Seeing is Believing: A Client-Centric Specification of Database Isolation

Natacha Crooks
The University of Texas at Austin and Cornell University

Youer Pu
Cornell University

Lorenzo Alvisi
The University of Texas at Austin and Cornell University

Allen Clement
Google, Inc.

ABSTRACT

This paper introduces the first *state-based* formalization of isolation guarantees. 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. Defining isolation guarantees in terms of these states frees definitions from implementation-specific assumptions. It makes immediately clear what anomalies, if any, applications can expect to observe, thus bridging the gap that exists today between how isolation guarantees are defined and how they are perceived. The clarity that results from definitions based on client-observable states brings forth several benefits. First, it allows us to easily compare the guarantees of distinct, but semantically close, isolation guarantees. We find that several well-known guarantees, previously thought to be distinct, are in fact equivalent, and that many previously incomparable flavors of snapshot isolation can be organized in a clean hierarchy. Second, freeing definitions from implementation-specific artefacts can suggest more efficient implementations of the same isolation guarantee. We show how a client-centric implementation of parallel snapshot isolation can be more resilient to *slowdown cascades*, a common phenomenon in large-scale datacenters.

ACM Reference format:
Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of PODC '17, Washington, DC, USA, July 25-27, 2017,* 30 pages.
http://dx.doi.org/10.1145/3087801.3087802

1 INTRODUCTION

Large-scale applications such as Facebook [1], or Twitter [56] 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 meet these demands, commercial databases or distributed storage systems like MySQL [45], Oracle [46], or SQL Server [42] often give up serializability [47] (the gold-standard correctness criterion: an interleaved execution of transactions must be equivalent to a serial schedule) and instead privilege weaker but more scalable correctness criteria [2, 15, 34, 45, 49, 50, 53, 61] (referred to as *weak isolation*) such as snapshot isolation [15] or read committed [15]. In fact, to the best of our knowledge, almost all SQL databases use read committed as their default isolation level [39, 42, 45, 46, 50, 51], with some only supporting read-committed or snapshot isolation [45, 51][1].

This trend poses an additional burden on the application programmer, as these weaker isolation guarantees allow for counter-intuitive application behaviors: relaxing the ordering of operations yields better performance, but introduces schedules and anomalies that could not arise if transactions executed atomically and sequentially. These anomalies may affect application logic: consider a bank account with a $50 balance and no overdraft allowed. If the application runs under read-committed [15], the underlying database may allow two transactions to concurrently withdraw $45, incorrectly leaving the account with a negative balance [15].

To mitigate this increased programming complexity, many commercial databases and distributed storage systems [14, 28, 29, 39–41, 45, 46, 50] interact with applications through a front-end that gives applications the illusion of querying or writing to a logically centralized, failure-free node that will scale as much as one's wallet will allow [28, 29, 39, 40, 46]. In practice, however, this abstraction is leaky: a careful understanding of the system that implements a given isolation level is oftentimes *necessary* to determine which anomalies the system will admit and how these will affect application correctness.

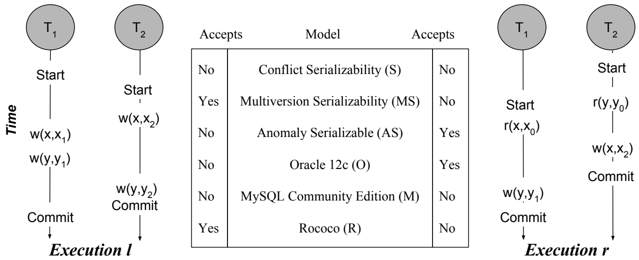
Indeed, the guarantees provided by isolation levels are often dependent on specific and occasionally implicit system properties—be it properties of storage (e.g., whether it is single or multiversioned [16]); of the chosen concurrency control (e.g., whether it is based on locking or timestamps [15]); or other system features (e.g., the existence of a centralized timestamp [27]).

Consider for example serializability [47]: the original ANSI SQL specification states that guaranteeing serializability is equivalent to preventing four phenomena [15]. This equivalence, however, only holds for lock-based, single version databases. Such implicit dependencies continue to have practical consequences: current multiversioned commercial databases that prevent these four phenomenas, such as Oracle 12c, claim to implement serializability, when they in fact implement the weaker notion of snapshot isolation [11, 27, 46]. As such, they accept non-serializable schedules akin to the schedule in Figure 1(r), an anomaly commonly referred to as *write skew* [15]. In contrast, a majority reject the (serializable)

[1] As of June 2017

PODC '17, July 25-27, 2017, Washington, DC, USA

ACM ISBN 978-1-4503-4992-5/17/07...$15.00
http://dx.doi.org/10.1145/3087801.3087802



Accepts	Model	Accepts
No	Conflict Serializability (S)	No
Yes	Multiversion Serializability (MS)	No
No	Anomaly Serializable (AS)	Yes
No	Oracle 12c (O)	Yes
No	MySQL Community Edition (M)	No
Yes	Rococo (R)	No

Figure 1: Serializability. Abbreviations refer to: S[47], MS[17], AS[15] O[46], M[45], R[43].

schedule of Figure 1(l) because, for performance reasons, these systems choose not reorder writes.

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

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

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

First, this model makes clear to developers what anomalies, if any, their applications can expect to observe, thus bridging the semantic gap between how isolation is experienced and how it is formalized. For example, we show (§5.1) how a state-based and client-centric definition brings immediately into focus the root cause of the write-skew anomaly, which distinguishes snapshot isolation from serializability.

Second, by removing the distorting effects of implementation artefacts, our approach makes it easy to compare the guarantees of distinct, but semantically close, isolation guarantees. The results are sometimes surprising. We prove (§5.2) that several well-known flavors of isolation in fact provide the same guarantees: *parallel snapshot isolation* (PSI) [20, 53] is equivalent to *lazy consistency* (PL-2+) [2, 3]; similarly, *generalized snapshot isolation* (GSI) [48] is actually equivalent to ANSI snapshot isolation (ANSI SI) [15], though GSI was proposed as a more scalable alternative to ANSI SI. Likewise, we also show that the lesser known *strong session SI* [24] and *prefix-consistent SI* [48] are also equivalent. Ultimately, the insights offered by state-based definitions enable us to organize in a clean hierarchy (§5.2) what used to be incomparable flavors of snapshot isolation [2, 9, 15, 24, 37, 48, 53].

Finally, by focusing on how clients perceive a given isolation guarantee, rather than on the mechanisms currently used to implement it, a state-based formalization can lead to a fresh, end-to-end perspective on how that guarantee should be implemented. Specifically, a state-based definition of parallel snapshot isolation (PSI) makes clear that the requirement of totally ordering transactions at each datacenter, which is baked into its current definition [53], is only an implementation artefact. Removing it offers the opportunity of an alternative implementation of PSI that makes it resilient to *slowdown cascades* [37], a common failure scenario in large-scale datacenters that has inhibited the adoption of stronger isolation models in industry [5].

After quickly reviewing in Section 2 the current approach to formalizing isolation guarantees, we introduce our state-based model in Section 3, and use it in Section 4 to define several isolation guarantees. We highlight the benefits of our approach in Section 5, summarize related work in Section 6 before outlining its limitations and concluding in Section 7.

2 BACKGROUND

Isolation guarantees have been formalized in many different ways: initial specifications of serializability [47] define correctness as a function of schedule equivalence, while weaker isolation guarantees have been defined using implementation-oriented operational specifications [13, 15, 53] or by relating the order in which transactions commit with the values that they observe [20, 21, 52]. The most prevalent approach, however, has been to formulate isolation guarantees as dependency graphs, with edges denoting conflicts between transactions: this method was introduced by Bernstein et al. to formalize serializability for both single-version [16] and multiversioned databases [17], and subsequently refined by Adya to specify weak isolation guarantees [2]. Adya's specification has since been adopted as the de-facto language for specifying isolation [25, 37, 54, 58]. We select Adya's model as a baseline and prove our definitions equivalent to his in §4.

Adya's formalism We summarize below some of the key definitions and results from Adya's treatment of isolation [2]; a more complete description can be found in Appendix A.

Adya's model is expressed in terms of *histories*, which consist of two parts: a partial order of events that reflect the operations of a set of transactions, and a version order that imposes a total order on committed object versions. Every history is associated with a *directed serialization graph* DSG(H) [17], whose nodes consist of committed transactions and whose edges mark the conflicts (read-write, write-write, or write-read) that occur between them. For specific isolation levels, Adya further augments the model with logical start and commit timestamps for transactions, leading

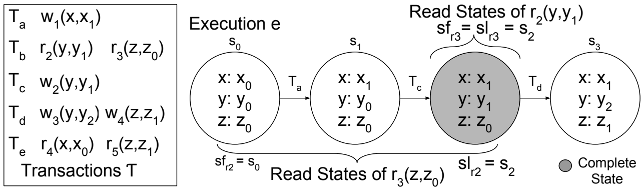

T_a $w_1(x,x_1)$
T_b $r_2(y,y_1)$ $r_3(z,z_0)$
T_c $w_2(y,y_1)$
T_d $w_3(y,y_2)$ $w_4(z,z_1)$
T_e $r_4(x,x_0)$ $r_5(z,z_1)$
Transactions T
Execution e
s_0
s_1
s_3
Read States of $r_2(y,y_1)$
$sf_{r3} = sl_{r3} = s_2$
x: x_0 y: y_0 z: z_0
T_a
x: x_1 y: y_0 z: z_0
T_c
x: x_1 y: y_1 z: z_0
T_d
x: x_1 y: y_2 z: z_1
$sf_{r2} = s_0$
Read States of $r_3(z,z_0)$
$sl_{r2} = s_2$
Complete State

Figure 2: Read States and execution.

to *start-ordered serialization graphs* ($SSG(H)$) that add start-dependency edges to the nodes and edges of the corresponding DSG(H) (two transactions T, T' are start-ordered if the commit timestamp of one precedes the start timestamp of the other).

An execution satisfies a given isolation level if it disallows *aborted reads*, *intermediate reads*, and *circularity*. The first two conditions prevent a transaction T_1 from reading, respectively, (*i*) values produced by an aborted transaction T_2 and (*ii*) a version of an object x written by a transaction T_2 that T_2 subsequently overwrites. The third condition is more complex. Disallowing circularity prevents cycles in the serialization graph; the specific edges that compose the cycle, however, are dependent on the particular isolation level: read-uncommitted disallows cycles consisting only of write-write edges in the DSG(H) (referred to by Adya as phenomenon G0)[2], while all remaining ANSI SQL isolation levels disallow cycles consisting of write-write/write-read edges (phenomenon G1). Serializability also disallows cycles that include read-write edges (G2). In contrast, snapshot isolation disallows write-write/write-read edges without corresponding start edges (G-SI(a)) as well as cycles containing a single read-write edge in the SSG(H) (G-SI(b)).

Towards a new formalism Adya's formalism, like its other existing counterparts, specifies isolation guarantees as constraints on the ordering of the read and write operations that the storage system performs, and relies on low-level implementation details like timestamps or version order. Unfortunately, applications cannot directly observe this ordering: to them, the storage system is a black box. All they can observe are the values returned by the read operations they issue: they experience the storage system as if it were going through a sequence of atomic state transitions, of which they observe a subset. To make it easier for applications to reason about different levels of isolation, we adopt the viewpoint of the applications that must ultimately use their guarantees and introduce a new formalization of isolation based on *application-observable states*.

3 A STATE-BASED MODEL

To the best of our knowledge, our model is the first to specify isolation without relying on some notion of history. Instead, it associates with each transaction the set of candidate states (called *read states*) from which the transaction may have retrieved the values it read during its execution. Read states perform a role similar to Kripke structures [35]: they inform the application of the set of possible worlds (i.e., states) consistent with what a transaction observed during its execution.

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

More formally, we define a *storage system* S with respect to a set $\mathcal{K}$ of keys and $\mathcal{V}$ of values; a *system state* s is a unique mapping from keys to values produced by writes from aborted or committed transactions. For simplicity, we assume that each value is uniquely identifiable, as is common practice both in existing formalisms [2, 17] and in practical systems (ETags in Azure [40] and S3 [7], timestamps in Cassandra [8]). There can thus be no ambiguity, when reading an object, as to which transaction wrote its content. In the initial system state, all keys have value $\perp$; later states similarly include every key, possibly mapped to $\perp$. As is common in database systems, we assume that applications modify the storage system's state using transactions. A *transaction* T is a tuple $(\Sigma_T, \xrightarrow{to})$, where Σ_T is the set of *operations* in T, and $\xrightarrow{to}$ is a total order on Σ_T. Operations can be either reads or writes. *Read* operations $r(k,v)$ retrieve value v by reading key k; *write* operations $w(k,v)$ update k to its new value v. The *read set* of T comprises the keys read by T: $\mathcal{R}_T = \{k | r(k,v) \in \Sigma_T\}$. Similarly, the *write set* of T comprises the keys that T updates: $\mathcal{W}_T = \{k | w(k,v) \in \Sigma_T\}$. For simplicity of exposition, we assume that a transaction only writes a key once. Finally, we assume the existence of a time oracle O that assigns distinct real-time *start* and *commit* timestamps ($T.start$ and $T.commit$) to every transaction $T \in \mathcal{T}$. A transaction T_1 time-precedes T_2 (we write $T_1 <_s T_2$) if $T_1.commit < T_2.start$. Applying a transaction T to a state s transitions the system to a state s' that is identical to s in every key except those written by T. Formally,

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

We refer to s as the *parent state* of T (denoted as $s_{p,T}$) [3]; to the transaction that generated s as T_s; and to the set of keys in which s and s' differ as $\Delta(s,s')$. An *execution* e for a set of transactions $\mathcal{T}$ is a totally ordered set defined by the pair $(\mathcal{S}_e, \xrightarrow{T \in \mathcal{T}})$, where $\mathcal{S}_e$ is the set of states generated by applying, starting from the system's initial state, a permutation of all the transactions in $\mathcal{T}$. We write $s \xrightarrow{*} s'$ (respectively, $s \xrightarrow{+} s'$) to denote a sequence of zero (respectively, one) or more state transitions from s to s' in e. For example, in Figure 2, $\mathcal{T}$ comprises five transactions, operating on a state that consists of the current version of keys x, y, and z.

Note that while e identifies the state transitions produced by each transaction $T \in \mathcal{T}$, it does not specify from which states in $\mathcal{S}_e$ each operation in T reads. In particular, reading a key in replicated distributed systems will not necessarily return the value produced by the latest write to that key, as writes may become visible in different orders at different replicas. In general, multiple states in

[2] We will use Adya's shorthand for this and other phenomena in §4, when we prove that our new state-based definitions of isolation guarantees are equivalent to his.

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

$\mathcal{S}_e$ may be compatible with the value returned by any given operation. We call this subset the operation's *read states*. To prevent operations from *reading from the future*, we restrict the valid read states for the operations in T to be no later than s_p. Further, once an operation in T writes v to k, we require all subsequent operations in T that read k to return v [2]: in this case, their set of read states by convention includes all states in $\mathcal{S}_e$ up to and including s_p.

Definition 2 *Given an execution e for a set of transactions $\mathcal{T}$, let $T \in \mathcal{T}$ and let s_p denote T's parent state. The read states for a read operation $o = r(k,v) \in \Sigma_T$ define the set of states*

$\mathcal{RS}_e(o) = \{s \in \mathcal{S}_e | s \xrightarrow{*} s_p \wedge$
$\Big((k,v) \in s \vee (\exists w(k,v) \in \Sigma_T : w(k,v) \xrightarrow{to} r(k,v))\Big)\}$.

