systems often give them implementation-specific twists that can only be understood by looking inside the box.

For example, the exact meaning of session guarantees, present in Bayou, Corba and, more recently, in Pileus and DocumentDB, depends on whether the system to which they apply implements a total order of write operations across client sessions. Consider the execution in Figure 2: 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 (like DocumentDB), 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.

Even the classic notion of serializability can fall prey to inconsistencies. In theory, its guarantee is clear: it states that an interleaved execution of transactions must be equivalent to a serial schedule. In practice, however, the system interpretations of that notion can differ. Consider the execution in Figure 2(a), consisting of two interleaved write-only transactions: is it serializable? It would appear so, as it is equivalent to the serial schedule (T₁; T₂). The answer, however, depends on whether the underlying databases allows for writes to be re-ordered, a feature that is expensive to implement. And indeed, a majority of the self-declared serializable databases surveyed in Figure 2(a) do actually reject the schedule. In contrast, Figure 2(b) depicts a non-serializable schedule that exhibits write-skew. Yet, that execution is allowed by systems that claim to be serializable, such as Oracle 12c. The source of the confusion is the very definition of serializability. Oracle 12c uses the definition of the original SQL standard, based on the four ANSI SQL phenomena that it disallows. In single-versioned systems, preventing these phenomena is indeed equivalent to serializability; but in a multiversion system (such as Oracle 12c) it no longer is.

Fundamentally, consistency guarantees are contracts between the storage system and its clients, specifying a well-defined set of admissible behaviors—i.e., the set of values that each read is allowed to return. To be useful, they need to be precise and unchanging. When implicit assumptions about the implementation of the system are allowed to encroach, however, these essential attributes can suffer.

3 Model

Existing consistency or isolation models are defined as constraints on the ordering of the read and write operations that the storage system performs. Applications, however, 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 consistency and isolation, we adopt the same viewpoint of the applications that must ultimately use these guarantees. We propose a model based on application-observable states rather than on the invisible history of the low-level operations performed by the system.

Intuitively, a storage system guarantees a specific isolation or consistency level if it can produce an execution (a sequence of atomic state transitions) that is 1) consistent with the values observed by applications and 2) valid, in that it satisfies the guarantees of the desired isolation/consistency level. In essence, the values returned by the systems constrain the set of states that the system can bring forth to demonstrate that it can produce a valid execution.

More formally, we define a storage system $S$ with respect to a set $K$ of keys, and $V$ of values; a system state $s$ is a unique mapping from key to values. For simplicity, we assume that each value is uniquely identifiable. In the initial system state, all keys have value $\bot$.

To unify our treatment of consistency and isolation, we assume that applications modify the storage system’s state using transactions; we model individual operations as transactions consisting of a single read or a single write. A transaction $t$ is a tuple $(\Sigma_t, \rightarrow_t)$, where $\Sigma_t$ is the set of operations in $t$, and $\rightarrow_t$ is a total order on $\Sigma_t$. Operation 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 $t$ contains the keys read by $t$: $R_t = \{k | r(k, v) \in \Sigma_t\}$. Similarly, the write set of $t$ contains the keys that $t$ updates: $W_t = \{k | w(k, v) \in \Sigma_t\}$. For simplicity of exposition, we assume that a transaction only writes a key once.

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 $t$. We refer to $s$ as the parent state of $t$, and refer to the transaction that generated $s'$ as $t_{s'}$. Formally,

**Definition 1.** $s \xrightarrow{t} s' = (((k, v) \in s' \land (k, v) \notin s) \Rightarrow k \in W_t) \land (w(k, v) \in \Sigma_t) \Rightarrow (k, v) \in s')$

We denote the set of keys in which $s$ and $s'$ differ as $\Delta(s, s')$.

An execution $e$ for a set of transactions $\mathcal{T}$ is a totally ordered set defined by the pair $(\mathcal{S}_e, \rightarrow_{\mathcal{T}})$, 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 $s \xrightarrow{e} s'$ (respectively, $s \xrightarrow{t} s'$) to denote a sequence of zero (respectively, one) or more state transitions from $s$ to $s'$ in $e$. Note that, while $e$ identifies the state transitions produced by each transaction $t \in \mathcal{T}$, it does not specify the subset of states in $\mathcal{S}_e$ that each operation in $t$ can read from. 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 candidate read states.

**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 candidate read states for a read operation $o = r(k, v) \in \Sigma_t$ is the set of states

$$\mathcal{R}\mathcal{S}_e(o) = \{s \in \mathcal{S}_e | s \xrightarrow{\mathcal{T}} s_p \land ((k, v) \in s \lor (\exists w(k, v) \in \Sigma_t : w(k, v) \xrightarrow{t} r(k, v)))\}$$

To prevent transactions from reading from the future, we restrict the set of valid candidate read states to those no later than $s_p$. Additionally, once $t$ writes $v$ to $k$, we require all subsequent read operations $o \in \Sigma_t$ to return $v$ [1].
By convention, the candidate read states of a write operation include all the states \( s \in S_e \) that \( s \xrightarrow{w} s_p \). It is easy to prove that the candidate read states of any operation define a subsequence of contiguous states in the total order that \( e \) defines on \( S_e \). We refer to the first state in that sequence as \( s_{f_0} \), and to the last state as \( s_{l_0} \). The predicate \( \text{PREREAD}_e(T) \) guarantees that such states exist:

**Definition 3.** Let \( \text{PREREAD}_e(t) \equiv \forall o \in \Sigma_t : \mathcal{R}_e(o) \neq \emptyset \). Then \( \text{PREREAD}_e(T) \equiv \forall t \in T : \text{PREREAD}_e(t) \).

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.** \( \text{COMPLETE}_e(t)(s) \equiv s \in \bigcap_{o \in \Sigma_t} \mathcal{R}_e(o) \)

Finally, we introduce the notion of *internal read consistency*: internal read consistency states that read operations that follow each other in the transaction order should read from a monotonically increasing state. We write:

**Definition 5.** \( \text{IRC}_e(t) \equiv \forall o, o' \in \Sigma_t : o' \xrightarrow{t_o} o \Rightarrow \neg (s_{l_o} \xrightarrow{t} s_{f_{o'}}) \)

### 4 Isolation

Isolation guarantees specify the valid set of executions for a given set of transactions \( T \). The long-established \([1, 14, 15]\) way to accomplish this has been to constrain the history of the low-level operations that the system is allowed to perform when processing transactions. Our new approach eschews this history, which is invisible to applications, in favor of application-observable states. Before we introduce our work, we provide some context for it by summarizing some of the key definitions and results from Adya’s classic, history-based, treatment of isolation \([1]\).

#### 4.1 A history-based specification of isolation guarantees

**Definition 6.** 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, \( \ll \), that totally orders committed object versions.

**Definition 7.** We consider three kinds of *direct read/write conflicts:*

- **Directly write-depends** \( T_i \) writes a version of \( x \) and \( T_j \) writes the next version of \( x \) \((T_i \xrightarrow{w} T_j)\)
- **Directly read-depends** \( T_i \) writes a version of \( x \) that \( T_j \) then reads \((T_i \xrightarrow{wr} T_j)\)
- **Directly anti-depends** \( T_i \) reads a version of \( x \), and \( T_j \) writes the next version of \( x \) \((T_i \xrightarrow{ru} T_j)\)

**Definition 8.** We say that \( T_j \) *start-depends* on \( T_i \) (denoted as \( T_i \xrightarrow{sd} T_j \)) if \( c_i <_t b_j \), where \( c_i \) denotes \( T_i \)’s commit timestamp and \( b_j \) \( T_j \)’s start timestamp, i.e., if \( T_j \) starts after \( T_i \) commits.

**Definition 9.** Each node in the *direct serialization graph* \( \text{DSG}(H) \) arising from a history \( H \) corresponds to a committed transaction in \( H \). Directed edges in \( \text{DSG}(H) \) 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 \( T_i \).

**Definition 10.** The *Started-ordered Serialization Graph* \( \text{SSG}(H) \) contains the same nodes and edges as \( \text{DSG}(H) \) along with start-dependency edges.

**Definition 11.** Adya identifies the following phenomena:

- **G0: Write Cycles** \( \text{DSG}(H) \) contains a directed cycle consisting entirely of write-dependency edges.
- **G1a: Dirty Reads** \( H \) contains an aborted transaction \( T_i \) and a committed transaction \( T_j \) such that \( T_j \) has read the same object (maybe via a predicate) modified by \( T_i \).
Table 1: ANSI SQL Commit Tests

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Commit Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serializability</td>
<td>$\text{COMPLETE}_{e,t}(s_p)$</td>
</tr>
<tr>
<td>Snapshot Isolation</td>
<td>$\exists s \in S_e : \text{COMPLETE}_{e,t}(s) \land (\Delta(s, s_p) \cap \mathcal{W}_e = \emptyset)$</td>
</tr>
<tr>
<td>Read Committed</td>
<td>$\text{PREREAD}_{e}(t)$</td>
</tr>
<tr>
<td>Read Uncommitted</td>
<td>True</td>
</tr>
</tbody>
</table>

G1b: Intermediate Reads  $H$ 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 first modification of $x$.

G1c: Circular Information Flow  $DSG(H)$ contains a directed cycle consisting entirely of dependency edges.

G1:  $G1a \lor G1b \lor G1c$

G2: Anti-dependency Cycles  $DSG(H)$ 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  $SSG(H)$ 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  contains a directed cycle with exactly one anti-dependency edge.

G-SI:  $G-SIa \lor G-SIb$

Definition 12. Adya defines the following isolation levels in terms of the phenomena in Definition 11:

Serializability (PL-3)  $\equiv \neg G1 \land \neg G2$

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

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

Snapshot Isolation  $\equiv \neg G1 \land \neg G-SI$

4.2 A state-based specification of isolation guarantees

In our approach based on observable states, isolation guarantees specify the valid set of executions for a given set of transactions $\mathcal{T}$ by constraining 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 constraint by means of a commit test: for an execution $e$ to be valid under a given isolation level $I$, each transaction $t$ in $e$ must satisfy the commit test for $I$, written $CT_I(t,e)$.

Definition 13. A storage system satisfies an isolation level $I \equiv \exists e : \forall t \in T : CT_I(t,e)$.

Table 1 shows the commit tests for the four most common ANSI SQL isolation levels. We informally motivate their rationale below.

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 operations of $t$ should read from the same state $s$, thereby ensuring that transactions never observe the effects of concurrently running transactions. Second, $s$ should be the parent state of $t$, i.e., the state that $t$ transitions from.

Snapshot isolation (SI). Like serializability, SI prevents every 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 a transaction that precedes $t$ in the execution $e$. However, SI no longer insists on $s$ being $t$’s parent state $s_p$: other transactions may commit in between $s$ and $s_p$, whose operations $t$ will not observe. The commit test only forbids $t$ from modifying any of the keys that changed value as the system’s state progressed from $s$ to $s_p$.

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 them to read from state that precedes $t$ in the execution $e$.

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. The reason for this seemingly excessive laxity is that isolation models in databases consider only
committed transactions and are therefore unable to distinguish between values produced by aborted transactions and by altogether imaginary writes. This distinction is not lost in environments, such as transactional memory, where correctness depends on providing guarantees such as opacity [31] for all live transactions. We discuss this further in Section 7.

Although these tests make no mention of histories, they each admit the same set of executions of the corresponding history-based condition formulated by Adya [1]. In Appendix B.1, Appendix B.2 and Appendix B.3, we prove the following theorems, respectively:

**Theorem 1.** Let \( I \) be Serializability (SER). Then \( \forall e : \forall t \in \mathcal{T} : CT_{SER}(t, e) \equiv \neg G1 \land \neg G2. \)

**Theorem 2.** Let \( I \) be Snapshot Isolation (SI). Then \( \forall e : \forall t \in \mathcal{T} : CT_{SI}(t, e) \equiv \neg G1 \land \neg G-SI \)

**Theorem 3.** Let \( I \) be Read Committed (RC). Then \( \forall e : \forall t \in \mathcal{T} : CT_{RC}(t, e) \equiv \neg G1 \)

**Theorem 4.** Let \( I \) be Read Uncommitted (RU). Then \( \forall e : \forall t \in \mathcal{T} : CT_{RU}(t, e) \equiv \neg G0 \)

### 4.2.1 Discussion

The above theorems establish that a specification of isolation guarantees based on client-observable states is as expressive as one based on histories. Adopting a client-centric perspective, however, has a distinct advantage: it makes it easier for application programmers to understand the anomalies allowed by weak isolation levels.

To illustrate this “intuition gap”, consider the simple banking example of Figure 2. Alice and Bob share checking (C) and saving (S) accounts, each holding $30. To avoid the bank’s wrath, 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 eventually cover any overdraft. Suppose Alice and Bob try concurrently to each withdraw $40 from, respectively, their checking and savings account, and issue transactions \( t_{w1} \) and \( t_{w2} \). Figure 2(a) shows an execution under serializability. Because transactions read from their parent state, \( t_{w2} \) observes \( t_{w1} \)’s withdrawal and, since the balance of Bob’s accounts is below $40, aborts.

In contrast, consider the execution under snapshot isolation in Figure 2(b). It is legal for both \( t_{w1} \) and \( t_{w2} \) to read the same state \( s_1 \), find that the combined funds in the two accounts exceed $40, and, unaware of each other, proceed to generate an execution whose final state \( s_3 \) will get Alice and Bob in trouble with the bank. This anomaly, commonly referred to as write-skew, arises because \( t_{w2} \) is allowed to read from a state other than the most recent state. Defining snapshot isolation in terms of observable states makes the source of this anomaly obvious, arguably to a greater degree than the standard history-based definition, which characterizes snapshot isolation as “disallowing all cycles consisting of direct (write-write and write-read) dependencies and at most a single anti-dependency”.

Though we have focused our discussion on ANSI SQL isolation levels that do not consider real-time, our model can straightforwardly be extended to support strict serializability [32] as follows.

Let \( O \) be a time oracle \( O \) that assigns distinct 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 \( t_1.commit < t_2.start \). Strict serializability can then be defined be adding the following condition to the serializability commit test: \( \forall t' \in \mathcal{T} : t' <_s t \Rightarrow s_{t'} \rightarrow s_t. \)

### 5 Consistency

Though isolation guarantees typically do not regulate how transactions from a given client should be ordered \(^1\), they tacitly assume that transactions from the same client will be executed in client-order, as they naturally would in a

---

\(^1\)Strict serializability is the exception to this rule.
centralized or synchronously replicated storage system. In weakly consistent systems, where transactions can be asynchronously replicated between sites, this assumption no longer holds: two transactions from the same client may be re-ordered if they happen to be executed on different replicas. To bring back order, distributed systems introduce the notion of *sessions*. Sessions encapsulate the sequence of transactions performed by a single entity (a thread, client, or application). Informally, their aim is to provide each entity with a view of the system consistent with its own actions; formally, a session $se$ is a tuple $(T_{se}, \rightarrow)$ where $\rightarrow$ is a total order over the transactions in $T_{se}$. The set of all sessions is denoted by $SE$.

To provide a foundation to a common theory of isolation and consistency, we define session-based consistency guarantees for transactions. Session guarantees have traditionally been defined for operations [17, 22, 53]; our definitions can be mapped back by considering single-operation transactions.

We first introduce the definition of sequential consistency [37]. Sequential consistency requires read operations within each transaction observe monotonically increasing states and have non-empty candidate read sets and, like previously defined isolation levels, demands that all sessions observe a single execution. Unlike isolation levels, however, sequential consistency also requires transactions to take effect in the order specified by their session. We define the commit test for sequential consistency as follows:

$$CT_{SC}(e, t) \equiv \text{PREREAD}_e(t) \land \text{IRC}_e(t) \land \left( \forall se \in SE : \forall t_i \xrightarrow{se} t_j : \left( s_{t_i} \xrightarrow{e} s_{t_j} \land \forall o \in \Sigma t_j : s_{t_i} \xrightarrow{e} s_{lo} \right) \right)$$

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 $T_{se}$ must satisfy the session test for $SG$, written $\text{SESSION}_{SG}(se, t, e)$.

**Definition 14.** A storage system satisfies a session guarantee $SG \equiv \forall se \in SE : \exists e : \forall t \in T_{se} : \text{SESSION}_{SG}(se, t, e)$.

Intuitively, session tests invert the order between the existential qualifier for execution and the universal quantifier for sessions. Table 2 shows the session tests for the most common session guarantees. We informally motivate their rationale below.

**Read-My-Writes** This session guarantee 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, nor does it provide any guarantee on the reads of a client who never writes, or say anything about 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 is simply content with asking for the read state of every operation in a transaction’s session to be after the commit state of all preceding update transactions in that session.

**Monotonic Reads** Monotonic reads, instead, constrains a client’s reads to observe an increasingly up-to-date state of the database: this applies to transactions in a session, and to operations within the transaction (by IRC). The notion of “up-to-date” here may vary by client, as the storage is free to arrange transactions differently for each session’s execution. The only way for the client to detect an MR violation is hence to read a 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 for clients to read from monotonically increasing but stale states.

**Monotonic Writes** In contrast, monotonic writes constrains the ordering of writes within a 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, at least when it comes to update transactions. The PREREAD requirement for read operations, instead, continues to apply only within each session.

**Writes-Follow-Reads** Like monotonic writes, writes-follow-reads 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 every execution that a client may observe.

