Efficient Black-box Checking of Snapshot Isolation in Databases

Kaile Huang
Nanjing University

Si Liu
ETH Zurich

Zhenge Chen, Hengfeng Wei
Nanjing University

David Basin
ETH Zurich

Haixiang Li, Anqun Pan
Tencent Inc.

ABSTRACT

Snapshot isolation (SI) is a prevalent weak isolation level that avoids the performance penalty imposed by serializability and simultaneously prevents various undesired data anomalies. Nevertheless, SI anomalies have recently been found in production cloud databases that claim to provide the SI guarantee. Given the complex and often unavailable internals of such databases, a black-box SI checker is highly desirable.

In this paper we present PolySI, a novel black-box checker that efficiently checks SI and provides understandable counterexamples upon detecting violations. PolySI builds on a novel characterization of SI using generalized polygraphs (GPs), for which we establish its soundness and completeness. PolySI employs an SMT solver and also accelerates SMT solving by utilizing the compact constraint encoding of GPs and domain-specific optimizations for pruning constraints. As demonstrated by our extensive assessment, PolySI successfully reproduces all of 2477 known SI anomalies, detects novel SI violations in three production cloud databases, identifies their causes, outperforms the state-of-the-art black-box checkers under a wide range of workloads, and can scale up to large-sized workloads.

PVLDB Reference Format:
Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, and Haixiang Li, Anqun Pan. Efficient Black-box Checking of Snapshot Isolation in Databases. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/anonymous-hippo/PolySI.

1 INTRODUCTION

Database systems are an essential building block of many software systems and applications. Transactional access to databases simplifies concurrent programming by providing an abstraction for executing concurrent computations on shared data in isolation [6]. The gold-standard isolation level, *serializability* (SER) [35], ensures that all transactions appear to execute serially, one after another. However, providing SER, especially in geo-replicated environments like modern cloud databases, is computationally expensive [2, 33].

Many databases provide weaker guarantees for transactions to balance the trade-off between data consistency and system performance. *Snapshot isolation* (SI) [5] is one of the prevalent weaker isolation levels used in practice, which avoids the performance penalty imposed by SER and simultaneously prevents undesired data anomalies such as fractured reads, causality violations, and lost updates [11]. In addition to classic centralized databases such as Microsoft SQL Server [39] and Oracle Database [18], SI is supported by numerous *production cloud* database systems like Google's Percolator [36], MongoDB [34], TiDB [46], YugabyteDB [50], Galera [13], and Dgraph [20].

Unfortunately, as recently reported in [30, 44, 45], data anomalies have been found in several production cloud databases that claim to provide SI.[1] This raises the question of whether such databases actually deliver the promised SI guarantee in practice. Given that their internals (e.g., source code) are often unavailable to the outsiders or hard to digest, a black-box SI checker is highly desirable.

A natural question then to ask is "What should an ideal black-box SI checker look like?" The SIEGE principle [30] has already provided a strong baseline: an ideal checker would be *sound* (return no false positives), *informative* (report understandable counterexamples), *effective* (detect violations in real-world databases), *general* (compatible with different patterns of transactions), and *efficient* (add modest checking time even for workloads of high concurrency). Additionally, (i) we expect an ideal checker to be *complete*, thus missing no violations; and (ii) we also augment the *generality* criterion by requiring the checker to be compatible not only with general (read-only, write-only, and read-write) transaction workloads but also with standard key-value/SQL APIs. We call this extended principle SIEGE+.

None of the existing SI checkers, to the best of our knowledge, satisfies SIEGE+ (see Section 7 for the detailed comparison). For example, dbcop [7] is incomplete, incurs exponentially increasing overhead under higher concurrency (Section 5.4), and returns no counterexamples upon finding a violation; Elle [30] relies on specific database APIs such as lists and the (internal) timestamps of transactions to infer isolation anomalies, thus not conforming to our black-box setting.

The PolySI Checker. We present PolySI, a novel, black-box SI checker designed to achieve all the SIEGE+ criteria. PolySI builds on three key ideas in response to three major challenges.

First, despite previous attempts to characterize SI [1, 5, 49], its semantics is usually explained in terms of low-level implementation choices invisible to the database outsiders. Consequently, one

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1] These anomalies, which we are also concerned with in this paper, are isolation violations purely in *database engines*. They may be tolerated by end users or higher-level applications, depending on their business logic [22, 48].

must *guess* the dependencies (aka uncertain/unknown dependencies) between client-observable data, for example, which of the two writes was first recorded in the database.

We introduce a novel dependency graph, called *generalized polygraph* (GP), based on which we present a new *sound* and *complete* characterization of SI. There are two main advantages of a GP: (i) it naturally models the guesses by capturing *all* possible dependencies between transactions in a single compacted data structure; and (ii) it enables the acceleration of SMT solving by compacting constraints (see below) as demonstrated by our experiments.

Second, there have been recent advances in SAT and SMT solving for checking *graph properties* such as the MonoSAT solver [4] and its successful application to the black-box checking of SER [42]. The idea is to *search* for an acyclic graph where the nodes are transactions in the history[2] and the edges meet certain constraints. We show that SMT techniques can also be applied to build an effective SI checker. This application is nontrivial as a brute-force approach would be inefficient due to the high computational complexity of checking SI [7]: the problem is NP-complete in general and $O(n^c)$ with c (resp. n) a fixed, yet in practice large, number of clients (resp. transactions), even for a single transaction history. In fact, checking SI is known to be asymptomatically more complex than checking SER [7]. In the context of SMT solving over graphs, SI leads to much larger search space due to its specific anomaly patterns [11] while checking SER simply requires finding a cycle.

Thanks to our GP-based characterization of SI, we leverage its compact encoding of constraints on transaction dependencies to accelerate MonoSAT solving. Moreover, we develop domain-specific optimizations that further prune constraints, thereby reducing the search space. For example, PolySI prunes a constraint if an associated uncertain dependency would result in an SI violation with known dependencies.

Finally, although MonoSAT outputs cycles upon detecting a violation, they are still *uninformative* with respect to understanding how the violation actually occurred. Locating the actual causes of violations would facilitate debugging and repairing the defective implementations. For example, if an SI checker were to identify a *lost update* anomaly from the returned counterexample, developers could then focus on investigating the write-write conflict resolution mechanism. Hence, we design and integrate into PolySI a novel interpretation algorithm that explains the counterexamples returned by MonoSAT. More specifically, PolySI (i) recovers the violating scenario by bringing back any potentially involved transactions and dependencies eliminated during pruning and solving and (ii) finalizes the core participants to highlight the violation cause.

Main Contributions. In summary, we provide:

1. a new GP-based characterization of SI that both facilitates the modeling of uncertain transaction dependencies inherent to black-box testing and also enables the acceleration of constraint solving (Section 3);
2. a sound and complete GP-based checking algorithm for SI with domain-specific optimizations for pruning constraints (Section 4);
3. the PolySI tool comprising both our new checking algorithm and the interpretation algorithm for debugging; and
4. an extensive assessment of PolySI that demonstrates its fulfilment of SIEGE+ (Section 5). In particular, PolySI successfully reproduces all of 2477 known SI anomalies, detects novel SI violations in three production cloud databases, identifies their causes, outperforms the state-of-the-art black-box checkers under a wide range of workloads, and can scale up to large-sized workloads.

