Simplifying Snapshot Isolation: A New Definition, Equivalence, and Efficient Checking

Jian Zhang and Cheng Tan
Northeastern University

Abstract

Snapshot Isolation (SI) is a popular isolation level, supported by many databases and is widely used by applications. Understanding and checking SI is essential. However, today's SI definitions can be obscure for non-experts to understand, or inefficient to verify, or dependent on specific implementations. In contrast, our goal is to offer a definition that is straightforward and easy to comprehend, enables efficient verification, and remains independent of the underlying implementation. In this paper, we introduce such an SI definition using a data structure called *BC-graphs*. We prove that our SI definition is equivalent to Adya SI [7], the de facto SI definition. We did an empirical study to show that our SI definition accelerates SI checking significantly compared to checking Adya SI.

CCS Concepts: • **General and reference** → **Verification**; • **Information systems** → **Database transaction processing**.

Keywords: Databases, Snapshot Isolation, Verification

ACM Reference Format:
Jian Zhang and Cheng Tan. 2024. Simplifying Snapshot Isolation: A New Definition, Equivalence, and Efficient Checking . In *Principles and Practice of Consistency for Distributed Data (PaPoC '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3642976.3653032

1 Introduction

Snapshot isolation (SI) is a widely used isolation level. Many of today's commercial databases, including Oracle, TiDB [22], MongoDB [27], SQL Server [4], and YugabyteDB [6], support snapshot isolation. How to implement SI has been well-studied [10, 20, 22, 25]: people propose many efficient concurrency protocols to implement SI and its variants.

However, limited efforts have been spent on studying and communicating the intuition behind SI's specifications. As a consequence, SI's correctness guarantees are difficult for ordinary database users (isolation-level non-experts) to comprehend. We argue this is due to the complicated SI definitions we use today [7, 9, 14, 16]. On the contrary, serializability's definition [11, 26] is (a) *easy to comprehend*: if a set of transactions (called a *history*) is serializable, then the history is equivalent to sequentially executing these transactions on a single-copy database; and (b) *easy to check*: a checker constructs a serialization graph in which nodes are committed transactions and edges are happen-before relationships between conflicting transactions. The history is serializable iff the serialization graph is acyclic.

Unlike serializability, the SI definition (§3.3) is less intuitive and harder to check (§5). The de facto SI [7] is defined on a graph called *Start-ordered Serialization Graph* (Definition 3, §3.3), similar to the serialization graph. But, instead of requiring acyclic graphs, SI allows certain types of cycles and disallows others: in a nutshell, there are four types of edges in start-ordered serialization graphs; and SI allows the cycles containing two (or more) consecutive edges of a specific type.

In this paper, we introduce a new SI definition that is easier to interpret and is algorithmic complexity-wise cheaper to check. The definition has a neat parallel to serializability (§2); this helps people who know serializability understand. Based on our definition, the SI checking algorithm (i.e., checking if a given history is SI) is simple and runs faster than the current algorithm that uses today's SI definition.

The proposed SI definition is based on a new type of graph called *BC-graphs*. We will elaborate BC-graphs in section 2 and formally define them in Definition 6 (§3.4). An analogy is that BC-graphs are to SI as serialization graphs are to serializability. Then, our SI definition is as follows: if the BC-graph of a history is acyclic, then the history is SI. This SI definition is equivalent to the state-of-the-art SI definition, which we prove in Theorem 8 (§4). With this new definition, the checking algorithm is straightforward: given a history h, construct h's BC-graph g and check if g is acyclic.

To evaluate the checking efficiency, we conduct an empirical study to compare our SI checking algorithm with a baseline checking today's SI. Results show that our algorithm outperforms the baseline by 114× on a mid-sized history (90s execution of Jepsen testing framework [2]).

The major technical contributions are borrowed from our prior paper [35]. The purpose of this paper is to explain and argue for a better SI definition. We strongly believe that *understanding and checking SI is essential and urgent* because many systems use SI [4, 6, 22], and misusing isolation levels has severe consequences [1, 33]. Application developers (who are also database users) must understand the guarantees provided by SI so that they can use SI (hence many today's databases)

PaPoC '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0544-1/24/04
https://doi.org/10.1145/3642976.3653032

correctly. We hope this new SI definition can clarify and disambiguate SI's specifications and further help developers reduce the incorrect usage of SI.

2 A new Snapshot Isolation definition

This section introduces BC-graphs, describes our SI definition, and provides an intuitive interpretation of SI. For simplicity, we describe these less formally. Rigorously defining them requires a series of prerequisite terms and notions. We describe prerequisites and formally define SI in section 3.