Figure 2 illustrates the notion of read states for the operations executed by transaction T_b. Since r_2 returns y_1, its only possible read state is s_2, i.e., the only state containing y_1. When it comes to r_3, however, z_0 could have been read from any of s_0, s_1, or s_2: from the perspective of the client executing T_b, these read states are indistinguishable. By convention, write operations have read states too: for a write operation in T, they include all states in $\mathcal{S}_e$ up to and including T's parent state. It is easy to prove that the read states of any operation o define a subsequence of contiguous states in the total order that e defines on $\mathcal{S}_e$. We refer to the first state in that sequence as sf_o and to the last state as sl_o. For instance, in Figure 2, sf_{r_3} is s_0 (the first state that contains z_0) and sl_{r_3} is s_2 (z_0 is overwritten in s_3). When the predicate $\textsc{preread}_e(\mathcal{T})$ holds, then such states exist for all transactions in $\mathcal{T}$:

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

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 $\textsc{complete}_{e,T}(s) \equiv s \in \bigcap_{o \in \Sigma_T} \mathcal{RS}_e(o)$.

Looking again at Figure 2, s_2 is a complete state for transaction T_b, as it is in the set of candidate read states of both $r_2(y,y_1)$ ($\{s_2\}$) and $r_3(z, z_0)$ ($\{s_0, s_1, s_2\}$). A complete state is not guaranteed to exist: no such state exists for T_e, as the sole candidate read states of r_4 and r_5 (respectively, s_0 and s_3) are distinct. As we will see in §4, complete states are key to determining whether transactions read from a consistent snapshot.

4 ISOLATION

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

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

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

Table 1 summarizes the commit tests that define the isolation guarantees most commonly-used in research and industry: the ANSI SQL isolation levels (serializability, read committed, read uncommitted, and snapshot isolation) as well as parallel snapshot isolation [20, 53], strict serializability [47], and the recently proposed read atomic [13] isolation level. Though our state-based definitions make no reference to histories, we prove that they are equivalent to those in Adya's classic treatment. As the proofs follow a similar structure, we provide an informal proof sketch only for serializability and snapshot isolation, deferring a more complete and formal treatment to Appendices A, B and E.

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

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

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

$(\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{SER}(T,e) \Leftarrow \neg \mathrm{G1} \wedge \neg \mathrm{G2})$. If no cycle exists in the DSG, we can construct an execution e' such that the parent state s_p of each transaction T is a complete state for T. We construct e' by topologically sorting the DSG (it is acyclic) and by applying every transaction in the resulting order. Thus, if a transaction T' writes a value that T subsequently reads (write-read edge), the state associated with T' is guaranteed to precede T's state in the execution e'.

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

Table 1: Commit Tests

Moreover, as there are no backpointing read-write edges, no other transaction in e' will update an object read by T between the state produced by T' and s_p. s_p is therefore a valid read state for every operation in T and, consequently, a complete state for T. □

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

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

Proof sketch. ($\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{SI}(T,e) \Leftarrow \neg\mathrm{G1} \wedge \neg\mathrm{G\text{-}SI}$). We construct a valid execution for any history satisfying $\neg$G1$\wedge\neg$G-SI using the time-precedes partial order introduced by Adya. First, we topologically sort transactions according to their commit point, and apply them in the resulting order to generate an execution e. Next, we prove that every transaction T satisfies the commit test: we first show that the state created by the last transaction T_{rs} on which T start-depends is a complete state. As T start-depends on T_{rs}, it must also start-depend on all transactions that precede T_{rs}, since, by construction, these transactions have a commit timestamp smaller than T_{rs}. Moreover, as T_{rs} is the last transaction that T starts-depend on, all subsequent transactions will either be concurrent with T, or start-depend on T. Adya's ($\neg$G-SI) requirement (formally, that there cannot be a write-read /write-write edge without also a start dependency edge and there cannot be a cycle consisting of a single read-write edge) implies that T can only read or overwrite a value written by a transaction T' if T start-depends on T'. Any such T' must either be T_{rs} or precede T_{rs} in e. Similarly, if another transaction T'' overwrites a value that T reads, T cannot start-depend on T'' as it would otherwise create a cycle with a start-edge and a single read-write edge. T'' is therefore ordered after T_{rs} in e. We conclude that $s_{T_{rs}}$ necessarily contains all the values that T reads: it is a complete state. Next, we show that $\Delta(s_{T_{rs}}, s_T) = \emptyset$. By construction, T cannot start-depend on any transaction T' that follows T_{rs} in the execution but precedes T. By G-SI, there cannot be a write-write dependency edge from T' to T, and their write-sets must therefore be distinct. Consequently: $\Delta(s_{T_{rs}}, s_T) = \emptyset$.

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

Unlike Adya's, however, the correctness of our state-based definition does not rely on using start and commit timestamps. This is a crucial difference. Including these low-level attributes in the definition has encouraged the development of variations of SI that differ in their use of timestamps, whose fundamental guarantees are, as a result, difficult to compare. In §5.2 we show that, when expressed in terms of application-observable states, several of these variations, thought to be distinct, are actually equivalent![4]

Read committed Read committed allows T to see the effects of concurrent transactions, as long as they are committed. The commit test therefore no longer constrains all operations in T to read from the *same* state; instead, it only requires each of them to

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

read from a state that precedes T in the execution e. We prove in Appendix A.4 that:

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

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

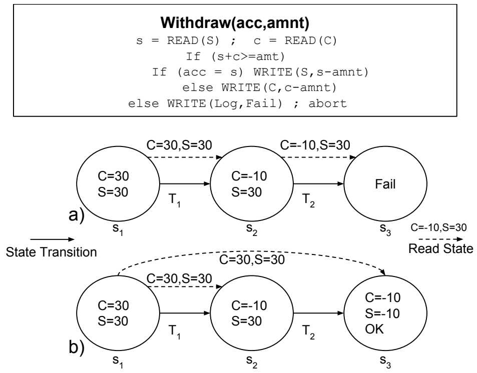
Theorem 4 $\exists e: \forall T \in \mathcal{T}: CT_{RU}(T,e) \equiv \neg G0.$

The reason behind the laxity of the state-based definition is that isolation models in databases consider only committed transactions and are therefore unable to distinguish values produced by aborted transactions from those produced by future transactions. This distinction, however, is not lost in environments, such as transactional memory, where correctness depends on providing guarantees such as opacity [30] for all live transactions. We discuss this further in Section 7.

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

Parallel Snapshot Isolation Parallel snapshot isolation (PSI) was recently proposed by Sovran et al [53] to address SI's scalability issues in geo-replicated settings. Snapshot isolation requires transactions to read from a snapshot (a *complete state* in our parlance) that reflects a single commit ordering of transactions. The coordination implied by this requirement is expensive to carry out in a geo-replicated system and must be enforced even when transactions do not conflict. PSI aims to offer a scalable alternative by allowing distinct geo-replicated sites to commit transactions in different orders. The specification of PSI is given as an abstract specification code that an implementation must emulate. Specifically, a PSI execution must enforce three properties. First, *site snapshot read*: all operations read the most recent committed version at the transaction's origin site as of the time when the transaction began (P1). Second, *no write-write conflicts*: the write sets of each pair of somewhere-concurrent committed transactions must be disjoint (two transactions are somewhere-concurrent if they are concurrent on site(T_1) or site(T_2)) (P2). And finally, *commit causality across sites*: if 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.

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



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

Figure 3: Simple Banking Application. Alice and Bob share checking and savings accounts. Withdrawals are allowed as long as the sum of both account is greater than zero.

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

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

Theorem 5 $\exists e: \forall T \in \mathcal{T}: CT_{PSI}(T,e) \equiv PSI.$

Read Atomic Read atomic [13], like PSI, aims to be a scalable alternative to snapshot isolation. It preserves atomic visibility (transactions observe either all or none of a committed transaction's effects) but does not preclude write-write conflicts nor guarantees that transactions will read from a causally consistent prefix of the execution. These weaker guarantees allow for efficient implementations and nonetheless ensure *synchronization independence*: one client's transactions cannot cause another client's transactions to fail or stall. Read atomic can be expressed in our state-based model as follows:

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

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

Theorem 6 $\exists e: \forall T \in \mathcal{T}: CT_{RA}(T,e) \equiv RA.$

5 BENEFITS OF A STATE-BASED APPROACH

Specifying isolation using client-observable states rather than histories is not only equally expressive, but brings forth several benefits: it gives application developers a clearer intuition for the implications of choosing a given isolation level (§5.1), brings additional clarity to how different isolation levels relate (§5.2), and opens up opportunities for performance improvements in existing implementations (§5.3).

5.1 Minimizing the intuition gap

A state-based model makes it easier for application programmers to understand the anomalies allowed by weak isolation levels, as

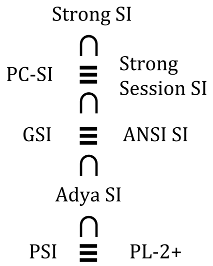


Figure 4: Snapshot-based isolation guarantees hierarchy. (ANSI SI [15, 24], Adya SI [2], Strong SI [24], GSI [48], PSI [53], Strong Session SI [24], PL-2+ [3], PC-SI [24]).

it precisely captures the *root cause* of these anomalies. Consider, for example, snapshot isolation: it allows for a non-serializable behavior called write-skew (see §1), illustrated in the simple banking example of Figure 3. Alice and Bob share checking (C) and savings (S) accounts, each holding \$30, the sum of which should never be negative. Before performing a withdrawal, they check that the total funds in their accounts allow for it. They then withdraw the amount from the specified account, using the other account to cover any overdraft. Suppose Alice and Bob try concurrently to withdraw \$40 from, respectively, their checking and savings account, and issue transactions T_1 and T_2. Figure 3(a) shows an execution under serializability. Because transactions read from their parent state (see Table 1), T_2 observes T_1's withdrawal and, since the balance of Bob's accounts is below \$40, aborts. In contrast, consider the execution under snapshot isolation in Figure 3(b). As it is is legal for both T_1 and T_2 to read from a complete but stale state s_1, Alice and Bob can both find that the combined funds in the two accounts exceed \$40, and, unaware of each other, proceed to generate an execution whose final state s_3 is illegal. The state-based definitions of snapshot isolation and serializability make both the causes and the danger of write-skew immediately clear: to satisfy snapshot isolation, it suffices that both transactions read from the same complete state s_1, even though this behavior is clearly not serializable, as s_1 is not the parent state of T_2. The link is, arguably, less obvious with the history-based definition of snapshot isolation, which requires "disallowing all cycles consisting of write-write and write-read dependencies and a single anti-dependency".

5.2 Removing implementation artefacts

By cleanly separating high-level properties from low-level implementation details, a state-based model makes the plethora of isolation guarantees introduced in recent years easier to compare. We leverage this newfound clarity below to systematize snapshot-based guarantees, including ANSI SI [15], Adya SI [2], Weak SI [24], Strong SI [24], generalized snapshot isolation (GSI) [48], parallel snapshot isolation (PSI) [53], Strong Session SI [24], PL-2+ (Lazy Consistency) [3], Prefix-Consistent SI (PC-SI) [24]. We find that several of these isolation guarantees, previously thought to be distinct, are in fact *equivalent* from a client's perspective, and establish a clean *hierarchy* that encompasses them.

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

In contrast, formulating isolation in terms of client-observable states *forces* definitions that specify guarantees according to how they are *perceived by clients*. It then becomes straightforward to understand what guarantees are provided, and to observe their differences and similarities. Specifically, it clearly exposes the three dimensions along which snapshot-based guarantees differ: (*i*) *time* (whether timestamps are logical [2, 37, 53] or based on real-time [15, 24, 48]); (*ii*) *snapshot recency* (whether the computed snapshot contains *all* transactions that committed before the transaction start time) [2, 24] or can be stale [9, 24, 48, 53]; and *state completeness* (in our parlance, whether snapshots must correspond to a complete state [2, 15, 24, 48] or whether a causally consistent [4] snapshot suffices [3, 9, 37, 53]).

Grouping isolation guarantees in this way highlights a clean hierarchy between these definitions, and suggests that many of the newly proposed isolation levels proposed are in fact equivalent to prior guarantees. We summarize the different commit tests in Table 2 and the resulting hierarchy in Figure 4. As the existence of the hierarchy follows straightforwardly from the commit tests, we defer the proof of its soundness to Appendix F, along with proofs of the corresponding equivalences.

At the top of the hierarchy is Strong SI [24]. It requires that a transaction T observe the effects of all transactions that have committed (in real-time) before T (in other words, read from the most recent database snapshot) and obtain a commit timestamp greater than any previously committed transaction. We express this (Table 2, first row) by requiring that the last state in the execution generated by a transaction that happens before T in real time must be complete ($\forall T' <_s T : s_{T'} \xrightarrow{*} s$), and that the total order defined by the execution respects commit order ($\textsc{c-ord}(T,T') \equiv T.commit < T'.commit$). We prove that this formulation is equivalent to its traditional implementation specification in Appendix C.4:

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

Skipping for a moment one level in the hierarchy, we consider next ANSI SI [15]. ANSI SI weakens Strong SI's requirement that the snapshot from which T reads include *all* transactions that precede T in real-time (including those that access objects that T does not access). This weakening, which improves scalability by avoiding the coordination needed to generate Strong SI's snapshot, can be expressed in our state-based approach by relaxing the requirement that the complete state be the most recent in real-time (Table 2, third line). An attractive consequence of this new formulation is that it clarifies the relationship between ANSI SI and *generalized snapshot isolation* [48], a refinement of ANSI SI for lazily replicated databases. We prove that these two decade-old guarantees are actually equivalent in Appendices C.2 and D.2:

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

Table 2: Commit tests for snapshot-based protocols

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

This is not an isolated case: we find that the two less popular notions of isolation that occupy the level of the hierarchy between Strong SI and ANSI SI (Strong Session SI (SSessSI) [48] and Prefix-Consistent SI (PC-SI) [24]) are also equivalent These guarantees seek to prevent *transaction inversions* [24] (a client c_1 executes a transaction T_1 followed, in real-time by T_2 without T_2 observing the effects of T_1) that can arise when transactions read from a stale snapshot—but without requiring all transactions to read from the most recent snapshot. To this effect, they strike a balance between ANSI SI and Strong SI: they introduce the notion of *sessions* and require a transaction T to read from a snapshot more recent than the commit timestamp of all transactions that precede T in a session (formally: a session *se* is a tuple $(\mathcal{T}_{se},\xrightarrow{se})$ where $\xrightarrow{se}$ is a total order over the transactions in $\mathcal{T}_{se}$ such that $T\xrightarrow{se}T'\Rightarrow T<_s T'$). Our model straightforwardly captures this definition (Table 2, second row) by requiring that the complete state from which a transaction reads follow the commit state of all transactions in a session. We prove the following theorem in Appendices C.3 and D.3:

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

Though ANSI SI or Strong Session SI are both more scalable than Strong SI, their definitions still include several red flags for efficient large-scale implementations. First, they require a total order on transactions ($\text{C-ORD}(s,s_p)$), forcing developers to implement expensive coordination mechanisms, even as transactions may access different objects. Second, they limit a transaction T to reading only from complete states that do not include transactions that committed in real time *after* T's start timestamp. This implementation choice often forces transactions to read further in the past than necessary, making them more prone to write-write conflicts with concurrent transactions. Moreover, it prevents transactions from reading uncommitted operations, precluding efficient implementations for high-contention workloads [26, 60]. Adya's reformulation of SI [2] side-steps many of these baked-in implementation decisions by removing the dependence on real-time, instead allocating *logical* timestamps consistent with the transactions' observations. Our model can capture this distinction by simply removing the two aforementioned clauses from the commit test (Table 2, fourth row), allowing for maximum flexibility for how snapshot isolation can be implemented without affecting client-side guarantees.