**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 $s$. 

---

**Table 2:** Session tests for the most common session guarantees. We informally motivate their rationale below.

<table>
<thead>
<tr>
<th>Session Guarantee</th>
<th>Test Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>$CT_{SC}(e, t)$</td>
</tr>
<tr>
<td>RMW</td>
<td>$\text{PREREAD}<em>e(t)$ $\land$ $\text{IRC}<em>e(t)$ $\land$ $\left( \forall se \in SE : \forall t_i \xrightarrow{se} t_j : \left( s</em>{t_i} \xrightarrow{e} s</em>{t_j} \land \forall o \in \Sigma t_j : s_{t_i} \xrightarrow{e} s_{lo} \right) \right)$</td>
</tr>
<tr>
<td>MR</td>
<td>$\forall se \in SE : \exists e : \forall t \in T_{se} : \text{SESSION}_{SG}(se, t, e)$</td>
</tr>
<tr>
<td>RMW</td>
<td>$\forall se \in SE : \exists e : \forall t \in T_{se} : \text{SESSION}_{SG}(se, t, e)$</td>
</tr>
<tr>
<td>MRW</td>
<td>$\forall se \in SE : \exists e : \forall t \in T_{se} : \text{SESSION}_{SG}(se, t, e)$</td>
</tr>
<tr>
<td>WFR</td>
<td>$\forall se \in SE : \exists e : \forall t \in T_{se} : \text{SESSION}_{SG}(se, t, e)$</td>
</tr>
</tbody>
</table>

---
<table>
<thead>
<tr>
<th>Specification Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSI</td>
<td>ENFORCEC(Ta) \land \forall o \in \Sigma_t : \forall t^e \xrightarrow{sf_1} t : W_{t^e} \neq \emptyset \Rightarrow s_i \leadsto s_{i, o}</td>
</tr>
<tr>
<td>Monotonic Reads (MR)</td>
<td>PREREADc(Ta) \land IRCc(t) \land \forall o \in \Sigma_t : \forall t^e \xrightarrow{sf_1} t : V_{t^e} \land W_{t^e} \neq \emptyset \Rightarrow s_i \leadsto s_{i, o}</td>
</tr>
<tr>
<td>Monotonic Writes (MW)</td>
<td>PREREADc(Ta) \land \forall se' \in SE : \forall t_i \xrightarrow{se'} t_j : (W_{se} \land W_{t^e} \neq \emptyset) \Rightarrow s_i \leadsto s_{i, j}</td>
</tr>
<tr>
<td>Writes-Follow-Reads (WFR)</td>
<td>PREREADc(T) \land \forall se' \in SE : \forall t_i \xrightarrow{se'} t_j : (V_{t_i} \land W_{t^e} \neq \emptyset \Rightarrow s_j \leadsto s_{i, j}</td>
</tr>
<tr>
<td>Causal Consistency (CC)</td>
<td>PREREADc(T) \land IRCc(t) \land \forall o \in \Sigma_t : \forall t^e \xrightarrow{sf_1} t : s_i \leadsto s_{i, o}</td>
</tr>
</tbody>
</table>

Table 2: Session Guarantees

6 Practical benefits

In addition to reducing the gap between how isolation and consistency guarantees are defined and how they are perceived by their users, definitions based on client-observable states provide further benefits.

6.1 Economy

Focusing on client-observable states frees definitions from implementation-specific assumptions. Removing these artefacts can bring out similarities and winnow out spurious distinctions in the increasingly crowded field of isolation and consistency.

Consider, for example, parallel snapshot isolation (PSI), recently proposed by Sovran et al [52] and lazy consistency, introduced by Adya et al [1, 2]. Both isolation levels are appealing as they are implementable at scale in geo-replicated settings. Indeed, PSI aims to offer a scalable alternative to snapshot isolation by relaxing the order in which transactions are allowed to commit on geo-replicated sites. The specification of PSI is given as an abstract specification code that an implementation must emulate.

Definition 15. PSI enforces three main properties:

- **P1 (Site Snapshot Read):** All operations read the most recent committed version at the transaction’s site as of the time when the transaction began.
- **P2 (No Write-Write Conflicts):** The write sets of each pair of committed somewhere-concurrent transactions must be disjoint (two transactions are somewhere concurrent if they are concurrent on site(T_1) or at site(T_2).
- **P3 (Commit Causality Across Sites):** If a transaction T_1 commits at a site A before a transaction T_2 starts at site A, then T_1 cannot commit after T_2 at any site.

PL-2+, on the other hand, guarantees consistent reads (transactions never partially observe the effects of other transactions) and disallows lost updates. Formally:

Definition 16. Lazy Consistency (PL-2+) ≡ ¬G1 ∧ ¬G-single

At first blush, these system-centric definitions bear little resemblance to each other. Yet, when their authors explain their intuitive meaning from a client’s perspective, similarities emerge. Cerone et al. [20] characterizes PSI as requiring that transactions read from a causally consistent state and that concurrent transactions do not write the same object. Adya describes PL-2+ in intriguingly similar terms: “PL-2+ provides a notion of causal consistency since it ensures that a transaction is placed after all transactions that causally affect it” [1]. The “intuition gap” between how these guarantees are formally expressed and how they are experienced by clients makes it hard to appreciate how these guarantees actually compare.

Formulating isolation and consistency in terms of client-observable states eliminates this gap by forcing definitions that inherently specify guarantees according to how they are perceived by clients. The client-centric definition of PSI given below, for example, makes immediately clear that a valid PSI execution must ensure that all transactions observe the effects of transactions that they depend on.

Definition 17. For each transaction t, let its precede-set contain the set of transactions after which t is ordered. A transaction t’ precedes t 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.
As we observed earlier, applications are encouraged to think of weakly consistent systems as benevolent black boxes that hide, among others, the details of replication. Since replicas are invisible to clients, our new client-centric definitions of consistency and isolation make no mention of them, giving developers full flexibility in how these guarantees should be implemented. In Appendix D we prove that this client-centric, state-based definition of PSI is equivalent to both the axiomatic formulation of PSI \((PSI_A)\) by Cerone et al. and to the cycle-based specification of PL-2+:

**Theorem 5.** Let \(I\) be PSI. Then \(\exists e : \forall t \in T : CT_{PSI}(t, e) \equiv \neg G1 \land \neg G\text{-single}\)

**Theorem 6.** Let \(I\) be PSI. Then \(\exists e : \forall t \in T : CT_{PSI}(t, e) \equiv PSI_A\)

### 6.2 Composition

Formulating isolation and consistency guarantees on the basis of client-observable states makes them not just easier to understand, but also to compose. Composing such guarantees is often desirable in practice to avoid counter-intuitive behaviors. For example, when considered in isolation, monotonic reads lets transactions take effect in an order inconsistent with session order, and monotonic writes puts no constraints on reads. Systems like DocumentDB thus compose multiple such guarantees in their implementation, but have no way of articulating formally the new guarantee that their implementations offer. Expressing guarantees as local session tests makes it easy to formalize their composition.

**Definition 18.** A storage system satisfies a set of session guarantees \(G\) iff

\[
\forall se \in SE : \exists e : \forall t \in T_{se} : \forall SG \in G : SESSION_{SG}(se, t, e)
\]

An analogous definition specifies the meaning of combining isolation levels. Once formalized, such combinations can be easily compared against existing standalone consistency guarantees. For example, generalizing a result by Chockler et al’s [22], we prove in Appendix C that also when using transactions the four session guarantees, taken together, are equivalent to causal consistency.

**Theorem 7.** Let \(G = \{RMW, MR, MW, WFR\}\), then

\[
\forall se \in SE : \exists e : \forall t \in T_{se} : SESSION_{G}(se, t, e) \equiv \forall se \in SE : \exists e : \forall t \in T_{se} : SESSION_{CC}(se, t, e)
\]

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. Sometimes, however, constraining the isolation level of transactions within a session may be useful: think, for example, of a large-scale distributed system that would like transactions to execute atomically, while preserving session order. Once again, formulating isolation and consistency guarantees in terms of observable states makes expressing such requirements straightforward: all that is needed to modularly combine guarantees is to combine their corresponding commit and session tests.

**Definition 19.** A storage system satisfies a session guarantee \(SG\) and isolation level \(I\) iff

\[
\forall se \in SE : \exists e : \forall t \in T_{se} : SESSION_{SG}(se, t, e) \land CT_{I}(t, e).
\]

### 6.3 Scalability

As we observed earlier, applications are encouraged to think of weakly consistent systems as benevolent black boxes that hide, among others, the details of replication. Since replicas are invisible to clients, our new client-centric definitions of consistency and isolation make no mention of them, giving developers full flexibility in how these guarantees should be implemented.

In contrast, in their current system-centric formulations, several consistency and isolation guarantees not only explicitly refer to replicas, but they subject operations within each replica to stronger requirements than what is called for by these guarantees’ end-to-end obligations. For example, the original definition of parallel snapshot
isolation [52] requires individual replicas to enforce snapshot isolation, even as it globally only offers (as we prove in Theorem 5) the guarantees of Lazy Consistency/PL-2+ [1, 2]. Similarly, several definitions of causal consistency interpret the notion of thread of execution [3] as serializing all operations that execute on the same site [8, 21, 42]. Requiring individual sites to offer stronger guarantees than what applications can observe not only runs counter to an intuitive end-to-end argument, but—as these guarantees translate into unnecessary dependencies between operations and transactions that must then be honored across sites—can have significant implications on a system’s ability to provide a given consistency guarantee at scale. We show in Figure 6.3 that, for a simple transactional workload, guaranteeing per-site PL-2+ rather than snapshot isolation can reduce the number of dependencies per transaction by two orders of magnitude ($175 \times$).

7 Related work and conclusion

Most past definitions of isolation and consistency [1, 7, 12, 14–17, 22, 28, 33, 49, 52, 53] 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 [8, 20, 41] distinguishes between concrete executions (the nuts-and-bolts implementations detail) and abstract executions (the system behaviour as perceived by the client) on its way to introduce observable causal consistency, a refinement of causal consistency where causality can be inferred by client observations. Attiya et al. introduce the notion of observable causal consistency [8], a refinement on causal consistency where causality can be inferred by client observations. Likewise, Cerone et al. [20] introduce the dual notions of visibility and arbitration to define, axiomatically, a large number of existing isolation levels. All continue, however, to reason about correctness as constraints on ordering of read and write operations. 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 [4, 10].

Limitations Our model has two main limitations. First, it does not constrain the behavior of ongoing transactions, as it assumes that applications never make externally visible decisions based on uncommitted data. It thus cannot express consistency models, like opacity [31] or virtual world consistency [34] designed to prevent STM transactions from accessing an invalid memory location. Second, our model enforces a total order on executions, which does not easily account with recently proposed forking consistency models [18, 24, 38, 42]. We leave extending the model in this direction as future work.

Conclusion We present a new way to reason about consistency and isolation based on application-observable states. This approach (i) maps more naturally to what applications can observe, in turn making it obvious what anomalies distinct isolation/consistency levels allow; (ii) provides a structure to compose and compare isolation and consistency guarantees; and (iii) enables performance optimizations by reasoning about consistency guarantees end-to-end.

References


Appendix A  Adya et al’s model for specifying weak isolation

Adya et al. [1] 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 20. 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, $\prec$, 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 [15]: 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 21. Direct conflicts:

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

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

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

Definition 22. 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 $c_i \prec_t s_j$; otherwise, we have $b_j \prec_t c_i$.

Definition 23. 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 24. Start-Depends. $T_j$ start-depends on $T_i$ if $c_i \prec_t s_j$, i.e., if it starts after $T_i$ commits. We write $T_i \xrightarrow{b} T_j$

Definition 25. 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 phenomenemas, of which isolation levels proscribe a subset.

Definition 26. Phenomenemas:

$G_0$: Write Cycles A history $H$ exhibits phenomenon $G_0$ if $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges.

$G_{1a}$: Dirty Reads A history $H$ exhibits phenomenon $G_{1a}$ if it contains an aborted transaction $T_i$ and a committed transaction $T_j$ such that $T_j$ has read the same object (maybe via a predicate) modified by $T_i$.

$G_{1b}$: Intermediate Reads A history $H$ exhibits phenomenon $G_{1b}$ 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 first modification of $x$.

$G_{1c}$: Circular Information Flow A history $H$ exhibits phenomenon $G_{1c}$ if $DSG(H)$ contains a directed cycle consisting entirely of dependency edges.

$G_2$: Anti-dependency Cycles A history $H$ exhibits phenomenon $G_2$ 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.

$G_{-Sla}$: Interference A history $H$ exhibits phenomenon $G_{-Sla}$ 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 $T_j$. 

14
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 27. Isolations:

Serializability (PL-3) $\neg G1 \land \neg G2$

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

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

Snapshot Isolation $\neg G1 \land \neg G-SI$

Appendix B State-based and cycle-based model equivalence

This section proves the following theorems:

**Theorem 1.** Let $I$ be Serializability (SER). Then $\exists e : \forall t \in \mathcal{T}. CT_{SER}(t, e) \equiv \neg G1 \land \neg G2$.

**Theorem 2.** Let $I$ be Snapshot Isolation (SI). Then $\exists e : \forall t \in \mathcal{T}. CT_{SI}(t, e) \equiv \neg G1 \land \neg G-SI$

**Theorem 3.** Let $I$ be Read Committed (RC). Then $\exists e : \forall t \in \mathcal{T}. CT_{RC}(t, e) \equiv \neg G1$

Appendix B.1 Serializability

**Theorem 1.** Let $I$ be Serializability (SER). Then $\exists e : \forall t \in \mathcal{T}. CT_{SER}(t, e) \equiv \neg G1 \land \neg G2$.

**Proof.** We first prove $\neg G1 \land \neg G2 \Rightarrow \exists e : \forall t \in \mathcal{T} : CT_{SER}(t, e)$.

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 $t_1, ..., t_n$ 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 $\forall t \in \mathcal{T}. CT_{SER}(t, e)$. Specifically, we show that for all $t = t_{ij}$, COMPLETE$_{e,t_{ij}}(s_{t_{ij-1}})$ where $s_{t_{ij-1}}$ is the parent state of $t_{ij}$.

Consider the three possible types of operations in $t_{ij}$:

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 parent state of $t_{ij}$ is included in the read set of each of those operation types:

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

   We first show that $s_{t_{ik}} \rightarrow* s_{t_{ij-1}}$. As $t_{ij}$ directly read-depends on $t_{ik}$, there must exist an edge $t_{ik} \xrightarrow{wr} t_{ij}$ in $DSG(H)$, and $t_{ik}$ must therefore be ordered before $t_{ij}$ 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 $s_{t_{ik}} \xrightarrow{\rightarrow} s_{t_{ij-1}}$.

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

2. **Internal Reads.** Let $r_{ij}(x_{ij})$ read $x_{ij}$ such that $w(x_{ij}) \xrightarrow{w} r(x_{ij})$. By definition, the read state set of such an operation consists of $\forall s \in S_s : s \rightarrow s_p$. Since $s_{t_{ij-1}}$ is $t_{ij}$’s parent state, it trivially follows that $s_{t_{ij-1}} \in RS_e(r_{ij}(x_{ij}))$.

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

15
Thus $s_{ij-1} \in \bigcap_{o \in \Sigma_{ij}} RSe(o)$. We have $\text{COMPLETE}_{e, t_{ij}}(s_{ij-1})$ for any $t_{ij} : \forall t \in T : \text{CT}_{SER}(t, e)$.

($\Leftarrow$) We next prove $\exists e : \forall t \in T : \text{CT}_{SER}(t, e) \Rightarrow \neg G1 \land \neg G2$.

To do so, we prove the contrapositive $G1 \lor G2 \Rightarrow \forall e \exists t \in T : \neg \text{CT}_{SER}(t, e)$. Let $H$ be a history that displays phenomena $G1$ or $G2$. We generate a contradiction. Consider any execution $e$ such that $\forall t \in T : \text{CT}_{SER}(t, e)$.

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 $x \in W_{t_i} \cap W_{t_j}$ and $s_{t_i} \xrightarrow{w} s_{t_j}$.

First, we show that:

**Claim 1.** $t_i \rightarrow t_j$ in $\text{DSG}(H) \Rightarrow s_{t_i} \xrightarrow{w} s_{t_j}$ in the execution $e$ $(i \neq j)$.

**Proof.** Consider the three edge types in $\text{DSG}(H)$:

1. Let $t_i \xrightarrow{w} t_j$ there exists an object $x$ s.t. $x_i << x_j$ (version order). By construction, we have $s_{t_i} \xrightarrow{w} s_{t_j}$.
2. Let $t_i \xrightarrow{r} t_j$ there exists an object $x$ s.t. $t_i$ reads version $x_i$ written by $t_i$. Let $s_{t_k}$ be the parent state of $s_{t_j}$, i.e. $s_{t_k} \rightarrow s_{t_j}$. By assumption $\text{CT}_{SER}(e, t) (t = t_j)$ i.e. $\text{COMPLETE}_{e, t_j}(s_{t_k})$, 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: $s_{t_i} \xrightarrow{w} s_{t_k} \rightarrow s_{t_j}$.
3. Let $t_i \xrightarrow{r} t_j$ 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 $s_{t_m} \xrightarrow{w} s_{t_j}$. Let $s_{t_k}$ be the parent state of $s_{t_j}$, i.e. $s_{t_k} \rightarrow s_{t_j}$. As $\text{CT}_{SER}(e, t)$, where $t = t_j$, holds by assumption, i.e. $\text{COMPLETE}_{e, t_j}(s_{t_k})$, the key-value pair $(x, x_m) \in s_{t_k}$, hence $s_{t_m} \xrightarrow{w} s_{t_k}$ as before. In contrast, $s_{t_i} \xrightarrow{w} s_{t_j}$. Hence, $t_j$ has not yet been applied. We thus have $s_{t_k} \rightarrow s_{t_i} \xrightarrow{w} s_{t_j}$.

We now derive a contradiction in all cases of the disjunction $G1 \lor G2$:

- 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. $T$ and any corresponding execution $e$, however, consists only of committed transactions. Hence $\forall e : \exists s \in S_e, s.t. s \in RSe(r_j(x_i))$: 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, $\exists s \in S_e, s.t. s \in RSe(r_{(x_{\text{intermediate}})})$. 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 $\text{DSG}$. Hence, there must exist a chain of transactions $t_1 \rightarrow t_i+1 \rightarrow ... \rightarrow t_j$ such that $i = j$ in $\text{DSG}(H)$. By Claim 1, we thus have $s_{t_i} \xrightarrow{w} s_{t_{i+1}} \xrightarrow{w} \ldots \xrightarrow{w} s_{t_j}$ 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 $G1 \lor G2 \Rightarrow \forall e : \exists t \in T : \neg \text{CT}_{SER}(e, t)$. This completes the proof.