In this paper, all operations are wrapped in transactions. Transactions start with a begin, come with a sequence of reads and writes, and ends with a commit or an abort.

BC-graph. BC-graphs are directed graphs; BC-graph's nodes are begin/commit operations of committed transactions; BC-graph's edges are happen-before relationships between begin/-commit operations. Edges come from five cases in a history. We describe them below.

Consider two transactions T_i and T_j. B_i and C_i represent the begin and commit operations, respectively (similar for T_j). x is a key in the database. For any two transactions, a BC-graph has the following edges:

1. T_i must begin then commit: $B_i \rightarrow C_i$.
2. If T_j reads a value of x written by T_i, then T_i should commit before T_j begins: $C_i \rightarrow B_j$.
3. If T_j overwrites a value of x written by T_i, then T_i should commit before T_j begins: $C_i \rightarrow B_j$.
4. If T_j writes a value of x after T_i reads its old value, then T_i should begin before T_j commits: $B_i \rightarrow C_j$.

Time-precedes order. SI definition also requires a *time-precedes order* $\prec_t$ [7, §4.3.1] that represents a partial order of begin and commit events (this can be used for dictating wall-clock time ordering). We use b_i and c_i to represent T_i's begin and commit events in the history (as opposed to B_i, C_i, which are nodes in the BC-graph).

5. T_i's commit time-precedes T_j's begin ($c_i \prec_t b_j$): $C_i \rightarrow B_j$.

SI definition. As stated in section 1, our SI definition is simple: *given a history h, if the BC-graph of h is acyclic, then h is SI.*

An intuitive interpretation of an SI history. Given our SI definition, if a history h is SI, there is a total order of transaction begins and commits, say $\hat{s}$. (One can get $\hat{s}$ by topological sorting the BC-graph of h.) Assume all reads in a transaction T_i happen at the moment when T_i begins and all writes happen at the moment when T_i commits. Then, $\hat{s}$ extends to a serial order of all reads and writes in h. Crucially, this serial order of reads/writes has two properties.

First, sequentially executing these reads and writes produces the same results in h (i.e., returning the same values for reads and resulting in the same final database state). For example, below is a well-known history, called write skew [21], which is SI but not serializability. The history has two transactions (we omit some obvious time-precedes order, like $b_1 \prec_t c_1$; x and y have an initial value of 0):

$$T_1 : b_1, r_1(x) \rightarrow 0, r_1(y) \rightarrow 0, w_1(x, 1), c_1$$
$$T_2 : b_2, r_2(x) \rightarrow 0, r_2(y) \rightarrow 0, w_2(y, 2), c_2$$
$$(\prec_t:\ b_1 \prec_t c_2,\ b_2 \prec_t c_1)$$

By our definition, if this history is SI, there exists a serial order of begins (delegating reads) and commits (delegating writes), such that sequentially executing reads and writes produces the same results for reads and ends up with the same final state:

$$[x = 0, y = 0]\ \underbrace{r_1(x), r_1(y)}_{b_1}, \underbrace{r_2(x), r_2(y)}_{b_2}, \underbrace{w_1(x, 1)}_{c_1}, \underbrace{w_2(y, 2)}_{c_2}$$

Second, the serial order of reads/writes generated from $\hat{s}$ guarantees that no conflicting writes are concurrent. This can be proved by contradiction. Assume $\hat{s}$ has two concurrent transactions T_i and T_j. To be concurrent, neither T_i nor T_j commits before the other begins; for the total order $\hat{s}$, that is $c_i \not\prec_{\hat{s}} b_j$ and $c_j \not\prec_{\hat{s}} b_i$. However, two conflicting writes have a "type-3 edge" above—one write happens before the other—indicating $c_i \prec_{\hat{s}} b_j$ or $c_j \prec_{\hat{s}} b_i$, a contradiction.

A note on "self-reads". The reads previously discussed assume reading from other transactions. When a transaction reads from its own writes—writing to x and then reading from x—we call these *self-reads*. Self-reads are simpler because they do not introduce dependencies between transactions. Our intuitive interpretation remains applicable: it suffices to posit such self-reads immediately after the corresponding writes in the serial order of reads/writes (instead of with other reads).

A neat parallel to serializability. Our SI definition has a nice parallel to the canonical serializability definition. Below we put serializability and SI together to highlight their connections:

- A history is serializability iff its serialization graph is acyclic; a history is SI iff its BC-graph is acyclic.
- Serialization graph and BC-graph are both simple directed graphs (no type is associated with edges).
- In serializability, a serialization graph include transactions as nodes and dependencies of transactions as edges; in SI, a BC-graph include begin/commit operations as nodes and dependencies of begin/commit operations as edges.
- A serializable history is equivalent to some sequential execution of transactions; an SI history is equivalent to some sequential execution of begins (delegating reads) and commits (delegating writes).