[2]A history collected from dynamically executing a system records the transactional requests to and responses from the database. See Section 2.2 for its formal definition.

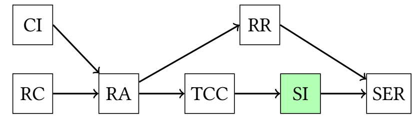


Figure 1: A hierarchy of isolation levels. $A \rightarrow B$: A is strictly weaker than B. CI: cut isolation [2]; RC: read committed [5]; RA: read atomicity [3]; RR: repeatable read [1]; TCC: transactional causal consistency [32]; SI: snapshot isolation [11]; SER: serializability [5].

2 PRELIMINARIES

2.1 Snapshot Isolation in a Nutshell

Snapshot isolation (SI) [5] is one of the most prominent weaker isolation levels that modern (cloud) databases usually provide to avoid the performance penalty imposed by *serializability* (SER). Figure 1 shows a hierarchy of isolation levels where SI sits inbetween *transactional causal consistency* [32] and SER, and is not comparable to *repeatable read* [1].

A transaction with SI always reads from a snapshot that reflects a single commit ordering of transactions and is allowed to commit if no concurrent transaction has updated the data that it intends to write. SI prevents various undesired data anomalies such as fractured reads, causality violations, lost updates, and long fork [5, 11]. The following examples illustrate two kinds of anomalies disallowed by SI. As we will see in Section 5, both anomalies have been detected by our PolySI checker in production cloud databases.

Example 1 (Causality Violation). Alice posts a photo of her birthday party. Bob writes a comment to her post. Later, Carol sees Bob's comment but not Alice's post.

Example 2 (Lost Update). Dan and Emma share a banking account with 10 dollars. Both simultaneously deposit 50 dollars. The resulting balance is 60, instead of 110, as one of the deposits is lost.

In this paper we focus on the prevalent *strong session* variant of SI [11, 19], which additionally requires a transaction to observe all the effects of the preceding transactions in the same *session* [43]. Many production databases, including DGraph [20], Galera [13], and CockroachDB [14], provide this isolation level in practice.

2.2 Snapshot Isolation: Formal Definition

We recall the formalization of SI over dependency graphs, which serves as the theoretical foundation of PolySI. The following account is standard, see for example [11], and Table 1 summarizes the notation used throughout the paper.

Table 1: Notation

Category	Notation	Meaning
KV Store	Key	set of keys
	Val	set of values
	Op	set of operations
Relations	$R?$	reflexive closure of R
	R^+	transitive closure of R
	$R\ ;\ S$	composition of R with S
Dependency	SO, WR, WW, RW	dependency relations/edges
Graph	$G = (V, E, C)$	(generalized) polygraph
	V_G, E_G, C_G	components of G
	$G\|_F$	digraph with set F of edges
Algorithm	$\mathcal{H} = (\mathcal{T}, \mathrm{SO})$	history to check
	I	SI induced graph
	BV	set of Boolean variables
	CL	set of clauses

We consider a distributed key-value store managing a set of keys $\mathrm{Key} = \{\mathrm{x}, \mathrm{y}, \mathrm{z}, \ldots\}$, which are associated with values from a set Val.[3] We denote by Op the set of possible read or write operations on keys: $\mathrm{Op} = \{\mathrm{R}_\iota(\mathrm{x}, \mathrm{v}), \mathrm{W}_\iota(\mathrm{x}, \mathrm{v}) \mid \iota \in \mathrm{OpId}, \mathrm{x} \in \mathrm{Key}, \mathrm{v} \in \mathrm{Val}\}$, where OpId is the set of operation identifiers. We omit operation identifiers when they are unimportant.

2.2.1 Relations, Orderings, Graphs, and Logics. A binary relation R over a given set A is a subset of $A \times A$, i.e., $R \subseteq A \times A$. For $a, b \in A$, we use $(a, b) \in R$ and $a \xrightarrow{R} b$ interchangeably. We use $R?$ and R^+ to denote the reflexive closure and the transitive closure of R, respectively. A relation $R \subseteq A \times A$ is *acyclic* if $R^+ \cap I_A = \emptyset$, where $I_A \triangleq \{(a, a) \mid a \in A\}$ is the identity relation on A. Given two binary relations R and S over set A, we define their composition as $R\ ;\ S = \{(a, c) \mid \exists b \in A : a \xrightarrow{R} b \xrightarrow{S} c\}$. A strict partial order is an irreflexive and transitive relation. A strict total order is a relation that is a strict partial order and total.

For a directed labeled graph $G = (V, E)$, we use V_G and E_G to denote the set of vertices and edges in G, respectively. For a set F of edges, $G|_F$ denotes the directed labeled graph that has the set V of vertices and the set F of edges.

In logical formulas, we write _ for irrelevant parts that are implicitly existentially quantified. We use $\exists!$ to mean "unique existence."

2.2.2 Transactions and Histories.

Definition 3. A ***transaction*** is a pair $(\mathrm{O}, \mathrm{po})$, where $\mathrm{O} \subseteq \mathrm{Op}$ is a finite, non-empty set of operations and $\mathrm{po} \subseteq \mathrm{O} \times \mathrm{O}$ is a strict total order called the ***program order***.

For a transaction T, we let $T \vdash \mathrm{W}(\mathrm{x}, \mathrm{v})$ if T writes to x and the last value written is v, and $T \vdash \mathrm{R}(\mathrm{x}, \mathrm{v})$ if T reads from x before writing to it and v is the value returned by the first such read. We also use $\mathrm{WriteTx}_\mathrm{x} = \{T \mid T \vdash \mathrm{W}(\mathrm{x}, \_)\}$.

Clients interact with the store by issuing transactions during *sessions*. We use a *history* to record the client-visible results of such interactions. For conciseness, we consider only committed transactions in the formalism [11]; see further discussions in Section 4.5.

[3] We discuss how to support SQL queries in Section 6. However, we do not support predicates in this work.

Definition 4. A ***history*** is a pair $\mathcal{H} = (\mathcal{T}, \mathrm{SO})$, where $\mathcal{T}$ is a set of transactions with disjoint sets of operations and the ***session order*** $\mathrm{SO} \subseteq \mathcal{T} \times \mathcal{T}$ is the union of strict total orders on disjoint sets of $\mathcal{T}$, which correspond to transactions in different sessions.

2.2.3 Dependency Graph-based Characterization of SI. A dependency graph extends a history with three relations (or typed edges, in terms of graphs): WR, WW, and RW, representing three possibility of dependencies between transactions in this history [11]. The WR relation associates a transaction that reads some value with the one that writes this value. The WW relation stipulates a strict total order (aka the version order [1]) among the transactions on the same key. The RW relation is derived from WR and WW, relating a transaction that reads some value to the one that overwrites this value, in terms of the version orders specified by the WW relation.