### Appendix B.2 Snapshot Isolation

**Theorem 2.** Let $I$ be Snapshot Isolation (SI). Then $\exists e : \forall t \in T : \text{CT}_{SI}(t, e) \equiv \neg G1 \land \neg G2 - SI$

**Proof.** We first prove $\neg G1 \land \neg G2 - SI \Rightarrow \exists e : \forall t \in T : \text{CT}_{SI}(t, e)$.

**LDG(H) properties** To do so, we introduce the notion of *logical order* between transactions and capture this relationship in a logic-order directed graph (LDG). This order refines the pre-existing start-order present in snapshot isolation to include transitive observations. $\text{LDG}(H)$ contains, as nodes, the set of committed transactions in $H$ and includes the following two edges:

- $t_i \xrightarrow{\text{SI}} t_j$ iff $t_i \xrightarrow{sd} t_j$ (and $t_i \neq t_j$). This is a simple renaming of the sd-edge, for clarity.
- $t_i \xrightarrow{\text{SI}} t_j$ iff $\exists t_k : t_i \xrightarrow{sd} t_k \xrightarrow{rw} t_j$. Intuitively, this edge captures the observation that if $t_1$ is ordered before $t_2$ and $t_2$’s reads entail that $t_3$ is before $t_3$, then $t_1$ must be before $t_3$. We note that, by construction $t_1 \neq t_j$ (otherwise, the history $H$ will display a G-SIb cycle).
Both l1 and l2 edges are transitive:

Claim 2. \(l1\) edge is transitive: \(t_1 \overset{l1}{\rightarrow} t_2 \overset{l1}{\rightarrow} t_3 \Rightarrow t_1 \overset{l1}{\rightarrow} t_3\)

Proof. We have \(t_1 \overset{l1}{\rightarrow} t_2 \iff t_1 \overset{sd}{\rightarrow} t_2 \iff c_1 \prec_t b_2\). By definition of \(\prec_t\), \(b_2 \prec_t c_2\). Hence, \(t_1 \overset{l1}{\rightarrow} t_2 \iff b_2 \prec_t c_2\). Similarly, \(t_2 \overset{l1}{\rightarrow} t_3 \iff c_2 \prec_t b_3\). Consequently, we have \(c_1 \prec_t b_2 \prec_t c_2 \prec_t b_3\). By definition \(c_1 \prec b_3 \Rightarrow t_1 \overset{sd}{\rightarrow} t_3 \Rightarrow t_1 \overset{l1}{\rightarrow} t_3\). The proof is complete.

Claim 3. \(l2\) edge is transitive: \(t_1 \overset{l2}{\rightarrow} t_2 \overset{l2}{\rightarrow} t_3 \Rightarrow t_1 \overset{l2}{\rightarrow} t_3\)

Proof. By definition of \(l2\), there must exist transactions \(t_{12}, t_{23}\), such that \(t_1 \overset{sd}{\rightarrow} t_{12} \overset{rw}{\rightarrow} t_2 \overset{sd}{\rightarrow} t_{23}\). We show that: \(t_1 \overset{sd}{\rightarrow} t_{23}\). By definition, \(t_1 \overset{sd}{\rightarrow} t_{12} \iff c_1 \prec_t b_{12}\) and \(t_2 \overset{sd}{\rightarrow} t_{23} \iff c_2 \prec_t b_{23}\). By assumption, the history \(H\) does not contain an G-SIb cycle, there thus cannot exists an edge \(t_{12} \overset{rw}{\rightarrow} t_2\) as \(t_2\) either happens after or is concurrent with \(t_{12}\). In both cases, we have \(b_{12} \prec_t c_2\). The following relation thus hold: \(c_1 \prec_t b_{12} \prec_t c_2 \prec_t b_{23}\). By definition, the edge \(t_1 \overset{sd}{\rightarrow} t_{23}\) is therefore included in the \(SSG(H)\), as is the path \(t_1 \overset{sd}{\rightarrow} t_{23} \overset{rw}{\rightarrow} t_3\). Hence, by construction, \(t_1 \overset{l2}{\rightarrow} t_3\) exists in \(LDG(H)\). This completes the proof.

Claim 4. \(t_1 \overset{l2}{\rightarrow} t_2 \overset{l1}{\rightarrow} t_3 \Rightarrow t_1 \overset{l1}{\rightarrow} t_3\)

Proof. If \(t_1 \overset{l2}{\rightarrow} t_2 \overset{l1}{\rightarrow} t_3\) exists in \(LDG(H)\), there must exist a transaction \(t_{12}\) such that \(t_1 \overset{sd}{\rightarrow} t_{12} \overset{rw}{\rightarrow} t_2 \overset{sd}{\rightarrow} t_{32}\) in \(SSG(H)\). By definition, \(t_1 \overset{sd}{\rightarrow} t_{12} \Rightarrow c_1 \prec_t b_{12}\) and \(t_2 \overset{sd}{\rightarrow} t_{32} \Rightarrow c_2 \prec_t b_{23}\). Moreover, by assumption, the history \(H\) does not contain an G-SIb cycle; there thus cannot exists an edge \(t_{12} \overset{rw}{\rightarrow} t_2\) as \(t_2\) either happens after or is concurrent with \(t_{12}\). In both cases, we have \(b_{12} \prec_t c_2\). Thus the following inequalities hold: \(c_1 \prec_t b_{12} \prec_t c_2 \prec_t b_{23}\). By definition, we thus have \(t_1 \overset{l1}{\rightarrow} t_3\).

Finally, we prove the following lemma:

Lemma 1. \(LDG(H)\) is acyclic if \(H\) disallows phenomena G1 and G-SI.

Proof. \(l1\)-edges First, we show that \(LDG(H)\) does not contain a cycle consisting only of \(l1\) edges. \(l1\) edges are transitive (Claim 2). Hence any cycle consisting only of \(l1\) edges will result in a self-loop, which we have argued is impossible. Thus no such cycle can exist.

\(l2\)-edges Next, we show that \(LDG(H)\) does not contain a cycle consisting only of \(l2\) edges. The proof proceeds as above: \(l2\) edges are transitive (Claim 3). Hence any cycle consisting of \(l2\) edges only will result in a self-loop, which we have argued is impossible. Thus no such cycle can exist.

\(l1/l2\)-edges Finally, we show that \(LDG(H)\) does not contain a cycle consisting of both \(l1\) and \(l2\) edges. We do this by contradiction. Assume such a cycle \(cyc_1\) exists. It must contain the following sequence of transactions: \(t_1 \overset{l2}{\rightarrow} t_2 \overset{l1}{\rightarrow} t_3\) in \(LDG(H)\). By Claim 4, there must exist an alternative cycle \(cyc_2\) containing the same set of edges as \(cyc_1\), but with the edges \(t_1 \overset{l2}{\rightarrow} t_2 \overset{l1}{\rightarrow} t_3\) replaced by the edge \(t_1 \overset{l1}{\rightarrow} t_3\). If \(t_1 \overset{l2}{\rightarrow} t_2\) was the only \(l2\) edge in \(cyc_1\), \(cyc_2\) is a cycle with only \(l1\) edges, which we previously proved could not arise. Otherwise, we apply the same reasoning, starting from \(cyc_2\), and generate an equivalent cycle \(cyc_3\) consisting of one fewer \(l2\) edge, until we obtain a cycle \(cyc_n\) with all \(n l2\) edges removed, and consisting only of \(l1\) edges. In all cases, we generate a contradiction, and no cycle consisting of both \(l1\) and \(l2\) edges can exist. This concludes the proof.

Commit Test Armed with an acyclic \(LDG(H)\), we can construct an execution \(e\) such that every committed transaction satisfies the commit test \(CT_{S}(t, e)\). Let \(i_1, \ldots, i_n\) be a permutation of \(1, 2, \ldots, n\) such that \(i_1, \ldots, i_n\) is a topological sort of \(LDG(H)\) \((LDG(H)\) is acyclic and can thus be topologically sorted). We construct an execution \(e\) according to the topological order defined above: \(e : s_0 \rightarrow s_{i_1} \rightarrow s_{i_2} \rightarrow \ldots \rightarrow s_{i_n}\) and show that \(\forall t \in T.CT_{S}(t, e)\). Specifically, we prove the following: let there be a largest \(k\) such that \(t_{ik} \overset{l1}{\rightarrow} t_{ij}\), then \(\text{COMPLETE}_{e,t_{ij}}(s_{ik}) \land (\Delta(s_{ik}, s_{i_{j-1}}) \cap \mathcal{W}_{s_{i_{j-1}}} = \emptyset)\).

Complete State We first prove that \(\text{COMPLETE}_{e,t_{ij}}(s_{ik})\). Consider the three possible types of operations in \(t_{ij}\):

1. \(\text{External Reads}\): an operation reads an object version that was created by another transaction.
2. \(\text{Internal Reads}\): an operation reads an object version that itself created.
3. \(\text{Writes}\): an operation creates a new object version.

We show that the \(s_{ik}\) is included in the read set of each of those operation types:
1. External Reads. Let $r_{ij}(x_{i_q})$ read the version for $x$ created by $t_{i_q}$, where $q \neq j$.

We first show that $s_{t_{ik}} \xrightarrow{w} s_{t_{iq}}$. As $t_{ij}$ directly read-depends on $t_{i_q}$, there must exist an edge $t_{i_q} \xrightarrow{wr} t_{ij}$ in $SSG(H)$. Given that $H$ disallows phenomenon G-SI by assumption, there must therefore exist a start-dependency edge $t_{i_q} \xrightarrow{sd} t_{ij}$ in $SSG(H)$. $LDG(H)$ will consequently contain the following edge: $t_{i_q} \xrightarrow{1} t_{ij}$. Given $e$ was constructed by applying every transaction $T$ in topological order, and that we select the largest $k$ such that $t_{i_k} \xrightarrow{1} t_{ij}$, it follows that $q < k < j$ and $s_{t_{ik}} \xrightarrow{w} s_{t_{i_k}}$.

Next, we argue that the state $s_{t_{ik}}$ contains the object value pair $(x, x_{i_q})$. Specifically, we argue that there does not exist a $s_{t_{im}}$, where $q < m \leq k$, such that $t_{im}$ writes a new version of $x$. We prove this by contradiction. Consider the smallest such $m$: $t_{ik}$ reads the version of $x$ written by $t_{iq}$ and $t_{im}$ writes the next version of $x$. $t_{ij}$ thus anti-depends on $t_{im}$, i.e., $t_{ij} \xrightarrow{rw} t_{im}$. In addition, it holds by assumption that $t_{i_k} \xrightarrow{1} t_{ij}$. Hence, the following sequence of edges exists in $SSG(H)$: $t_{ik} \xrightarrow{sd} t_{ij} \xrightarrow{rw} t_{im}$. Equivalently, $LDG(H)$ will contain the edge $t_{i_k} \xrightarrow{12} t_{im}$. As the topological sort of $LDG(H)$ from which we constructed $e$ respects l1 edges, we finally have $s_{t_{ik}} \xrightarrow{+} s_{t_{im}} (k < m)$, a contradiction. We conclude: $(x, x_{i_q}) \in s_{t_{ik}}$ and therefore $s_{t_{ik}} \in \mathcal{RS}_e(r_{ij}(x_{i_q}))$.

2. Internal Reads. Let $r_{ij}(x_{i_q})$ read $x_{i_q}$ such that $w(x_{i_q}) \xrightarrow{IO} r(x_{i_q})$. By definition, the read state set of such an operation consists of $\forall s \in \mathcal{S}_e: s \xrightarrow{w} s_r$. Since $s_{t_{ij}}$ is precedes $s_{t_{ij}}$ in the topological order ($t_{i_j} \xrightarrow{11}$-precedes $t_{ij}$ and $e$ respects l1 edges), it trivially follows that $s_{t_{ik}} \in \mathcal{RS}_e(r_{ij}(x_{i_q}))$.

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

Thus $s_{t_{ik}} \in \bigcap_{e \in \Sigma e_{i_j}} \mathcal{RS}_e(o)$.

**Distinct Write Sets** We now prove the second half of the commit test: $(\Delta(s_{t_{ik}}, s_{t_{ij-1}}) \cap W_{s_{t_{ij}}}) = \emptyset$ We prove this by contradiction. Consider the largest $m$, where $k < m < j$ such that $W_{s_{t_{im}}} \cap W_{s_{t_{ij}}} \neq \emptyset$. $t_{im}$ thus directly write-depends on $t_{ij}$, i.e., $t_{im} \xrightarrow{w} t_{ij}$. By assumption, $H$ proscribes phenomenon G-SI. Hence, there must exist an edge $t_{im} \xrightarrow{sd} t_{ij}$ in $SSG(H)$, and equivalently $t_{im} \xrightarrow{1} t_{ij}$ in $LDG(H)$. As $e$ respects the topological order of $LDG(H)$, it follows that $s_{im} \xrightarrow{+} s_{ij} (m < j)$.

By assumption however, $t_{ik}$ is the latest transaction in $e$ such that $t_{i_k} \xrightarrow{1} t_{ij}$, so $m \leq k$. Since we had assumed that $k < m < j$, we have a contradiction. Thus, $\forall m, k < m < j, W_{s_{t_{im}}} \cap W_{s_{t_{ij}}} = \emptyset$. We conclude that $(\Delta(s_{t_{ik}}, s_{t_{ij-1}}) \cap W_{s_{t_{ij}}}) = \emptyset$ for any $t_{ij} \in T$.\(\forall t \in T : CT_{SI}(t, e)\).

(\(\Leftarrow\)) We next prove $\exists e: \forall t \in T : CT_{SI}(t, e) \Rightarrow \neg G1 \land \neg G-SI$.

Let $e$ be an execution such that $\forall t \in T : CT_{SI}(t, e)$, and $H$ be a history for committed transactions $T$. 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 $x \in W_{s_{t_{mi}}} \cap W_{s_{t_{ij}}} \land s_{t_{mi}} \xrightarrow{w} s_{t_{ij}}$. It follows that, for any two states such that $(x, x_i) \in t_{im} \land (x, x_j) \in t_{ij} \Rightarrow s_{t_{im}} \xrightarrow{+} s_{t_{ij}}$. 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 $s_{t_0} \rightarrow s_{t_1} \rightarrow s_{t_2} \rightarrow \cdots \rightarrow s_{t_n}$, our timestamp assignment logic is then the following:

1. Let $i = 0$.
2. Set $s = s_{t_0}$; if $i = 0$, $s = s_0$.
3. Assign a commit timestamp to $t_{si}$ if $i \neq 0$.
4. Assign a start timestamp to all transactions $t_k$ such that $t_k$ satisfies $\text{COMPLETE}_{e, t_k}(s) \land (\Delta(s, s_{t_k}(t_k)) \cap W_{s_{t_k}}) = \emptyset$ 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 5.** $\forall t_{ij}, t_{j} \in T : s_{t_{ij}} \xrightarrow{w} s_{t_{ij}} \Rightarrow \neg t_{ij} \xrightarrow{sd} t_{ij}$
Proof. We have $t_i \xrightarrow{s_{ij}} t_j \Rightarrow c_i \prec_t b_j$ 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 $s_{ti} \triangleright s_{tj}$. We conclude: $t_i \xrightarrow{s_{ij}} t_j \Rightarrow s_{ti} \triangleright s_{tj}$. Taking the contrapositive of this implication completes the proof. **\[\square\]**

**G1** We first prove that: $\forall t \in T : CT_{SI}(t, e) \Rightarrow \neg G1$. We do so by contradiction.

**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. $T$ and any corresponding execution $e$, however, consists only of committed transactions. Hence $\forall e' : E \in E_0 : s \in RS_e(r(x_{Intermediate}))$ 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, $\exists s \in S_e : s \in RS_e(r(x_{Intermediate}))$. 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:

- $t_i \xrightarrow{w_t} t_j$ There exist an object $x$ such that $x_i \ll x_j$ (version order). By construction, version in $H$ is consistent with the execution order $e$: we have $s_{ti} \triangleright s_{tj}$.
- $t_i \xrightarrow{r_t} t_j$ There exist an object $x$ such that $t_j$ reads version $x_i$ written by $t_i$. By assumption, $CT_{SI}(e, t_j)$ holds. There must therefore exists a state $s_{tk} \in S_e$ such that $COMPLETE_{e, t_i}(s_{tk})$. If $s_{tk}$ is a complete state for $t_j$, $s_{tk} \in RS_e(r(x_i))$ and $(x, x_i) \in s_{tk}$. For the effects of $t_i$ to be visible in $s_{tk}$, $t_i$ must have been applied at an earlier point in the execution. Hence we have: $s_{ti} \triangleright s_{tk}$. Moreover, by definition of the candidate read states, $s_{tk} \triangleright s_p(t_j) \rightarrow s_{tj}$ (Definition 2). It follows that $s_{ti} \triangleright s_{tj}$.

If a history $H$ displays phenomenon G1c, there must exist a chain of transactions $t_i \rightarrow t_{i+1} \rightarrow \ldots \rightarrow t_j$ such that $i = j$. A corresponding cycle must thus exist in the execution $e$: $s_{ti} \triangleright s_{t_{i+1}} \triangleright \ldots \triangleright s_{t_j}$. 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 $\forall t \in T : CT_{SI}(t, e) \Rightarrow \neg G-SI$.

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

- $t_i \xrightarrow{w_r} t_j$ There must exist an object $x$ such that $t_j$ reads version $x_i$ written by $t_i$. Let $s_{tk}$ be the first state in $e$ such that $COMPLETE_{e, t_i}(s_{tk}) \land (\Delta(s_{tk}, s_p(t_j)) \cap W_{s_{tj}} = \emptyset)$. Such a state must exist since $CT_{SI}(e, t_j)$ holds by assumption. As $s_{tk}$ is complete, we have $(x, x_i) \in s_{tk}$. For the effects of $t_i$ to be visible in $s_{tk}$, $t_i$ must have been applied at an earlier point in the execution. Hence we have: $s_{ti} \triangleright s_{tk} \triangleright s_{tj}$. It follows from our timestamp assignment logic that $c_i \leq c_k$. Similarly, the start point of $t_j$ must have been assigned after $t_k$'s commit point (as $s_{tk}$ 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 $t_i \xrightarrow{s_{ij}} t_j$. $H$ will not display G-SIa for write-read dependencies.

- $t_i \xrightarrow{w_w} t_j$ There must exist an object $x$ such that $t_j$ writes the version $x_j$ that follows $x_i$. By construction, it follows that $s_{ti} \triangleright s_{tj}$. Let $s_{tk}$ be the first state in the execution such that $COMPLETE_{e, t_i}(s_{tk}) \land (\Delta(s_{tk}, s_p(t_j)) \cap W_{s_{tj}} = \emptyset)$. We first show that: $s_{ti} \triangleright s_{tk}$. Assume by way of contradiction that $s_{ti} \triangleright s_{tk}$. The existence of a write-write dependency between $t_i$ and $t_j$ implies that $W_{t_i} \cap W_{t_j} \neq \emptyset$, and consequently, that $\Delta(s_{tk}, s_p(t_j)) \cap W_{s_{tj}} \neq \emptyset$, contradicting our assumption that $CT_{SI}(e, t_j)$. We conclude that: $s_{ti} \triangleright s_{tk}$. It follows from our timestamp assignment logic that $c_i \leq c_k$. Similarly, the start point of $t_j$ must have been assigned after $t_k$'s commit point (as $s_{tk}$ 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 $t_i \xrightarrow{s_{ij}} t_j$. $H$ 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 (Claim 2), 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 \(s_{tk}\) be the earliest complete state of \(t_i\). Such a state must exist as \(CT_{S_j}(t_i)\) by assumption. Hence, by definition of read state \((x, x_m) \in s_{tk}\). Similarly, \((x, x_j) \in s_{tj}\) by the definition of state transition (Definition 1).