3 Snapshot Isolation

3.1 Prerequisite

In this section, we define the setup and common notations. A *database* stores a set of *data objects*, $\{x, y, z, \dots\}$. *Clients* issue *transactions* to access these objects concurrently. A transaction T_i is an ordered set of *operations* on data objects. An

operation is a basic unit that can be executed by the database. In our setup, an operation can be a *read* or a *write* on a data object. A read in a transaction T_i is written as $r_i(x_j)$, which means T_i reads from a version of x written by transaction T_j, A write in the transaction T_i is written as $w_i(x_i)$, where the subscript i of w_i represents the write operation belongs to T_i, and the subscript of x_i represents which version of x value is. Formally, a transaction is defined as follows.

Definition 1 (Transaction)**.** A transaction T is a 2-tuple (M, TO). M is a set of read/write operations on the objects stored in a database. TO is a total order of the operations in M.

Each transaction is associated with a begin and a commit event. We use b_i and c_i to denote the begin and commit event of T_i. For simplicity, we assume each transaction only writes an object once, a common assumption used by prior work [30, 35]. A transaction has two possible states: committed or aborted. A *history* summarizes a total order on committed object versions (called *version order*, denoted as $\ll$) over all the executed transactions.

Definition 2 (History)**.** A history h is a set of transactions with a total committed version order ($\ll$) for all the data objects.

3.2 SI definitions

In 1995, Berenson et al. [9] critiqued ANSI SQL isolation levels' ambiguities and introduced snapshot isolation as a new isolation level. We refer to this SI definition as *SI'95*. SI'95 is defined on action rules, which focuses on the implementation mechanisms. This definition is good for implementing SI but is challenging for checking if a history is SI.

In 1999, Adya proposed a graph-based definition [7, 8]. Adya SI is defined as proscribing certain cycles in graphs (§3.3). To check Adya SI, a checker needs to find all cycles in a graph; for each cycle, the checker examines if the cycle possesses the forbidden patterns. Meanwhile, people proposed various SI variants to trade off consistency and performance, including Generalized SI [20], Prefix-Consistent SI [20], Strong SI [19] and Strong session SI [19]. Later, in 2016, Cerone et al. [13] proposed an axiom-based SI definition and proved that it is equivalent to Adya SI.

In 2017, Crooks et al. [16, 18] proposed a state-based definition of SI, which is the first work to define SI based on client-observable states. They also analyzed the existing SI definitions, and proved that some SI variants are equivalent from the perspective of clients and summarized a hierarchy of these SI variants [18, Figure 4]:

$$\begin{aligned} \textit{Strong SI} \subset (&\textit{Prefix-Consistent SI} \equiv \textit{Strong Session SI}) \\ \subset (&\textit{Generalized SI} \equiv \textit{ANSI SI}) \\ \subset &\textit{Adya SI} \end{aligned}$$

where $SI_A \equiv SI_B$ indicates SI_A and SI_B are equivalent; $SI_A \subset SI_B$ means that SI_A is stricter than SI_B, and SI_B can accept all the histories accepted by SI_A.

3.3 Adya SI

Adya SI is widely used and is regarded as the state-of-the-art SI definition [3]. This section formally defines Adya SI.

To capture the ordering of begin and commit events, SI includes the *time-precedes order* ($\prec_t$) over the begin/commit events, which is a partial order specifies $b_i \prec_t c_i$ for any transaction T_i, and the ordering between b_i and c_j for any pair transactions $T_i, T_j, i \neq j$. Each key in the data store has a version order which specifies in which order the different versions of values are installed. In Adya's definition, SI [7] has a natural interpretation:

- Snapshot read: $\forall\ r_i(x_j)$ in a history h, $c_j \prec_t b_i$; and $\forall\ w_k$ in $h(j \neq k)$, either $b_i \prec_t c_k$ or $c_k \prec_t b_i \wedge x_k \ll x_j$, where $\ll$ means x_k is installed before x_j in the version order.
- Snapshot write: if T_i and T_j are concurrent and they modify the same object, then either $c_i \prec_t b_j$ and $c_j \prec_t b_i$.

Adya proposed a graph-based specification and proved its equivalence to these two rules. Adya SI defines four dependencies between two transactions T_i and T_j that correspond to four types of edges:

- *direct read-dependency*: T_j directly read-depends on T_i if T_j reads from the value that is written by T_i.
- *direct write-dependency*: T_j directly write-depends on T_i if T_j writes the next version of value after T_i.
- *direct anti-dependency*: T_j anti-depends on T_i if T_i reads a version of value and T_j updates it to be the next version.
- *start-dependency*: T_j start-depends on T_i if T_i commits before T_j according to the time-precedes order.