Definition 5. A ***dependency graph*** is a tuple $\mathcal{G} = (\mathcal{T}, \mathrm{SO}, \mathrm{WR}, \mathrm{WW}, \mathrm{RW})$, where $(\mathcal{T}, \mathrm{SO})$ is a history and

(1) $\mathrm{WR} : \mathrm{Key} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that
- $\forall \mathrm{x} \in \mathrm{Key}.\ \forall S \in \mathcal{T}.\ S \vdash \mathrm{R}(\mathrm{x}, \_) \implies \exists! T \in \mathcal{T}.\ T \xrightarrow{\mathrm{WR}(\mathrm{x})} S$
- $\forall \mathrm{x} \in \mathrm{Key}.\ \forall T, S \in \mathcal{T}.\ T \xrightarrow{\mathrm{WR}(x)} S \implies T \neq S \land \exists \mathrm{v} \in \mathrm{Val}.\ T \vdash \mathrm{W}(\mathrm{x}, \mathrm{v}) \land S \vdash \mathrm{R}(\mathrm{x}, \mathrm{v})$.

(2) $\mathrm{WW} : \mathrm{Key} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that for every $\mathrm{x} \in \mathrm{Key}$, $\mathrm{WW}(\mathrm{x})$ is a strict total order on the set $\mathrm{WriteTx}_\mathrm{x}$;

(3) $\mathrm{RW} : \mathrm{Key} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that $\forall T, S \in \mathcal{T}.\ \forall \mathrm{x} \in \mathrm{Key}.\ T \xrightarrow{\mathrm{RW}(\mathrm{x})} S \iff T \neq S \land \exists T' \in \mathcal{T}.\ T' \xrightarrow{\mathrm{WR}(\mathrm{x})} T \land T' \xrightarrow{\mathrm{WW}(\mathrm{x})} S$.

We denote a component of $\mathcal{G}$, such as WW, by $\mathrm{WW}_\mathcal{G}$. We write $T \xrightarrow{\mathrm{WR/WW/RW}} S$ when the key x in $T \xrightarrow{\mathrm{WR(x)/WW(x)/RW(x)}} S$ is irrelevant or the context is clear.

Intuitively, a history satisfies SI if and only if it can be extended to a dependency graph that contains only cycles (if any) with at least two adjacent RW edges. Formally,

Theorem 6 (Dependency Graph-based Characterization of SI (Theorem 4.1 of [11])). *For a history* $\mathcal{H} = (\mathcal{T}, \mathrm{SO})$,

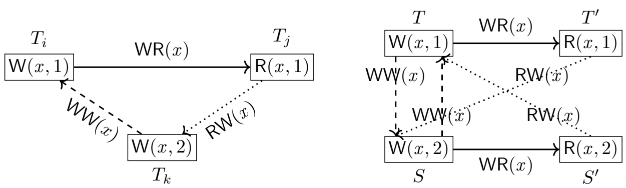
$$\begin{aligned}\mathcal{H} \models \mathrm{SI} \iff\ & \mathcal{H} \models \textsc{Int}\ \land \\ & \exists \mathrm{WR}, \mathrm{WW}, \mathrm{RW}.\ \mathcal{G} = (\mathcal{H}, \mathrm{WR}, \mathrm{WW}, \mathrm{RW})\ \land \\ & (((\mathrm{SO}_\mathcal{G} \cup \mathrm{WR}_\mathcal{G} \cup \mathrm{WW}_\mathcal{G})\ ;\ \mathrm{RW}_\mathcal{G}?)\ \ \textit{is acyclic}).\end{aligned}$$

The *internal consistency axiom* Int ensures that, within a transaction, a read from a key returns the same value as the last write to or read from this key in the transaction.

2.3 The SI Checking Problem

Definition 7. The ***SI checking problem*** is the decision problem of determining whether a given history $\mathcal{H}$ satisfies SI, i.e., is $\mathcal{H} \models \mathrm{SI}$?

We take the common "UniqueValue" assumption on histories [1, 7, 9, 16, 42]: for each key, every write to the key assigns a unique value. For database testing, we can produce such histories by ensuring the uniqueness of the values written on the client side (or workload generator) using, e.g., the client identifier and local counter.



(a) A polygraph **(b) A generalized polygraph**

Figure 2: Examples of polygraphs and generalized polygraphs. WR, WW, and RW relations are represented by solid, dashed, and dotted arrows, respectively.

Under this assumption, each read can be associated with the transaction that issues the corresponding write [11].

Theorem 6 provides a brute-force approach to the SI checking problem: enumerate all possible WW relations and check whether any of them results in a dependency graph that contains only cycles with at least two adjacent RW edges. This approach is, however, prohibitively expensive.

2.4 Polygraphs

A dependency graph extending a history represents *one* possibility of dependencies between transactions in this history. To capture *all* possible dependencies between transactions in a single structure, we rely on polygraphs [35]. Intuitively, a polygraph can be viewed as a family of dependency graphs.

Definition 8. A ***polygraph*** $G = (V, E, C)$ associated with a history $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$ is a directed labeled graph (V, E) called the *known graph*, together with a set C of *constraints* such that

- V corresponds to all transactions in the history $\mathcal{H}$;
- $E = \{(T, S, \mathsf{SO}) \mid T \xrightarrow{\mathsf{SO}} S\} \cup \{(T, S, \mathsf{WR}) \mid T \xrightarrow{\mathsf{WR}} S\}$, where SO and WR, when used as the third component of an edge, are edge labels (i.e., types); and
- $C = \{\langle (T_k, T_i, \mathsf{WW}), (T_j, T_k, \mathsf{RW})\rangle \mid (T_i \xrightarrow{\mathsf{WR(x)}} T_j) \land T_k \in \mathsf{WriteTx_x} \land T_k \neq T_i \land T_k \neq T_j\}$.

As shown in Figure 2a, for a pair of transactions T_i and T_j such that $T_i \xrightarrow{\mathsf{WR(x)}} T_j$ and a transaction T_k that writes x, the constraint $\langle (T_k, T_i, \mathsf{WW}), (T_j, T_k, \mathsf{RW})\rangle$ captures the unknown dependencies that "either T_k happened before T_i or T_k happened after T_j."

3 CHARACTERIZING SI USING GENERALIZED POLYGRAPHS

In this section we introduce generalized polygraphs with generalized constraints and use them to characterize SI. By compacting several constraints together, using generalized constraints leads to a compact encoding which in turn helps accelerate the solving process (see Section 5.4.3).

3.1 Generalized Polygraphs

In a polygraph, a constraint involves only a single pair of transactions related by WR, like T_i and T_j on x in Figure 2a. Thus, several constraints are needed when there are multiple transactions reading the value of x from T_i. To *compact* these constraints, we introduce *generalized polygraphs* with generalized constraints.

Definition 9. A ***generalized polygraph*** $G = (V, E, C)$ associated with a history $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$ is a directed labeled graph (V, E) called the *known graph*, together with a set C of *generalized constraints* such that

- V corresponds to all transactions in the history $\mathcal{H}$;
- $E \subseteq V \times V \times \mathcal{L}$ is a set of edges with labels (i.e., types) from $\mathcal{L} = \{\mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}\}$; and
- $C = \Big\{\Big\langle \mathrm{either} \triangleq \{(T, S, \mathsf{WW})\} \cup \bigcup\limits_{T' \in \mathsf{WR(x)}(T)} \{(T', S, \mathsf{RW})\},$ $\mathrm{or} \triangleq \{(S, T, \mathsf{WW})\} \cup \bigcup\limits_{S' \in \mathsf{WR(x)}(S)} \{(S', T, \mathsf{RW})\}\Big\rangle \mid T \in \mathsf{WriteTx_x} \land S \in \mathsf{WriteTx_x} \land T \neq S\Big\}$.