By construction, we have \(s_{tk} \xrightarrow{} s_{tj}\). Our timestamp assignment logic maintains the following invariant: given a state \(s_t\), \(\forall t_k : COMPLETE_{c,j_k}(s_k) : \forall s_{tk} : s_t \xrightarrow{} s_{tk} \Rightarrow b_k \prec \pi c_n\). Intuitively, the start timestamp of all transactions associated with a particular complete state \(s_t\) smaller than the commit timestamp of any transaction that follows \(s_t\) in the execution. We previously showed that \(s_{tk} \xrightarrow{} s_{tj}\) is a complete state for \(t_i\). We conclude \(b_i \prec \pi c_j\). However, the edge \(t_j \xrightarrow{sd} t_i\) implies that \(c_j \prec \pi c_i\). 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 \(\forall t \in T : CT_{S_j}(t, e) \Rightarrow \neg G-SI \land G1\). This completes the proof.

\[\Box\]

**Appendix B.3 Read Committed**

**Theorem 3.** Let \(T\) be Read Committed (RC). Then \(\exists e : \forall t \in T.\) CT\(_{RC}(t, e) \equiv \neg G1\).

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

Let \(H\) define a history over \(T = \{t_1, t_2, \ldots, t_n\}\) and let \(DSG(H)\) be the corresponding direct serialization graph. \(\neg G1c\) 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 \(t_1, \ldots, t_n\) be a permutation of \(1, 2, \ldots, n\) such that \(t_{t_1}, \ldots, t_{t_n}\) 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: \(e : s_0 \rightarrow s_{t_1} \rightarrow s_{t_2} \rightarrow \ldots \rightarrow s_{t_m}\) and show that \(\forall t \in T.\) CT\(_{RC}(t, e)\). Specifically, we show that for all \(t = t_i,\) PR\(_{READ}(t)\).

Consider the three possible types of operations in \(t_i\):

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 \(r_{ij}(x_{ik})\) read the version for \(x\) created by \(t_{ik}\), where \(k \neq j\).

   We first show that \(s_{t_{ik}} \xrightarrow{} s_{t_{ij}}\). As \(t_{ij}\) directly read-depends on \(t_{ik}\), there must exist an edge \(t_{ik} \xrightarrow{wt} t_{ij}\) in \(SDSG(H)\), and \(t_{ik}\) must therefore be ordered before \(t_{ij}\) in the topological sort of \(SDSG(H)\) (\(k < j\)). It follows that \(s_{t_{ik}} \xrightarrow{} s_{t_{ij}}\).

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

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

Thus \(\forall o \in S_e : RS\(_e(o)\) \neq \emptyset\). We have PR\(_{READ}(t_{ij})\) for any \(t_{ij} : \forall t \in T : CT\(_{RC}(t, e)\).

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

To do so, we prove the contrapositive \(G1 \Rightarrow \forall e \exists t \in T : \neg 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 T : 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 \(x \in W_{t_i} \cap W_{t_j} \land s_{t_i} \xrightarrow{} s_{t_j}\).

First, we show that:

**Claim 6.** \(t_i \rightarrow t_j\) in \(SDSG(H) \Rightarrow s_{t_i} \xrightarrow{} s_{t_j}\) in the execution \(e\) (\(i \neq j\)).

**Proof.** Consider the three edge types in \(DSG(H)\):
We prove the following theorems: All cases generate a contradiction. We have $G_1 \implies 8$

We now derive a contradiction in all cases of $G_1$:

- Let us assume that $H$ exhibits phenomenon $G_1a$ (aborted reads). There must exist events $w_i(x_i), r_j(x_i)$ in $H$ such that $t_i$ subsequently aborted. $T$ and any corresponding execution $e$, however, consists only of committed transactions. Hence $\forall e : \exists s \in S_e, s.t. s \in RS_e(r_j(x_i))$: 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 $G_1b$ (intermediate reads). In an execution $e$, only the final writes of a transaction are applied. Hence, $\exists s \in S_e, s.t. s \in RS_e(r(x_{intermediate}))$. 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 $G_1c$. Any history that displays $G_1c$ will contain a cycle in the SDSG($H$). Hence, there must exist a chain of transactions $t_i \to t_{i+1} \to \ldots \to t_j$ such that $i = j$. By Claim 6, we thus have $s_{t_i} \xrightarrow{s} s_{t_{i+1}} \xrightarrow{\ldots} s_{t_j}$, $i = j$ 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 $G_1 \implies \exists e : \exists t \in T : \neg CTRC(e,t)$. This completes the proof.

\section*{Appendix C \ Causality and Session Guarantees}

We prove the following theorems:

\textbf{Theorem 7} Let $G = \{RMW, MR, MWM, WFR\}$, then 
$\forall e \in SE : \exists e : \forall t \in T_{se} : SESSION_{G}(se, t, e) \equiv \exists e \in SE : \exists t \in T_{se} : SESSION_{CC}(se, t, e)$

We first state a number of useful lemmas about the PREREAD, ($T$) predicate (Definition 3): if PREREAD,($T$) holds, then the candidate read set of all operations in all transactions in $T$ 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.

\textbf{Lemma 2.} For any $T'$ such that $T' \subseteq T$, $\text{PREREAD}_c(T') \iff \forall o \in T' : \forall o \in \Sigma_{t} : sf_{o} \xrightarrow{\downarrow s_{t}} t_{t}$. 

\textbf{Proof.} (\implies) We first prove $\text{PREREAD}_c(T') \Rightarrow \forall t \in T' : \forall o \in \Sigma_{t} : sf_{o} \xrightarrow{\downarrow s_{t}} t_{t}$. 

By the definition of $\text{PREREAD}_c(T')$, we have $\forall t \in T' : \forall o \in \Sigma_{t} : RS_{c}(o) \neq \emptyset$. 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 $RS_{c}(o) = \{s \in S_{c} : s \xrightarrow{s} s_{p} \land ((k, v) \in s \lor \exists w(k, v) \in \Sigma_{t} : w(k, v) \xrightarrow{t} r(k, v))\}$. The disjunction considers two cases:

1. **Internal Reads** if $\exists w(k, v) \in \Sigma_{t} : w(k, v) \xrightarrow{t} r(k, v)$, $RS_{c}(o) = \{s \in S_{c} : s \xrightarrow{s} s_{p}\}$. Hence $s_{0} \in RS_{c}(o)$. It follows that $sf_{o} = s_{0} \xrightarrow{\downarrow s_{t}} t_{t}$.

2. **External Reads** By $\text{PREREAD}_c(T')$, we have that $RS_{c}(o) \neq \emptyset$. There must therefore exist a state $s \in S_{c}$ such that $s \xrightarrow{s} s_{p} \land (k, v) \in s$. Since $sf_{o}$ is, by definition, the first such $s$, we have that $sf_{o} \xrightarrow{s} s_{p} \rightarrow t_{t}$. We conclude: $sf_{o} \xrightarrow{\downarrow s_{t}} t_{t}$.

**Writes** The candidate read states set for write operations $o = w(k, v)$, is defined as $RS_{c}(o) = \{s \in S_{c} : s \xrightarrow{s} s_{p}\}$. Hence $s_{0} \in RS_{c}(o)$. It trivially follows that $(sf_{o} = s_{0}) \xrightarrow{\downarrow s_{t}} t_{t}$.

We conclude: $\text{PREREAD}_c(T') \Rightarrow \forall t \in T' : \forall o \in \Sigma_{t} : sf_{o} \xrightarrow{\downarrow s_{t}} t_{t}$.

(\implies) Next, we prove that $\text{PREREAD}_c(T') \iff \forall t \in T' : \forall o \in \Sigma_{t} : sf_{o} \xrightarrow{\downarrow s_{t}} t_{t}$.

By assumption, $\forall t \in T' : \forall o \in \Sigma_{t} : sf_{o} \xrightarrow{\downarrow s_{t}} t_{t}$. By definition, $sf_{o} \in RS_{c}(o)$. It trivially follows that $\forall t \in T' : \forall o \in \Sigma_{t} : RS_{c}(o) \neq \emptyset$, i.e. $\text{PREREAD}_c(T')$ holds.

\hfill \Box
Lemma 3. For any $\mathcal{T}'$ that $\mathcal{T}' \subseteq \mathcal{T}$, PREREAD$_e(\mathcal{T}) \Rightarrow$ PREREAD$_e(\mathcal{T}')$.

Proof. Given PREREAD$_e(\mathcal{T})$, by definition we have that $\forall t \in \mathcal{T}, \forall o \in \Sigma_t, RS_e(o) \neq \emptyset$. Since $\mathcal{T}' \subseteq \mathcal{T}$, $\forall t \in \mathcal{T}' \Rightarrow \exists t \in \mathcal{T}$, it follows that $\forall t \in \mathcal{T}', \forall o \in \Sigma_t, RS_e(o) \neq \emptyset$, i.e. PREREAD$_e(\mathcal{T}')$.

We now begin in earnest our proof of Theorem 7.

(\Rightarrow) We first prove that $\forall se \in SE : \exists e : \forall t \in \mathcal{T}_se : SESSIONCC(se, t, e) \Rightarrow \forall se \in SE : \exists e : \forall t \in \mathcal{T}_se : SESSION_G(se, t, e)$.

For any $se \in SE$, consider the execution $e$, such that $\forall t \in \mathcal{T}_se : SESSIONCC(se, t, e)$. We show that this same execution satisfies the session test of all four session guarantees.

- **CC \Rightarrow RMW:** By assumption, SESSION$_{CC}(se, t, e)$ for all $\mathcal{T}_se$. Hence: $\forall t \in \mathcal{T}_se : \forall o \in \Sigma_t : \forall t' \overset{se}{\rightarrow} t : s_{t'} \overset{e}{\rightarrow} s_{lo}$. Weakening this statement gives the following implication: $\forall o \in \Sigma_t : \forall t' \overset{se}{\rightarrow} t : (W_i \neq \emptyset \land W_j \neq \emptyset) \Rightarrow s_{t'} \overset{e}{\rightarrow} s_{lo}$. Additionally, $e$ satisfies PREREAD$_e(\mathcal{T})$ (by assumption) and therefore PREREAD$_e(\mathcal{T}_se)$ as $\mathcal{T}_se \subseteq \mathcal{T}$ (by Lemma 3). Putting it all together: $e$ satisfies PREREAD$_e(\mathcal{T}_se) \land \forall o \in \Sigma_t : \forall t' \overset{se}{\rightarrow} t : (W_i \neq \emptyset \land W_j \neq \emptyset) \Rightarrow s_{t'} \overset{e}{\rightarrow} s_{lo}$. We conclude that $\forall se \in SE : \exists e : \forall t \in \mathcal{T}_se : SESSION_{RMW}(se, t, e)$.

- **CC \Rightarrow MW:** By assumption, SESSION$_{CC}(se, t, e)$ for all $t \in \mathcal{T}_se$. Hence, it holds that $\forall se' \in SE : \forall t_i \overset{se'}{\rightarrow} t_j : s_{t_i} \overrightarrow{s} s_{t_j}$. Weakening this statement gives the following implication: $\forall se' \in SE : \forall t_i \overset{se'}{\rightarrow} t_j : (W_i \neq \emptyset \land W_j \neq \emptyset) \Rightarrow s_{t_i} \overrightarrow{s} s_{t_j}$. Additionally, $e$ satisfies PREREAD$_e(\mathcal{T})$ (by assumption) and therefore PREREAD$_e(\mathcal{T}_se)$ as $\mathcal{T}_se \subseteq \mathcal{T}$ (by Lemma 3). Putting it all together: $e$ satisfies PREREAD$_e(\mathcal{T}_se) \land \forall se' \in SE : \forall t_i \overset{se'}{\rightarrow} t_j : (W_i \neq \emptyset \land W_j \neq \emptyset) \Rightarrow s_{t_i} \overrightarrow{s} s_{t_j}$. We conclude that $\forall se \in SE : \exists e : \forall t \in \mathcal{T}_se : SESSION_{MW}(se, t, e)$.

- **CC \Rightarrow MR:** By assumption, SESSION$_{CC}(se, t, e)$, hence $e$ ensures that $\forall t \in \mathcal{T}_se : \forall o \in \Sigma_t : \forall t' \overset{se}{\rightarrow} t : s_{t'} \overrightarrow{s} s_{lo}$. Moreover, by assumption, we have that PREREAD$_e(\mathcal{T})$. It follows that $\forall t' \in \mathcal{T} : \forall o' \in \Sigma_{t'} : sf_{o'} \overrightarrow{s} s_{t'}$ (Lemma 2). Combining the two statements, we have $\forall o \in \Sigma_t : \forall t' \overset{se}{\rightarrow} t : \forall o' \in \Sigma_{t'} : sf_{o'} \overrightarrow{s} s_{t'} \Rightarrow s_{t'} \overrightarrow{s} s_{lo}$, i.e. $sf_{o'} \overrightarrow{s} s_{lo}$. Finally, we have that $e$ satisfies IRC$_e(t)$ by assumption, and PREREAD$_e(\mathcal{T}_se)$ by Lemma 3: we have PREREAD$_e(\mathcal{T})$ and $\mathcal{T}_se \subseteq \mathcal{T}$. Putting it all together, $e$ satisfies PREREAD$_e(\mathcal{T}_se) \land IRC_e(t) \land \forall o \in \Sigma_t : \forall t' \overset{se}{\rightarrow} t : \forall o' \in \Sigma_{t'} : sf_{o'} \overrightarrow{s} s_{lo}$. We conclude that $\forall se \in SE : \exists e : \forall t \in \mathcal{T}_se : SESSION_{MR}(se, t, e)$.

- **CC \Rightarrow WFR:** By assumption, SESSION$_{CC}(se, t, e)$. Hence $e$ satisfies $\forall se' \in SE : \forall t_i \overset{se'}{\rightarrow} t_j : s_{t_i} \overrightarrow{s} s_{t_j}$. By assumption, $e$ respects PREREAD$_e(\mathcal{T})$. It follows from Lemma 2 that $\forall t_i \in \mathcal{T} : \forall o_i \in \Sigma_{t_i} : sf_{o_i} \overrightarrow{s} s_{t_i}$. We have, by combining these two statements, that: $\forall se' \in SE : \forall t_i \overset{se'}{\rightarrow} t_j : \forall o_i \in \Sigma_{t_i} : sf_{o_i} \overrightarrow{s} s_{t_i}$. Weakening this statement results in the following implication: $\forall se' \in SE : \forall t_i \overset{se'}{\rightarrow} t_j : \forall o_i \in \Sigma_{t_i} : W_i \neq \emptyset \Rightarrow sf_{o_i} \overrightarrow{s} s_{t_i}$. Putting it all together, $e$ satisfies PREREAD$_e(\mathcal{T}) \land \forall se' \in SE : \forall t_i \overset{se'}{\rightarrow} t_j : \forall o_i \in \Sigma_{t_i} : W_i \neq \emptyset \Rightarrow sf_{o_i} \overrightarrow{s} s_{t_i}$. We conclude that $\forall se \in SE : \exists e : \forall t \in \mathcal{T}_se : SESSION_{WFR}(se, t, e)$.

(\Leftarrow) We now prove that, given $G = \{RMW, MR, MW, WFR\}$, the following implication holds: $\forall se \in SE : \exists e : \forall t \in \mathcal{T}_se : SESSION_G(se, t, e) \Rightarrow \forall se \in SE : \exists e : \forall t \in \mathcal{T}_se : SESSION_{CC}(se, t, e)$.

To this end, we prove that for any session $se$, given the execution $e$ such that $\forall t \in \mathcal{T}_se : SESSION_G(se, t, e)$, we can construct an equivalent execution $e'$ that satisfies all four session guarantees, such that $\forall t \in \mathcal{T}_se : SESSION_{CC}(se, t, e')$.

The need for constructing an alternative but equivalent 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 equivalent 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. This will in turn imply that $e'$ satisfies 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 $W_i \neq \emptyset$, 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 $e'$: $\forall t \in \mathcal{T} \land W_i \neq \emptyset : (k, v) \in s_{e,t} \Leftrightarrow (k, v) \in s_{e',t}$. 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_{s \in \Sigma_t} \{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: \(\forall t \in T \land W_t \neq \emptyset : s_{e,t} \equiv s_{e',t}\).

We now proceed to prove that \(e'\) satisfies PREREAD\(_e\)(\(T\)) and the session test for all session guarantees.

**Preread** First, we show that PREREAD\(_e\)(\(T\)) 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 \(s_p(t_i) = \max\{\max_{s \in \Sigma_t} \{sf_o\}, s_{t_{i-1}}\}\). It follows that \(\forall o \in \Sigma_t, s_p(t) \geq sf_o \implies sf_o \rightarrow sf_{o(t)} \rightarrow s_t\). We have \(\forall o \in \Sigma_t : sf_o \rightarrow sf_{o(t)} \rightarrow s_t\) in \(e'\). Hence, \(sf_o = s_0\) and \(sf_o \rightarrow sf_{o(t)} \rightarrow s_t\) in \(e'\) trivially holds. The state corresponding to \(sf_o\) for read operations \(o = r(x)\) 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 PREREAD\(_e\)(\(T\)), hence by Lemma 2, we have \(s_{t_{i-1}} \rightarrow s_t \rightarrow s_t\) in \(e'\). By construction (update transactions are applied in \(e'\) in the same order as \(e\)), it follows that \(s_{t_{i-1}} \rightarrow s_{t_{i-1}} \rightarrow s_t\) in \(e'\).

By Lemma 2, we conclude that PREREAD\(_e\)(\(T\)) holds.

**MW** We next show that \(e'\) satisfies SESSION\(_{MW}\)(\(se, t, e'\)) for all sessions \(se \in SE\) and \(\forall t \in T_{se}\). Consider any session \(se'\) and two transactions \(t_i, t_j \in T_{se'}\) such that \(t_i \rightarrow e; t_j, W_{t_i} \neq \emptyset \land W_{t_j} \neq \emptyset\). As \(e\), by assumption, satisfies \(\forall t \in T_{se} : \text{SESSION}_{MW}(se, t, e)\), we have \(s_{se,t_i} \rightarrow s_{se,t_j}\). Since \(t_i\) and \(t_j\) are update transactions, they are applied in the same order in \(e'\) as in \(e\): hence, \(s_{se,t_i} \rightarrow s_{se,t_j}\). Further, recall that we previously showed that PREREAD\(_e\)(\(T\)) (and consequently PREREAD\(_e\)(\(T_{se}\)) by Lemma 3).