The lowest level of the hierarchy covers snapshot-based isolation guarantees intended for large-scale geo-replicated systems. When transactions may be asynchronously replicated for performance and availability, it is challenging to require that transactions read a database snapshot that corresponds to a single moment in time (and hence read from a *complete state*) as it would require transactions to become visible atomically across all (possibly distant) datacenters. PSI [53] (introduced in §4) and PL-2+ [2, 3] consequently weaken Adya's SI to address these new challenges: PSI requires that transactions read from a committed snapshot but allows concurrent transactions to commit in a different order at different sites, while PL-2+ disallows cycles consisting of either write-write/write-read dependencies, or containing a single anti-dependency edge. Unlike what these widely different low-level definitions suggest, taking a client-centric view of these guarantees indicates that PSI and PL-2+ in fact weaken Adya's snapshot isolation in an identical fashion: they no longer require transactions to read from a complete state, and instead require that operations read from a (possibly different) state that includes the effects of all previously observed transactions. Our model cleanly captures the shared guarantee provided by PL-2+/PSI: that a transaction T must observe the effects of all transactions that it is not concurrent with (Table 2, fifth line). We write: for every transaction T' that a transaction T depends on: $\forall o\in\Sigma_T:o.k\in\mathcal{W}_{T'}\Rightarrow s_{T'}\xrightarrow{*}sl_o\equiv\text{CAUS-VIS}(e,T)$. From this client-centric formulation, we prove the following theorem in Appendix E:

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

5.3 Identifying performance opportunities

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

In its original specification, the definition of parallel snapshot isolation [53] requires datacenters to enforce snapshot isolation, even as it globally only offers (as we prove in Theorem 10) the guarantees of lazy consistency/PL-2+. This baked-in implementation decision makes the *very definition* of PSI unsuitable for large-scale partitioned datacenters as it makes the definition (and therefore any system that implements it) susceptible to slowdown cascades. Slowdown cascades (common in large-scale systems [5]) arise when a slow or failed node/partition delays operations that do not access that node itself, and have been identified by industry [5] as the primary barrier to adoption of stronger consistency guarantees. By enforcing SI on every site, the history-based definition of PSI creates a total commit order across all transactions within a datacenter, even as they may access different keys. Transactions thus become dependent on all previously committed transactions on that datacenter, and cannot be replicated to other sites until all these transactions have been applied. If a single partition is slow, all transactions that artificially

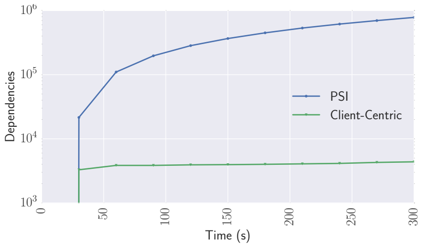


Figure 5: Number of dependencies per transaction as a function of time. TARDiS [23] runs with three replicas on a shared local cluster (2.67GHz Intel Xeon CPU X5650, 48GB memory and 2Gbps network).

depend on transactions on that node will be unnecessarily delayed, creating a cascading slowdown.

An approach based on client-observable states, in contrast, makes no such assumptions: the depend-set of a transaction is computed using client observations and read states only, and thus consists exclusively of transactions that the application itself can *perceive* as ordered with respect to one another. Every dependency created stems from an actual *observation*: the number of dependencies that a client-centric definition creates is consequently minimal (and the fewer dependencies a system creates, the less likely it will be subject to slowdown cascades). To illustrate this potential benefit, we simulated the number of transactional dependencies created at each datacenter by the traditional definition of PSI as compared to the "true" dependencies generated by the proposed client-centric definition, using an asynchronously replicated transactional key-value store, TARDiS [23]. On a workload consisting of read-write transactions (three reads, three writes) accessing data uniformly over 10,000 objects (Figure 5), we found that a client-centric approach decreased dependencies, per transaction, by two orders of magnitude (175×), a reduction that can yield significant dividends in terms of scalability and robustness.

State-based specifications of isolation guarantees can also benefit performance, as they abstract away the details of specific mechanisms used to enforce isolation, and instead focus on how different flavors of isolation constrain permissible read states. A case-in-point is Ardekani et al.'s non-monotonic snapshot isolation (NMSI) [9]: NMSI logically moves snapshots forward in time to minimize the risk of seeing stale data (and consequent aborts due to write-write conflicts), without violating any consistency guarantees. This technique is premised on the observation that, given the values read by the client, the states at the earlier and later snapshot are indistinguishable.

6 RELATED WORK

Most past definitions of isolation and consistency [2, 9, 15–19, 27, 31, 47, 53, 55] 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 [10, 20, 36, 57] distinguishes between *concrete* executions (the nuts-and-bolts implementations details) and *abstract* executions (the system behaviour as perceived by the client). Attiya et al., for instance, introduce the notion of observable causal consistency [10], a refinement of causal consistency where causality can be inferred by client observations. Likewise, Cerone et al. [20, 21] introduce the dual notions of visibility and arbitration to define, axiomatically, a large number of existing isolation levels. The simplicity of their formulation, however, relies on restricting their model to consider only isolation levels that guarantee atomic visibility [13], which prevents them from expressing guarantees like read-committed, the default isolation level of most common database systems [39, 42, 45, 45, 46, 50, 51, 58], and the only supported level for some [45][5]. Shapiro and Ardekani [52] adopt a similar approach to identify three orthogonal dimensions (total order, visibility and transaction composition) that they use to classify consistency and isolation guarantees. All continue, however, to characterize correctness by constraining the ordering of read and write operations and often let system specific details (e.g., system replicas) leak through definitions. Our model takes their approach a step further: it *directly defines* consistency and isolation in terms of the observable states that are routinely used by developers to express application invariants [6, 12, 23]. Finally, several practical systems have recognized the benefits of taking a client-centric approach to system specification and development. These systems target very different concerns, from file I/O [44] to cloud storage [38], and from Byzantine fault-tolerance [33] to efficient Paxos implementations [49]. In the specific context of databases and key-value stores, in addition to Ardekani et al.'s work [9], Mehdi et al. [37] recently proposed a client-centric implementation of causal consistency that is both scalable and resilient to slowdown cascades (§5.3).

7 CONCLUSION

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

Limitations Nonetheless, our model currently has two main limitations, which we plan to address as future work. First, it does not constrain the behavior of ongoing transactions. It thus cannot express consistency models, like opacity [30] or virtual world consistency [32] designed to prevent STM transactions from accessing an invalid memory location. This limitation is consistent with the assumption, made in most isolation and consistency research, that applications never make externally visible decisions based on uncommitted data, so that their actions can be rolled back if the transaction aborts. Second, our model focuses on the traditional transactional/read/write model, predominant in database theory and modern distributed storage systems. To support semantically rich operations, abstract data types, and commutativity, we will start from Weikum et al's theory of multi-level serializability [59], which maps higher-level operations to reads and writes.

Acknowledgements We thank Peter Bailis, Phil Bernstein, Ittay Eyal, Idit Keidar, Fred Schneider, and Immanuel Trummer

[5] as of June 2017

for their valuable feedback on earlier drafts of this work. This work was supported by the National Science Foundation under grants CNS-1409555 and CNS-1718709, and by a Google Faculty Research Award. Natacha Crooks was partially funded by a Google Fellowship in Distributed Computing.

REFERENCES

[1] Facebook. http://www.facebook.com/.

[2] Adya, A. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, Cambridge, MA, USA, Mar. 1999. Also as Technical Report MIT/LCS/TR-786.

[3] Adya, A., and Liskov, B. Lazy consistency using loosely synchronized clocks. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1997), PODC '97, ACM, pp. 73–82.

[4] Ahamad, M., Neiger, G., Burns, J. E., Kohli, P., and Hutto, P. Causal memory: Definitions, implementation and programming. Tech. rep., Georgia Institute of Technology, 1994.

[5] Ajoux, P., Bronson, N., Kumar, S., Lloyd, W., and Veeraraghavan, K. Challenges to adopting stronger consistency at scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (2015), HOTOS'15.

[6] Alvaro, P., Bailis, P., Conway, N., and Hellerstein, J. M. Consistency without borders. In *Proceedings of the 4th ACM Symposium on Cloud Computing* (2013), SOCC '13, pp. 23:1–23:10.

[7] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[8] Apache. Cassandra. http://cassandra.apache.org/.

[9] Ardekani, M. S., Sutra, P., and Shapiro, M. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 32nd International Symposium on Reliable Distributed Systems* (2013), SRDS '13, pp. 163–172.

[10] Attiya, H., Ellen, F., and Morrison, A. Limitations of highly-available eventually-consistent data stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (2015), PODC '15, ACM, pp. 385–394.

[11] Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Highly available transactions: Virtues and limitations. *PVLDB 7*, 3 (2013), 181–192.

[12] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, pp. 1327–1342.

[13] Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems 41*, 3 (July 2016), 15:1–15:45.

[14] Basho. Riak. http://basho.com/products/.

[15] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. A critique of ansi sql isolation levels. *SIGMOD Rec. 24*, 2 (May 1995), 1–10.

[16] Bernstein, P. A., and Goodman, N. Concurrency control in distributed database systems. *ACM Computing Survey 13*, 2 (June 1981), 185–221.

[17] Bernstein, P. A., and Goodman, N. Multiversion concurrency control;theory and algorithms. *ACM Transactions on Database Systems 8*, 4 (1983), 465–483.

[18] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency control and recovery in database systems*. 1987.

[19] Brzezinski, B., Sobaniec, C., and D., W. From session causality to causal consistency. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network based Processing* (2004), PDP 2004.

[20] Cerone, A., Bernardi, G., and Gotsman, A. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015*, (2015).

[21] Cerone, A., and Gotsman, A. Analysing snapshot isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (2016), PODC '16, ACM, pp. 55–64.

[22] Cerone, A., Gotsman, A., and Yang, H. *Transaction Chopping for Parallel Snapshot Isolation*. DISC'15. 2015, pp. 388–404.

[23] Crooks, N., Pu, Y., Estrada, N., Gupta, T., Alvisi, L., and Clement, A. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data* (2016), SIGMOD '16, ACM, pp. 1615–1628.

[24] Daudjee, K., and Salem, K. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (2006), VLDB '06, VLDB Endowment, pp. 715–726.

[25] Escriva, R., Wong, B., and Sirer, E. G. Warp: Lightweight multi-key transactions for key-value stores. *CoRR abs/1509.07815* (2015).

[26] Faleiro, J. M., Abadi, D., and Hellerstein, J. M. High performance transactions via early write visibility. *PVLDB 10*, 5 (2017), 613–624.

[27] Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. Making snapshot isolation serializable. *ACM Transactions on Database Systems 30*, 2 (June 2005), 492–528.

[28] Google. Bigtable - massively scalable nosql. https://cloud.google.com/bigtable/.

[29] Google. Cloud sql - fully managed sql service. https://cloud.google.com/sql/.

[30] Guerraoui, R., and Kapalka, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP '08, ACM, pp. 175–184.

[31] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions Programming Language Systems 12*, 3 (July 1990), 463–492.

[32] Imbs, D., and Raynal, M. Virtual world consistency: A condition for stm systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science 444* (July 2012), 113–127.

[33] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems 27*, 4 (Jan. 2010), 7:1–7:39.

[34] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. Mdcc: multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, pp. 113–126.

[35] Kripke, S. A. Semantical considerations on modal logic. *Acta Philosophica Fennica 16*, 1963 (1963), 83–94.

[36] Mahajan, P., Alvisi, L., and Dahlin, M. Consistency, availability, convergence. Tech. Rep. TR-11-22, Computer Science Department, UT Austin, May 2011.

[37] Mehdi, A., Littley, C., Crooks, N., Alvisi, L., and Lloyd, W. I can't believe it's not causal. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation* (2017), NSDI '17.

[38] Mickens, J., Nightingale, E. B., Elson, J., Gehring, D., Fan, B., Kadav, A., Chidambaram, V., Khan, O., and Nareddy, K. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation* (2014), NSDI'14.

[39] Microsoft. Azure sql database. https://https://azure.microsoft.com/en-us/services/sql-database/?v=16.50.

[40] Microsoft. Azure storage - secure cloud storage. https://azure.microsoft.com/en-us/services/storage/.

[41] Microsoft. Documentdb - nosql service for json. https://azure.microsoft.com/en-us/services/documentdb/.

[42] Microsoft. SQL Server. https://https://www.microsoft.com/en-cy/sql-server/sql-server-2016.

[43] Mu, S., Cui, Y., Zhang, Y., Lloyd, W., and Li, J. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 479–494.

[44] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the Sync. *ACM Transactions on Computer Systems 26*, 3 (Sept. 2008), 6:1–6:26.

[45] Oracle. MySQL Cluster. https://www.mysql.com/products/cluster/.

[46] Oracle. Oracle 12c. https://docs.oracle.com/database/121/.

[47] Papadimitriou, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.

[48] Pedone, F., Zwaenepoel, W., and Elnikety, S. Database replication using generalized snapshot isolation. *24th IEEE Symposium on Reliable Distributed Systems* (2005), 73–84.

[49] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, pp. 43–57.

[50] Postgres. Postgresql. http://www.postgresql.org/.

[51] SAP. Hana. https://www.sap.com/products/hana.html.

[52] Shapiro, M., Ardekani, M. S., and Petri, G. Consistency in 3d (invited paper). In *27th International Conference on Concurrency Theory (CONCUR 2016)* (2016).

[53] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), SOSP '11, pp. 385–400.

[54] Su, C., Crooks, N., Ding, C., Alvisi, L., and Xie, C. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, pp. 283–297.

[55] Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M. J., Theimer, M. M., and Welch, B. B. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on on Parallel and Distributed Information Systems* (1994), PDIS '94, pp. 140–150.

[56] Twitter. Twitter. https://www.twitter.com/.

[57] Viotti, P., and Vukolić, M. Consistency in non-transactional distributed storage systems. *ACM Computing Survey 49*, 1 (June 2016), 19:1–19:34.

[58] Warszawski, T., and Bailis, P. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, pp. 5–20.

[59] Weikum, G. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems 16*, 1 (Mar. 1991), 132–180.

[60] Xie, C., Su, C., Littley, C., Alvisi, L., Kapritsos, M., and Wang, Y. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 279–294.

[61] Zhang, I., Sharma, N. K., Szekeres, A., Krishnamurthy, A., and Ports, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 263–278.

Appendices - Table of Contents

A EQUIVALENCE TO ADYA ET AL.