A generalized constraint is a pair of sets of edges of the form $\langle \mathrm{either}, \mathrm{or}\rangle$. The either part handles the possibility of T being ordered before S via a WW edge. This forces each transaction T' that reads the value of x from T to be ordered before S via an RW edge. Symmetrically, the or part handles the possibility of S being ordered before T via a WW edge. This forces each transaction S' that reads the value of x from S to be ordered before T via an RW edge.

Example 10 (Generalized Polygraphs vs. Polygraphs). In Figure 2b, both transactions T and S write to x, and T' and S' read the values of x from T and S, respectively. The possible dependencies between these transactions can be compactly expressed as a single generalized constraint $\langle\{(T, S, \mathsf{WW}), (T', S, \mathsf{RW})\}, \{(S, T, \mathsf{WW}), (S', T, \mathsf{RW})\}\rangle$, which corresponds to two constraints: $\langle (T, S, \mathsf{WW}), (S', T, \mathsf{RW})\rangle$ and $\langle (S, T, \mathsf{WW}), (T', S, \mathsf{RW})\rangle$.

Note that a generalized polygraph may contain edges of any type in E such that a "pruned" generalized polygraph (Section 4.3) is still a generalized polygraph. For a generalized polygraph $G = (V, E, C)$ and a label $L \in \mathcal{L}$, we use V_G, E_G, C_G, and L_G to denote the set V of vertices, the set E of known edges, the set C of constraints, and the set of known edges with label L in E, respectively. For a generalized polygraph $G = (V, E, C)$ and a set F of edges, we use $G|_F$ to denote the directed labeled graph that has the set V of vertices and the set F of edges. For any two subsets $R, S \subseteq E$, we define their composition as $R \,;\, S = \{(a, c, L_1 \,;\, L_2) \mid \exists b \in V : (a, b, L_1) \in R \land (b, c, L_2) \in S\}$, where $L_1 \,;\, L_2$ is a newly introduced label used when composing edges. In the sequel, we use generalized polygraphs, but sometimes we still refer to them as polygraphs.

3.2 Characterizing SI

According to Theorem 6, we are interested in the *induced* graph of a generalized polygraph G, obtained by composing the edges of G according to the rule $((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?)$.

Definition 11. The ***induced SI graph*** of a polygraph $\mathrm{G} = (V, E, C)$ is the graph $G' = (V, E, C, \mathcal{R})$, where $\mathcal{R} = (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?$ is the induce rule.

The concept of *compatible* graphs gives a meaning to polygraphs and their induced SI graphs. A graph is compatible with a polygraph when it is a resolution of the constraints of the polygraph. Thus, a polygraph corresponds to a family of its compatible graphs.

Definition 12. A directed labeled graph $G' = (V', E')$ is ***compatible with a generalized polygraph*** $G = (V, E, C)$ if

- $V' = V$;
- $E' \supseteq E$; and
- $\forall \langle \text{either}, \text{or} \rangle \in C.\ (\text{either} \subseteq E' \land \text{or} \cap E' = \emptyset) \lor (\text{or} \subseteq E' \land \text{either} \cap E' = \emptyset)$.

By applying the induce rule $\mathcal{R}$ to a compatible graph of a polygraph, we obtain a compatible graph with the induced SI graph of this polygraph.

Definition 13. Let $G' = (V', E')$ be a compatible graph with a polygraph G. Then $G'|_{(\mathrm{SO}_{G'} \cup \mathrm{WR}_{G'} \cup \mathrm{WW}_{G'})\ ;\ \mathrm{RW}_{G'}?}$ is a ***compatible graph with the induced SI graph*** of G.

Example 14 (Compatible Graphs). There are two compatible graphs with the generalized polygraph of Figure 2b: one is with the edge set $\{(T, T', \mathrm{WR}), (S, S', \mathrm{WR}), (T, S, \mathrm{WW}), (T', S, \mathrm{RW})\}$, and the other is with $\{(T, T', \mathrm{WR}), (S, S', \mathrm{WR}), (S, T, \mathrm{WW}), (S', T, \mathrm{RW})\}$.

Accordingly, there are also two compatible graphs with the induced SI graph of the polygraph of Figure 2b: one is with the edge set $\{(T, T', \mathrm{WR}), (S, S', \mathrm{WR}), (T, S, \mathrm{WW}), (T, S, \mathrm{WR}\ ;\ \mathrm{RW})\}$. The edge $(T, S, \mathrm{WR}\ ;\ \mathrm{RW})$ is obtained from $(T, T', \mathrm{WR})\ ;\ (T', S, \mathrm{RW})$. It is identical to (T, S, WW) if the edge types are ignored. The other is with $\{(T, T', \mathrm{WR}), (S, S', \mathrm{WR}), (S, T, \mathrm{WW}), (S, T, \mathrm{WR}\ ;\ \mathrm{RW})\}$. Similarly, $(S, T, \mathrm{WR}\ ;\ \mathrm{RW})$ is identical to (S, T, WW) if the edge types are ignored.

We are concerned with the acyclicity of polygraphs and their induced SI graphs.

Definition 15. An induced SI graph is ***acyclic*** if there exists an acyclic compatible graph with it, *when the edge types are ignored.* A polygraph is ***SI-acyclic*** if its induced SI graph is acyclic.

Finally, we present the generalized polygraph-based characterization of SI. Its proof can be found in [25, Appendix B]. The key lies in the correspondence between compatible graphs of polygraphs and dependency graphs.

Theorem 16 (Generalized Polygraph-based Characterization of SI). *A history $\mathcal{H}$ satisfies* SI *if and only if $\mathcal{H} \models$* Int *and the generalized polygraph of $\mathcal{H}$ is SI-acyclic.*

4 THE CHECKING ALGORITHM FOR SI

Given a history $\mathcal{H}$, PolySI encodes the induced SI graph of the generalized polygraph of $\mathcal{H}$ into an SAT formula and utilizes the MonoSAT solver [4] to test its acyclicity. We choose MonoSAT mainly because, compared to conventional SMT solvers such as Z3, it is more efficient in checking graph properties [4].

The main challenge is that the size (measured as the number of variables and clauses) of the resulting SAT formula may be too large for MonoSAT to solve in reasonable time. Hence, PolySI prunes constraints of the generalized polygraph of $\mathcal{H}$ before encoding. As we will see in Section 5.4, this pruning process is crucial to PolySI's high performance. Additionally, solving is accelerated by utilizing, instead of original polygraphs, generalized polygraphs with compact generalized constraints.

4.1 Overview

The procedure CheckSI (line 1 of Algorithm 1) outlines the checking algorithm. First, if $\mathcal{H}$ does not satisfy the Int axiom, the checking algorithm terminates and returns false (line 2; see Section 4.5 for the predicates AbortedReads and IntermediateReads). The algorithm proceeds otherwise in the following three steps:

- construct the generalized polygraph G of $\mathcal{H}$ (lines 4 and 5);
- prune constraints in the polygraph G (line 6); and
- encode the induced SI graph, denoted I, of G after pruning into an SAT formula (line 8), and call MonoSAT to test whether I is acyclic (line 9).