Putting it all together, we conclude that PREREAD\(_e\)(\(T_{se}\)) \land \forall se' \in SE : \forall t_i \rightarrow e; t_j : (W_{t_i} \neq \emptyset \land W_{t_j} \neq \emptyset) \implies s_{t_{i-1}} \rightarrow s_{t_{j-1}}\), i.e., \(\forall t \in T_{se} : \text{SESSION}_{MW}(se, t, e')\).

**WFR** Consider all update transactions such that \(\forall se' \in SE : \forall t_i \rightarrow e; t_j : \forall o \in \Sigma_t : W_{t_i} \neq \emptyset\). We prove that \(sf_{o(t)} \rightarrow sf_{o(t)}\). Consider the two types of operations that arise in an update transaction:

- **Reads.** The state corresponding to \(sf_o\) for read operations \(o = r(x, x_k)\) is the state created by the transaction \(t_k\) that wrote version \(x_k\) of object \(x\): \(sf_o = s_{t_k}\). By assumption, \(e\) satisfies \(\forall t \in T_{se} : \text{SESSION}_{WFR}(se, t, e)\), we have \(sf_{o(t)} \rightarrow sf_{o(t)}\) in \(e\), i.e., \(s_{t_k} \rightarrow s_{t_k}\) in \(e'\). Since we apply update transactions in \(e'\) the same order as in \(e\), it follows that \(s_{t_k} \rightarrow s_{t_k}\) in \(e'\), i.e., \(sf_{o(t)} \rightarrow sf_{o(t)}\) in \(e'\).

- ** Writes.** The candidate read states set for write operations \(o = w(x)\), is defined as \(RS_e(o) = \{s \in S_e \mid s \rightarrow s_p\}\). It trivially follows that \(sf_o = s_0 \rightarrow s_t\).

We conclude: \(\forall se \in SE : \forall t \in T_{se} : \text{SESSION}_{WFR}(se, t, e')\).

Before proving that the remaining session guarantees hold, we prove an intermediate result:

**Claim 7.** \(\forall se' \in SE : \forall t_i \rightarrow e; t_j : s_{t_{i-1}} \rightarrow s_{t_{j-1}}\). 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 \(T_u\) be the set of all update transactions in \(T_{se'}\), and let \(t_j\) be an arbitrary transaction in \(T_u\). It thus holds that \(t_j \in T_{se'} \land W_{t_j} \neq \emptyset\). We associate with each such \(t_j\) two further sets: \(T_{pre, u}\) and \(T_{pre, u}\) \(T_{pre, u}\) contains all update transactions \(t_i\) such that \(t_i \rightarrow e; t_j\). Similarly, \(T_{pre, u}\) contains all read-only transactions \(t_i\) such that \(t_i \rightarrow e; t_j\). \(T_{pre, u}\) is the union of those two sets. We prove that \(\forall t_i \in T_{pre, u} : s_{t_{i-1}} \rightarrow s_{t_{i-1}}\). If \(t_i \in T_{pre, u}\), hence \(W_{t_i} \neq \emptyset \land W_{t_j} \neq \emptyset\), the result trivially follows from monotonic writes. We previously proved that \(\forall t \in T_{se} : \text{SESSION}_{MW}(se', t, e')\). As such, the conjunction \(W_{t_i} \neq \emptyset \land W_{t_j} \neq \emptyset\) implies \(s_{t_{i-1}} \rightarrow s_{t_{j-1}}\) in \(e'\).

The proof is more complex if \(t_i \in T_{pre, u}\) (read-only transaction). We proceed by induction: **Base Case** Consider the first read-only transaction \(t_i \in T_{pre, u}\) 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 \(s_p(t_i) = \max\{\max_{o \in \Sigma_t} \{sf_o\}, s_{t_{i-1}}\}\), 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 \(s_p(t_i) = \max_{o \in \Sigma_t} \{sf_o\}, s_{t_{i-1}}\) or \(s_p(t_i) = s_{t_{i-1}}\).
1. If \( s_p(t_i) = \max_{s_{o, e} \in \Sigma_{t_i}} \{s_{f_o}\} \): We previously proved that \( \forall t \in T_{sc} : \text{SESSION}_{WFR}(se, t, e') \). It follows that \( \forall o_i \in \Sigma_{t_i} : s_{o_i} \searrow s_{t_j} \) in \( e' \) and consequently, \( \max_{o_i \in \Sigma_{t_i}} \{s_{o_i}\} \searrow s_{t_j} \) in \( e' \). Given that \( s_p(t_i) = \max_{o_i \in \Sigma_{t_i}} \{s_{f{o_i}}\} \), the following then holds \( s_p(t_i) \searrow s_{t_j} \) 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 \( s_p(t_i) \searrow s_{t_j} \) 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 \( t_{i-1} \searrow t_i \), \( t_{i-1} \) is necessarily an update transaction, where \( W_{t_{i-1}} \neq \emptyset \). Consider the pair of transactions \( (t_{i-1}, t_j) \). The session order is transitive, hence \( t_{i-1} \searrow t_j \) given that \( t_{i-1} \searrow t_i \) and \( t_i \searrow t_j \) both hold. By construction, we have \( W_{t_{i-1}} \neq \emptyset \wedge W_{t_j} \neq \emptyset \). We previously proved that \( \forall t \in T_{sc} : \text{SESSION}_{MW}(se, t, e') \). Hence, if \( W_{t_{i-1}} \neq \emptyset \wedge W_{t_j} \neq \emptyset \), it follows that \( s_{t_{i-1}} \searrow s_{t_j} \). As above, we conclude that: \( s_p(t_i) \searrow s_{t_j} \) in \( e' \), and finally \( s_p(t_i) \searrow s_{t_j} \).

To complete the base case, we note that \( t_i \neq t_j \) as \( t_i \searrow t_j \). We conclude: \( s_{t_i} \searrow s_{t_j} \).

**Induction Step** Consider the k-th read-only transaction \( t_i \) in \( se' \) such that \( t_i \searrow t_j \). We assume that it satisfies the induction hypothesis \( s_{t_i} \searrow s_{t_j} \). Now consider the (k+1)-th read-only transaction \( t_i \) in \( se' \), such that \( t_i \searrow t_j \). By construction, we once again distinguish two cases: \( t_i \)'s parent state is either \( s_p(t_i) = \max_{o_i \in \Sigma_{t_i}} \{s_{f{o_i}}\} \), or \( s_p(t_i) = s_{t_{i-1}} \), where \( t_{i-1} \) denotes the transaction directly preceding \( t_i \) in the session.

1. If \( s_p(t_i) = \max_{o_i \in \Sigma_{t_i}} \{s_{f{o_i}}\} \): We previously proved that \( \forall t \in T_{sc} : \text{SESSION}_{WFR}(se, t, e') \). It follows that \( \forall o_i \in \Sigma_{t_i} : s_{o_i} \searrow s_{t_j} \) in \( e' \) and consequently, \( \max_{o_i \in \Sigma_{t_i}} \{s_{o_i}\} \searrow s_{t_j} \) in \( e' \). Given that \( s_p(t_i) = \max_{o_i \in \Sigma_{t_i}} \{s_{f{o_i}}\} \), the following then holds \( s_p(t_i) \searrow s_{t_j} \) 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 \( s_p(t_i) \searrow s_{t_j} \) in \( e' \), concluding the proof for this subcase.

2. If \( s_p(t_i) = s_{t_{i-1}} \): First, we note that \( t_{i-1} \searrow t_j \) holds, as the session order is transitive and we have both \( t_{i-1} \searrow t_i \) and \( t_i \searrow t_j \). 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: \( W_{t_{i-1}} \neq \emptyset \wedge W_{t_j} \neq \emptyset \). Given that we previously proved \( \forall t \in T_{sc} : \text{SESSION}_{MW}(se, t, e') \), we can infer that \( s_{t_{i-1}} \searrow s_{t_j} \). 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 \( s_p(t_i) \searrow s_{t_j} \) 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 \( s_{t_i} \searrow s_{t_j} \). It thus follows that \( s_p(t_i) \searrow s_{t_j} \) in \( e' \). As previously, we conclude that: \( s_p(t_i) \searrow s_{t_i} \) in \( e' \).

To complete the induction step, we note that \( t_i \neq t_j \) as \( t_i \searrow t_j \). We conclude: \( s_{t_i} \searrow s_{t_j} \).

We proved the desired result for both the base case and induction step. By induction, we conclude that: given any \( t_j \) such that \( W_{t_j} \neq \emptyset \), and any read-only transactions \( t_i \) such that \( t_i \searrow t_j \), \( s_{t_j} \searrow s_{t_j} \) holds.

We conclude that given any \( t_j \) such that \( W_{t_j} \neq \emptyset \), and any transaction \( t_i \) such that \( t_i \searrow t_j \), \( s_{t_i} \searrow s_{t_j} \) holds.

**Read-Only Transactions** We now generalise the result to both update and read-only transactions. Specifically, we prove that in \( e' \), \( \forall t_i \searrow t_j : s_{t_i} \searrow s_{t_j} \). 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 two pair of transactions \( t_i, t_{i-1} \) in \( T_{se} \) such that \( t_{i-1} \) directly precede \( t_i \) in \( se' (t_{i-1} \searrow t_i) \). If \( W_{t_i} \neq \emptyset, s_{t_{i-1}} \searrow s_{t_i} \) as proven above. If \( t_i \) is a read-only transaction, its parent state, by construction, is equal to \( s_p(t_i) = \max_{o_i \in \Sigma_{t_i}} \{s_{f{o_i}}\}, s_{t_{i-1}} \). Since \( \max_{o_i \in \Sigma_{t_i}} \{s_{f{o_i}}\}, s_{t_{i-1}} \) \( \geq s_{t_{i-1}} \) by definition, it follows that \( s_{t_{i-1}} \searrow s_p(t_i) \). As \( s_p(t_i) \searrow s_{t_i} \), it follows that \( s_{t_{i-1}} \searrow s_{t_i} \). Together each such pair of consecutive transactions \( t_{i-1}, t_i \) defines a sequence: \( t_1 \searrow t_2 \searrow \ldots \searrow t_k \), where \( T_{se} = \{t_1, \ldots, t_k\} \). From the implication derived in the previous paragraph, it follows that \( s_{t_1} \searrow s_{t_2} \searrow \ldots \searrow s_{t_k} \). Noting that session order is transitive, we conclude: \( \forall t_i \searrow t_j : s_{t_i} \searrow s_{t_j} \).

This completes the proof of Claim 7.  

\[\square\]
RMW We now return to session guarantees and prove that \( \forall t \in T_{se} : \text{SESSION}_{RMW}(se, t, e') \). Specifically, we show that \( \text{PREREAD}_{e'}(T_{se}) \land \forall o \in \Sigma_t : \forall \ell' \xrightarrow{se} t : W_{\ell'} \neq \emptyset \Rightarrow s_{t \ell} \xrightarrow{s} s_{t \ell} \).

We proceed to prove that each of the two clauses holds true.

By assumption, \( e \) guarantees read-my-writes: \( \forall t \in T_{se} : \text{SESSION}_{RMW}(se, t, e) \). Consider an arbitrary transaction \( t \), and all update transactions \( t' \) that precede \( t \) in the session: \( \forall o \in \Sigma_t : \forall \ell' \xrightarrow{se} t : W_{t'} \neq \emptyset \). We distinguish between read operations and write operations:

- Let \( o \) be a read operation \( o = r(k, v) \). Its candidate read set is \( \mathcal{RS}_{e'}(o) = \{ s \in \mathcal{S}_e | s \xrightarrow{t} s_{p}(t) \land ((k, v) \in s \vee (\exists w(k, v) \in \Sigma_{t} : w(k, v) \xrightarrow{t} r(k, v))) \} \). \( s_{o} \), the last state in \( \mathcal{RS}_{e'}(o) \) can have one of two values: \( s_{o} = s_{p}(t) \), disallowing states created after \( t \)'s commit point, or \( s_{o} = s_{p}(\ell) \), where \( \ell \) is the update transaction that writes the next version of \( k \).
  - \( s_{o} = s_{p}(t) \). We previously proved that \( \forall se' \in SE : \forall t \xrightarrow{se'} t : s_{t \ell} \xrightarrow{s} s_{t \ell} \). By construction, \( t' \xrightarrow{se} t \). It follows that \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) and consequently that \( s_{t \ell} \xrightarrow{s} s_{t \ell} \). Given \( s_{p}(t) = s_{o} \), we conclude: \( s_{t \ell} \xrightarrow{s} s_{o} \).
  - \( s_{o} = s_{p}(\ell) \). Consider first the relationships between read states and commit states in \( e \). By assumption, \( e \) satisfies \( \forall t \in T_{se} : \text{SESSION}_{RMW}(se, t, e) \), i.e. \( s_{t \ell} \xrightarrow{s} s_{o} \) in \( e \). Since \( t \) wrote the next version of the object that \( t \) read, we have that \( s_{o} \xrightarrow{s} s_{p}(t) \xrightarrow{s} s_{t \ell} \). Combining the guarantee given by read-my-writes \( s_{t \ell} \xrightarrow{s} s_{o} \) and \( s_{o} \xrightarrow{s} s_{t \ell} \), we obtain \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) in \( e \). Returning to the execution \( e' \), since \( t' \) and \( \ell \) are both update transactions, \( W_{t'} \neq \emptyset \land W_{\ell} \neq \emptyset \), if \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) in \( e \), then \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) in \( e' \). Given \( e' \) is a total order and \( s_{p}(\ell) \rightarrow s_{t \ell} \rightarrow s_{o} \), we conclude: \( s_{t \ell} \xrightarrow{s} s_{o} \).

- Let \( o = w(k, v) \) be a write operation. By Claim 7, it holds that \( \forall se' \in SE : \forall t \xrightarrow{se'} t \rightarrow s_{t \ell} \xrightarrow{s} s_{t \ell} \). As \( t' \xrightarrow{se} t \), it follows that \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) and consequently that \( s_{t \ell} \xrightarrow{s} s_{t \ell} \). We conclude: \( s_{t \ell} \xrightarrow{s} s_{o} \).

Finally, as \( \text{PREREAD}_{e}(T) \) and \( T_{se} \subseteq T \), by Lemma 3, \( \text{PREREAD}_{e'}(T_{se}) \) holds.

We conclude that \( \text{PREREAD}_{e'}(T_{se}) \land \forall o \in \Sigma_t : \forall \ell' \xrightarrow{se} t : W_{\ell'} \neq \emptyset \Rightarrow s_{t \ell} \xrightarrow{s} s_{o} \), i.e. \( \forall t \in T_{se} : \text{SESSION}_{RMW}(se, t, e) \).

MR Finally, we prove that \( e' \) satisfies the final session guarantee: \( \text{SESSION}_{MR}(se, t, e') \). Specifically, we show that:
\( \text{PREREAD}_{e'}(T_{se}) \land \forall o \in \Sigma_t : \forall \ell' \xrightarrow{se} t : \forall \alpha' \in \Sigma_{t} : s_{\alpha'} \xrightarrow{s} s_{a} \). Intuitively, this states that the read state of \( o \) must include any write seen by \( \alpha' \).

We proceed to prove that each of the three clauses holds true.

The first clause follows directly from Lemma 3 and the fact that \( T_{se} \subseteq T \). We can then conclude that \( s_{o}, s_{o'}, s_{a}, s_{o'} \) must exist.

We can now proceed to prove that the third clause 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 \( \ell' \in T \) that writes version \( v' \) of object \( k' \), i.e \( s_{\alpha'} = s_{\ell'}, k \in W_{\ell'} \). 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 \( \mathcal{RS}_{e'}(o) = \{ s \in \mathcal{S}_e | s \xrightarrow{t} s_{p}(t) \land ((k, v) \in s \vee (\exists w(k, v) \in \Sigma_{t} : w(k, v) \xrightarrow{t} r(k, v))) \} \). \( s_{o} \), the last state in \( \mathcal{RS}_{e'}(o) \) can be one of two cases: \( s_{o} = s_{p}(t) \), disallowing states created after \( t \)'s commit point, or \( s_{o} = s_{p}(\ell) \), where \( \ell \) is the update transaction that writes the next version of \( k \).
  - \( s_{o} = s_{p}(t) \). We previously proved that \( \forall se' \in SE : \forall t \xrightarrow{se'} t : s_{t \ell} \xrightarrow{s} s_{t \ell} \). By construction, \( t' \xrightarrow{se} t \). It follows that \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) and consequently that \( s_{t \ell} \xrightarrow{s} s_{t \ell} \). Given \( s_{p}(t) = s_{o} \), we conclude: \( s_{t \ell} \xrightarrow{s} s_{o} \).
  - \( s_{o} = s_{p}(\ell) \). Consider first the relationships between read states and commit states in \( e \). By assumption, \( e \) satisfies \( \forall t \in T_{se} : \text{SESSION}_{MR}(se, t, e) \), i.e. \( s_{t \ell} \xrightarrow{s} s_{o} \) in \( e \). Since \( t \) wrote the next version of the object that \( t \) read, we have that \( s_{o} \xrightarrow{s} s_{p}(t) \xrightarrow{s} s_{t \ell} \). Combining the guarantee given by monotonic reads \( s_{t \ell} \xrightarrow{s} s_{o} \) and \( s_{o} \xrightarrow{s} s_{t \ell} \), we obtain \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) in \( e \). Returning to the execution \( e' \), since \( t' \) and \( \ell \) are both update transactions, \( W_{t'} \neq \emptyset \land W_{\ell} \neq \emptyset \), if \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) in \( e \), then \( s_{t \ell} \xrightarrow{s} s_{t \ell} \) in \( e' \). By definition \( s_{\alpha'} \xrightarrow{s} s_{\ell} \).
  - \( s_{o} = s_{p}(\ell) \) in \( e' \). Moreover, by assumption, \( s_{o} = s_{p}(\ell) \). Putting this together, we obtain the desired result \( s_{\alpha'} \xrightarrow{s} s_{o} \).
Let $o = w(k, v)$ be a write operation. The write set of a write operation is defined as $\mathcal{RS}_c(o) = \{s \in \mathcal{S}_c | s \xrightarrow{w} s_p\}$. It follows that: $sl_o = s_p(t)$. We previously proved that in $e'$, $\forall s \in SE: \forall t_i \xrightarrow{se} t_j : s_{t_i} \xrightarrow{s} s_{t_j}$; given $t \xrightarrow{se} t$, it thus follows that $s_{v'} \xrightarrow{s} s_p(t)$. Noting that $s_p(t) = sl_o$, we write $s_{v'} \xrightarrow{s} sl_o$.