A *start-ordered serialization graph* (SSG) is defined given a set of transactions and all the dependencies above:

Definition 3 (Start-ordered Serialization Graph)**.** Given a history h, its start-ordered serialization graph SSG, represented as $SSG(h)$, is defined as a directed graph. Each node represents a committed transaction in h. Each edge represents a type of dependency between two transactions. Edges $T_i \xrightarrow{\text{wr}} T_j$, $T_i \xrightarrow{\text{ww}} T_j$, $T_i \xrightarrow{\text{rw}} T_j$, $T_i \xrightarrow{\text{start}} T_j$ correspond to the read-dependency, write-dependency, anti-dependency, and start-dependency, respectively.

Next, we define SI based on SSG.

Definition 4 (Adya SI)**.** Given a history h together with all the dependencies and a time-precedes order $\prec_t$, h is SI if $SSG(h)$ proscribes G1, G-SIa and G-SIb, where G1, G-SIa and G-SIb are defined as:

1. *G1* includes three rules:
   - *G1a (Aborted Read)*: read from an aborted transaction;
   - *G1b (Intermediate Read)*: T_i reads a version of x from T_j which is not the final write of x by T_j;
   - *G1c (Circular Information Flow)*: SSG(h) contains cycles with only read/write/start dependency edges.

2. *G-SIa*: SSG(h) contains a read/write-dependency edge $T_i \to T_j$ without a start-dependency edge $T_i \to T_j$.
3. *G-SIb*: SSG(h) contains a directed cycle with exactly one anti-dependency edge.

Implicit assumptions. Adya [7] has made two implicit assumptions. First, a read will read from the most recent write with respect to the time-precedes order $\prec_t$. That is, if T_j reads x from T_i, then (a) T_i happens before T_j (i.e., $c_i \prec_t b_j$), and (b) no such a transaction T_k writing to x happens "in-between" them (i.e., $\nexists b_k, c_k : \ c_i \prec_t b_k \wedge c_k \prec_t b_j$). Second, if a transaction T_j overwrite a value written by T_i, then T_i committed before T_j begin (i.e., $c_i \prec_t b_j$). These implicit assumptions are used in Adya's proof; we also use them in ours.

Corollary 5 (Dependencies and time-precedes order)**.** By the implicit assumptions, we conclude that the time-precedes order does not conflict with the dependencies:

$$T_j \text{ read/write/start-depends on } T_i \Rightarrow c_i \prec_t b_j$$
$$\text{and} \quad T_j \text{ anti-depends on } T_i \Rightarrow b_i \prec_t c_j$$

3.4 Our definition of snapshot isolation

Adya SI suffers from the intrinsic complexity of finding and checking the types of cycles. We mitigate the problem by defining BC-graph.

Definition 6 (BC-graph)**.** Given a history h together with its dependencies and a time-precedes order $\prec_t$ over begin/commit events, a BC-graph is constructed as follows:

1. if any transaction has aborted/intermediate reads (G1a/G1b), the h is invalid for BC-graph and the construction stops.
2. create two nodes (B_i and C_i) for each transaction (T_i).
3. create edges as follows:
   - *intra-txn edges*: for each committed transaction T_i in h, add an edge $B_i \to C_i$.
   - *read-dependency and write-dependency edges*: for each read-dependency and write-dependency $T_i \to T_j$ in h, create an edge from $C_i \to B_j$.
   - *anti-dependency edges*: for each anti-dependency $T_i \to T_j$, create an edge $B_i \to C_j$.
   - *start-dependency edges*: for all $c_i \prec_t b_j$, create $C_i \to B_j$.

Read-dependency, write-dependency, anti-dependency and start-dependency edges are also called *inter-txn edges*. Next, we define SI based on the acyclic property of a BC-graph:

Definition 7 (Our SI)**.** Given a history h and write-dependencies, read-dependencies, anti-dependencies, and a time-precedes order $\prec_t$, h is SI if h's BC-graph exists and is acyclic.