Algorithm 1 depicts the core procedures of pruning and encoding. The remaining procedures are given in [25, Appendix A].

Before diving into details, we illustrate our algorithm using the example history in Figure 3a, which exemplifies the well-known "long fork" anomaly in SI [11, 41]. Specifically, transaction T_0 writes to both keys x and y. Transactions T_1 and T_2 concurrently write to x and y, respectively. Transaction T_3 sees the write by T_1, but not the write by T_2, while T_4 sees the write by T_2, but not the write by T_1. The session committing T_0 then issues T_5 to update x.

Pruning Constraints. To check whether this history satisfies SI, we must determine the order between T_0, T_1, and T_5 (on x) and the order between T_0 and T_2 (on y). Consider first the constraint on the order between T_0 and T_5 shown in Figure 3b. Due to $T_0 \xrightarrow{\mathrm{SO}} T_5$, the $T_5 \xrightarrow{\mathrm{WW(x)}} T_0$ case would introduce an undesired cycle. Therefore, this case can be safely pruned and the other case of $T_0 \xrightarrow{\mathrm{WW(x)}} T_5$, along with the edge $T_4 \xrightarrow{\mathrm{RW(x)}} T_5$, become known.

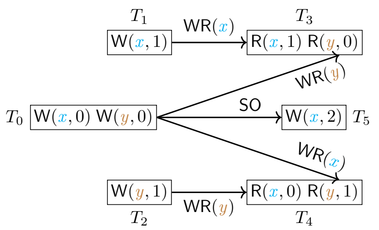
Figure 3c shows the constraint on the order between T_0 and T_1: $\langle \text{either} = \{(T_1, T_0, \mathrm{WW}), (T_3, T_0, \mathrm{RW}\}, \text{or} = \{(T_0, T_1, \mathrm{WW}), (T_4, T_1, \mathrm{RW})\}\rangle$. Consider first the either case. Note that the edge $T_3 \xrightarrow{\mathrm{RW(x)}} T_0$ is in an undesired cycle $T_3 \xrightarrow{\mathrm{RW(x)}} T_0 \xrightarrow{\mathrm{WR(y)}} T_3$, which contains only a single RW edges. Hence, the either case could be safely pruned, without SAT encoding and solving. Conversely, the or case does not introduce undesired cycles. Thus, the edges in the or case become known, before SAT encoding and solving.

Similarly, the either case of the constraint on the order between T_0 and T_2, namely $\langle \text{either} = \{(T_2, T_0, \mathrm{WW}), (T_4, T_0, \mathrm{RW}\}, \text{or} = \{(T_0, T_2, \mathrm{WW}), (T_4, T_2, \mathrm{RW})\}\rangle$, could be safely pruned (not shown in Figure 3d), and the edges in the or case become known.

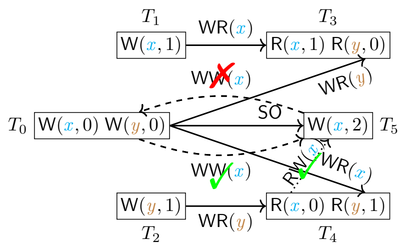
SAT Encoding. The order between T_1 and T_5 is still uncertain after pruning in Figure 3d. We encode the constraint $\langle \text{either} = \{(T_1, T_5, \mathrm{WW}), (T_3, T_5, \mathrm{RW})\}, \text{or} = \{(T_5, T_1, \mathrm{WW})\}\rangle$ on the order as a SAT formula

$$(\mathrm{BV}_{1,5} \land \mathrm{BV}_{3,5} \land \neg \mathrm{BV}_{5,1}) \lor (\mathrm{BV}_{5,1} \land \neg \mathrm{BV}_{1,5} \land \neg \mathrm{BV}_{3,5}),$$

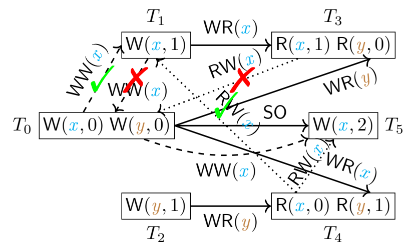
where $\mathrm{BV}_{i,j}$ is a Boolean variable indicating the existence of the edge from T_i to T_j in the pruned polygraph. We then encode the induced SI graph, denoted I. Since $T_2 \xrightarrow{\mathrm{WR(y)}} T_4 \xrightarrow{\mathrm{RW(x)}} T_5$, we have $\mathrm{BV}^{\mathrm{I}}_{2,5} = \mathrm{BV}_{2,4} \land \mathrm{BV}_{4,5}$, where $\mathrm{BV}^{\mathrm{I}}_{i,j}$ is a Boolean variable indicating the existence of the edge from T_i to T_j in I. Similarly, we have $\mathrm{BV}^{\mathrm{I}}_{1,2} = \mathrm{BV}_{1,3} \land \mathrm{BV}_{3,2}$ and $\mathrm{BV}^{\mathrm{I}}_{2,1} = \mathrm{BV}_{2,4} \land \mathrm{BV}_{4,1}$. In



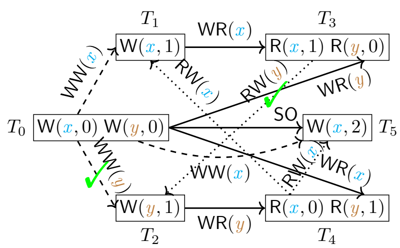
(a) A "long fork" history with SO and WR edges. T_0 and T_5 are on the same session.



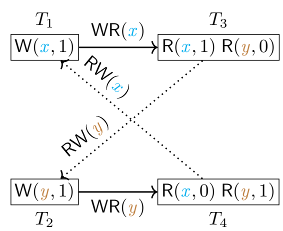
(b) The $T_5 \xrightarrow{\text{WW(x)}} T_0$ case is pruned due to the cycle $T_0 \xrightarrow{\text{SO}} T_5 \xrightarrow{\text{WR(x)}} T_0$.



(c) The $T_1 \xrightarrow{\text{WW(x)}} T_0$ case is pruned due to the cycle $T_3 \xrightarrow{\text{RW(x)}} T_0 \xrightarrow{\text{WR(y)}} T_3$.



(d) The $T_2 \xrightarrow{\text{WW(y)}} T_0$ case is pruned and the $T_0 \xrightarrow{\text{WW(y)}} T_2$ case become known.



(e) The undesired and violating cycle found by MonoSAT.

Figure 3: The "long fork" anomaly: an illustrating example of PolySI.

contrast, since it is possible that $T_3 \xrightarrow{\text{RW(x)}} T_5$, we have $\text{BV}^{\text{I}}_{1,5} = \text{BV}_{1,3} \wedge \text{BV}_{3,5}$.

MonoSAT Solving. Finally, we feed the SAT formula to MonoSAT for an acyclicity test of the graph I. MonoSAT successfully finds an undesired cycle $T_1 \xrightarrow{\text{WR(x)}} T_3 \xrightarrow{\text{RW(y)}} T_2 \xrightarrow{\text{WR(y)}} T_4 \xrightarrow{\text{RW(x)}} T_1$, which contains two *non-adjacent* RW edges; see Figure 3e. Therefore, this history violates SI.