In this section, we prove the following theorems:

Theorem 1 $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{SER}(T,e) \equiv \neg\mathrm{G1} \wedge \neg\mathrm{G2}$ (§A.2).
Theorem 2 $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{SI}(T,e) \equiv \neg\mathrm{G1} \wedge \neg\text{G-SI}$ (§A.3).
Theorem 3 $\exists e : \forall t \in \mathcal{T} : \mathrm{CT}_{RC}(T,e) \equiv \neg\mathrm{G1}$ (§A.4).
Theorem 4 $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{RU}(T,e) \equiv \neg\mathrm{G0}$ (§A.5).

A.1 Adya et al. model [2] summary

Adya et al. [2] 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 A.1. A history H over a set of transactions consists of two parts: i) a partial order of events E that reflects the operations (e.g., read, write, abort, commit) of those transactions, and ii) a version order, $<<$, that is a total order on committed object versions.

We note that the version-order associated with a history is implementation specific. As stated in Bernstein et al [17]: 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 A.2. Direct conflicts:

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

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

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

Definition A.3. Time-Precedes Order. The time-precedes order, $\prec_t$, is a partial order specified for history H such that:

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

Definition A.4. 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 A.5. Start-Depends. T_j start-depends on T_i if $c_i \prec_t b_j$ i.e., if it starts after T_i commits. We write $T_i \xrightarrow{sd} T_j$

Definition A.6. Start-ordered Serialization Graph or SSG. For a history H, SSG(H) contains the same nodes and edges as DSG(H) along with start-dependency edges.

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

Definition A.7. Phenomena:

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

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

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

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

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

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

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

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

Definition A.8. Each isolation level is defined as proscribing one or more of these phenomena

Serializability (PL-3) $\neg\mathrm{G1} \wedge \neg\mathrm{G2}$
Read Committed (PL-2) $\neg\mathrm{G1}$
Read Uncommitted (PL-1) $\neg\mathrm{G0}$
Snapshot Isolation $\neg\mathrm{G1} \wedge \neg\text{G-SI}$

A.2 Serializability

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

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

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 ¬G1c and ¬G2 state that the $DSG(H)$ must not contain anti-dependency or dependency cycles: $DSG(H)$ must therefore be acyclic. Let $i_1,...i_n$ be a permutation of $1,2,...,n$ such that $T_{i_1},...,T_{i_n}$ is a topological sort of $DSG(H)$ ($DSG(H)$ is acyclic and can thus be topologically sorted). We construct an execution e according to the topological order defined above: $e : s_0 \rightarrow s_{T_{i_1}} \rightarrow s_{T_{i_2}} \rightarrow ... \rightarrow s_{T_{i_n}}$ and show that $\forall t \in \mathcal{T} . \mathrm{CT}_{SER}(T,e)$. Specifically, we show that for all $T = T_{i_j}$, $\text{COMPLETE}_{e,T_{i_j}}(s_{T_{i_{j-1}}})$ where $s_{T_{i_{j-1}}}$ is the parent state of T_{i_j}.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

- Let us assume that H exhibits phenomenon G1a (aborted reads). There must exists events $w_i(x_i), r_j(x_i)$ in H such that T_i subsequently aborted. $\mathcal{T}$ and any corresponding execution e, however, consists only of committed transactions. Hence $\forall e : \nexists s \in \mathcal{S}_e$, s.t. $s \in \mathcal{RS}_e(r_j(x_i))$: no complete state can exists for T_j. There thus exists a transaction for which the commit test cannot be satisfied, for any e. We have a contradiction.
- Let us assume that H exhibits phenomenon G1b (intermediate reads). In an execution e, only the final writes of a transaction are applied. Hence, $\nexists s \in \mathcal{S}_e$, s.t. $s \in \mathcal{RS}_e(r(x_{intermediate}))$. There thus exists a transaction, which for all e, will not satisfy the commit test. We once again have a contradiction.
- Finally, let us assume that the history H displays one or both phenomena G1c or G2. Any history that displays G1c or G2 will contain a cycle in the DSG. Hence, there must exist a chain of transactions $T_i \to T_{i+1} \to \ldots \to T_j$ such that $i = j$ in $DSG(H)$. By Claim 1, we thus have $s_{T_i} \xrightarrow{*} s_{T_{i+1}} \xrightarrow{*} \ldots \xrightarrow{*} s_{T_j}$ for any e. By definition however, a valid execution must be totally ordered. We have our final contradiction.

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

A.3 Snapshot Isolation

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

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

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

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

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

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

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

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

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

We first show that $s_{T_{i_q}} \xrightarrow{*} s_{T_{i_k}}$. As T_{i_j} directly read-depends on T_{i_q}, there must exist an edge $T_{i_q} \xrightarrow{wr} T_{i_j}$ in *SSG(H)*. Given that H disallows phenomenon G-SIa by assumption, there must therefore exist a start-dependency edge $T_{i_q} \xrightarrow{sd} T_{i_j}$ in *SSG(H)*. Therefore we have $c_{T_{i_q}} \prec_t b_{T_{i_j}}$. By definition of time-precedes order, $b_{T_{i_j}} \prec_t c_{T_{i_j}}$. By transitivity of the partial order $c_{T_{i_q}} \prec_t c_{T_{i_j}}$. Given e was constructed by applying every transaction $\mathcal{T}$ in topological order of c, and that we select the largest k such that $T_{i_k} \xrightarrow{sd} T_{i_j}$, it follows that $q \leq k < j$ and $s_{T_{i_q}} \xrightarrow{*} s_{T_{i_k}} \xrightarrow{+} s_{T_{i_j}}$. Next, we argue that the state $s_{T_{i_k}}$ contains the object value pair (x, x_{i_q}). Specifically, we argue that there does not exist a $s_{T_{i_m}}$, where $q < m \leq k$, such that T_{i_m} writes a new version of x. We prove this by contradiction. Consider the smallest such m: T_{i_k} reads the version of x written by T_{i_q} and T_{i_m} writes the next version of x. T_{i_j} thus directly anti-depends on T_{i_m}– i.e., $T_{i_j} \xrightarrow{rw} T_{i_m}$. Given that in time-precedes order, for any two transactions, the start point of one is always comparable to the commit point of the other, we necessarily have $b_{T_{i_j}} \prec_t c_{T_{i_m}}$. Otherwise we would have $c_{T_{i_j}} \prec_t b_{T_{i_m}} \prec_t c_{T_{i_m}}$, i.e. $c_{T_{i_j}} \prec_t c_{T_{i_m}}$, which is inconsistent with the order defined by the execution. In addition, it holds by assumption that $T_{i_k} \xrightarrow{sd} T_{i_j}$. We can conclude that $c_{T_{i_k}} \prec_t b_{T_{i_j}}$. Combined with $b_{T_{i_j}} \prec_t c_{T_{i_m}}$, we will have $c_{T_{i_k}} \prec_t c_{T_{i_m}}$. However, we constructed the execution respecting the time-precedes order of commit point. We have a contradiction. Hence we conclude: $(x, x_{i_q}) \in s_{T_{i_k}}$ and therefore $s_{T_{i_k}} \in \mathcal{RS}_e(r_{i_j}(x_{i_q}))$.

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

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

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

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

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

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

(1) Let $i = 0$.
(2) Set $s = s_{T_i}$; if $i = 0$, $s = s_0$.
(3) Assign a commit timestamp to T_{s_i} if $i \neq 0$.
(4) Assign a start timestamp to all transactions T_k such that T_k satisfies $\text{COMPLETE}_{e,T_k}(s) \wedge (\Delta(s, s_p(T_k)) \cap \mathcal{W}_{s_{T_k}} = \emptyset)$ and T_k does not already have a start timestamp.
(5) Let $i = i + 1$. Repeat 1-4 until the final state in e is reached.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.4 Read Committed

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

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

Let H define a history over $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ 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 $i_1, \ldots i_n$ be a permutation of $1, 2, \ldots, n$ such that $T_{i_1}, \ldots, T_{i_n}$ is a topological sort of $SDSG(H)$ ($SDSG(H)$ is acyclic and can thus be topologically sorted). We construct an execution e according to the topological order defined above: $e : s_0 \rightarrow s_{T_{i_1}} \rightarrow s_{T_{i_2}} \rightarrow \ldots \rightarrow s_{T_{i_n}}$ and show that $\forall T \in \mathcal{T}.\text{CT}_{RC}(T,e)$. Specifically, we show that for all $T = T_{i_j}$,$\textsc{preread}_e(T)$. Consider the three possible types of operations in T_{i_j}:

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

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

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

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

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

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

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

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

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

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

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

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

□

We now derive a contradiction in all cases of G1:

- Let us assume that H exhibits phenomenon G1a (aborted reads). There must exists events $w_i(x_i), r_j(x_i)$ in H such that T_i subsequently aborted. $\mathcal{T}$ and any corresponding execution e, however, consists only of committed transactions. Hence $\forall e : \nexists s \in \mathcal{S}_e, s.t.\ s \in \mathcal{RS}_e(r_j(x_i))$: no complete state can exists for T_j. There thus exists a transaction for which the commit test cannot be satisfied, for any e. We have a contradiction.
- Let us assume that H exhibits phenomenon G1b (intermediate reads). In an execution e, only the final writes of a transaction are applied. Hence,$\nexists s \in \mathcal{S}_e, s.t.\ s \in \mathcal{RS}_e(r(x_{intermediate}))$. There thus exists a transaction, which for all e, will not satisfy the commit test. We once again have a contradiction.
- Finally, let us assume that the history H displays G1c. Any history that displays G1c will contain a cycle in the SDSG(H). Hence, there must exist a chain of transactions $T_i \rightarrow T_k \rightarrow \ldots \rightarrow T_j$ such that $i = j$. By Claim 3, we thus have $s_{T_i} \xrightarrow{+} s_{T_k} \xrightarrow{+} \ldots \xrightarrow{+} s_{T_j}$, $i = j$ for any e. By definition however, a valid execution must be totally ordered. We have our final contradiction.

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

A.5 Read Uncommitted

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

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

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

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

□

B EQUIVALENCE TO READ-ATOMIC

Read atomic [13], like PSI, was introduced a scalable alternative to snapshot isolation. Read atomic preserves atomic visibility (transactions observe either all or none of a committed transaction's effects) but does not preclude write-write conflicts nor guarantee that transactions will read from a causally consistent prefix of the execution. These weaker guarantees allow for efficient implementations in which one client's transactions cannot cause another client's transactions to fail (synchronization independence). We can express read atomic in our state-based model as follows:

Definition B.1. $\mathrm{CT}_{RA}(T,e) \equiv \textsc{preread}_e(T) \wedge \forall r_1(k_1,v_1), r_2(k_2,v_2) \in \Sigma_T \wedge k_2 \in \mathcal{W}_{T_{sf_{r_1}}} \Rightarrow sf_{r_1} \xrightarrow{*} sf_{r_2}$

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

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

B.1 Bailis et al. [13] model summary

We summarize the key definitions of the model here (an alternative formalization was given by Cerone et al. [20]).

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

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

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

B.2 Read Atomic

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

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

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

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

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

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

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

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

C EQUIVALENCE TO ANSI, STRONG AND SESSION SI

In this section, we prove the following theorems:

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

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

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

C.1 Berenson/Daudjee et al. [15, 24] model summary

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

Definition C.1. **ANSI SI** A history H, consisting of the set of transactions $\mathcal{T}$, satisfies ANSI Snapshot Isolation (or weak snapshot isolation) iff, for every T:

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

Definition C.2. **Strong Session SI (R5)** A transaction execution history H is strong session SI under labeling L_H iff it is ANSI SI, and if, for every pair of committed transactions T_i and T_j in H such that $L_H(T_i) = L_H(T_j)$ and T_i's commit precedes the first operation of T_j, $T_i <_s T_j \Rightarrow \mathrm{commit}(T_i) \leq \mathrm{start}(T_j)$.

Definition C.3. **Strong SI (R6)** A transaction execute history H is strong SI iff it is weak SI and if, for every pair of committed transaction T_i and T_j in H such that T_i's commit precedes the first operation of T_j, $T_i <_s T_j \Rightarrow \mathrm{commit}(T_i) \leq \mathrm{start}(T_j)$

C.2 ANSI SI

We now prove the following theorem:
Theorem 8 (a) $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{ANSI\ SI}(T,e) \equiv ANSI\ SI.$

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

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

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

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

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

This concludes the proof.

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

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

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

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

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

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

We first prove the following lemma:

Lemma C.4. $\forall T, T'.commit(T) \leq commit(T') \equiv s_T \xrightarrow{+} s_{T'}$

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

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

R2 Second, we prove that R2 holds. To prove R2a, consider an arbitrary transaction T in e and let s_c be its selected complete state. By assumption, we have that c-ord(T_{s_p},T). By induction, one can easily prove that $\forall T, T' \in \mathcal{T} : T.commit < T'.commit \Leftrightarrow s_T \xrightarrow{+} s_{T'}$ (1). Combined with Lemma C.4, we have $\forall T, T' \in \mathcal{T} : T.commit < T'.commit \Rightarrow s_T \xrightarrow{+} s_{T'} \Rightarrow$ commit(T) $<$ commit(T') so (R2a) holds. Moreover, as start(T) = commit(T_c) and the selected read state of T necessarily precedes T in e, it also follows that start(T) $<$ commit(T). Hence R2c also holds. Finally, we show that R2b holds by contradiction. Assume that commit(T') $\leq$ start(T) and that $T.start < T'.commit$. As start(T) is equal (modulo ϵ) to the timestamp of the selected commit state of T, s_c we have commit(T') $\leq$ commit(T_c) By Lemma C.4 and the aforementioned property (1), it follows that $T'.commit \leq T_c.commit$. By assumption, we have $T_c <_s T$, ak $T_c.commit < T.start$, and consequently $T'.commit \leq T_c.commit \leq T.start$. But we had $T.start < T'.commit$. We have a contradiction and thus R2b holds.

R3 Third, we prove that R3 holds. We first show that a transaction T observes the effects of all transactions with commit(T') $<$ start(T) by contradiction. Consider this transaction T which generates state s when committing. Let s_c and T_c be the selected read state for T. Assume that there exists a transaction T' with timestamp commit(T') $<$ start(T) whose effects T does not observe: there exists a key k that is written by two transactions T' (WRITE(k,v')) and T_v (WRITE(k,v)) such that $s_{T_v} \xrightarrow{+} s_{T'} \xrightarrow{*} s$ in e and T reads value v. By construction, we know that start(T) = commit(T_c) so commit(T') $\leq$ commit(T_c). It follows that $s_{T'} \xrightarrow{*} s_c$ by Lemma C.4 and $s_{T'} \xrightarrow{*} s_c \xrightarrow{+} s$ (as the selected read state necessarily precedes s). As T misses the effect of T' by reading v, we can extend this to $s_{T_v} \xrightarrow{+} s_{T'} \xrightarrow{*} s_c \xrightarrow{+} s$ We know, however, that $(k,v) \in s_c$ as s_c is a complete read state for T so $k \notin \Delta(s_{T_v}, s_c)$. Yet, we had T' write k. We have a contradiction: T observes the effects of all transactions T' with commit(T') $<$ start(T).

Next, we show that T does not observe the writes of transactions T' with commit(T') $>$ start(T). Assume that T observes the effect of transaction T' with commit(T') $>$ start(T). As before, let s_c and T_c be the selected read state for T. By construction, start(T) = commit(T_c) + ϵ, hence commit(T_c) $\leq$ commit(T') and consequently $s_{T_c} \xrightarrow{+} s_{T'}$ by Lemma C.4. We know by assumption that s_{T_c} is the latest complete state such that no-conf$_T(s) \wedge T_s <_s T$. Let o =WRITE(k,v) be the write from T' that T observes. As T' created version v and $s_{T_c} \xrightarrow{+} s_{T'}$ holds, $(k,v) \notin s_c$, so s_c cannot be a read state for o, and as such cannot be a complete state for T. We have a contradiction. R3 holds.