Moreover, as PREREAD$_{c'}(T_{se})$ holds for $e'$, by Lemma 2, we have $s_{f{o'}} \xrightarrow{s} s_{v'}$ in $e'$. Combining the relationships, we conclude: $s_{f{o'}} \xrightarrow{s} sl_o$ 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 $s_{f{o'}} = s_0$. Thus $s_{f{o'}} \xrightarrow{s} sl_o$ trivially holds.

We can now prove that the second clause, IRC$_{c'}(t)$, holds—namely, that $\forall o, o' \in \Sigma_t : o' \xrightarrow{t} o \Rightarrow s_{f{o'}} \xrightarrow{s} sl_o$. Once again we consider two cases, depending on whether $o'$ is a read or a write operation.

**Read** The presumed read operation $o' = r(k', v')$ entails the existence of an update transaction $\hat{t} \in T$ that writes version $v'$ of object $k'$, i.e $s_{f{o'}} = s_{\hat{t}}$, $k \in W_{\hat{t}}$.

- Let us first assume that $o$ is a read operation $o = r(k, v)$, Its candidate read set $\mathcal{RS}_c(o) = \{s \in \mathcal{S}_c | s \xrightarrow{w} s_p(t) \land \{(k, v) \in s \land (\exists w(k, v) \in \mathcal{S}_c : (w(k, v) \xrightarrow{t} r(k, v)))\}$. $sl_o$, the last state in $\mathcal{RS}_c(o)$ 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 PREREAD$_{c'}(T)$. Given that $o'$, like $o$ is in $\Sigma_t$, it follows by Lemma 2 that $s_{f{o'}} \xrightarrow{s} s_{t_i} \in e'$, and consequently that $s_{f{o'}} \xrightarrow{s} s_p(t)$. Setting $s_p(t)$ to $sl_o$, we conclude $s_{f{o'}} \xrightarrow{s} sl_o$ in $e'$.

  - $sl_o = s_p(\hat{t})$: Consider first the relationships between read states and commit states in $e$. By assumption, $e$ satisfies IRC$_{c}(t)$, i.e. $s_{f{o'}} \xrightarrow{s} sl_o$ holds in $e$. Since $\hat{t}$ wrote the next version of the object that $o$ read, we have that $sl_o \xrightarrow{s} s_p(\hat{t}) \xrightarrow{s} s_{t_i}$ in $e$. Combining the guarantee given by monotonic reads $s_{f{o'}} \xrightarrow{s} sl_o$ and $sl_o \xrightarrow{s} s_{t_i}$, it follows that $s_{f{o'}} \xrightarrow{s} s_{t_i}$ in $e$ i.e. $s_{f{o'}} \xrightarrow{s} s_{t_i}$. Returning to the execution $e'$, since $\hat{t}$ and $\hat{t}$ are both update transactions, $W_{\hat{t}} \neq \emptyset \land W_{\hat{t}} \neq \emptyset$, if $s_{f{o'}} \xrightarrow{s} s_{t_i}$ in $e$, then $s_{f{o'}} \xrightarrow{s} s_{t_i}$ and consequently $s_{f{o'}} \xrightarrow{s} s_{t_i}$. Given $e'$ is a total order and $s_{p(\hat{t})} \xrightarrow{s} s_{t_i}$, we conclude $s_{f{o'}} \xrightarrow{s} s_p(\hat{t})$, and $s_{f{o'}} \xrightarrow{s} sl_o$ 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 $\mathcal{RS}_c(o) = \{s \in \mathcal{S}_c | s \xrightarrow{w} s_p\}$, where, consequently, $sl_o = s_p(t)$. We previously showed that PREREAD$_{c'}(T)$. Given that $o'$, like $o$ is in $\Sigma_t$, it follows by Lemma 2 that $s_{f{o'}} \xrightarrow{s} s_{t_i} \in e'$, and consequently that $s_{f{o'}} \xrightarrow{s} s_p(t)$. Setting $s_p(t)$ to $sl_o$, we conclude $s_{f{o'}} \xrightarrow{s} sl_o$ 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 $s_{f{o'}} = s_0$. Thus $s_{f{o'}} \xrightarrow{s} sl_o$ trivially holds.

We conclude that IRC$_{c'}(t)$ 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: $\forall t \in T_{se} : SESSION_c(se, t, e')$. More specifically, we must prove that: PREREAD$_{c'}(T) \land IRC$_{c'}(t)$ \land $(\forall o \in \Sigma_t : \forall t' \xrightarrow{se} t : s_{t'} \xrightarrow{s} sl_o) \land (\forall s \in SE : \forall t_i \xrightarrow{se} t_j : s_{t_i} \xrightarrow{s} s_{t_j})$. We proceed by proving that each of the clauses holds.

The first two clauses are easy to establish. We previously proved that PREREAD$_{c'}(T)$ holds. Likewise, IRC$_{c'}(t)$ holds as $\forall t \in T_{se} : SESSION_{MR}(se, t, e')$. To establish the fourth clause, we note that we previously proved that $\forall se' \in SE : \forall t_i \xrightarrow{se'} t_j : s_{t_i} \xrightarrow{s} s_{t_j}$. We proceed by proving that each of the clauses holds.

We are then left to prove only the third clause; namely we must prove that $\forall t_j \in T_{se} : (\forall o_j \in \Sigma_{t_j} : \forall t_i \xrightarrow{se} t_j : s_{t_i} \xrightarrow{s} sl_o)$. We distinguish between two cases: $t_j$ 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: $\forall t_j \in T_{se} : (\forall o_j \in \Sigma_{t_j} : \forall t_i \xrightarrow{se} t_j : W_{t_i} \neq \emptyset \Rightarrow s_{t_i} \xrightarrow{s} sl_o)$. 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 $sp(t_i) = \max \{ \max_{o_i \in \Sigma_i} \{ sf_{o_i} \}, s_{t_i-1} \}$, 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 $sp(t_i) = \max_{o_i \in \Sigma_i} \{ sf_{o_i} \}$, or $sp(t_i) = s_{t_i-1}$.

- If $sp(t_i) = \max_{o_i \in \Sigma_i} \{ sf_{o_i} \}$: We previously proved that $\forall t_i \in T_{se} : \text{SESSION}_{MR}(se, t, e')$, hence that $\forall t_i \xrightarrow{se'} t_j$ : $\forall o_i \in \Sigma_i : sf_{o_i} \xrightarrow{t_j} sl_{o_i}$, and consequently $\max_{o_i \in \Sigma_i} \{ sf_{o_i} \} \xrightarrow{t_j} sl_{o_i}$ in $e'$. Noting that $sp(t_i) = \max_{o_i \in \Sigma_i} \{ sf_{o_i} \}$, we obtain the desired result $sp(t_i) \xrightarrow{t_j} sl_{o_i}$ in $e'$.

- If $sp(t_i) = s_{t_i-1}$, 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 $W_{t_{i-1}} \neq \emptyset$. Given that, by transitivity $t_{i-1} \xrightarrow{se} t_j$, and that $e'$ guarantees read-my-writes $\forall t_i \in T_{se} : \text{SESSION}_{RW}(se, t, e')$, we have $s_{t_i-1} \xrightarrow{t_j} sl_{o_i}$ in $e'$. Noting that $sp(t_i) = s_{t_i-1}$, we conclude: $sp(t_i) \xrightarrow{t_j} sl_{o_i}$ in $e'$.

Finally, we argue that $sp(t_i) \neq sl_{o_i}$ (and therefore that $s_{t_i} \xrightarrow{t_j} sl_{o_i}$ as $sp(t_i) \xrightarrow{t_j} s_{t_i}$). Read-only transactions, like $t_i$ do not change the state on which they are applied, hence $\forall (k, v) \in sp(t_i) \Rightarrow (k, v) \in s_{t_i}$. Moreover, by Claim 7, transactions commit in session order: $\forall se' \in SE : \forall t_i \xrightarrow{se'} t_j : s_{t_i} \xrightarrow{t_j} s_{t_i}$. We thus have $s_{t_i} \xrightarrow{t_j} s_{t_i}$ and consequently $sp(t_i) \in RS_{e'}(o_j) \Rightarrow s_{t_i} \in RS_{e'}(o_j)$, i.e. $sp(t_i) \neq sl_{o_i}$: if $sp(t_i)$ is in $RS_{e'}(o_j)$, so is $s_{t_i}$. As $s_{t_i}$ follows $sp(t_i)$ in the execution, $sp(t_i)$ will never be $sf_{o_i}$.

The following thus holds in the base case: $sp(t_i) \xrightarrow{t_j} sl_{o_i}$, i.e. $sp(t_i) \rightarrow s_{t_i} \xrightarrow{t_j} sl_{o_i}$.

**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 $s_{t_i} \xrightarrow{t_j} sl_{o_j}$. 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 $sp(t_{i'}) = \max_{o_{i'} \in \Sigma_{i'}} \{ sf_{o_{i'}} \}$, or $sp(t_{i'}) = s_{t_{i'}-1}$, where $t_{i'-1}$ denotes the transaction directly preceding $t_{i'}$ in a session.

1. If $sp(t_{i'}) = \max_{o_{i'} \in \Sigma_{i'}} \{ sf_{o_{i'}} \}$: We previously proved that $\forall t_i \in T_{se} : \text{SESSION}_{MR}(se, t, e')$, hence that $\forall t_i \xrightarrow{se'} t_j : \forall o_{i'} \in \Sigma_{i'} : sf_{o_{i'}} \xrightarrow{t_j} sl_{o_{i'}}$, and consequently $\max_{o_{i'} \in \Sigma_{i'}} \{ sf_{o_{i'}} \} \xrightarrow{t_j} sl_{o_{i'}}$ in $e'$. Noting that $sp(t_{i'}) = \max_{o_{i'} \in \Sigma_{i'}} \{ sf_{o_{i'}} \}$, we obtain the desired result $sp(t_{i'}) \xrightarrow{t_j} sl_{o_{i'}}$ in $e'$.

2. If $sp(t_{i'}) = s_{t_{i'}-1}$: First, we note that $t_{i'-1} \xrightarrow{se'} t_{i'}$ holds, as the session order is transitive and we have both $t_{i-1} \xrightarrow{se} t_i$ and $t_i \xrightarrow{se'} t_{i'}$. 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 $s_{t_i} \xrightarrow{t_j} sl_{o_j}$, and consequently $s_{t_{i'-1}} \xrightarrow{t_j} sl_{o_j}$. Noting that $sp(t_{i'}) = s_{t_{i'}-1}$, we obtain $sp(t_{i'}) \xrightarrow{t_j} sl_{o_j}$.

If $t_{i'-1}$ is an update transaction, we note that $e'$ guarantees read-my-writes: $\forall t_i \in T_{se} : \text{SESSION}_{RW}(se, t, e')$. As $t_{i'-1} \xrightarrow{se'} t_{i'}$, we have $s_{t_{i'-1}} \xrightarrow{t_j} sl_{o_j}$ in $e'$. Noting that $sp(t_{i'}) = s_{t_{i'}-1}$, we conclude: $sp(t_{i'}) \xrightarrow{t_j} sl_{o_j}$ in $e'$.

Finally, we argue that $sp(t_{i'}) \neq sl_{o_j}$ (and therefore that $s_{t_{i'}} \xrightarrow{t_j} sl_{o_j}$ as $sp(t_{i'}) \rightarrow s_{t_{i'}}$). Read-only transactions, like $t_{i'}$ do not change the state on which they are applied, hence $\forall (k, v) \in sp(t_{i'}) \Rightarrow (k, v) \in s_{t_{i'}}$. Moreover, by Claim 7, transactions commit in session order: $\forall se' \in SE : \forall t_i \xrightarrow{se'} t_j : s_{t_i} \xrightarrow{t_j} s_{t_i}$. We thus have $s_{t_i} \xrightarrow{t_j} s_{t_i}$ and consequently $sp(t_{i'}) \in RS_{e'}(o_j) \Rightarrow s_{t_i} \in RS_{e'}(o_j)$, i.e. $sp(t_{i'}) \neq sl_{o_j}$: if $sp(t_{i'})$ is in $RS_{e'}(o_j)$, so is $s_{t_{i'}}$. As $s_{t_{i'}}$ succeeds $sp(t_{i'})$ in the execution, $sp(t_{i'})$ will never be $sf_{o_j}$. The following thus holds in the induction case: $sp(t_{i'}) \xrightarrow{t_j} sl_{o_j}$, i.e. $sp(t_{i'}) \rightarrow s_{t_{i'}} \xrightarrow{t_j} sl_{o_j}$. We proved the desired result for both the base case and induction step. By induction, we conclude that, for read-only transactions: $\forall t_{i'} \in T_{se}, \forall o_{i'} \in \Sigma_{i'} : \forall t_i \xrightarrow{se'} t_{i'} : W_{t_{i'}} = \emptyset \Rightarrow s_{t_{i'}} \xrightarrow{t_{i'}} sl_{o_j}$. Hence, the desired result holds for both read-only and update transactions $\forall t_{i'} \in T_{se} : (\forall o_{i'} \in \Sigma_{i'} : \forall t_i \xrightarrow{se'} t_{i'} : s_{t_{i'}} \xrightarrow{t_{i'}} sl_{o_j})$.

**Conclusion** Putting everything together, if $e'$ guarantees all four session guarantees, there exists an equivalent execution $e'$ such that $e'$ also satisfies the session guarantees and is causally consistent: $\text{PREREAD}_{c}(T) \land \text{IRC}_{c}(T) \land (\forall o \in \Sigma : \forall t_i \xrightarrow{t_j} s_{t_i} \xrightarrow{t_{j'}} sl_{o_j}) \land (\forall se' \in SE : \forall t_i \xrightarrow{se'} t_j : s_{t_i} \xrightarrow{t_{j'}} s_{t_j})$, i.e. $\forall t \in T_{se} : \text{SESSION}_{CC}(se, t, e')$. This completes the second part of the proof.

Consequently, $\forall se \in SE : \exists e : \forall t \in T_{se} : \text{SESSION}_{P}(se, t, e) \equiv \forall se \in SE : \exists e : \forall t \in T_{se} : \text{SESSION}_{CC}(se, t, e)$ holds.
Appendix D  Equivalence of PL-2+ and PSI

In this section, we prove that our state-based definition of PSI is equivalent to both the axiomatic formulation of PSI (PSIA) by Cerone et al. and to the cycle-based specification of PL-2+:

**Theorem 5** Let \( \mathcal{T} \) be PSI. Then \( \exists e : \forall t \in \mathcal{T} : CT_{PSI}(t,e) \equiv \neg G1 \land \neg G\)-single

**Theorem 6** Let \( \mathcal{T} \) be PSI. Then \( \exists e : \forall t \in \mathcal{T} : CT_{PSI}(t,e) \equiv PSIA \)

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 \( PREREAD_e(\mathcal{T}) \), then any transaction \( t \) that depends on a transaction \( t' \) 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 4), and then extend that result to all the transaction’s transitive dependencies (Lemma 5). Formally

**Lemma 4.** \( PREREAD_e(\mathcal{T}) \Rightarrow \forall t' \in \mathcal{T} : s_t \not\rightarrow s_{t'} \)

**Proof.** Consider any \( t \in \mathcal{T} \) and any \( e \in D-PREC_e(\hat{t}) \). \( t \) is included in \( D-PREC_e(\hat{t}) \) if one of two cases hold: if \( \exists o \in \Sigma_t : t = t_{o} \rightarrow s_{t} \land W_t \cap W_i \neq \emptyset \) (t and \( \hat{t} \) write the same objects and \( t \) commits before \( \hat{t} \).

1. \( t \in \{ t \in \Sigma_t : t = t_{s} \rightarrow s_{t} \land W_t \cap W_i \neq \emptyset \} \) Let \( o \) be the operation such that \( t = t_{s} \rightarrow s_t \). By assumption, we have \( PREREAD_e(\mathcal{T}) \) It follows that \( \forall o, s_{t} \rightarrow s_t \) and consequently \( s_{t} \rightarrow s_t \).

2. \( t \in \{ t \in \Sigma_t : t = t_{s} \rightarrow s_t \land W_t \cap W_i \neq \emptyset \} \), trivially we have \( s_t \rightarrow s_t \).

We now generalise the result to hold transitivevly.

**Lemma 5.** \( PREREAD_e(\mathcal{T}) \Rightarrow \forall t' \in \mathcal{T} : s_t \not\rightarrow s_{t'} \)

**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 \( s_t \rightarrow s_{t_1} \) \( \forall t' \in \mathcal{T} : s_{t'} \rightarrow s_t \). As \( t_1 \) is the first transaction in the execution, \( D-PREC_e(t_1) = \emptyset \) and consequently \( PREC_e(t) = \emptyset \). We see this by contradiction: assume there exists a transaction \( t \in D-PREC_e(t_1) \), by implication \( s_t \rightarrow s_{t_1} \) (Lemma 4), violating 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. \( s_t \rightarrow s_{t_1} \) the property \( \forall t' \in \mathcal{T} : s_{t'} \rightarrow s_t \) holds. In otherwords, 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 \( \forall t' \in \mathcal{T} : s_{t'} \rightarrow s_{t_i} \). A transaction \( t' \) belongs to \( D-PREC_e(t_{i+1}) \) if one of two conditions holds: either \( t' \in D-PREC_e(t_{i+1}) \), or \( \exists t_k \in \mathcal{T} : t' \in \mathcal{T} \land t_k \in D-PREC_e(t_{i+1}) \). We consider each in turn:

- If \( t' \in D-PREC_e(t_{i+1}) \): by Lemma 4., we have \( s_{t'} \rightarrow s_{t_{i+1}} \).

- If \( \exists t_k \in D-PREC_e(t_{i+1}) : t' \in D-PREC_e(t_k) \): As \( t_k \in D-PREC_e(t_{i+1}) \), by Lemma 4., we have \( s_{t_k} \rightarrow s_{t_{i+1}} \), i.e. \( s_{t_k} \rightarrow s_{t_i} \) (\( s_{t_k} \) 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 \( \forall t \in \mathcal{T} : s_{t_k} \rightarrow s_{t_i} \). As \( t' \in D-PREC_e(t_k) \) by construction, it follows that \( s_{t_k} \rightarrow s_{t_i} \). Putting everything together, we have \( s_{t'} \rightarrow s_{t_k} \rightarrow s_{t_{i+1}} \), and consequently \( s_{t'} \rightarrow s_{t_{i+1}} \). This completes the induction step of the proof.