Runtime complexity of checking SI. Our SI definition converts the problem of determining whether a history is SI into a problem of checking if the history's BC-graph has cycles, which can be easily solved in a linear time. Specifically, two classic algorithms of cycle detection are back-edge-based depth-first search (DFS) and topological sort, both of which have a total complexity of $O(|V| + |E|)$. ($|V|$ and $|E|$ represent the number of nodes and edges in the BC-graph, respectively.) Adya SI definition requires determining if the cycle has zero or one anti-dependency edges for all the cycles in the BC-graph. So it needs to find all the cycles in the dependency graph first, and then iterate over all the edges of each cycle to check if it violates the condition. A naive implementation is to use DFS to find all the cycles, which costs exponential time. Some more advanced algorithms for finding all the cycles cost linear time: Szwarcfiter and Lauer algorithm [29] costs $O(V + EC)$, Tarjan algorithm [31, 32] costs $O(VEC)$, and Johnson algorithm [23] costs $O((V + E)C)$ where C is the number of simple cycles. However, the constant C varies and can be exponential if the graph is dense. By dense, we mean the number of edges is close to the maximal number of edges. Our definition helps stabilize the performance and accelerates determining whether a history is SI by removing the dependency on C in theory.

4 Equivalence to Adya SI

Next, we prove that our SI definition is equivalent to Adya SI.

Theorem 8. *Given a history h together with all the dependencies and a time-precedes order $\prec_t$ of begin/commit events, h's BC-graph exists and is acyclic $\Leftrightarrow$ h is Adya SI.*

Before proving the main Theorem 8, we need two helper lemmas to build the connection between start-ordered serialization graphs (SSGs) and BC-graphs.

Lemma 9. *Given a history h together with all the dependencies, for any path $p = T_1 \to T_2 \rightsquigarrow T_n$ in SSG(h) with non-consecutive anti-dependency edges, $b_1 \prec_t c_n$.*

Proof. We show this by induction. The Base step proves that for zero or one anti-dependency edge, the claim ($b_1 \prec_t c_n$) holds. The induction step proves that for a path having k non-consecutive anti-dependency edges, the claim holds.

Base step: (1) for a path p that has zero anti-dependency edges, according to the Corollary 5, $c_1 \prec_t b_2$; plus a transaction's begin always time-precedes $\prec_t$ its commit, we get $b_1 \prec_t c_1 \prec_t b_2 \prec_t c_2$. By repeating this step, we prove the claim $b_1 \prec_t c_n$. (2) for a path p that has one anti-dependency edge, say $T_i \xrightarrow{\text{rw}} T_{i+1}$, then by Corollary 5, we know $b_i \prec_t c_{i+1}$. Because $T_1 \rightsquigarrow T_i$ and $T_{i+1} \rightsquigarrow T_n$ have zero anti-dependency edge and the above point (1), we have $b_1 \prec_t \cdots \prec_t b_i$ and $c_{i+1} \prec_t \cdots \prec_t c_n$. Combined with $b_i \prec_t c_{i+1}$, we prove the claim $b_1 \prec_t c_n$.

Inductive step: Assume that $b_1 \prec_t c_n$ holds for any path p with $\leq k$ non-consecutive anti-dependency edges. Consider a path with $k + 1$ non-consecutive anti-dependency edges and assume $T_i \to T_{i+1}$ is the first anti-dependency edge in p. We see the path as three pieces: (a) because $T_1 \rightsquigarrow T_{i+1}$ has one anti-dependency edge, according to the base step, $b_1 \prec_t c_{i+1}$. (b) because of the non-consecutive anti-dependency edges

constraint, $T_{i+1} \rightarrow T_{i+2}$ must be a non-anti-dependency edge hence $c_{i+1} \prec_t b_{i+2}$. (c) finally, the path $T_{i+2} \rightsquigarrow T_n$ has k non-consecutive anti-dependency edges. By induction hypothesis, $b_{i+2} \prec_t c_n$. Combine the above (a)–(c) and we get $b_1 \prec_t c_{i+1} \prec_t b_{i+2} \prec_t c_n$. Therefore, $b_1 \prec_t c_n$ holds for the $k+1$ case. □

Lemma 10. *Given a history h together with all the dependencies, and a time-precedes order $\prec_t$ of begin/commit events, SSG(h) has a cycle with non-consecutive anti-dependency edges ⇔ h's BC-graph g has a cycle.*

Proof. Note that $SSG(h)$ and BC-graph g has a one-to-one mapping per Definition 3 and Definition 6: (1) for each node T_i in $SSG(h)$, there are two nodes B_i and C_i in BC-graph; and (2) for each inter-txn edge in g, $SSG(h)$ have an edge of the same type because the edges in both graphs are built according to the same dependencies and time-precedes order.

"⇐" Assume BC-graph has a cycle C. C may have two types of edges: intra-txn edges and inter-txn edges. Note that $SSG(h)$ contains all the inter-txn edges of C. $\forall C_i \rightarrow B_j$ or $B_i \rightarrow C_j$ in C, find the edge $T_i \rightarrow T_j$ in $SSG(h)$. All the found inter-txn edges form a cycle C' in $SSG(h)$.