R4 Fourth, we prove that R4 holds. Consider an arbitrary transaction T in e and let s_c be its selected complete state. A transaction T overlaps with committed T' if start(T) $\leq$ commit(T') and commit(T') $\leq$ commit(T). By construction, start(T) = commit(T_c), so we have commit(T_c) $\leq$ commit(T'). By Lemma C.4, commit(T_c) $\leq$ commit(T') $\leq$ commit(T) $\Rightarrow s_{T_c} \xrightarrow{*} s_{T'} \xrightarrow{*} s_T$. By no-conf$_T(s)$ it follows that the writeset of T' does not intersect the writeset of T, so R4 holds.

□

C.3 Strong Session SI

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

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

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

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

(⇒) **Next, we prove:** $(\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{Session\ SI}(T,e) \Rightarrow$ Session SI). (R1), (R2), (R3), (R4) hold by an identical proof to that of Theorem 8(a)(⇒). We consider the same execution e for which the start/commit timestamps are assigned according to the algorithm described in the proof. We do not repeat the proof here and simply show that (R5) holds. To do so, we consider two transactions T and T' such that $T' \xrightarrow{se} T$ and show that $\mathrm{commit}(T') \leq \mathrm{start}(T)$. As T and T' both satisfy the commit test, we have that $\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s_c$ where s_c is the selected read state for T. By Lemma C.4, $s_{T'} \xrightarrow{*} s_c \Rightarrow \mathrm{commit}(T') \leq \mathrm{commit}(T_c)$. Moreover, we have by construction that $\mathrm{start}(T) = \mathrm{commit}(T_c) + \epsilon$. Hence $\mathrm{commit}(T') \leq \mathrm{start}(T)$. This completes the proof.

□

C.4 Strong SI

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

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

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

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

(⇒) **Next, we prove:** $(\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{Strong\ SI}(T,e) \Rightarrow Strong\ SI)$. (R1), (R2), (R3), (R4) hold by an identical proof to that of Theorem 8(a)(⇒). We consider the same execution e for which the start/commit timestamps are assigned according to the algorithm described in the proof. We do not repeat the proof here and simply show that R6 holds. To do so, we consider two transactions T and T' such that $T' <_s T$ and show that $\mathrm{commit}(T') \leq \mathrm{start}(T)$. As T and T' both satisfy the commit test, we have that $\forall T' <_s T : s_{T'} \xrightarrow{*} s_c$ where s_c is the selected read state for T. By Lemma C.4, $s_{T'} \xrightarrow{*} s_c \Rightarrow \mathrm{commit}(T') \leq \mathrm{commit}(T_c)$. Moreover, we have by construction that $\mathrm{start}(T) = \mathrm{commit}(T_c) + \epsilon$. Hence $\mathrm{commit}(T') \leq \mathrm{start}(T)$. This completes the proof.

□

D EQUIVALENCE TO PC-SI AND GSI

In this section, we prove the following theorems:

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

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

D.1 Elnikety et al. [48] model summary

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

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

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

Definition D.1. **Generalized Snapshot Isolation (GSI)** For any history H created by GSI, the following two properties hold (where i, j, and k are distinct)

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

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

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

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

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

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

Definition D.2. **Prefix-consistent Snapshot Isolation (PC-SI)** For any history H created by PC-SI, the following two properties hold (where i,j,k are distinct)

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

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

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

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

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

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

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

D.2 Generalized Snapshot Isolation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Distinct Write Sets We now prove the third part of the commit test: $\text{NO-CONF}_{T_{i_j}}(s_{T_{i_k}})$, i.e. $(\Delta(s_{T_{i_k}}, s_{T_{i_{j-1}}}) \cap \mathcal{W}_{T_{i_j}} = \emptyset)$. We prove this by contradiction. Consider any m, where $k < m <$

j such that $\mathcal{W}_{T_{i_m}} \cap \mathcal{W}_{T_{i_j}} \neq \emptyset$. Since k is the largest index such that $commit(T_{i_k}) < snapshot(T_{i_j})$, we have $commit(T_{i_m}) \geq snapshot(T_{i_j})$. Furthermore, we have $commit(T_{i_m}) < commit(T_{i_j})$ by construction. Combining the two inequalities, we have $snapshot(T_{i_j}) \leq commit(T_{i_m}) < commit(T_{i_j})$ and consequently $writeset(T_{i_m}) \cap writeset(T_{i_j}) = \emptyset$ by D2(4). Yet, we assumed $\mathcal{W}_{T_{i_m}} \cap \mathcal{W}_{T_{i_j}} \neq \emptyset$. We have a contradiction. Thus, $\forall m, k < m < j, \mathcal{W}_{T_{i_m}} \cap \mathcal{W}_{T_{i_j}} = \emptyset$. We conclude that $\Delta(s_{T_{i_k}}, s_{T_{i_{j-1}}}) \cap \mathcal{W}_{T_{i_j}} = \emptyset$.

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

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

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

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

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

D.3 Prefix-consistent Snapshot Isolation

We now prove **Theorem** 9 $\exists e : \forall T \in \mathcal{T} : \text{CT}_{Session\ SI}(T,e) \equiv PC\text{-}SI$ i.e. $\textsc{c-ord}(T_{s_p}, T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s) \equiv P1 \wedge P2$.

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

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

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

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

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

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

E EQUIVALENCE TO PL-2+ AND PSI

In this section, we prove that our state-based definition of PSI is equivalent to the axiomatic formulation of PSI (PSI_A) by Cerone et al. [20] and to the cycle-based specification of PL-2+. Specifically, we prove the following theorems:

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

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

Before beginning, we first prove a useful lemma: if an execution e, written $s_0\rightarrow s_{T_1}\rightarrow s_{T_2}\rightarrow\cdots\rightarrow s_{T_n}$ satisfies the predicate $\mathrm{PREREAD}_e(\mathcal{T})$, then any transaction T that depends on a transaction T' ($T\in\mathrm{PREC}_e(T')$) will always commit after T' and all its dependents in the execution. We do so in two steps: we first prove that T will commit after the transactions that it directly reads from (Lemma E.1), and then extend that result to all the transaction's transitive dependencies (Lemma E.2). Formally

Lemma E.1. $\mathit{PREREAD}_e(\mathcal{T})\Rightarrow\forall\hat{T}\in\mathcal{T}:\forall T\in\mathit{D\text{-}PREC}_e(\hat{T}),s_T\xrightarrow{+}s_{\hat{T}}$

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

(1) $T\in\{T|\exists o\in\Sigma_{\hat{T}}:T=T_{sf_o}\}$ Let o_i be the operation such that $T=T_{sf_{o_i}}$. By assumption, we have $\mathrm{PREREAD}_e(\mathcal{T})$. It follows that $\forall o,sf_o\xrightarrow{+}s_{\hat{T}}$. and consequently that $sf_{o_i}\xrightarrow{+}s_{\hat{T}}$ and $s_T\xrightarrow{+}s_{\hat{T}}$.

(2) $T\in\{T|s_T\xrightarrow{+}s_{\hat{T}}\wedge\mathcal{W}_{\hat{T}}\cap\mathcal{W}_T\neq\emptyset\}$, trivially we have $s_T\xrightarrow{+}s_{\hat{T}}$.

□

We now generalize the result to hold transitively.

Lemma E.2. $\mathit{PREREAD}_e(\mathcal{T})\Rightarrow\forall T'\in\mathit{PREC}_e(T):s_{T'}\xrightarrow{+}s_T$.

Proof. We prove this implication by induction.

Base Case Consider the first transaction T_1 in the execution. We want to prove that for all transactions T that precede T_1 in the execution $s_T\xrightarrow{*}s_{T_1}:\forall T'\in\mathrm{PREC}_e(T):s_{T'}\xrightarrow{*}s_T$. As T_1 is the first transaction in the execution, $\mathrm{D\text{-}PREC}_e(T_1)=\emptyset$ and consequently $\mathrm{PREC}_e(T)=\emptyset$. We see this by contradiction: assume there exists a transaction

$T\in\mathrm{D\text{-}PREC}_e(T_1)$, by implication $s_T\xrightarrow{+}s_{T_1}$ (Lemma E.1), which violates our assumption that T_1 is the first transaction in the execution. Hence the desired result trivially holds.

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

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

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

E.1 Cerone et al. [20]'s model summary

We note that this axiomatic specification, defined by Cerone et al. [20?] is proven to be equivalent to the operational specification of Sovran et al. [53], modulo an additional assumption: that each replica executes each transaction sequentially. The authors state that this is for syntactic elegance only, and does not change the essence of the proof. We provide a brief summary and explanation of the main terminology introduced in Cerone et al.'s framework. We refer the reader to [20] for the full set of definitions. The authors consider a database storing a set of objects $Obj=\{x,y,\ldots\}$, with operations $Op=\{read(x,n),write(x,n)|x\in Obj,n\in\mathbb{Z}\}$. For simplicity, the authors assume the value space to be $\mathbb{Z}$.

Definition E.3. History events are tuples of the form (ι,op), where ι is an identifier from a countably infinite set EventId and $op\in Op$. Let $WEvent_x=\{(\iota,write(x,n))|\iota\in EventId,n\in\mathbb{Z}\}$, $REvent_x=\{(\iota,read(x))|\iota\in EventId,n\in\mathbb{Z}\}$, and $HEvent_x=REvent_x\cap WEvent_x$.

Definition E.4. A transaction T is a pair (E,po), where $E\subseteq HEvent$ is an non-empty set of events with distinct identifiers, and the program order po is a total order over E. A history $\mathcal{H}$ is a set of transactions with disjoint sets of event identifiers.

Definition E.5. An abstract execution is a triple $A=(\mathcal{H},VIS,AR)$ where visibility $VIS\subseteq\mathcal{H}\times\mathcal{H}$ is an acyclic relation; and arbitration $AR\subseteq\mathcal{H}\times\mathcal{H}$ is a total order such that $AR\supseteq VIS$.

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

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

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

Definition E.6. $Hist_\Phi=\{\mathcal{H}|\exists Vis,AR.(\mathcal{H},AR)\models\Phi\}$

The authors define the axioms in Table E1. PSI_A is then defined with the following set of consistency axioms.

Definition E.7. PSI_A allows histories for which there exists an execution that satisfies INT, EXT, TRANSVIS and NOCONFLICT: $Hist_{PSI}=\{H|\exists VIS,AR.(\mathcal{H},VIS,AR)\models$ INT, EXT, TRANSVIS, NOCONFLICT$\}$.

INT	$\forall (E,po) \in \mathcal{H}. \forall event \in E. \forall x,n.(event = (\_, read(x,n)) \wedge (po^{-1}(event) \cap HEvent_x \neq \emptyset)) \Rightarrow \max_{po}(po^{-1}(event) \cap HEvent_x) = (\_,\_(x,n))$
EXT	$\forall T \in \mathcal{H}. \forall x,n. T \vdash Read\ x : n \Rightarrow ((VIS^{-1}(T) \cap \{S \mid S \vdash Write\ x : \_\} = \emptyset \wedge n = 0) \vee \max_{AR}((VIS^{-1}(T) \cap \{S \mid S \vdash Write\ x : \_\}) \vdash Write\ x : n)$
TRANSVIS	VIS is transitive
NOCONFLICT	$\forall T,S \in \mathcal{H}. (T \neq S \wedge T \vdash Write\ x : \_ \wedge S \vdash Write\ x : \_) \Rightarrow (T \xrightarrow{VIS} S \vee S \xrightarrow{VIS} T)$

Table E1: PSI Axioms

E.2 PL-2+

Before beginning, we first prove a useful lemma. Let us consider a history H that contains the same set of transactions $\mathcal{T}$ as an execution e. The version order for H , denoted as $<<$, is instantiated as follows: given an execution e and an object x, $x_i << x_j$ if and only if $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \wedge s_{T_i} \xrightarrow{*} s_{T_j}$. We show that, if a transaction T' is in the depend set of a transaction T ($T' \in \textsc{prec}_e(T)$), then there exists a path of write-read/write-write dependencies from T' to T in the $DSG(H)$. Formally:

Lemma E.8. *$\textit{PREREAD}_e(\mathcal{T}) \Rightarrow \forall T' \in \textit{PREC}_e(T) : T' \xrightarrow{ww/wr}{}^{+} T$ in $DSG(H)$.*

Proof. We improve this implication by induction.

Base Case Consider the first transaction T_1 in the execution. We want to prove that for all transactions T that precede T_1 in the execution $\forall T \in \mathcal{T}$ such that $s_T \xrightarrow{*} s_{T_1}$, the following holds: $\forall T' \in \textsc{prec}_e(T) : T' \xrightarrow{ww/wr}{}^{+} T$ in $DSG(H)$. As T_1 is the first transaction in the execution, $\textsc{d-prec}_e(T_1) = \emptyset$ and consequently $\textsc{prec}_e(T) = \emptyset$. We see this by contradiction: assume there exists a transaction $T \in \textsc{d-prec}_e(T_1)$, by implication $s_T \xrightarrow{+} s_{T_1}$ (Lemma E.1), violating our assumption that T_1 is the first transaction in the execution. ence the implication trivially holds.

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

- If $T' \in \textsc{d-prec}_e(T_{i+1})$: There are two cases: $T' \in \{T \mid \exists o \in \Sigma_{T_{i+1}} : t = T_{sf_o}\}$ or, $T' \in \{T \mid s_T \xrightarrow{+} s_{T_{i+1}} \wedge \mathcal{W}_{T_{i+1}} \cap \mathcal{W}_T \neq \emptyset\}$. If $T' \in \{T \mid \exists o \in \Sigma_{T_{i+1}} : t = T_{sf_o}\}$, T_{i+1} reads the version of an object that T' wrote, hence T_{i+1} read-depends on T', i.e. $T' \xrightarrow{wr} T$.

  If $T' \in \{T \mid s_T \xrightarrow{+} s_{T_{i+1}} \wedge \mathcal{W}_{T_{i+1}} \cap \mathcal{W}_T \neq \emptyset\}$: trivially, $s_{T'} \xrightarrow{+} s_{T_{i+1}}$. Let x be the key that is written by T and T_{i+1}: $x \in \mathcal{W}_{T_{i+1}} \cap \mathcal{W}_T$. By construction, the history H's version order for x is $x_{T'} << x_{T_{i+1}}$. By definition of version order, there must therefore a chain of ww edges between T' and T_{i+1} in $DSG(H)$, where all of the transactions in the chain write the next version of x. Thus: $T' \xrightarrow{ww}{}^{+} T_{i+1}$ holds.
- If $\exists T_k : T' \in \textsc{prec}_e(T_k) \wedge T_k \in \textsc{d-prec}_e(T_{i+1})$. As $T_k \in \textsc{d-prec}_e(T_{i+1})$, we conclude , as above that $T_k \xrightarrow{ww/wr}{}^{+} T_{i+1}$. Moreover, by Lemma E.1, we have $s_{T_k} \xrightarrow{+} s_{T_{i+1}}$, i.e. $s_{T_k} \xrightarrow{*} s_{T_i}$ (s_{T_i} directly precedes $s_{T_{i+1}}$ in e by construction). The induction hypothesis holds for every transaction that precedes T_{i+1} in e, hence $\forall T_{k'} \in \textsc{prec}_e(T_k)$: $T_{k'} \xrightarrow{ww/wr}{}^{+} T_k$. Noting $T' \in \textsc{prec}_e(T_k)$, we see that $T' \xrightarrow{ww/wr}{}^{+} T_k$. Putting everything together, we obtain $T' \xrightarrow{ww/wr}{}^{+} T_k \xrightarrow{ww/wr}{}^{+} T_{i+1}$, i.e. $T' \xrightarrow{ww/wr}{}^{+} T_{i+1}$ by transitivity.