Combining the base case, and induction step, we conclude: \( PREREAD_e(\mathcal{T}) \Rightarrow \forall t' \in \mathcal{T} : s_{t'} \rightarrow s_t \). □

Appendix D.1  PL-2+

**Theorem 5** Let \( \mathcal{T} \) be PSI. Then \( \exists e : \forall t \in \mathcal{T} : CT_{PSI}(t,e) \equiv \neg G1 \land \neg G\)-Single

Let us consider a history \( H \) that contains the same set of transactions \( \mathcal{T} \) as \( e \). 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 \( i < j \) and \( e \) commits after \( x_i \). We show that, if a transaction \( t \) is in the depend set of a transaction \( t' \), then there exists a path of write-read/write-write dependencies from \( t \) to \( t' \) in the \( DSG(H) \). Formally:

28
We consider each possible edge in the cycle in turn: action are applied. Hence, there exists a transaction \( t \) the execution, \( \text{D-PREC}_e(t_1) = \emptyset \) and consequently \( \text{PRE}_c(t) = \emptyset \). We see this by contradiction: assume there exists a transaction \( t \in \text{D-PREC}_e(t_1) \), by implication \( s_t \xrightarrow{\text{w}} s_{t_i} \) (Lemma 4), violating our assumption that \( t_1 \) is the first transaction in the execution. The implication trivially holds.

**Induction Step** Consider the \( i \)-th transaction in the execution. We assume that \( \forall t \), s.t. \( s_t \xrightarrow{\text{w}} s_{t_i} \), \( \forall t' \in \text{PRE}_c(t) \): \( t' \xrightarrow{\text{w}} t \). 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 \( \forall t' \in \text{PRE}_c(t_{i+1}) \): \( t' \xrightarrow{\text{w}} t_{i+1} \). A transaction \( t' \) belongs to \( \text{PRE}_c(t_{i+1}) \) if one of two conditions holds: either \( t' \in \text{D-PREC}_c(t_{i+1}) \), or \( \exists t_k \in T : t' \in \text{PRE}_c(t_k) \wedge t_k \in \text{D-PREC}_c(t_{i+1}) \). We consider each in turn:

- If \( t' \in \text{D-PREC}_c(t_{i+1}) \): There are two cases: \( t' \in \{ t | \exists o \in \Sigma_{t_{i+1}} : t = t_{\text{to}} \} \) or \( t' \in \{ t | s_t = s_{t_{i+1}} \wedge W_{t_{i+1}} \cap W_t \neq \emptyset \} \). If \( t' \in \{ t | \exists o \in \Sigma_{t_{i+1}} : t = t_{\text{to}} \} \), \( t_{i+1} \) reads the version of an object that \( t' \) wrote, hence \( t_{i+1} \) read-depends on \( t' \), i.e. \( t' \xrightarrow{\text{w}} t \).

  If \( t' \in \{ t | s_t = s_{t_{i+1}} \wedge W_{t_{i+1}} \cap W_t \neq \emptyset \} \): trivially, \( s_{t'} \xrightarrow{\text{w}} s_{t_{i+1}} \). Let \( x \) be the key that is written by \( t \) and \( t_{i+1} \): \( x \in W_{t_{i+1}} \cap W_t \). By construction, the history \( H \)'s version order for \( x \) is \( x_{t'} \prec x_{t_{i+1}} \). By definition of version order, there must therefore a chain of \( w \) 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: \( t' \xrightarrow{\text{w}} t_{i+1} \).

- If \( \exists t_k \in \text{PRE}_c(t_k) \wedge t_k \in \text{D-PREC}_c(t_{i+1}) \). As \( t_k \in \text{D-PREC}_c(t_{i+1}) \), we conclude, as above that \( t_k \xrightarrow{\text{w}} t_{i+1} \). Moreover, by Lemma 4, we have \( s_{t_k} \xrightarrow{\text{w}} s_{t_{i+1}} \), i.e. \( s_{t_k} \xrightarrow{\text{w}} s_{t_i} \) (\( s_{t_i} \) 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 \( \forall t_{k'} \in \text{PRE}_c(t_k) \): \( t_{k'} \xrightarrow{\text{w}} t_{i+1} \). Noting \( t' \in \text{PRE}_c(t_k) \), we see that \( t' \xrightarrow{\text{w}} t_{i+1} \). Putting everything together, we obtain \( t' \xrightarrow{\text{w}} t_{i+1} \).

Combining the base case, and induction step, we conclude: \( \forall t \): \( \forall t' \in \text{PRE}_c(t) \): \( t' \xrightarrow{\text{w}} t \).

**Equivalence** We now prove Theorem 5.

**Theorem 5** Let \( \mathcal{I} \) be PSI. Then \( \exists e \in \mathcal{T} : \text{CT}_{PSI}(t,e) \equiv \neg \text{G1} \land \neg \text{G-Single} \)

**Proof** Let us recall the definition of PSI’s commit test:

\[
\text{PREREAD}_c(T) \land \forall o \in \Sigma : \forall t' \in \text{PRE}_c(t) : o,k \in W_{t'} \Rightarrow s_t \xrightarrow{\text{w}} s_{t_o}
\]

(\( \Rightarrow \)) **First we prove** \( \exists e \in \mathcal{T} : \text{CT}_{PSI}(t,e) \Rightarrow \neg \text{G1} \land \neg \text{G-Single} \).

Let \( e \) be an execution that \( \forall t \in \mathcal{T} : \text{CT}_{PSI}(t,e) \), and \( H \) be a history for committed transactions \( T \).

We first instantiate the version order for \( H \), denoted as \( \prec \), as follows: given an execution \( e \) and an object \( x, x_i \prec x_j \) if and only if \( x \in W_{t_{i+1}} \cap W_{t_j} \) and \( s_{t_i} \xrightarrow{\text{w}} s_{t_j} \). It follows that, for any two states such that \( (x, x_i) \in T_m \land (x, x_j) \in T_n \Rightarrow s_{T_m} \xrightarrow{\text{w}} s_{T_n} \).

**G1** We next prove that \( \forall t \in \mathcal{T} : \text{CT}_{PSI}(t,e) \Rightarrow \neg \text{G1} \):

**G1-a** 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. \( T \) and any corresponding execution \( e \), however, consists only of committed transactions. Hence \( \forall e : \exists s \in \mathcal{S}, s.t. s \in R\mathcal{S}_c(r_j(x_i)) \): i.e. \( \neg \text{PREREAD}_c(t_j) \), therefore \( \neg \text{PREREAD}_c(T) \). There thus exists a transaction for which the commit test cannot be satisfied, for any \( e \). We have a contradiction.

**G1-b** Let us assume that \( H \) exhibits phenomenon G1b(intermediate reads). In an execution \( e \), only the final writes of a transaction are applied. Hence, \( \forall e : \exists s \in \mathcal{S}, s.t. s \in R\mathcal{S}_c(r_j(x_{\text{intermediate}})) \): i.e. \( \neg \text{PREREAD}_c(t) \), therefore \( \neg \text{PREREAD}_c(T) \). There thus exists a transaction \( t \), which for all \( e \), will not satisfy the commit test. We once again have a contradiction.

**G1-c** 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:

29
We distinguish between two cases: D-PREC $H$ 

Hence, there exist a permutation of $\pi$, we have $s_{t_i} \rightarrow s_{t_{i+1}} \rightarrow \ldots \rightarrow s_{t_j}$. By definition however, a valid execution must be totally ordered. We once again have a contradiction.

**G-Single** We now prove that $\forall t \in T : CT_{PSI}(t, e) \Rightarrow \neg G$-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 $t_1 \xrightarrow{wr} t_2 \xrightarrow{wr} \ldots \xrightarrow{wr} t_k \xrightarrow{wr} t_1$ 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 succeeds $t_1$. We then show that there exist a $t' \in PREC_e(t_k)$ such that $o.k \in W_{t'} \Rightarrow s_{t'} \rightarrow s_{t_1}$ does not hold.

**Base case** We prove that $t_1 \in PREC_e(t_2)$. We distinguish between two cases $t_1 \xrightarrow{wr} t_2$, and $t_1 \xrightarrow{wr} t_2$.

- If $t_1 \xrightarrow{wr} t_2$, there must exist an object $k$ that $t_1$ and $t_2$ both write: $k \in W_{t_1}$ and $k \in W_{t_2}$, therefore $W_{t_1} \cap W_{t_2} \neq \emptyset$. By construction, $t_1 \xrightarrow{wr} t_2$ if $s_{t_1} \rightarrow s_{t_2}$. Hence we have $s_{t_1} \rightarrow s_{t_2}$. 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 $t_1 \xrightarrow{wr} t_2$ if $s_{t_1} \rightarrow s_{t_2}$ and $s_f = s_{t_1}$, i.e. $t_1 = t_{s_{t_1}}$. 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 $t_i \in PREC_e(t_{i+1})$. To do so, we first prove that $t_i \in D$-PREC$_e(t_{i+1})$. We distinguish between two cases: $t_i \xrightarrow{wr} t_{i+1}$, and $t_i \xrightarrow{wr} t_{i+1}$.

- If $t_i \xrightarrow{wr} t_{i+1}$, there must exist an object $k$ that $t_i$ and $t_{i+1}$ both write: $k \in W_{t_i}$ and $k \in W_{t_{i+1}}$, therefore $W_{t_i} \cap W_{t_{i+1}} \neq \emptyset$. By construction, $t_i \xrightarrow{wr} t_{i+1}$ if $s_{t_i} \rightarrow s_{t_{i+1}}$. Hence we have $s_{t_i} \rightarrow s_{t_{i+1}}$. 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 $t_i$: $o = r(k_1)$. We previously proved that $t_i \xrightarrow{wr} t_{i+1}$ if $s_{t_i} \rightarrow s_{t_{i+1}}$. It follows that $s_{t_i} \rightarrow s_{t_{i+1}}$ and $s_f = s_{t_i}$, i.e. $t_i = t_{s_{t_i}}$. 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: $t_i \in PREC_e(t_{i+1})$.

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 $t_k \xrightarrow{wr} t_1$ 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 $r_k(x_m) \in \Sigma_0$, $w_1(x_1) \in \Sigma_1$, 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 $s_{t_1} \xrightarrow{wr} s_{l_{r}(x_m)}$.

However, from $x_m << x_1$, we have $\forall s, s', s.t.(x, x_m) \in s \land (x, x_1) \in s' \Rightarrow s \xrightarrow{wr} s'$. Since $(x, x_m) \in s_{l_{r}(x_m)} \land (x, x_1) \in s_{t_1}$, we have $s_{l_{r}(x_m)} \xrightarrow{wr} s_{t_1}$. We previously proved that $T s_{t_1} \xrightarrow{wr} s_{l_{r}(x_m)}$. We have a contradiction: $H$ does not exhibit phenomenon G-Single, i.e. $\exists e : \forall t \in T : CT_{PSI}(t, e) \Rightarrow \neg G$-Single.

$(\Leftarrow)$ We now prove the other direction $\neg G$-Single $\Rightarrow \exists e : \forall t \in T : CT_{PSI}(t, e)$.

We construct $e$ as follows: Consider only dependency edges in the DSG($H$), by $\neg G$, there exist no cycle consisting of only dependency edges, therefore the transactions can be topologically sorted respecting only dependency edges. Let $i_1, \ldots, i_n$ be a permutation of $1, 2, \ldots, n$ such that $i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_n$ is a topological sort of DSG($H$) with only dependency edges. We construct an execution $e$ according to the topological order defined above: $e : s_0 \rightarrow s_{i_1} \rightarrow s_{i_2} \rightarrow \ldots \rightarrow s_{i_n}$.

First we show that PREREAD$_e(T)$ 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 $s_0 \in RS_e(o)$ hence $RS_e(o) \neq \emptyset$ follows trivially. Consider the case now
We now prove the following theorem: We show that \( o \) where \( \circ \)

The authors state that this is for syntactic elegance only, and does not change the essence of the proof.

Next, we prove that \( \forall o \in \Sigma_t : \forall t' \in \text{PRECE}(t) : o.k \in \mathcal{W}_t \Rightarrow s_{t'} \xrightarrow{\circ} s_{o} \) holds. For any \( t' \in \text{PRECE}(t) \), by Lemma 5, \( s_{t'} \xrightarrow{\circ} s_t \). Consider any \( o \in \Sigma_t \), let \( t' \) be a transaction such that \( t' \in \text{PRECE}(t) \land o.k \in \mathcal{W}_t \), we now prove that \( s_{t'} \xrightarrow{\circ} s_{o} \).

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 \( s_{t'} \xrightarrow{\circ} s_{o} \) for each of those operation types:

1. **External Reads**. Let \( o = r(x, t) \in \Sigma_t \) read the version for \( x \) created by \( t \), where \( t \neq t \). Since \( \text{PRECE}(t) \) is true, we have \( RS_{e}(o) \neq \emptyset \), therefore \( s_{t} \xrightarrow{\circ} s_{t} \) and \( t = t_{s} \). From \( t = t_{s} \), we have \( t \in D-\text{PRECE}(t) \). Now consider \( t \) and \( t \), we have currently proved that \( s_{t} \xrightarrow{\circ} s_{t} \) and \( s_{t} \xrightarrow{\circ} s_{t} \). There are two cases:
   - \( s_{t} \xrightarrow{\circ} s_{t} \): Consequently \( s_{t} \xrightarrow{\circ} s_{t} \) and \( s_{t} \xrightarrow{\circ} s_{t} \). It follows that \( s_{t} \xrightarrow{\circ} s_{t} \).
   - \( s_{t} \xrightarrow{\circ} s_{t} \): We prove that this cannot happen by contradiction. Since \( o.k \in \mathcal{W}_t \), \( t \) also writes key \( x \). By construction, \( s_{t} \xrightarrow{\circ} s_{t} \) in \( e \) implies \( x << x \). There must consequently exist a chain of \( w \) edges between \( t \) 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 \( t \), denoted as \( t_{s} \), where \( t \xrightarrow{w} t_{s} \) \( t_{s} \) overwrites the version of \( x \). Consequently, \( t_{s} \) directly anti-depends on \( t_{s} \), i.e., \( t_{s} \xrightarrow{w} t_{s} \). Moreover \( t' \in \text{PRECE}(t) \), by Lemma 6, we have \( t' \xrightarrow{w} t \). There thus exists a cycle consists of only one anti dependency edges as \( t \xrightarrow{w} t_{s} \) \( t_{s} \xrightarrow{w} t \), in contradiction with G-Single. \( s_{t} \xrightarrow{\circ} s_{t} \) holds.

2. **Internal Reads**. Let \( o = r(x, t) \) read \( x \) such that \( w(x, t) \xrightarrow{t} r(x, t) \). By definition of \( RS_{e}(o) \), we have \( s_{o} = s_{p} \).

Since we have proved that \( s_{t} \xrightarrow{\circ} s_{t} \), therefore we have \( s_{t} \xrightarrow{\circ} s_{p} \) (as \( s_{p} \) is the value space to be \( Z \)).

3. **Writes**. Let \( o = w(x, t) \) be a write operation. By definition of \( RS_{e}(o) \), we have \( s_{o} = s_{p} \).

We previously proved that \( s_{t} \xrightarrow{\circ} s_{t} \). Consequently we have \( s_{t} \xrightarrow{\circ} s_{p} \) (as \( s_{p} \) is the value space to be \( Z \)).

We conclude, in all cases, \( \text{CT}_{PSI}(t, e) \equiv \text{PRECE}(t) \land \forall o \in \Sigma_t : \forall t' \in \text{PRECE}(t) : o.k \in \mathcal{W}_t \Rightarrow s_{t'} \xrightarrow{\circ} s_{o} \). \( \square \)

### Appendix D.2 PSI \(_{IA} \)

We now prove the following theorem:

**Theorem 6** Let \( \mathcal{I} \) be PSI. Then \( \exists e : \forall t \in \mathcal{T} : \text{CT}_{PSI}(t, e) \equiv PSI_{A} \)

We note that this axiomatic specification, defined by Cerone et al. [20, 21] is proven to be equivalent to the operational specification of Sovran et al. [52], 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.

### Appendix D.2.1 Model Summary

We provide a brief summary and explanation of the main terminology introduced in Cerone et al.’s framework for reasoning about concurrency. We refer the reader to [20] for the full set of definitions.

The authors consider a database storing a set of objects \( \text{Obj} = \{x, y, \ldots\} \), with operations \( \text{Op} = \{\text{read}(x, n), \text{write}(x, n) | x \in \text{Obj}, n \in \mathbb{Z}\} \). For simplicity, the authors assume the value space to be \( \mathbb{Z} \).

**Definition 28.** History events are tuples of the form \( (i, op) \), where \( i \) is an identifier from a countably infinite set \( \text{EventId} \) and \( op \in \text{Op} \). Let \( \text{LEvent}_{x} = \{(i, \text{write}(x, n)) | i \in \text{EventId}, n \in Z\} \), \( \text{REvent}_{x} = \{(i, \text{write}(x, n)) | i \in \text{EventId}, n \in Z\} \), and \( \text{HEvent}_{x} = \text{REvent}_{x} \cap \text{LEvent}_{x} \).

**Definition 29.** A transaction \( T \) is a pair \( (E, po) \), where \( E \subseteq \text{HEvent} \) 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.
We first relate Cerone et al.'s notion of transactions to transactions in our model: Cerone defines transactions as a tuple \( (E, \max_R(A)) \) where \( E \) is the set of events and \( \max_R(A) \) is the element \( u \in A \) such that \( \forall v \in A . v = u \lor (v, u) \in R \). \( R_{-1}(u) \) For a relation \( R \subseteq A \times A \) and an element \( u \in A \), we let \( R_{-1}(u) = \{ v | (v, u) \in R \} \).

\( T \vdash Write \_ x : n \) T writes to \( x \) and the last value written is \( n: \max_{po}(E \cap WEvent_{x}) = (\_ , write(x, n)) \).

\( T \vdash Read \_ x : n \) 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: \( \min_{po}(E \cap Hevent_{x}) = (\_ , read(x, n)) \).