Next, we prove that C' does not have consecutive anti-dependency edges; that is, for any $T_i \xrightarrow{\text{rw}} T_j$ in C', the immediate precedent and immediate succedent edges must not be anti-dependency ($\xrightarrow{\text{rw}}$). By Definition 6, all edges in BC-graph must contain both begin and commit. Therefore, any path in BC-graph g, including the cycle C, has commit vertex and begin vertex appear alternately. Now, consider the "one-to-one mapped" edge regarding $T_i \xrightarrow{\text{rw}} T_j$ in g: $B_i \xrightarrow{\text{rw}} C_j$. In cycle C, its immediate precedent edge must be $C_{prev} \rightarrow B_i$; its immediate succedent edge must be $C_j \rightarrow B_{succ}$. Neither is an anti-dependency edge because anti-dependency requires pointing from B to C. Thus, for the cycle C' in $SSG(h)$, the corresponding precedent and succedent edges $T_{prev} \rightarrow T_i$ and $T_j \rightarrow T_{succ}$ are not anti-dependency edges.

"⇒" Assume $SSG(h)$ has a cycle C' which has no consecutive anti-dependency edges: $T_1 \rightarrow T_2 \rightsquigarrow T_n \rightarrow T_1$.

Case 1: C' has no anti-dependency edges, and all the edges are one of the other three edges (i.e., write-dependency, read-dependency, and start-dependency). By the one-to-one edge mapping and Definition 6, for each read-dependency, write-dependency and start-dependency edge $T_i \rightarrow T_j$, there is an edge $C_i \rightarrow B_j$ in g. Hence, there is a cycle $(B_1 \rightarrow C_1) \rightarrow (B_2 \rightarrow C_2) \rightsquigarrow (B_n \rightarrow C_n) \rightarrow B_1$ in g.

Case 2: C' has exactly one anti-dependency edge. Assume $T_i \rightarrow T_j$ is the only anti-dependency, then g has an edge $B_i \rightarrow C_j$. For the remaining path $T_j \rightsquigarrow T_i$ in the cycle C', all the edges are non-anti-dependency, hence all pointing from B to C. Therefore, we have $C_j \rightarrow (B_{j+1} \rightarrow C_{j+1}) \rightsquigarrow B_i$, which forms a cycle in g together with the edge $B_i \rightarrow C_j$.

Case 3: C' has multiple anti-dependency edges but neither two of them are consecutive. We can apply the argument of Case 2 to each anti-dependency edge and get a cycle in g. □

Finally, we prove Theorem 8.

Proof. "⇒" By construction in Definition 6, BC-graph (h) exists naturally proscribes G1a and G1b. By Lemma 10, an acyclic BC-graph g implies that $SSG(h)$ does not have cycles with zero or one anti-dependency edges, which means $SSG(h)$ proscribes G1c and G-SIb in Definition 4. Further, Corollary 5 implies that for any read/write dependency $T_i \rightarrow T_j$, there is $c_i \prec_t b_j$, then by construction, a start-dependency edge $C_i \rightarrow B_j$ exists in g. Therefore, $SSG(h)$ has a corresponding edge, hence G-SIa is proscribed. By Definition 4, the history h is Adya SI.

"⇐" Given h is Adya SI, h proscribes G1a and G1b ⇒ BC-graph (h) exists by construction.

Next, we only need to prove BC-graph (h) is acyclic. h is SI ⇒ $SSG(h)$ does not have cycles with zero, one anti-dependency edges. If we can further prove that Adya SI disallows cycles with non-consecutive anti-dependency edges, then by Lemma 10, g is acyclic and the claim holds. We prove this by contradiction. Consider a cycle $T_1 \rightsquigarrow T_n \rightarrow T_1$ containing non-consecutive anti-dependency edges. Pick one non-anti-dependency edge, say $T_i \rightarrow T_j$. By Corollary 5, $c_i \prec_t b_j$. On the contrary, the rest of the cycle $T_j \rightsquigarrow T_n \rightarrow T_1 \rightsquigarrow T_i$ is a path with non-consecutive anti-dependency edges; by Lemma 9, $b_j \prec_t c_i$, which is a contradiction. □

5 Empirical study of SI checking performance

SI-checking algorithm implementations. For our checking algorithm, we use a depth-first search algorithm to detect cycles in BC-graph. If BC-graph is acyclic, then our algorithm accepts the history as SI. Otherwise, it rejects.