Combining the base case, and induction step, we conclude: $\forall t : \forall T' \in \textsc{prec}_e(T) : T' \xrightarrow{ww/wr}{}^{+} T$. □

Now, we prove Theorem 10 (a) Let $\mathcal{I}$ be PSI. Then $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T,e) \equiv \neg \mathrm{G1} \wedge \neg$G-Single

Proof. Let us recall the definition of PSI's commit test: $\textsc{preread}_e(\mathcal{T}) \wedge \forall o \in \Sigma_T : \forall T' \in \textsc{prec}_e(T) : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$

($\Rightarrow$) **First we prove** $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T,e) \Rightarrow \neg \mathrm{G1} \wedge \neg$G-Single. Let e be an execution that $\forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T,e)$, and H be a history for committed transactions $\mathcal{T}$. We first instantiate the version order for H, denoted as $<<$, as follows: given an execution e and an object x, $x_i << x_j$ if and only if $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \wedge s_{T_i} \xrightarrow{*} s_{T_j}$. It follows that, for any two states such that $(x,x_i) \in T_m \wedge (x,x_j) \in T_n \Rightarrow s_{T_m} \xrightarrow{+} s_{T_n}$.

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

G1a Let us assume that H exhibits phenomenon G1a (aborted reads). There must exist events $w_i(x_i)$, $r_j(x_i)$ in H such that T_i subsequently aborted. $\mathcal{T}$ and any corresponding execution e, however, consists only of committed transactions. Hence $\forall e : \nexists s \in \mathcal{S}_e, s.t.\ s \in \mathcal{RS}_e(r_j(x_i))$: i.e. $\neg\textsc{preread}_e(T_j)$, therefore $\neg\textsc{preread}_e(\mathcal{T})$. There thus exists a transaction for which the commit test cannot be satisfied, for any e. We have a contradiction.

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

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

- $T_i \xrightarrow{ww} T_j$ There must exist an object x such that $x_i << x_j$ (version order). By construction, version order in H is consistent with the execution order e: we have $s_{T_i} \xrightarrow{*} s_{T_j}$.
- $T_i \xrightarrow{wr} T_j$ There must exist a read $o = r_j(x_i) \in \Sigma_{T_j}$ such that T_j reads version x_i written by T_i. By assumption, $\mathrm{CT}_{PSI}(e,T_j)$ holds. By $\textsc{preread}_e(\mathcal{T})$, we have $sf_o \xrightarrow{+} s_{T_j}$; and since sf_o exists, $sf_o = s_{T_i}$. It follows that $s_{T_i} \xrightarrow{+} s_{T_j}$.

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

G-Single We now prove that $\forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T,e) \Rightarrow \neg$G-Single

By way of contradiction, let us assume that H exhibits phenomenon G-Single: $DSG(H)$ must contain a directed cycle with exactly one anti-dependency edge. Let $T_1 \xrightarrow{ww/wr} T_2 \xrightarrow{ww/wr} \ldots \xrightarrow{ww/wr} T_k \xrightarrow{rw} T_1$ be the cycle in $DSG(H)$. We first prove by induction that $T_1 \in \textsc{prec}_e(T_k)$, where T_k denotes the $k-th$ transaction that succedes T_1. We then show that there exist a $T' \in \textsc{prec}_e(T_k)$ such that $o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$ does not hold.

Base case We prove that $T_1 \in \textsc{prec}_e(T_2)$. We distinguish between two cases $T_1 \xrightarrow{ww} T_2$, and $T_1 \xrightarrow{wr} T_2$.

- If $T_1 \xrightarrow{ww} T_2$, there must exist an object k that T_1 and T_2 both write: $k \in \mathcal{W}_{T_1}$ and $k \in \mathcal{W}_{T_2}$, therefore $\mathcal{W}_{T_1} \cap \mathcal{W}_{T_2} \neq \emptyset$. By construction, $T_i \xrightarrow{ww} T_j \Leftrightarrow s_{T_i} \xrightarrow{*} s_{T_j}$. Hence we have $s_{T_1} \xrightarrow{*} s_{T_2}$. By definition of $\textsc{d-prec}_e(T)$, it follows that $T_1 \in \textsc{d-prec}_e(T_2)$.
- If $T_1 \xrightarrow{wr} T_2$, there must exist an object k such that T_2 reads the version of the object created by transaction T_1: $o = r(k_1)$. We previously proved that $T_i \xrightarrow{wr} T_j \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$. It follows that $s_{T_1} \xrightarrow{+} s_{T_2}$ and $sf_o = s_{T_1}$, i.e. $T_1 = T_{sf_o}$. By definition, $T_1 \in \textsc{d-prec}_e(T_2)$.

Since $\textsc{d-prec}_e(T_2) \subseteq \textsc{prec}_e(T_2)$, it follows that $T_1 \in \textsc{prec}_e(T_2)$.

Induction step Assume $T_1 \in \textsc{prec}_e(T_i)$, we prove that $T_1 \in \textsc{prec}_e(T_{i+1})$. To do so, we first prove that $T_i \in \textsc{d-prec}_e(T_{i+1})$. We distinguish between two cases: $T_i \xrightarrow{ww} T_{i+1}$, and $T_i \xrightarrow{wr} T_{i+1}$.

- If $T_i \xrightarrow{ww} T_{i+1}$, there must exist an object k that T_i and T_{i+1} both write: $k \in \mathcal{W}_{T_i}$ and $k \in \mathcal{W}_{T_{i+1}}$, therefore $\mathcal{W}_{T_i} \cap \mathcal{W}_{T_{i+1}} \neq \emptyset$. By construction, $T_i \xrightarrow{ww} T_j \Leftrightarrow s_{T_i} \xrightarrow{*} s_{T_j}$. Hence we have $s_{T_i} \xrightarrow{*} s_{T_{i+1}}$. By definition of $\textsc{d-prec}_e(T)$, it follows that $T_i \in \textsc{d-prec}_e(T_{i+1})$.
- If $T_i \xrightarrow{wr} T_{i+1}$, there must exist an object k such that T_{i+1} reads the version of the object created by transaction T_i: $o = r(k_i)$. We previously proved that $T_i \xrightarrow{wr} T_j \Rightarrow s_{T_i} \xrightarrow{+} s_{T_j}$. It follows that $s_{T_i} \xrightarrow{+} s_{T_{i+1}}$ and $sf_o = s_{T_i}$, i.e. $T_i = T_{sf_o}$. By definition, $T_i \in \textsc{d-prec}_e(T_{i+1})$.

Hence, $T_i \in \textsc{d-prec}_e(T_{i+1})$. The depends set includes the depend set of every transaction that it directly depends on: consequently $\textsc{prec}_e(T_i) \subseteq \textsc{prec}_e(T_{i+1})$. We conclude: $T_1 \in \textsc{prec}_e(T_{i+1})$. Combining the base step and the induction step, we have proved that $T_1 \in \textsc{prec}_e(T_k)$.

We now derive a contradiction. Consider the edge $T_k \xrightarrow{rw} T_1$ in the G-Single cycle: T_k reads the version of an object x that precedes the version written by T_1. Specifically, there exists a version x_m written by transaction T_m such that $r_k(x_m) \in \Sigma_{T_k}$, $w_1(x_1) \in \Sigma_{T_1}$ and $x_m << x_1$. By definition of the PSI commit test for transaction T_k, if $T_1 \in \textsc{prec}_e(T_k)$ and T_1's write set intersect with T_k's read set, then $s_{T_1} \xrightarrow{*} sl_{r_k(x_m)}$. However, from $x_m << x_1$, we have $\forall s, s', s.t. (x, x_m) \in s \wedge (x, x_1) \in s' \Rightarrow s \xrightarrow{+} s'$. Since $(x, x_m) \in sl_{r_k(x_m)} \wedge (x, x_1) \in s_{T_1}$, we have $sl_{r_k(x_m)} \xrightarrow{+} s_{T_1}$. But, we previously proved that T $s_{T_1} \xrightarrow{*} sl_{r_k(x_m)}$. We have a contradiction: H does not exhibit phenomenon G-Single, i.e. $\exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T,e) \Rightarrow \neg G1 \wedge \neg$G-Single.

($\Leftarrow$) We now prove the other direction $\neg$G1 $\wedge \neg$G-Single $\Rightarrow \exists e : \forall T \in \mathcal{T} : \mathrm{CT}_{PSI}(T,e)$. We construct e as follows: Consider only dependency edges in the DSG(H), by $\neg G1$, there exist no cycle consisting of only dependency edges, therefore the transactions can be topologically sorted respecting only dependency edges. Let $i_1, \ldots i_n$ be a permutation of $1, 2, \ldots, n$ such that $T_{i_1}, \ldots, T_{i_n}$ is a topological sort of DSG(H) with only dependency edges. We construct an execution e according to the topological order defined above: $e : s_0 \rightarrow s_{T_{i_1}} \rightarrow s_{T_{i_2}} \rightarrow \ldots \rightarrow s_{T_{i_n}}$.

First we show that $\textsc{preread}_e(\mathcal{T})$ is true: consider any transaction T, for any operation $o \in \Sigma_T$. If o is a internal read operation or o is a write operation, by definition $s_0 \in \mathcal{RS}_e(o)$ hence $\mathcal{RS}_e(o) \neq \emptyset$ follows trivially. Consider the case now where o is a read operation that reads a value written by another transaction T'. Since the topological order includes wr edges and e respects the topological order, $T' \xrightarrow{wr} T$ in $DSG(H)$ implies $s_{T'} \xrightarrow{*} s_T$, then for any $o = r(x, x_{T'}) \in \Sigma_T$, $s_{T'} \in \mathcal{RS}_e(o)$. It follows that $\mathcal{RS}_e(o) \neq \emptyset$ and $\textsc{preread}_e(T)$ is true. In conclusion: $\textsc{preread}_e(\mathcal{T})$ holds.

Next, we prove that $\forall o \in \Sigma_T : \forall T' \in \textsc{prec}_e(T) : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$ holds. For any $T' \in \textsc{prec}_e(T)$, by Lemma E.2, $s_{T'} \xrightarrow{+} s_T$. Consider any $o \in \Sigma_T$, let T' be a transaction such that $T' \in \textsc{prec}_e(T) \wedge o.k \in \mathcal{W}_{T'}$, we now prove that $s_{T'} \xrightarrow{*} sl_o$. Consider the three possible types of operations in T:

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

We show that $s_{T'} \xrightarrow{*} sl_o$ for each of those operation types:

(1) *External Reads.* Let $o = r(x, x_{\hat{T}}) \in \Sigma_T$ read the version for x created by $\hat{T}$, where $\hat{T} \neq T$. Since $\textsc{preread}_e(T)$ is true, we have $\mathcal{RS}_e(o) \neq \emptyset$, therefore $s_{\hat{T}} \xrightarrow{+} s_T$ and $\hat{T} = T_{sf_o}$. From $\hat{T} = T_{sf_o}$, we have $\hat{T} \in \textsc{d-prec}_e(T)$. Now consider T' and $\hat{T}$, we have that $s_{T'} \xrightarrow{+} s_T$ and $s_{\hat{T}} \xrightarrow{+} s_T$. There are two cases:
   - $s_{T'} \xrightarrow{*} s_{\hat{T}}$: Consequently $s_{T'} \xrightarrow{*} s_{\hat{T}} = sf_o \xrightarrow{*} sl_o$ It follows that $s_{T'} \xrightarrow{*} sl_o$.
   - $s_{\hat{T}} \xrightarrow{+} s_{T'}$: We prove that this cannot happen by contradiction. Since $o.k \in \mathcal{W}_{T'}$, T' also writes key $x_{T'}$. By construction, , $s_{\hat{T}} \xrightarrow{+} s_{T'}$ in e implies $x_{\hat{T}} << x_{T'}$. There must consequently exist a chain of ww edges between $\hat{T}$ and T' in $DSG(H)$, where all the transactions on the chain writes a new version of key x. Now consider the transaction in the chain directly after to $\hat{T}$, denoted as $\hat{T}_{+1}$, where $\hat{T} \xrightarrow{ww} \hat{T}_{+1} \xrightarrow{ww}{}^{*} T'$. $\hat{T}_{+1}$ overwrites the version of x T reads. Consequently, T directly anti-depends on $\hat{T}_{+1}$, i.e. $T \xrightarrow{rw} \hat{T}_{+1}$. Moreover $T' \in \textsc{prec}_e(T)$, by Lemma E.8, we have $T' \xrightarrow{ww/wr}{}^{+} T$. There thus exists a cycle consists of only one anti dependency edges as $T \xrightarrow{rw} \hat{T}_{+1} \xrightarrow{ww}{}^{*} T' \xrightarrow{ww/wr}{}^{+} T$, in contradiction with G-Single. $s_{T'} \xrightarrow{*} s_{\hat{T}}$ holds.

   $s_{T'} \xrightarrow{*} s_{\hat{T}}$ holds in all cases. Noting that $s_{\hat{T}} = sl_o$, we conclude $s_{T'} \xrightarrow{*} sl_o$.

(2) *Internal Reads.* Let $o = r(x, x_T)$ read x_T such that $w(x, x_T) \xrightarrow{to} r(x, x_T)$. By definition of $\mathcal{RS}_e(o)$, we have $sl_o = s_p$. Since we have proved that $s_{T'} \xrightarrow{+} s_T$, therefore we have $s_{T'} \xrightarrow{*} s_p = sl_o$ (as $s_p \rightarrow s_T$).

(3) *Writes.* Let $o = w(x,x_T)$ be a write operation. By definition of $\mathcal{RS}_e(o)$, we have $sl_o = s_p$. We previously proved that $s_{T'} \xrightarrow{+} s_T$. Consequently we have $s_{T'} \xrightarrow{*} s_p = sl_o$ (as $s_p \rightarrow s_T$).

We conclude that, in all cases, $\text{CT}_{PSI}(T,e) \equiv \textsc{preread}_e(T) \wedge \forall o \in \Sigma_T : \forall T' \in \textsc{prec}_e(T) : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$. □

E.3 PSI

We now prove the following theorem:

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

We first relate Cerone et al.'s notion of transactions to transactions in our model: Cerone defines transactions as a tuple (E,po) where E is a set of events and po is a program order over E. Our model similalry defines transactions as a tuple $(\Sigma_T, \xrightarrow{to})$, where Σ_T is a set of operations, and $\xrightarrow{to}$ is the total order on Σ_T. These definitions are equivalent: events defined in Cerone are extensions of operations in our model (events include a unique identifier), while the partial order in Cerone maps to the program order in our model. Finally, we relate our notion of versions to Cerone's values.

($\Rightarrow$) **We first prove** $\exists e : \forall T \in \mathcal{T} : \text{CT}_{PSI}(T,e) \Rightarrow PSI_A$.