The authors introduce a number of consistency axioms. A consistency model specification is a set of consistency axioms \( \Phi \) constraining executions. The model allows a choice that satisfies the axioms:

**Definition 31.** \( Hist_{\Phi} = \{ \| \forall \text{vis}, \text{AR}, (\mathcal{H}, V I S, AR) \models \Phi \} \)

The authors define several consistency axioms:

**Definition 32.** \( INT \) \( \forall (E, po) \in \mathcal{H}, \forall \text{event} \in E. \forall x, n. (\text{event} = (\_ , \text{read}(x, n)) \land (po^{-1}(\text{event}) \cap HEvent_{x}) \neq \emptyset) \)

\( \Rightarrow \) \( \max_{po}(po^{-1}(\text{event}) \cap HEvent_{x}) = (\_ , (x, n)) \)

\( EXT \) \( \forall T \in \mathcal{H}, \forall x, n. T \vdash \text{Read} x : n \Rightarrow (\text{V I S}^{-1}(T) \cap \{ S | S \vdash \text{Write} x : \_ \} = \emptyset \land n = 0) \lor \max_{AR}(\text{V I S}^{-1}(T) \cap \{ S | S \vdash \text{Write} x : \_ \}) \vdash \text{Write} x : n \)

\( \text{TRANSVIS} \) \( V I S \) is transitive

\( \text{NOCONFLICT} \) \( \forall T, S \in \mathcal{H}. (T \neq S \land T \vdash \text{Write} x : \_ \land S \vdash \text{Write} x : \_ \Rightarrow (T \xrightarrow{V I S} S \lor S \xrightarrow{V I S} T) \)

\( PSI_{A} \) is then defined with the following set of consistency axioms.

**Definition 33.** \( PSI \) allows histories for which there exists an execution that satisfies \( INT \), \( EXT \), \( TRANSVIS \) and \( NOCONFLICT \):

\( Hist_{PSI} = \{ H | \exists \text{vis}, \text{AR}, (\mathcal{H}, V I S, AR) \models INT, EXT, TRANSVIS, NOCONFLICT \} \)

**Appendix D.2.2 Equivalence**

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 similarly defines transactions as a tuple \( (S_{\Sigma}, \Sigma_{\Sigma}) \), where \( \Sigma_{\Sigma} \) is a set of operations, and \( \Sigma_{\Sigma} \) is the total order on \( \Sigma_{\Sigma} \). 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. For clarity, we denote transactions in Cerone’s model as \( T \) and transactions in our model as \( t \). Finally, we relate our notion of versions to Cerone’s values.

\( \Rightarrow \) **We first prove** \( \exists e : \forall t \in T : CT_{PSI}(t, e) \Rightarrow PSI_{A} \).

**Construction** Let \( e \) be an execution such that \( \forall t \in T : CT_{PSI}(t, e) \). We construct \( AR \) and \( V I S \) as follows: \( AR \) is defined as \( T_{i} \xrightarrow{AR} T_{j} \iff s_{t_{i}} \to s_{t_{j}} \) while \( V I S \) order is defined as \( T_{i} \xrightarrow{V I S} T_{j} \iff t_{i} \in \text{PRECE}(t_{j}) \). By definition, our execution is a total order, hence our constructed \( AR \) is also a total order. \( V I S \) defines an acyclic partial order that is a subset of \( AR \) (by \( \text{PREREAD}(T) \) and Lemma 5).

We now prove that each consistency axiom holds:

\( INT \) \( \forall (E, po) \in \mathcal{H}. \forall \text{event} \in E. \forall x, n. (\text{event} = (\_ , \text{read}(x, n)) \land (po^{-1}(\text{event}) \cap HEvent_{x}) \neq \emptyset) \)

\( \Rightarrow \) \( \max_{po}(po^{-1}(\text{event}) \cap HEvent_{x}) = (\_ , (x, n)) \)

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 \( \text{event} \) and \( x \) such that \( (\text{event} = (\_ , \text{read}(x, n)) \land (po^{-1}(\text{event}) \cap HEvent_{x}) \neq \emptyset) \). We prove that \( \max_{po}(po^{-1}(\text{event}) \cap HEvent_{x}) = (\_ , (x, n)) \). By assumption, \( (po^{-1}(\text{event}) \cap HEvent_{x}) \neq \emptyset \) holds, there must exist an event such that \( \max_{po}(po^{-1}(\text{event}) \cap HEvent_{x}) = (\_ , (x, n)) \). This event is either a read operation, or a write operation:

1. If \( op = \max_{po}(po^{-1}(\text{event}) \cap HEvent_{x}) \) is a write operation: given \( \text{event} = (\_ , \text{read}(x, n)) \) and \( \text{op} \xrightarrow{po} \text{event} \), the equivalent statement in our model is \( w(x, \text{op}) \xrightarrow{\Sigma_{\Sigma}} r(x, n) \). By definition, our model enforces that \( w(k, v) \xrightarrow{\Sigma_{\Sigma}} r(k, v) \Rightarrow v = v' \). Hence \( \text{v}_{\text{op}} = n \), i.e., \( \text{op} = (\_ , \text{write}(x, n)) \), therefore \( \text{op} = (\_ , (x, n)) \). Hence \( INT \) holds.
2. If \( op = \max_{po}(po^{-1}(\text{event}) \cap HEvent_x) \) is a read operation, We write \( op = (\_read(x, v_{op})) \). The equivalent formulation in our model is as follows. For \( \text{event} = (\_read(x, n)) \), we write \( o_1 = r(x, n) \), and for \( op \), we write \( o_2 = r(x, v_{op}) \) with \( o_2 \rightarrow o_1 \) 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 \( w(k, v) \rightarrow o_2 \rightarrow o_1 \) (all three operations belong to the same transaction). Given that \( o_0 \) is a total order, we have \( w(k, v) \rightarrow o_1 \) and \( w(k, v) \rightarrow o_2 \). It follows by definition of candidate read state that \( w(k, v') \rightarrow o_0 \rightarrow r(k, v) \Rightarrow v = v' \), where \( v = n \land v = v_{op}, i.e. v_{op} = n \). Hence \( op = (\_\_read(x, n)) \) and \( INT \) holds. Second, let us next assume that there does not exist an operation \( w(k, v) \rightarrow o_2 \rightarrow o_1 \). We prove by contradiction that \( v_{op} = n \) nonetheless. Assume that \( v_{op} \neq n \), and consider transactions \( t_i \) that writes \( v_{op} \), and \( t_i \) that writes \( v_{op} \), by \( \text{PREREAD}_x(T) \), we know that \( \forall \text{so}_1, \text{so}_2 \) exist. We have \( t_1 = t_{\text{so}_1} \) and \( t_2 = t_{\text{so}_2} \). By definition of \( \text{PRECE}_x(t) \), we have \( t_1, t_2 \in D\text{-PRECE}_x(t) \subseteq \text{PRECE}_x(t) \), i.e. \( t_1, t_2 \in \text{PRECE}_x(t) \).

We note that the sequence of states containing \((x, n)\) is disjoint from states containing \((x, v_{op})\). In other words, the sequence of states bounded by \( \text{so}_1 \) and \( \text{so}_1 \) and \( \text{so}_2 \) and \( \text{so}_2 \) are disjoint. Hence, we have either \( s_{t_1} \rightarrow \text{so}_1 \rightarrow s_{t_2} \rightarrow \text{so}_2 \) or \( s_{t_2} \rightarrow \text{so}_2 \rightarrow s_{t_1} \rightarrow \text{so}_1 \). Equivalently, either \( t_2 \in \text{PRECE}_x(t) \land o_1 k \in \text{Wt}_2 \land \text{so}_1 \rightarrow s_{t_2} \), or \( t_1 \in \text{PRECE}_x(t) \land o_2 k \in \text{Wt}_1 \land \text{so}_2 \rightarrow s_{t_1} \). In both cases, this violates \( \text{CT}_{\text{PSI}}(t, e) \), a contradiction. We conclude \( \forall \text{so}_1, \text{so}_1 \) and use this result to prove that \( \text{VIS} \) is transitive.

We proceed by induction, let \( e = s_0 \rightarrow s_{t_1} \rightarrow s_{t_2} \rightarrow \cdots \rightarrow s_{t_i} \).

**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 \( \forall t' \in \text{PRECE}_x(t_1) : \text{PRECE}_x(t') \subseteq \text{PRECE}_x(t_1) \). As \( t_1 \) is the first transaction in the execution, \( D\text{-PRECE}_x(t_1) = \emptyset \) and consequently \( \text{PRECE}_x(t) = \emptyset \). We see this by contradiction: assume there exists a transaction \( t \in D\text{-PRECE}_x(t_1) \), by implication \( s_{t'} \rightarrow s_{t_1} \) (Lemma 4), violating 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 \( \forall t \text{ s.t. } s_t \rightarrow s_{t'} \text{ the property } \forall t' \in \text{PRECE}_x(t) : \text{PRECE}_x(t') \subseteq \text{PRECE}_x(t) \). In other words, we assume that the property holds for the first \( t \) transactions. We now prove that the property holds for the first \( i+1 \) transactions, specifically, we show that \( \forall t' \in \text{PRECE}_x(t_{i+1}) : \text{PRECE}_x(t') \subseteq \text{PRECE}_x(t_{i+1}) \). A transaction \( t' \) belongs to \( \text{PRECE}_x(t_{i+1}) \) if one of two conditions holds: either \( t' \in D\text{-PRECE}_x(t_{i+1}) \), or \( \exists k \in T : t' \in \text{PRECE}_x(t_k) \land t_k \in D\text{-PRECE}_x(t_{i+1}) \). We consider each in turn:

- **If** \( t' \in D\text{-PRECE}_x(t_{i+1}) \): by definition of \( \text{PRECE}_x(t_{i+1}) \), \( \text{PRECE}_x(t') \subseteq \text{PRECE}_x(t_{i+1}) \).

- **If** \( \exists k \in D\text{-PRECE}_x(t_{i+1}) : t' \in \text{PRECE}_x(t_k) \): As \( t_k \in D\text{-PRECE}_x(t_{i+1}) \), by definition of \( \text{PRECE}_x(t_{i+1}) \), \( \text{PRECE}_x(t_k) \subseteq \text{PRECE}_x(t_{i+1}) \). Moreover, by Lemma 4, we have \( s_{t_k} \rightarrow s_{t_{i+1}} \), i.e. \( s_{t_k} \rightarrow s_{t_1} \) (\( s_{t_k} \) directly precedes \( s_{t_{i+1}} \) in \( e \)). Moreover, by Lemma 4, we have \( s_{t_{i+1}} \rightarrow s_{t_{i+1}} \), i.e. \( s_{t_k} \rightarrow s_{t_1} \) (\( s_{t_k} \) directly precedes \( s_{t_{i+1}} \) in \( e \)).

This completes the induction step of the proof.
Combining the base case, and induction step, we conclude: \( \forall t' \in \text{PREC}_e(t) : \text{PREC}_e(t') \subseteq \text{PREC}_e(t) \).

If \( T, i \xrightarrow{V} T_j \wedge T_j \xrightarrow{V} T_k \), by construction we have \( t_i \in \text{PREC}_e(t_j) \wedge t_j \in \text{PREC}_e(t_k) \). From \( t_j \in \text{PREC}_e(t_k) \), we know, by induction, that \( \text{PREC}_e(t_j) \subseteq \text{PREC}_e(t_k) \), and consequently that \( t_i \in \text{PREC}_e(t_k) \). By construction, we have \( T \xrightarrow{V} T_k \), hence we conclude: \( V \text{IS} \) is transitive.

**NOCONFLICT** \( \forall T, S \in \mathcal{H}(T \neq S \wedge T \vdash \text{Write} x : \_ \wedge S \vdash \text{Write} x : \_) \Rightarrow (T \xrightarrow{V} S \wedge S \xrightarrow{V} T) \)

Consider any \( T, S \in \mathcal{H}(T \neq S \wedge T \vdash \text{Write} x : \_ \wedge S \vdash \text{Write} x : \_ \) and let \( t_i, t_j \) be the equivalent transactions in our model such that \( w(x, i) \in \Sigma_i \) and \( w(x, j) \in \Sigma_j \) and consequently \( x \in \mathcal{W}_i \cap \mathcal{W}_j \). Since \( e \) totally orders all the committed transactions, we have either \( s_{t_i} \xrightarrow{t_i} s_{t_j} \) or \( s_{t_j} \xrightarrow{t_i} s_{t_i} \). If \( s_{t_i} \xrightarrow{t_i} s_{t_j} \), it follows from \( s_{t_i} \xrightarrow{t_i} s_{t_j} \wedge \mathcal{W}_i \cap \mathcal{W}_j \neq \emptyset \) 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 \( T \xrightarrow{V} S \).

Similarly, if \( s_{t_i} \xrightarrow{t_i} s_{t_j} \), it follows from \( s_{t_i} \xrightarrow{t_i} s_{t_i} \wedge \mathcal{W}_i \cap \mathcal{W}_j \neq \emptyset \) 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 \( S \xrightarrow{V} T \).

We conclude: \( T \xrightarrow{V} S \wedge S \xrightarrow{V} T \), NOCONFLICT is true.

\((\Leftarrow)\) Now we prove that \( \text{PSI} \Rightarrow \exists e : \forall t \in T : \text{CT}_{PSI}(t, e) \).

By assumption, \( AR \) is a total order over \( T \). We construct an execution \( e \) by applying transactions in the same order as \( AR \), i.e. \( s_{t_i} \xrightarrow{t_i} s_{t_j} \Rightarrow T_i \xrightarrow{AR} T_j \) and subsequently prove that \( e \) satisfies \( \forall t \in T : \text{CT}_{PSI}(t, e) \).

**Preread** First we show that \( \text{PREREAD}_e(T) \) 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 \( sf_o = s_t \) hence \( sf_o \xrightarrow{t} s_t \) 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 transaction in Cerone’s model. We have \( T \vdash \text{Read} x : n \) and \( T' \vdash \text{Write} x : n \). Assuming that values are uniquely identifiable, we have \( T' \xrightarrow{V} \text{MAX}_A((V \text{IS}^{-1}(T)) \cap \{S | S \vdash \text{Write} x : \_ \}) \) by EXT, and consequently \( T' \in V \text{IS}^{-1}(T) \). As \( V \text{IS} \subseteq AR \), \( T' \xrightarrow{V} T \) and consequently \( T' \xrightarrow{AR} T \). Recall that we apply transactions in the same order as \( AR \), hence we have \( s_{t'} \xrightarrow{t} s_t \).

Since we have \((x, n) \in \Sigma_t \) and \( s_{t'} \xrightarrow{t} s_t \), it follows that \( s_{t'} \in R_S(o) \), hence \( R_S(o) \neq \emptyset \). We conclude: for any transaction \( t \), for any operation \( o \in \Sigma_t \), \( R_S(o) \neq \emptyset \), therefore \( \text{PREREAD}_e(T) \) is true.

Now consider any \( t \in T \), we want to prove that \( \forall o \in \Sigma_t : \forall t' \in \text{PREC}_e(t) : o.k \in \mathcal{W}_t \Rightarrow s_{t'} \xrightarrow{t} s_{o} \).

First we prove that \( \forall t' \in \text{PREC}_e(t) \Rightarrow T' \xrightarrow{V} T \).

We previously proved that \( \text{PREREAD}_e(T) \) is true. Hence, by Lemma 6 we know that there is a chain \( t' \xrightarrow{wr/wu} T \).

Consider any edge on the chain: \( t_i \xrightarrow{wr/wu} t_j \):

1. \( t_i \xrightarrow{wu} t_j \): We have \( T_i, T_j \in \mathcal{H} \) and \( T_i \neq T_j \wedge T_j \vdash \text{Write} x : \_ \wedge T_j \vdash \text{Write} x : \_ \), therefore by NOCONFLICT, we have \( t_i \xrightarrow{V} t_j \wedge t_j \xrightarrow{V} t_i \). Note that \( s_{t_i} \xrightarrow{t_i} s_{t_j} \), we know that \( t_i \xrightarrow{AR} t_j \), and since \( V \text{IS} \subseteq AR \), we have \( t_i \xrightarrow{V} t_j \).

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 empty value, i.e. \( \perp \), to a key. It follows that \( T_j \vdash \text{Read} x : n \) and \( n \neq 0 \). By EXT, 
    \[ \text{max}_A((V \text{IS}^{-1}(T_j)) \cap \{S | S \vdash \text{Write} x : \_ \}) \xrightarrow{t} \text{Write} x : n. \]
    Since \( T_i \vdash \text{Write} x : n \), we have \( T_i = \text{max}_A((V \text{IS}^{-1}(T_j)) \cap \{S | S \vdash \text{Write} x : \_ \}) \), hold, and consequently \( T_i \in V \text{IS}^{-1}(T_j) \), i.e. \( T_i \xrightarrow{V} T_j \).

Now we consider the chain \( t' \xrightarrow{wr/wu} t \), and we have that \( T' \xrightarrow{V} T \), by TRANSVIS, we have \( T' \xrightarrow{V} T \).

Now, consider any \( o \in \Sigma_t \) such that \( o.k \in \mathcal{W}_t \), let \( o.k = x \), therefore \( T' \vdash \text{Write} x : \_ \). We previously proved that \( T' \xrightarrow{V} T \). Hence we have \( T' \in V \text{IS}^{-1}(T) \cap \{S | S \vdash \text{Write} x : \_ \} \). Now we consider the following two cases.

If \( o \) is an external read and \( o \) reads the value \((x, \bar{x})\) written by \( \bar{t} \). As transactions cannot write an empty value, i.e. \( \perp \), to a key, we have \( T \vdash \text{Read} x : \bar{x} \) and \( x \neq \bar{x} \). By EXT, 
    \[ \text{max}_A((V \text{IS}^{-1}(T_j)) \cap \{S | S \vdash \text{Write} x : \_ \}) \xrightarrow{t} \text{Write} x : n. \]
    Since \( \bar{T} \vdash \text{Write} x : n \), we have \( \bar{T} = \text{max}_A((V \text{IS}^{-1}(T_j)) \cap \{S | S \vdash \text{Write} x : \_ \}) \), therefore \( T' \xrightarrow{AR} \bar{T} \). Note that we apply transactions in the same order as \( AR \), therefore we have \( s_{t'} \xrightarrow{t} s_{t_j} \) or \( s_{t'} = s_{t_j} \), i.e. \( s_{t'} \xrightarrow{t} s_{t_j} \). Since we proved that \( \text{PREREAD}_e(t) \) is true, we have \( sf_o \) exists and \( s_t = sf_o \), note that by definition \( sf_o \xrightarrow{t} s_{o} \). Now we have \( s_{t'} \xrightarrow{t} s_t = sf_o \xrightarrow{t} s_{o} \), therefore \( s_{t'} \xrightarrow{t} s_{o} \).

If \( o \) is an internal read operation or write operation, then \( s_{o} = s_p(t) \). Since \( t' \in \text{PREC}_e(t) \), by Lemma 5, we have \( s_{t'} \xrightarrow{t} s_t \), therefore \( s_{t'} \xrightarrow{t_p(t) = s_{o}} \), i.e. \( s_{t'} \xrightarrow{t} s_{o} \).