For Adya's SI checking algorithm (which we call baseline), we use Kosaraju's algorithm [28], a linear algorithm, to identify all the strongly connected components (SCCs). We then search cycles in each SCC using depth-first search. If finding any cycle disallowed by Adya SI (i.e., cycles with non-consecutive anti-dependency edges), the baseline rejects the history. If no such cycle is found, it accepts. Both checking algorithms are implemented in Java.

Experiment setup. We use Jepsen [2], a consistency testing framework, to run an append benchmark (described below) on TiDB [5] (configured to be SI). In our setup, 24 concurrent clients keep issuing transactions to the database, and we collect their responses to form a history. TiDB runs on a Ubuntu 20.04 machine from Google Cloud, with 16-core vCPUs (3.10GHz Intel Xeon) and 64GB RAM. We run the checking algorithms on a machine with a 12-core processor (AMD Ryzen 9 5900, 3.0GHz) and 64GB RAM. The OS is Ubuntu 20.04.

Benchmark. We use the append benchmark from Jepsen. There are two types of operations, append operations and read operations. An append operation appends a value to a keyed list of integers. A read operation reads the integer list of a key, by which the checking algorithms know the write order of

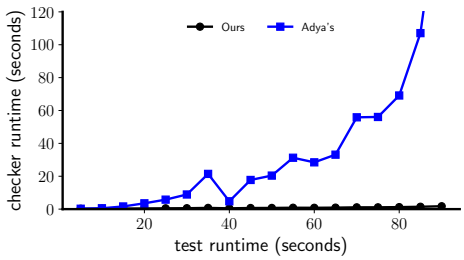

Figure 1. The checking algorithm using our SI definition outperforms the baseline based on Adya's specification [7].

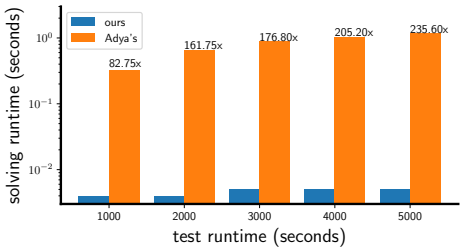

Figure 2. Our checking algorithm outperforms the baseline plus the optimization of start-dependency edges. Notice that the y-axis is the solving time (instead of the checker runtime) in log-scale.

appends. In the construction of graphs, it concatenates consecutive appends by creating write-dependency edges. Each transaction consists of one to eight operations. Each key is written at most 20 times.

BC-graph checking outperforms the baseline. Figure 1 shows the results of our experiments. The x-axis is the history generating time (i.e., how long the benchmark runs). It controls the size of the history: the rate of issuing transactions is approximately stable so that the sizes of histories grow approximately linearly in time. From Figure 1, the checking algorithm based on our SI definition outperforms that based on Adya's specification [7] consistently. Our checking algorithm runs faster for several reasons. First, it only needs to detect whether BC-graph has cycles, but does not need to find all the cycles or check whether the cycles are forbidden by SI. Second, it costs exponential time to find all the cycles in *SSG* using DFS, especially when the *SSG* is dense. This is reflected in Figure 1 that the baseline grows exponentially. Third, the *SSG*s, in our experiment, have a large amount of start-dependency edges makes the performance gap significant. The reason why the 40-second history takes less time is that it has fewer dependency edges, mainly start-dependencies, due to the randomness in the append workload. The runtime of our checking algorithm is stable and grows linearly. This is expected because the back-edge-based DFS cycle detection algorithm costs linear time.

Comparing with the baseline plus an optimization. People can apply an optimization to significantly reduce the number of start-dependency edges. That is, instead of adding start-dependency edges for each pair of transactions that has a start-dependency relation, one can only add start-dependency edges for consecutive transactions. For example, if $T_1 \xrightarrow{\text{start}} T_2 \xrightarrow{\text{start}} T_3$, we do not include the edge $T_1 \xrightarrow{\text{start}} T_3$.

Figure 2 shows the *solving time*, not the checker runtime, after applying this optimization. We only show the solving time because the time of building graphs dominates the end-to-end runtime for the large histories. For the largest (5000s) history, our checker takes 12.00s to build the graph and finishes in 12.16s; the baseline takes 11.79s to build the graph and finishes in 13.10s. In production, the graph building and history collection can be pipelined, hence is not on the critical path. Therefore, Figure 2 only shows the time of the solving phase, the phase of cycle detections on a graph. From the figure, we can see that our checking algorithm outperforms the baseline consistently even having applied the optimization.