Construction Let e be an execution such that $\forall T \in \mathcal{T}$: $\text{CT}_{PSI}(T,e)$. We construct AR and VIS as follows: AR is defined as $T_i \xrightarrow{AR} T_j \Leftrightarrow s_{T_i} \rightarrow s_{T_j}$ while VIS order is defined as $T_i \xrightarrow{VIS} T_j \Leftrightarrow T_i \in \textsc{prec}_e(T_j)$. By definition, our execution is a total order, hence our constructed AR is also a total order. VIS defines an acyclic partial order that is a subset of AR (by $\textsc{preread}_e(\mathcal{T})$ and Lemma E.2). We now prove that each consistency axiom holds:

INT $\forall (E,po) \in \mathcal{H}. \forall event \in E. \forall x,n.(event = (\_, read(x,n)) \wedge (po^{-1}(event) \cap HEvent_x \neq \emptyset)) \Rightarrow \max_{po}(po^{-1}(event) \cap HEvent_x) = (\_,\_(x,n))$ 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 $event$ and x such that $(event = (\_,read(x,n)) \wedge (po^{-1}(event) \cap HEvent_x \neq \emptyset))$. We prove that $\max_{po}(po^{-1}(event) \cap HEvent_x) = (\_,\_(x,n))$. By assumption, $(po^{-1}(event) \cap HEvent_x \neq \emptyset))$ holds, there must exist an event such that $\max_{po}(po^{-1}(event) \cap HEvent_x)$. This event is either a read operation, or a write operation:

(1) If $op = \max_{po}(po^{-1}(event) \cap HEvent_x)$ is a write operation: given $event = (\_,read(x,n))$ and $op \xrightarrow{po} event$, the equivalent statement in our model is $w(x,v_{op}) \xrightarrow{to} r(x,n)$. By definition, our model enforces that $w(k,v') \xrightarrow{to} r(k,v) \Rightarrow v = v'$. Hence $v_{op} = n$, i.e. $op = (\_, write(x,n))$, therefore $op = (\_,\_(x,n))$. Hence INT holds.

(2) If $op = \max_{po}(po^{-1}(event) \cap HEvent_x)$ is a read operation, We write $op = (\_, read(x, v_{op}))$. The equivalent formulation in our model is as follows. For $event = (\_,read(x,n))$, we write $o_1 = r(x,n)$, and for op, we write $o_2 = r(x,v_{op})$ with $o_2 \xrightarrow{to} o_1$ where $o_1, o_2 \in \Sigma_T$. Now we consider the following two cases. First, let us assume that there exists an operation $w(k,v)$ such that $w(k,v) \xrightarrow{to} o_2 \xrightarrow{to} o_1$ (all three operations belong to the same transaction). Given that $\xrightarrow{to}$ is a total order, we have $w(k,v) \xrightarrow{to} o_1$ and $w(k,v) \xrightarrow{to} o_2$. It follows by definition of candidate read state that $w(k,v') \xrightarrow{to} r(k,v) \Rightarrow v = v'$, where $v = n \wedge v = v_{op}$, i.e. $v_{op} = n$. Hence $op = (\_,\_(x,n))$ and INT holds. Second, let us next assume that there does not exist an operation $w(k,v) \xrightarrow{to} o_2 \xrightarrow{to} o_1$. We prove by contradiction that $v_{op} = n$ nonetheless. Assume that $v_{op} \neq n$, and consider transactions T_1 that writes (x,n), and T_2 that writes (x,v_{op}), by $\textsc{preread}_e(\mathcal{T})$, we know that sf_{o_1}, sf_{o_2} exist. We have $T_1 = T_{sf_{o_1}}$ and $T_2 = T_{sf_{o_2}}$. By definition of $\textsc{prec}_e(T)$, we have $T_1,T_2 \in \textsc{d-prec}_e(T) \subseteq \textsc{prec}_e(T)$, i.e. $T_1,T_2 \in \textsc{prec}_e(T)$. We note that the sequence of states containing (x,n) is disjoint from states containing (x, v_{op}): in otherwords, the sequence of states bounded by sf_{o_1} and sl_{o_1} and sf_{o_2} and sl_{o_2} are disjoint. Hence, we have either $s_{T_1} \xrightarrow{*} sl_{o_1} \xrightarrow{+} s_{T_2} \xrightarrow{*} sl_{o_2}$, or $s_{T_2} \xrightarrow{*} sl_{o_2} \xrightarrow{+} s_{T_1} \xrightarrow{*} sl_{o_1}$. Equivalently either $T_2 \in \textsc{prec}_e(T) \wedge o_1.k \in \mathcal{W}_{T_2} \wedge sl_{o_1} \xrightarrow{+} s_{T_2}$, or $T_1 \in \textsc{prec}_e(T) \wedge o_2.k \in \mathcal{W}_{T_1} \wedge sl_{o_2} \xrightarrow{+} s_{T_1}$. In both cases, this violates $\text{CT}_{PSI}(T,e)$, a contradiction. We conclude $v_{op} = n$, i.e. $op = (\_,read(x,n))$, therefore $op = (\_,\_(x,n))$.

We proved that $\max_{po}(po^{-1}(event) \cap HEvent_x) = (\_,\_(x,n))$, hence INT holds.

EXT We now prove that EXT holds for $\mathcal{H}$. Specifically, $\forall T \in \mathcal{H}. \forall x,n.T \vdash Read\ x : n \Rightarrow ((VIS^{-1}(T) \cap \{S | S \vdash Write\ x : \_\} = \emptyset \wedge n = 0) \vee \max_{AR}((VIS^{-1}(T) \cap \{S | S \vdash Write\ x : \_\}) \vdash Write\ x : n)$

We proceed in two steps, we first show that there exist a transaction T that wrote (x,n), and next we show that T is the most recent such transaction. Consider any $T \in \mathcal{H}. \forall x,n.T \vdash Read\ x : n$ (a external read). Equivalently, we consider a transaction T in our model such that $r(x,n) \in \Sigma_T$. Let T_n be the transaction that writes (x,n). By assumption, $\textsc{preread}_e(\mathcal{T})$ holds hence sf_o exists and $sf_o = s_{T_n}$, i.e. $T_n = T_{sf_o}$, as T_n created the first state from which o could read from. By definition of $\textsc{prec}_e(T)$, we have $T_n \in \textsc{d-prec}_e(T) \subseteq \textsc{prec}_e(T)$, i.e. $T_n \in \textsc{prec}_e(T)$. Moreover, we defined VIS as $T_i \xrightarrow{VIS} T_j \Leftrightarrow T_i \in \textsc{prec}_e(T_j)$. Hence, we have $T_n \xrightarrow{VIS} T$, and consequently $T_n \in VIS^{-1}(T)$. Since $write(x,n) \in \Sigma_{T_n}$, $T_n \vdash Write\ x : n$.

Next, we show that T_n is larger than any other transaction T' in AR: $T' \xrightarrow{VIS} T \wedge T' \vdash Write\ x : \_$. Consider the equivalent transaction T' in our model, we know that $T' \in \textsc{prec}_e(T)$ ($T' \xrightarrow{VIS} T$) and $x \in \mathcal{W}_{T'}$. As $o = r(x,n) \in \Sigma_T$ and $T' \in \textsc{prec}_e(T) \wedge o.k \in \mathcal{W}_{T'}$, $\text{CT}_{PSI}(T,e)$ implies that $s_{T'} \xrightarrow{*} sl_o$. We note that the sequence of states containing (x,n) is disjoint from states containing $(x,x_{T'})$. It follows that $s_{T'} \xrightarrow{*} sf_o = s_{T_n}$. We can strengthen this to say $s_{T'} \xrightarrow{+} sf_o = s_{T_n}$ as $T' \neq T_n$. By construction, we have $T' \xrightarrow{AR} T_n$, i.e. $T_n = \max_{AR}((VIS^{-1}(T) \cap \{S | S \vdash Write\ x : \_\})$. We conclude, EXT holds.

TRANSVIS If $T_i \xrightarrow{VIS} T_j \wedge T_j \xrightarrow{VIS} T_k$, by construction we have $T_i \in \textsc{prec}_e(T_j) \wedge T_j \in \textsc{prec}_e(T_k)$. From $T_j \in \textsc{prec}_e(T_k)$, we know, since precede-set is maintained transitively, that $\textsc{prec}_e(T_j) \subseteq \textsc{prec}_e(T_k)$, and consequently that $T_i \in \textsc{prec}_e(T_k)$. By construction, we have $T_i \xrightarrow{VIS} T_k$., hence we conclude: VIS is transitive.

NOCONFLICT Recall that this axiom is defined as: $\forall T,S \in \mathcal{H}.(T \neq S \wedge T \vdash \mathit{Write}\ x:\_ \wedge S \vdash \mathit{Write}\ x:\_) \Rightarrow (T \xrightarrow{VIS} S \vee S \xrightarrow{VIS} T)$ Consider any $T,S \in \mathcal{H}.(T \neq S \wedge T \vdash \mathit{Write}\ x:\_ \wedge S \vdash \mathit{Write} x:\_)$ and let T_i, T_j be the equivalent transactions in our model such that $w(x,x_i) \in \Sigma_{T_i}$ and $w(x,x_j) \in \Sigma_{T_j}$ and consequently $x \in \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j}$. Since e totally orders all the committed transactions, we have either $s_{T_i} \xrightarrow{+} s_{T_j}$ or $s_{T_j} \xrightarrow{+} s_{T_i}$. If $s_{T_i} \xrightarrow{+} s_{T_j}$, it follows from $s_{T_i} \xrightarrow{+} s_{T_j} \wedge \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \neq \emptyset$ that $T_i \in \text{D-PREC}_e(T_j) \subseteq \mathrm{PREC}_e(T_j)$, i.e. $T_i \in \mathrm{PREC}_e(T_j)$, and consequently $T \xrightarrow{VIS} S$. Similarly, if $s_{T_j} \xrightarrow{+} s_{T_i}$, it follows from $s_{T_j} \xrightarrow{+} s_{T_i} \wedge \mathcal{W}_{T_i} \cap \mathcal{W}_{T_j} \neq \emptyset$ that $T_j \in \text{D-PREC}_e(T_i) \subseteq \mathrm{PREC}_e(T_i)$, i.e. $T_j \in \mathrm{PREC}_e(T_i)$, and consequently $S \xrightarrow{VIS} T$. We conclude: $T \xrightarrow{VIS} S \vee S \xrightarrow{VIS} T$, NOCONFLICT is true.

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

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

Preread First we show that $\mathrm{PREREAD}_e(\mathcal{T})$ is true: consider any transaction T, for any operation $o \in \Sigma_T$. If o is a internal read operation or o is a write operation, by definition $sf_o = s_0$ hence $sf_o \xrightarrow{*} s_T$ follows trivially. On the other hand, consider the case where o is a read operation that reads a value written by another transaction T': let T and T' be the corresponding transactions in Cerone's model. We have $T \vdash \mathit{Read}\ x:n$ and $T' \vdash \mathit{Write}\ x:n$. Assuming that values are uniquely identifiable, we have $T' = \max_{AR}(VIS^{-1}(T) \cap \{S|S \vdash \mathit{Write}\ x:\_\})$ by EXT, and consequently $T' \in VIS^{-1}(T)$. As $VIS \subseteq AR$, $T' \xrightarrow{VIS} T$ and consequently $T' \xrightarrow{AR} T$. Recall that we apply transactions in the same order as AR, hence we have $s_{T'} \xrightarrow{+} s_T$. Since we have $(x,n) \in s_{T'}$ and $s_{T'} \xrightarrow{+} s_T$, it follows that $s_{T'} \in \mathcal{RS}_e(o)$, hence $\mathcal{RS}_e(o) \neq \emptyset$. We conclude: for any transaction T, for any operation $o \in \Sigma_T$, $\mathcal{RS}_e(o) \neq \emptyset$, hence $\mathrm{PREREAD}_e(\mathcal{T})$ is true. Now consider any $T \in \mathcal{T}$, we want to prove that $\forall o \in \Sigma_T : \forall T' \in \mathrm{PREC}_e(T) : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$. First we prove that $\forall T' \in \mathrm{PREC}_e(T) \Rightarrow T' \xrightarrow{VIS} T$. We previously proved that $\mathrm{PREREAD}_e(\mathcal{T})$ is true. Hence, by Lemma E.8 we know that there is a chain $T' {\xrightarrow{wr/ww}}^{+} T$. Consider any edge on the chain: $T_i \xrightarrow{ww/wr} T_j$:

(1) $T_i \xrightarrow{ww} T_j$: We have $T_i, T_j \in \mathcal{H}$ and $(T_i \neq T_j \wedge T_i \vdash \mathit{Write}\ x : \_ \wedge T_j \vdash \mathit{Write}\ x:\_)$, therefore by NOCONFLICT, we have $T_i \xrightarrow{VIS} T_j \vee T_j \xrightarrow{VIS} T_i$. Note that $s_{T_i} \xrightarrow{*} s_{T_j}$, we know that $T_i \xrightarrow{AR} T_j$, and since $VIS \subseteq AR$, we have $T_i \xrightarrow{VIS} T_j$.

(2) $T_i \xrightarrow{wr} T_j$. We map the initial values in Cerone et al from 0 to $\perp$. Let n be the value that T_i writes and T_j reads. A transaction cannot write the empty value, i.e. $\perp$, to a key. It follows that $T_j \vdash \mathit{Read}\ x : n$ and $n \neq 0$. By EXT, $\max_{AR}(VIS^{-1}(T_j) \cap \{S|S \vdash \mathit{Write}\ x:\_\}) \vdash \mathit{Write}\ x:n$. Since $T_i \vdash \mathit{Write}\ x:n$, $T_i = \max_{AR}(VIS^{-1}(T_j) \cap \{S|S \vdash \mathit{Write}\ x:\_\})$ hold, and consequently $T_i \in VIS^{-1}(T_j)$, i.e. $T_i \xrightarrow{VIS} T_j$.

Now we consider the chain $T' {\xrightarrow{wr/ww}}^{+} t$, and we have that $T' {\xrightarrow{VIS}}^{+} T$, by TRANSVIS, we have $T' \xrightarrow{VIS} T$. Now, consider any $o \in \Sigma_T$ such that $o.k \in \mathcal{W}_{T'}$, let $o.k = x$, therefore $T' \vdash \mathit{Write}\ x:\_$. We previously proved that $T' \xrightarrow{VIS} T$. Hence we have $T' \in VIS^{-1}(T) \cap \{S|S \vdash \mathit{Write}\ x:\_\}$. Now we consider the following two cases. If o is an external read, and reads the value $(x,\hat{x})$ written by $\hat{T}$. As transactions cannot write an empty value, i.e. $\perp$, to a key, we have $T \vdash \mathit{Read}\ x:\hat{x}$ and $\hat{x} \neq 0$. By EXT, $\max_{AR}(VIS^{-1}(T_j) \cap \{S|S \vdash \mathit{Write}\ x:\_\}) \vdash \mathit{Write}\ x:\hat{x}$. Since $\hat{T} \vdash \mathit{Write}\ x:n$, we have $\hat{T} = \max_{AR}(VIS^{-1}(T_j) \cap \{S|S \vdash \mathit{Write}\ x:\_\})$, therefore $T' \xrightarrow{AR} \hat{T}$ or $T' = \hat{T}$. Note that we apply transactions in the same order as AR, therefore we have $s_{T'} \xrightarrow{+} s_{\hat{T}}$ or $s_{T'} = s_{\hat{T}}$, i.e. $s_{T'} \xrightarrow{*} s_{\hat{T}}$. Since we proved that $\mathrm{PREREAD}_e(T)$ is true, we have sf_o exists and $s_{\hat{T}} = sf_o$, note that by definition $sf_o \xrightarrow{*} sl_o$. Now we have $s_{T'} \xrightarrow{*} s_{\hat{T}} = sf_o \xrightarrow{*} sl_o$, therefore $s_{T'} \xrightarrow{*} sl_o$. If o is an internal read operation or write operation, then $sl_o = s_p(T)$. Since $T' \in \mathrm{PREC}_e(T)$, by Lemma E.2, we have $s_{T'} \xrightarrow{+} s_T$, therefore $s_{T'} \xrightarrow{*} s_p(T) = sl_o$, i.e. $s_{T'} \xrightarrow{*} sl_o$.