4.2 Constructing the Generalized Polygraph

We construct the generalized polygraph G of the history $\mathcal{H}$ in two steps. First, we create the known graph of G by adding the known edges of types SO and WR to E_G. Second, we generate the generalized constraints of G on possible dependencies between transactions. Specifically, for each key x and each pair of transactions T and S that both write x, we generate a generalized constraint of the form $\langle \text{either}, \text{or} \rangle$ according to Definition 9.

4.3 Pruning Constraints

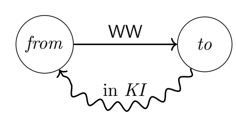
To accelerate MonoSAT solving, we prune as many constraints as possible before encoding (line 10). A constraint can be pruned if either of its two possibilities, represented by *either* or *or*, cannot happen, i.e., adding the edges in one of the two possibilities would create a cycle in the reduced SI graph. If neither of the two possibilities in a constraint can happen, PolySI immediately returns False. This process is repeated until no more constraints can be pruned (line 31).

In each iteration, we first construct the *currently known part* of the induced SI graph, denoted KI, of G. To do this, we define two auxiliary graphs, namely $\text{Dep} \leftarrow \text{G}|_{\text{SO}_\text{G} \cup \text{WR}_\text{G} \cup \text{WW}_\text{G}}$ and $\text{AntiDep} \leftarrow \text{G}|_{\text{RW}_\text{G}}$. By Definition 13, KI is $\text{Dep} \cup (\text{Dep} \ ; \ \text{AntiDep})$ (line 14). Then, we compute the reachability relation of KI. Next, for each constraint cons of the form $\langle \text{either}, \text{or} \rangle$, we check if either or or would create cycles in KI (line 16). Consider an edge (from, to, type) in either (line 17). By construction, it must be of type WW or RW. Note that KI does not contain any RW edges by definition. Therefore, an RW edge from from to to, together with a path from to to from in KI, does *not* necessarily create a cycle in KI. This fails the simple reachability-based strategy used in Cobra [42].

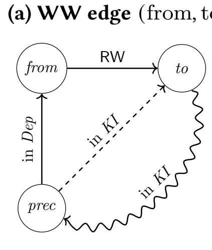
Suppose first that (from, to) is a WW edge; see Figure 4a. If there is already a path from to to from in KI (line 19), adding the WW edge would create a cycle in KI. Thus, we can prune the constraint cons and the edges in the other possibility or become known.

Now suppose that (from, to) is an RW edge; see Figure 4b. We check if there is a path in KI from to to any immediate predecessor prec of from in Dep (line 24). If there is a path, adding this RW edge would introduce, via composition with the edge from prec to from, an edge from prec to to in KI (the dashed arrow in Figure 4b). Then, with the path from to to prec, we obtain a cycle in KI.

The pruning process of the or possibility is same with that for either, except that it returns False



(a) WW edge (from, to)



(b) RW edge (from, to)

Figure 4: Two cases for pruning constraints.

Algorithm 1 The PolySI algorithm for checking SI

1: **procedure** CheckSI($\mathcal{H}$)
2:   **if** $\mathcal{H} \not\models$ Int $\lor$ AbortedReads $\lor$ IntermediateReads
3:     **return** false
4:   CreateKnownGraph($\mathcal{H}$) ▷ see [25, Appendix A]
5:   GenerateConstraints($\mathcal{H}$) ▷ see [25, Appendix A]
6:   **if** $\neg$PruneConstraints()
7:     **return** false
8:   SAT-Encode()
9:   **return** MonoSAT-Solve() ▷ see [25, Appendix A]

10: **procedure** PruneConstraints()
11:   **repeat**
12:     $\mathrm{Dep} \gets \mathrm{G}|_{\mathrm{SO_G} \cup \mathrm{WR_G} \cup \mathrm{WW_G}}$
13:     $\mathrm{AntiDep} \gets \mathrm{G}|_{\mathrm{RW_G}}$
14:     $\mathrm{KI} \gets \mathrm{Dep} \cup (\mathrm{Dep} \,;\, \mathrm{AntiDep})$
15:     $\mathrm{reachability} \gets$ Reachability(KI) ▷ using Floyd-Warshall algorithm [15].
16:     **for all** $\mathrm{cons} \gets \langle \mathrm{either}, \mathrm{or} \rangle \in \mathrm{C_G}$
17:       **for all** $(\mathrm{from}, \mathrm{to}, \mathrm{type}) \in \mathrm{either}$ ▷ for the "either" possibility
18:         **if** $\mathrm{type} = \mathrm{WW}$
19:           **if** $(\mathrm{to}, \mathrm{from}) \in \mathrm{reachability}$
20:             $\mathrm{C_G} \gets \mathrm{C_G} \setminus \{\mathrm{cons}\}$
21:             $\mathrm{E_G} \gets \mathrm{E_G} \cup \mathrm{or}$
22:             **break** the "**for all** $(\mathrm{from}, \mathrm{to}, \mathrm{type}) \in \mathrm{either}$" loop
23:         **else** ▷ type = RW
24:           **for all** $\mathrm{prec} \in \mathrm{V_{Dep}}$ such that $(\mathrm{prec}, \mathrm{from}, \_) \in \mathrm{E_{Dep}}$
25:             **if** $(\mathrm{to}, \mathrm{prec}) \in \mathrm{reachability}$
26:               $\mathrm{C_G} \gets \mathrm{C_G} \setminus \{\mathrm{cons}\}$
27:               $\mathrm{E_G} \gets \mathrm{E_G} \cup \mathrm{or}$
28:               **break** the "**for all** $(\mathrm{from}, \mathrm{to}, \mathrm{type}) \in \mathrm{either}$" loop
29:       **for all** $(\mathrm{from}, \mathrm{to}, \mathrm{type}) \in \mathrm{or}$ ▷ for the "or" possibility
30:         the same with the "either" possibility except that it returns False if both either and or possibilities of a constraint are pruned
31:   **until** $\mathrm{C_G}$ remains unchanged
32:   **return** True

