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,* 10 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PODC '17, July 25-27, 2017, Washington, DC, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4992-5/17/07…$15.00
http://dx.doi.org/10.1145/3087801.3087802

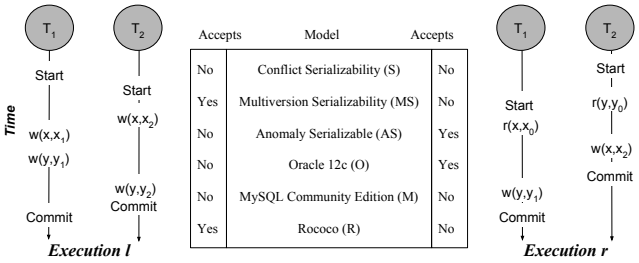

Figure 1: Serializability. Abbreviations refer to: S[47], MS[17], AS[15] O[46], M[45], R[43].

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 our technical report [22].

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

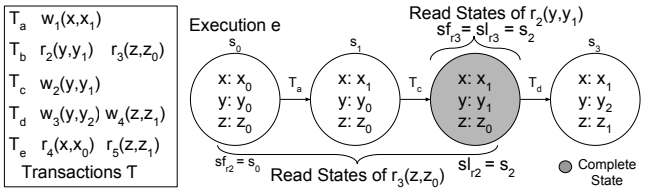

Figure 2: Read States and execution.

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 $\mathcal{O}$ that assigns distinct real-time *start* and *commit* timestamps ($T.start$ and $T.commit$) to every transaction $T \in \mathcal{T}$. A transaction T_1 time-precedes T_2 (we write $T_1 <_s T_2$) if $T_1.commit < T_2.start$. 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*

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

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 our extended technical report [22].

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 :

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'. Moreover, as there are no backpointing read-write edges, no other

Serializability ($\text{CT}_{SER}(T,e)$)	$\text{COMPLETE}_{e,T}(s_p)$
Snapshot Isolation ($\text{CT}_{SI}(T,e)$)	$\exists s \in S_e . \text{COMPLETE}_{e,T}(s) \wedge \text{NO-CONF}_T(s)$
Read Committed ($\text{CT}_{RC}(T,e)$)	$\text{PREREAD}_e(T)$
Read Uncommitted ($\text{CT}_{RU}(T,e)$)	True
Parallel Snapshot Isolation ($\text{CT}_{PSI}(e,T)$)	$\text{PREREAD}_e(T) \wedge \forall T' \triangleright T : \forall o \in \Sigma_T : o.k \in \mathcal{W}_{T'} \Rightarrow s_{T'} \xrightarrow{*} sl_o$
Strict Serializability ($\text{CT}_{SSER}(e,T)$)	$\text{COMPLETE}_{e,T}(s_p) \wedge \forall T' \in \mathcal{T} : T' <_s T \Rightarrow s_{T'} \xrightarrow{*} s_T$
Read Atomic ($\text{CT}_{RA}(e,T)$)	$\forall r_1(k_1,v_1), r_2(k_2,v_2) \in \Sigma_T \wedge k_2 \in \mathcal{W}_{T_{sfr_1}} \Rightarrow sfr_1 \xrightarrow{*} sfr_2$

Table 1: Commit Tests

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 $\text{NO-CONF}_T(s) \equiv \Delta(s,s_p) \cap \mathcal{W}_T = \emptyset$. We prove that this definition is equivalent to Adya's:

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

Proof sketch. ($\exists e : \forall T \in \mathcal{T} : \text{CT}_{SI}(T,e) \Leftarrow \neg\text{G1} \wedge \neg\text{G-SI}$). We construct a valid execution for any history satisfying $\neg$G1$\wedge\neg$G-SI using the time-precedes partial order introduced by Adya. First, we topologically sort transactions according to their commit point, and apply them in the resulting order to generate an execution e. Next, we prove that every transaction T satisfies the commit test: we first show that the state created by the last transaction T_{rs} on which T start-depends is a complete state. As T start-depends on T_{rs}, it must also start-depend on all transactions that precede T_{rs}, since, by construction, these transactions have a commit timestamp smaller than T_{rs}. Moreover, as T_{rs} is the last transaction that T starts-depend on, all subsequent transactions will either be concurrent with T, or start-depend on T. Adya's ($\neg$G-SI) requirement (formally, that there 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: $\Delta(s_{T_{rs}},s_T) = \emptyset$.

($\exists e : \forall T \in \mathcal{T} : \text{CT}_{SI}(T,e) \Rightarrow \neg\text{G1} \wedge \neg\text{G-SI}$). We show that the serialization graph $SSG(H)$ corresponding to e does not exhibit phenomena G1 or G-SI. Every transaction in e reads from some previous state and commits in the total order defined by e. It follows that all write-write and write-read edges follow the total order introduced by e: there can be no cycle consisting of write-write/write-read dependencies. $\neg$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 $\text{COMPLETE}_{e,T}(s) \wedge (\Delta(s,s_p) \cap \mathcal{W}_T = \emptyset)$. It follows that T_{rs} (and all the transactions that precede it in e) start-precede T. Proving $\neg$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 read from a state that precedes T in the execution e. We prove:

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

[4] As proofs follow a similar structure, we defer all subsequent proofs to our technical report [22]

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:

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:

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

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 :

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

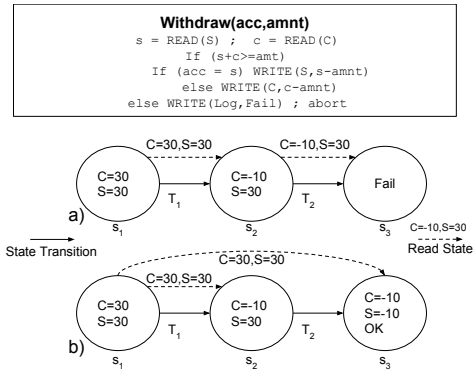

Figure 3: Simple Banking Application. Alice and Bob share checking and savings accounts. Withdrawals are allowed as long as the sum of both account is greater than zero.

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:

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 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

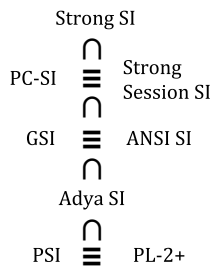

Figure 4: Snapshot-based isolation guarantees hierarchy. (ANSI SI [15, 24], Adya SI [2], Strong SI [24], GSI [48], PSI [53], Strong Session SI [24], PL-2+ [3], PC-SI [24]).

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 our extended technical report [22], 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:

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:

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

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

Table 2: Commit tests for snapshot-based protocols

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 $(T_{se}, \xrightarrow{se})$ 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:

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:

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 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

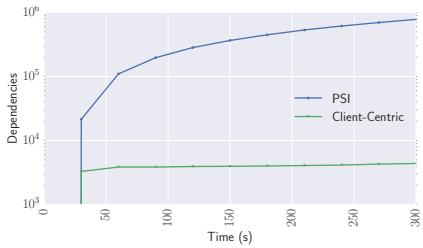

Figure 5: Number of dependencies per transaction as a function of time. TARDiS [23] runs with three replicas on a shared local cluster (2.67GHz Intel Xeon CPU X5650, 48GB memory and 2Gbps network).

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 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.

[5] as of June 2017

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] Crooks, N., Pu, Y., Alvisi, L., and Clement, A. Seeing is believing: A client-centric specification of database isolation. Tech. Rep. TR-17-06, Computer Science Department, UT Austin, May 2017.

[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.