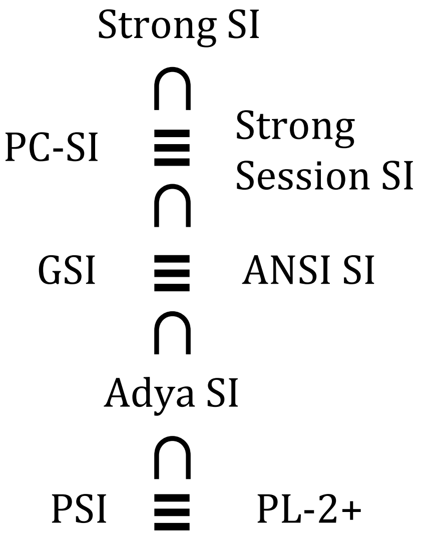


Figure F1: Snapshot-based isolation guarantees hierarchy. Equivalences are new results (ANSI SI [15], Adya SI [2], Weak SI [24], Strong SI [24], generalized snapshot isolation (GSI) [48], parallel snapshot isolation (PSI) [53], Strong Session SI [24], PL-2+ (Lazy Consistency) [3], prefix-consistent SI (PC-SI) [24])

F HIERARCHY

In this section, we prove the existence of the strict hierarchy described in Figure F1. Specifically, we prove:

Theorem F.1. *Adya SI⊂PSI.*

Theorem F.2. *ANSI SI⊂Adya SI.*

Theorem F.3. *Strong Session SI⊂ANSI SI.*

Theorem F.4. *Strong SI⊂Strong Session SI.*

The equivalence results derived from previous appendices complete the proof.

F.1 Adya SI ⊂ PSI

Theorem F.1 Adya SI⊂PSI.

Proof. **Adya SI ⊆ PSI** First we prove that Adya SI ⊆ PSI. Specifically, we prove that, if there exists an e such that $\forall T \in \mathcal{T}$: $\text{CT}_{AdyaSI}(T,e)$, that same e also satisfies $\forall T \in \mathcal{T} : \text{CT}_{PSI}(T,e)$ where $\text{CT}_{PSI}(T,e) = \textsc{preread}_e(T) \land \forall T' \triangleright T : \forall o \in \Sigma_T : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$ Consider any T that satisfies the commit test $\text{CT}_{AdyaSI}(T,e) = \exists s \in S_e : \textsc{complete}_{e,T}(s) \land \textsc{no-conf}_T(s)$ and let s_c be the state that satisfies $\textsc{complete}_{e,T}(s) \land \textsc{no-conf}_T(s)$. Since $\textsc{complete}_{e,t}(s_c)$, we have $s_c \in \bigcap_{o \in \Sigma_T} \mathcal{RS}_e(o)$. It follows that $\forall o \in \Sigma_T : \mathcal{RS}_e(o) \neq \emptyset$ and consequently that $\textsc{preread}_e(T)$ is satisfied. Now we consider any T' such that $T' \blacktriangleright T$, or equivalently, $T' \in \textsc{d-prec}_e(T)$ where $\textsc{d-prec}_e(\hat{T}) = \{T | \exists o \in \Sigma_{\hat{T}} : T = T_{sf_o}\} \cup \{T | s_T \xrightarrow{+} s_{\hat{T}} \land \mathcal{W}_{\hat{T}} \cap \mathcal{W}_T \neq \emptyset\}$. Let us first assume that $T' \in \{\hat{T} | \exists o \in \Sigma_T : \hat{T} = T_{sf_o}\}$ such that $\exists o \in \Sigma_T : T' = sf_o$. Since $T' = sf_o$, we have that $s_{T'} \in \mathcal{RS}_e(o)$ and consequently that $s_{T'} \xrightarrow{*} s_c \xrightarrow{*} sl_o$ Assuming instead that $T' \in \{\hat{T} | s_{\hat{T}} \xrightarrow{+} s_T \land \mathcal{W}_T \cap \mathcal{W}_{\hat{T}} \neq \emptyset\}$. As $\textsc{no-conf}_T(s_c)$, i.e. $\Delta(s_c, sp_T) \cap \mathcal{W}_T = \emptyset$ implies that either $s_{T'} \xrightarrow{*} s$ or $s_T \xrightarrow{*} s_{T'}$ is true, we can conclude that $s_{T'} \xrightarrow{*} s_c$ holds, and consequently that $s_{T'} \xrightarrow{*} sl_o$. Combining these two results, we can conclude that if $T' \blacktriangleright T$, $s_{T'} \xrightarrow{*} s_c$. Strengthening this result using the definition of read state, we have $T' \blacktriangleright T \Rightarrow s_{T'} \xrightarrow{+} s_T$. Taking the transitive closure, we have that $T' \triangleright T \Rightarrow s_{T'} \xrightarrow{+} s_T$. Now, considering the definition of $\triangleright$: for any $T' \triangleright T$, either $T' \blacktriangleright T$, or $\exists \hat{T} : T' \triangleright \hat{T} \land \hat{T} \blacktriangleright T$. If $T' \blacktriangleright T$, we already proved that $s_{T'} \xrightarrow{*} s_c$. Now considering $\exists \hat{T} : T' \triangleright \hat{T} \land \hat{T} \blacktriangleright T$. We previously proved that $T' \triangleright \hat{T} \Rightarrow s_{T'} \xrightarrow{+} s_{\hat{T}}$ and that $\hat{T} \blacktriangleright T \Rightarrow s_{\hat{T}} \xrightarrow{*} s_c$. Combining these two implications, we have $\exists \hat{T} : T' \triangleright \hat{T} \land \hat{T} \blacktriangleright T \Rightarrow s_{T'} \xrightarrow{*} s_c$, and consequently that $T' \triangleright T \Rightarrow s_{T'} \xrightarrow{*} s_c$. Since $s_c \in \bigcap_{o \in \Sigma_T} \mathcal{RS}_e(o)$, we have $\forall o \in \Sigma_T : s_c \xrightarrow{*} sl_o$. We have proved $\forall T' \triangleright T : \forall o \in \Sigma_t : s_{T'} \xrightarrow{*} sl_o$, which trivially implies $\forall T' \triangleright T : \forall o \in \Sigma_t : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$, i.e. $\textsc{caus-vis}(e,T)$ is satisfied. Combining all previous results, we have Adya SI ⊆ PSI.

Adya SI ≠ PSI Second, we prove Adya SI ≠ PSI by describing a set of transactions that satisfy PSI but not Adya SI Consider the five following transactions T_1, T_2, T_3, T_4, T_5, where $T_1 : w(x, x_1) w(y, y_1)$, $T_2 : r(x, x_1) w(x, x_2)$, $T_3 : r(y, y_1) w(y, y_2)$, $T_4 : r(x, x_2) r(y, y_1)$, $T_5 : r(x, x_1) r(y, y_2)$. This set of transactions satisfies PSI as it admits the following execution e such that all transactions satisfy the commit test: $s_0 \xrightarrow{T_1} s_1 \xrightarrow{T_2} s_2 \xrightarrow{T_3} s_3 \xrightarrow{T_4} s_4 \xrightarrow{T_5} s_5$. In contrast, the aforementioned transactions do not satisfy Adya SI as there does not exist an execution such that their commit tests are satisfied. Indeed, to satisfy the commit test of all these transactions, there should exist complete states s and s' for T_4 and T_5 respectively, where s should contain values (x, x_2) and (y, y_1) , and s' values (x, x_1) and (y, y_2). Generating s requires applying transactions T_1 and T_2 before applying transaction T_3, while generating s' requires applying T_1 and T_3 before applying T_2. As the execution e is totally ordered, satisfying both these constraints is impossible, hence there cannot exist complete states for both T_4 and T_5. This set of transactions thus satisfies PSI but not Adya SI. We conclude: Adya SI⊂PSI □

F.2 ANSI SI ⊂ Adya SI

Theorem F.2 ANSI SI⊂Adya SI.

ANSI SI ⊆ Adya SI First we prove that ANSI SI ⊆ Adya SI. The result follows trivially from the definition of the definitions' commit tests: $\exists s \in S_e : \textsc{complete}_{e,T}(s) \land \textsc{no-conf}_T(s) \implies \textsc{c-ord}(T_{s_p}, T) \land \exists s \in S_e : \textsc{complete}_{e,T}(s) \land \textsc{no-conf}_T(s) \land (T_s <_s T)$.

ANSI SI ≠ Adya SI Now we prove that ANSI SI ≠ Adya SI. Consider the set of transactions: $T_1 : w(x, x_1)$, T_1 starts at real time 1 and commits at real time 4; $T_2 : r(x, x_1)$, T_2 starts at real time 2 and commits at real time 3. The set of transactions satisfy Adya SI as there exists an execution for which the commit test of all transactions is satisfied: $s_0 \xrightarrow{T_1} s_{T_1} \xrightarrow{T_2} s_{T_2}$, with s_0 being the selected complete state for T_1 and s_1 being the selected complete state for T_2. However, it does not satisfy ANSI SI. Given that $\textsc{c-ord}(T_{s_p}, T)$ must hold, the only possible execution is $s_0 \xrightarrow{T_2} s_{T_2} \xrightarrow{T_1} s_{T_1}$. But s_0 is not a valid complete state for T_2. As this is the only possible execution, the aforementioned set of transactions does not satisfy ANSI SI, i.e. ANSI SI ≠ Adya SI. Therefore, we conclude that ANSI SI ⊂ Adya SI.

F.3 Strong Session SI ⊂ ANSI SI

Theorem F.3 Strong Session SI ⊂ ANSI SI.

First, we prove that Strong Session SI ⊆ ANSI SI. The result follows trivially from the definition of the definitions' commit tests: $\textsc{c-ord}(T_{s_p},T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (T_s <_s T) \implies \textsc{c-ord}(T_{s_p},T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (T_s <_s T) \wedge (\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s)$. Next, we prove that Strong Session SI ≠ ANSI SI. Consider the set of transactions: $T_1 : w(x,x_1)$, T_1 starts at real time 1 and commits at real time 2; $T_2 : r(x,x_1)w(x,x_2)$, T_2 starts at real time 3 and commits at real time 4; $T_3 : r(x,x_1)$, T_3 starts at real time 5 and commits at real time 6. T_2 and T_3 are in the same session. These transactions satisfy ANSI SI as there exists an execution e such that the commit test of all transactions is satisfied: $s_0 \xrightarrow{T_1} s_{T_1} \xrightarrow{T_2} s_{T_2} \xrightarrow{T_3} s_{T_3}$. The execution satisfies $\textsc{c-ord}(T_{s_p},T)$ with s_0 being the satisfying state for T_1, s_1 being the satisfying state for T_2, s_1 being the satisfying state for T_3. Due to $\textsc{c-ord}(T_{s_p},T)$, this is the only possible execution. This execution, however, does not satisfy Strong Session SI. In this execution, the only possible complete state for T_3 is s_{T_1}. Since $T_2 \xrightarrow{se} T_3$, satisfying Strong Session SI would require that $s_{T_2} \xrightarrow{*} s_{T_1}$, contradicting the order of state transitions in the execution. Since no other execution satisfies the commit test for all transactions, the aforementioned set of transactions does not satisfy Strong Session SI, i.e. Strong Session SI ≠ ANSI SI. We conclude: Strong Session SI ⊂ ANSI SI.

F.4 Strong SI ⊂ Strong Session SI

Theorem F.4 Strong SI ⊂ Strong Session SI.

First we prove that Strong SI ⊆ Strong Session SI. Specifically, we prove that if there exists an execution e such that $\textsc{c-ord}(T_{s_p},T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (T_s <_s T) \wedge (\forall T' <_s T : s_{T'} \xrightarrow{*} s)$, that same e also satisfies $\textsc{c-ord}(T_{s_p},T) \wedge \exists s \in S_e : \textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (T_s <_s T) \wedge (\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s)$. Let $\mathcal{T}$ denote a set of transactions satisfying Strong SI. Consider an execution e that satisfy $\text{CT}_{StrongSI}(\mathcal{T},e)$ (such e must exist by definition). For each transaction T, consider the state s that satisfies $\textsc{complete}_{e,T}(s) \wedge \textsc{no-conf}_T(s) \wedge (T_s <_s T) \wedge (\forall T' <_s T : s_{T'} \xrightarrow{*} s)$, we prove that s also satisfies $\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s$. We know by assumption that $\forall T' <_s T : s_{T'} \xrightarrow{*} s$ for every T in e as it is Strong SI. Moreover, by definition, $T' \xrightarrow{se} T \Rightarrow T'.commit < T.start$, i.e. $T' <_s T$. It thus trivially follows that $\forall T' \xrightarrow{se} T : s_{T'} \xrightarrow{*} s$ and consequently that every transaction in e satisfies $\text{CT}_{Session\ SI}(\mathcal{T},e)$, so Strong SI ⊆ Strong Session SI. Now, we prove that Strong SI ≠ Strong Session SI. Consider the set of transactions: $T_1 : w(x,x_1)$, T_1 starts at real time 1 and commits at real time 2; $T_2 : r(x,x_1)w(x,x_2)$, T_2 starts at real time 3 and commits at real time 4; $T_3 : r(x,x_1)$, T_3 starts at real time 5 and commits at real time 6. No two transactions belong to the same session. These transactions satisfy Strong Session SI as there exist an execution e such that the commit test of all transactions is satisfied: $s_0 \xrightarrow{T_1} s_{T_1} \xrightarrow{T_2} s_{T_2} \xrightarrow{T_3} s_{T_3}$. The execution satisfies $\textsc{c-ord}(T_{s_p},T)$, with s_0 being the selected complete state for T_1, s_1 the selected complete state for T_2 and T_3. However, the set of transactions does not satisfy Strong SI: as $\textsc{c-ord}(T_{s_p},T)$ must hold, the only possible execution is $s_0 \xrightarrow{T_1} s_{T_1} \xrightarrow{T_2} s_{T_2} \xrightarrow{T_3} s_{T_3}$. The only possible complete state for T_3 is s_{T_1}. Since $T_2 <_s T_3$, satisfying Strong SI would require $s_{T_2} \xrightarrow{*} s_{T_1}$, contradicting the order of state transitions in the execution. Since no other execution satisfies the commit test for all transactions, the aforementioned set of transactions does not satisfy Strong SI, i.e. Strong Session SI ≠ ANSI SI. We conclude: Strong SI ⊂ Strong Session SI.