33: **procedure** SAT-Encode()
34:   **for all** $\mathrm{v}_i, \mathrm{v}_j \in \mathrm{V_G}$ such that $i \neq j$
35:     $\mathsf{BV} \gets \mathsf{BV} \cup \{\mathsf{BV}_{i,j}, \mathsf{BV}^{\mathrm{I}}_{i,j}\}$
36:   **for all** $(\mathrm{v}_i, \mathrm{v}_j) \in \mathrm{E_G}$ ▷ encode the known graph of G
37:     $\mathsf{CL} \gets \mathsf{CL} \cup \{\mathsf{BV}_{i,j} = \mathsf{True}\}$
38:   **for all** $\langle \mathrm{either}, \mathrm{or} \rangle \in \mathrm{C_G}$ ▷ encode the constraints of G
39:     $\mathsf{CL} \gets \mathsf{CL} \cup \Big\{\big(\bigwedge_{(\mathrm{v}_i,\mathrm{v}_j,\_)\in\mathrm{either}} \mathsf{BV}_{i,j} \land \bigwedge_{(\mathrm{v}_i,\mathrm{v}_j,\_)\in\mathrm{or}} \neg\mathsf{BV}_{i,j}\big) \lor \big(\bigwedge_{(\mathrm{v}_i,\mathrm{v}_j,\_)\in\mathrm{or}} \mathsf{BV}_{i,j} \land \bigwedge_{(\mathrm{v}_i,\mathrm{v}_j,\_)\in\mathrm{either}} \neg\mathsf{BV}_{i,j}\big)\Big\}$
40:   $\mathrm{Dep} \gets \mathrm{G}|_{\mathrm{SO_G} \cup \mathrm{WR_G} \cup \mathrm{WW_G}}$
41:   $\mathrm{E_{Dep}} \gets \mathrm{E_{Dep}} \cup \{(\_,\_,\mathrm{WW}) \in \mathrm{either} \cup \mathrm{or} \mid \langle \mathrm{either}, \mathrm{or} \rangle \in \mathrm{C_G}\}$
42:   $\mathrm{AntiDep} \gets \mathrm{G}|_{\mathrm{RW_G}}$
43:   $\mathrm{E_{AntiDep}} \gets \mathrm{E_{AntiDep}} \cup \{(\_,\_,\mathrm{RW}) \in \mathrm{either} \cup \mathrm{or} \mid \langle \mathrm{either}, \mathrm{or} \rangle \in \mathrm{C_G}\}$
44:   $\mathsf{CL} \gets \mathsf{CL} \cup \Big\{\mathsf{BV}^{\mathrm{I}}_{i,j} = \big(\mathsf{BV}_{i,j} \land (\mathrm{v}_i, \mathrm{v}_j, \_) \in \mathrm{E_{Dep}}\big) \lor \big(\bigvee_{\substack{(\mathrm{v}_i,\mathrm{v}_k,\_)\in\mathrm{E_{Dep}} \\ (\mathrm{v}_k,\mathrm{v}_j,\_)\in\mathrm{E_{AntiDep}}}} \mathsf{BV}_{i,k} \land \mathsf{BV}_{k,j}\big) \;\Big|\; \mathrm{v}_i, \mathrm{v}_j \in \mathrm{V_G}\Big\}$ ▷ encode the induced SI graph I of G

if both either and or possibilities of a constraint are pruned.

The following theorem states that PruneConstraints is correct in that (1) it preserves the SI-(a)cyclicity of polygraphs; and (2) it does not introduce new undesired cycles, which ensures that any violation found in the pruned polygraph using MonoSAT later also exists in the original polygraph. This is crucial to the informativeness of PolySI. The theorem's proof can be found in [25, Appendix B].

Theorem 17 (Correctness of PruneConstraints). *Let G and G_p be the generalized polygraphs before and after* PruneConstraints*, respectively. Then,*

(1) G is SI-acyclic if and only if PruneConstraints *returns* True *and G_p is SI-acyclic.*
(2) Suppose that G_p is not SI-acyclic. Let C be a cycle in a compatible graph with the induced SI graph of G_p. Then there is a compatible graph with the induced SI graph of G that contains C.

Combining Theorems 16 and 17, we prove PolySI's soundness.

Theorem 18 (Soundness of PolySI). PolySI *is sound, i.e., if* PolySI *returns* False*, then the input history indeed violates SI.*

4.4 SAT Encoding

In this step we encode the induced SI graph, denoted I, of the pruned polygraph G into an SAT formula (line 33). We use BV and CL to denote the set of Boolean variables and the set of clauses of the SAT formula, respectively. For each pair of vertices v_i and v_j, we create two Boolean variables $\mathsf{BV}_{i,j}$ and $\mathsf{BV}^{\mathrm{I}}_{i,j}$: one for the polygraph G, and the other for its induced SI graph I. An edge $(\mathrm{v}_i, \mathrm{v}_j)$ is in the compatible graph with I (resp., G) if and only if $\mathsf{BV}^{\mathrm{I}}_{i,j}$ (resp., $\mathsf{BV}_{i,j}$) is assigned to True by MonoSAT in testing the acyclicity of I.

We first encode the polygraph G. For each edge $(\mathrm{v}_i, \mathrm{v}_j)$ in the known graph of G, we add a clause $\mathsf{BV}_{i,j} = \mathsf{True}$. For each constraint $\langle \mathrm{either}, \mathrm{or} \rangle$, the clause $\big(\bigwedge_{(\mathrm{v}_i,\mathrm{v}_j,\_)\in\mathrm{either}} \mathsf{BV}_{i,j} \land \bigwedge_{(\mathrm{v}_i,\mathrm{v}_j,\_)\in\mathrm{or}} \neg\mathsf{BV}_{i,j}\big) \lor \big(\bigwedge_{(\mathrm{v}_i,\mathrm{v}_j,\_)\in\mathrm{or}} \mathsf{BV}_{i,j} \land \bigwedge_{(\mathrm{v}_i,\mathrm{v}_j,\_)\in\mathrm{either}} \neg\mathsf{BV}_{i,j}\big)$ expresses that exactly one of either or or happens.

Then we encode the induced SI graph I of G. The auxiliary graph Dep contains all the known and potential SO, WR, and WW edges of G (lines 40 and 41), while AntiDep contains all the known and potential RW edges of G (lines 42 and 43). The clauses defined on BV^{I} at line 44 state that I is the union of Dep and the composition of Dep with AntiDep.

4.5 Completing the SI Checking

Theorem 6 assumes histories with only committed transactions and considers the WR, WW, and RW relations over transactions rather than read/write operations inside them. This would miss non-cycle anomalies. Hence, for completeness, PolySI also checks whether a history exhibits AbortedReads or IntermediateReads anomalies [1, 30] (line 2):

- *Aborted Reads*: a committed transaction cannot read a value from an aborted transaction.
- *Intermediate Reads*: a transaction cannot read a value that was overwritten by the transaction that wrote it.

Note that PolySI's completeness relies on a common assumption about *determinate* transactions [1, 7, 10, 11, 16, 30], i.e., the status of each transaction, whether committed or aborted, is legitimately decided. Indeterminate transactions are inherent to black-box testing: it is difficult for a client to justify the status of a transaction due to the invisibility of system internals. Together with the completeness of the dependency-graph-based characterization of SI in Theorem 6, we prove PolySI's completeness.

Theorem 19 (Completeness of PolySI). PolySI *is complete with respect to a history that contains only determinate transactions, i.e., if such a history indeed violates* SI*, then* PolySI *returns* false.

5 EXPERIMENTS

We have presented our SI checking algorithm PolySI and established its *soundness* and *completeness*. In this section, we conduct a comprehensive assessment of PolySI to answer the following questions with respect to the remaining criteria of SIEGE+ (Section 1):

(1) Effective: Can PolySI find SI violations in (production) databases?

(2) Informative: Can PolySI provide understandable counterexamples for SI violations?

(3) Efficient: How efficient is PolySI (and its components)? Can PolySI outperform the state of the art under *various* workloads and scale up to large-sized workloads?

Our answer to (1) is twofold (Section 5.2): (i) PolySI successfully reproduces all of 2477 known SI anomalies in production databases; and (ii) we use PolySI to detect novel SI violations in three cloud databases of different kinds: the graph database Dgraph [20], the relational database MariaDB-Galera [13], and YugabyteDB [50] supporting multiple data models. To answer (2) we provide an algorithm that recovers the violating scenario, highlighting the cause of the violation found (Section 5.3). Regarding (3), we (i) show that PolySI outperforms several competitive baselines including the most performant SI and serializability checkers to date; (ii) measure the contributions of its different components/optimizations to the overall performance under both general and specific transaction workloads (Section 5.4); and (iii) demonstrate its scalability for large-sized workloads with one billion keys and one million transactions. Note that we demonstrate PolySI's **generality** along with the answers to questions (1) and (3).