6 Related work

The most related work to our SI definition is Adya's SI definition [7, 8], both of which are based on graphs. The difference is that Adya SI is defined on serialization graphs (where nodes are transactions), whereas our SI definition is based on BC-graphs (in which a node is either a begin or a commit event). We prove that the two definitions are equivalent (§4). Adya et al. [8] choose serialization graphs because the graphs cover other isolation levels, hence a unified framework is preferable, despite complicating the definition of SI. Our SI definition however targets SI only and aims at simplicity and checking performance. It is our future work to study if we can extend BC-graphs to other isolation levels.

Crooks et al. [17, 18] propose the first client-centric isolation level definition, which treat databases as black boxes and do not require any internal information from the databases. Our SI definition is "white-box", as we assume having the knowledge of all internal information, including the time-precedes order ($\prec_t$) and the version order ($\ll$).

There are several other isolation level definitions [12, 24, 34]. In terms of SI, they are equivalent to Adya SI [15], thus are equivalent to ours. These definitions are based on axioms and/or operations (instead of graphs), hence are easier to integrate with program verification and can be used to prove application properties end-to-end. Meanwhile, our SI definition focuses on checking SI of databases, and application logic is out of our scope. Therefore, checking our SI is likely faster than checking their definitions.

7 Conclusion

This paper presents a new definition of Snapshot Isolation (SI) based on BC-graphs. This new definition aligns well with Serializability, making it more accessible. We prove that our definition is equivalent to Adya SI. In addition, our definition introduces a fast SI checking algorithm, which, according to our empirical studies, outperforms an implementation of checking Adya SI.

Acknowledgement

This work was supported by NSF CAREER Awards #2237295.

References

[1] Flexcoin. https://web.archive.org/web/20160408190656/http://www.flexcoin.com/.

[2] Jepsen: Distributed systems safety research. https://jepsen.io/.

[3] Jepsen: Snapshot isolatin. https://jepsen.io/consistency/models/snapshot-isolation.

[4] Snapshot isolation in sql server. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server.

[5] Tidb github repo. https://github.com/pingcap/tidb.

[6] Yugabytedb: Transaction isolation levels. https://docs.yugabyte.com/preview/architecture/transactions/isolation-levels/.

[7] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[8] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering*, 2000.

[9] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD*, May 1995.

[10] J. M. Bernabé-Gisbert and F. D. Munoz-Escoı. A compoundable specification of the snapshot isolation level. Technical report, Technical Report ITI-SIDI-2012/007, Instituto Tecnológico de Informática, 2012.

[11] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *TSE*, SE-5(3), May 1979.

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

[13] A. Cerone and A. Gotsman. Analysing snapshot isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 55–64, 2016.

[14] A. Cerone and A. Gotsman. Analysing snapshot isolation. *Journal of the ACM (JACM)*, 65(2):1–41, 2018.

[15] A. Cerone, A. Gotsman, and H. Yang. Algebraic laws for weak consistency. In *28th International Conference on Concurrency Theory (CONCUR 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[16] N. Crooks. A client-centric approach to transactional datastores. https://repositories.lib.utexas.edu/bitstream/handle/2152/81352/CROOKS-DISSERTATION-2019.pdf?sequence=1&isAllowed=y, 2019.

[17] N. Crooks. *A client-centric approach to transactional datastores*. PhD thesis, The University of Texas at Austin, 2020.

[18] N. Crooks, Y. Pu, L. Alvisi, and A. Clement. Seeing is believing: a client-centric specification of database isolation. In *Proc. PODC*, July 2017.

[19] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *Proc. ICDE*, 2004.

[20] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, 2005.

[21] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33(3):12–14, 2004.

[22] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[23] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[24] G. Kaki, K. Nagar, M. Najafzadeh, and S. Jagannathan. Alone together: compositional reasoning and inference for weak isolation. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[25] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. E. Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Transactions on Database Systems (TODS)*, 34(2):1–49, 2009.

[26] C. H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4), Oct. 1979.

[27] W. Schultz, T. Avitabile, and A. Cabral. Tunable consistency in mongodb. *Proceedings of the VLDB Endowment*, 12(12):2071–2081, 2019.

[28] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[29] J. L. Szwarcfiter and P. E. Lauer. A search strategy for the elementary cycles of a directed graph. *BIT Numerical Mathematics*, 16(2):192–204, 1976.

[30] C. Tan, C. Zhao, S. Mu, and M. Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[31] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[32] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[33] T. Warszawski and P. Bailis. ACIDRain: Concurrency-related attacks on database-backed web applications. In *Proc. SIGMOD*, May 2017.

[34] S. Xiong. *Parametric Operational Semantics for Consistency Models*. PhD thesis, Imperial College London, 2019.

[35] J. Zhang, Y. Ji, S. Mu, and C. Tan. Viper: A fast snapshot isolation checker. In *Proc. EuroSys*, May 2023.