5.1 Workloads, Benchmarks, and Setup

5.1.1 Workloads and Benchmarks. To evaluate PolySI on *general* read-only, write-only, and read-write transaction workloads, we have implemented a parametric workload generator. Its parameters are: the number of client sessions (#sess; 20 by default), the number of transactions per session (#txns/sess; 100 by default), the number of read/write operations per transaction (#ops/txn; 15 by default), the percentage of reads (%reads; 50% by default), the total number of keys (#keys; 10k by default), and the key-access distribution (dist) including uniform, zipfian (by default), and hotspot (80% operations touching 20% keys). Note that the default 2k transactions with 30k operations issued by 20 sessions are sufficient to distinguish PolySI from competing tools (see Section 5.4.1).

Among such general workloads, we also consider three representatives, each with 10k transactions and 80k operations in total (#sess=25, #txns/sess=400, and #ops/txn=8), in the comparison with Cobra and the decomposition and differential analysis of PolySI: (i) GeneralRH, read-heavy workload with 95% reads; (ii) GeneralRW, medium workload with 50% reads; and (iii) GeneralWH, write-heavy workloads with 30% reads.

We also use three synthetic benchmarks with only serializable histories of at least 10k transactions (which also satisfy SI):

- RUBiS [38]: an eBay-like bidding system where users can, for example, register and bid for items. The dataset archived by [42] contains 20k users and 200k items.
- TPC-C [47]: an open standard for benchmarking online transaction processing with a mix of five different types of transactions (e.g., for orders and payment) portraying the activity of a wholesale supplier. The dataset includes one warehouse, 10 districts, and 30k customers.
- C-Twitter [29]: a Twitter clone where users can, for example, tweet and follow or unfollow other users (following the zipfian distribution).

To assess PolySI's scalability, we also consider large-sized workloads with one billion keys and one million transactions (#sess=20; #txns/sess=50k). The workloads contain both short and long transactions; the default sizes are 15 and 150, respectively.

5.1.2 Setup. We use a PostgreSQL (v15 Beta 1) instance to produce *valid* histories without isolation violations: for the performance comparison with other SI checkers and the decomposition and differential analysis of PolySI itself, we set the isolation level to *repeatable read* (implemented as SI in PostgreSQL [37]); for the runtime comparison with Cobra (Section 5.4.1), we use the *serializability* isolation level to produce serializable histories. We colocate the client threads and PostgreSQL (or other databases for testing; see Section 5.2.2) on a local machine. Each client thread issues a stream of transactions produced by our workload generator to the database and records the execution history. All histories are saved to a file to benchmark each tool's performance.

We have implemented PolySI in 2.3k lines of Java code, and the workload generator, including the transformation from generated key-value operations to SQL queries (for the interactions with relational databases such as PostgreSQL), in 2.2k lines of Rust code. We ensure unique values written for each key using counters. We use a simple database schema of a two-column table storing keys and values, which is effective to find real violations in three production databases (see Section 5.2).

We conducted all experiments with a 4.5GHz Intel Xeon E5-2620 (6-core) CPU, 48GB memory, and an NVIDIA K620 GPU.

5.2 Finding SI Violations

5.2.1 Reproducing Known SI Violations. PolySI successfully reproduces *all* known SI violations in an extensive collection of 2477 anomalous histories [7, 17, 28]. These histories were obtained from the earlier releases of three different production databases, i.e., CockroachDB, MySQL-Galera, and YugabyteDB; see Table 2 for details. This set of experiments provides supporting evidence for PolySI's *soundness* and *completeness*, established in Section 4.

5.2.2 Detecting New Violations. We use PolySI to examine recent releases of three well-known cloud databases (of different kinds) that claim to provide SI: Dgraph [20], MariaDB-Galera [13], and YugabyteDB [50]. See Table 2 for details. We have found and reported

Table 2: Summary of tested databases. Multi-model refers to relational DBMS, document store, and wide-column store.

Database	GitHub Stars	Kind	Release
New violations found:			
Dgraph	18.2k	Graph	v21.12.0
MariaDB-Galera	4.4k	Relational	v10.7.3
YugabyteDB	6.7k	Multi-model	v2.11.1.0
Known bugs [7, 17, 28]:			
CockroachDB	25.1k	Relational	v2.1.0
			v2.1.6
MySQL-Galera	381	Relational	v25.3.26
YugabyteDB	6.7k	Multi-model	v1.1.10.0

novel SI violations in all three databases which, as of the time of writing, are being investigated by the developers. In particular, as communicated with the developers, (i) our finding has helped the DGraph team confirm some of their suspicions about their latest release; and (ii) Galera has confirmed the incorrect claim on preventing lost updates for transactions issued on different cluster nodes and thereafter removed any claims on SI or "partially supporting SI" from the previous documentation.[4]

5.3 Understanding Violations

MonoSAT reports cycles, constructed from its output logs, upon detecting an SI violation. However, such cycles are *uninformative* with respect to understanding how the violation actually occurred. For instance, Figure 5(a) depicts the original cycle returned by MonoSAT for an SI violation found in MariaDB-Galera, where it is difficult to identify the cause of the violation.

Hence, we have designed an algorithm to interpret the returned cycles. The key idea is to (i) bring back any potentially involved transactions and the associated dependencies, (ii) restore the violating scenario by identifying the core participants and dependencies, and (iii) remove the "irrelevant" dependencies to simplify the scenario. We have integrated into PolySI the algorithm written in 300 lines of C++ code. The pseudocode is given in [25, Appendix C]. We have also integrated the Graphviz tool [23] into PolySI to visualize the final counterexamples (e.g., Figure 5).

Minimal Counterexample. A "minimal" counterexample would facilitate understanding how the violation actually occurred. We define a minimal violation as a polygraph where no dependency can be removed; otherwise, the resulting polygraph would pass the verification of PolySI. Given a polygraph G (constructed from a collected history) and a cycle C (returned by MonoSAT), there may however be more than one minimal violation with respect to G and C due to different interpretations of uncertain dependencies. We call the one with the least number of dependencies *the minimal counterexample with respect to G and C*.

PolySI guarantees the minimality of returned counterexample:

[4]https://github.com/codership/documentation/commit/cc8d6125f1767493eb61e2cc82f5a365ecee6e7a and https://github.com/codership/documentation/commit/d87171b0d1b510fe59973cb7ce5892061ce67b80

Theorem 20 (Minimality). *PolySI always returns a minimal counterexample with respect to G and C, with G the polygraph built from a history and C the cycle output by MonoSAT.*

We defer to [25, Appendix E] for the formal definitions of the minimal violation and counterexample and the proof of Theorem 28.

Violation Found in MariaDB-Galera. We present an example violation detected in MariaDB-Galera. In particular, we illustrate how the interpretation algorithm helps us locate the violation cause: *lost update*. We defer the Dgraph and YugabyteDB anomalies (causality violations) to [25, Appendix D]. In the following example, we use T:(s, n) to denote the nth transaction issued by session s.

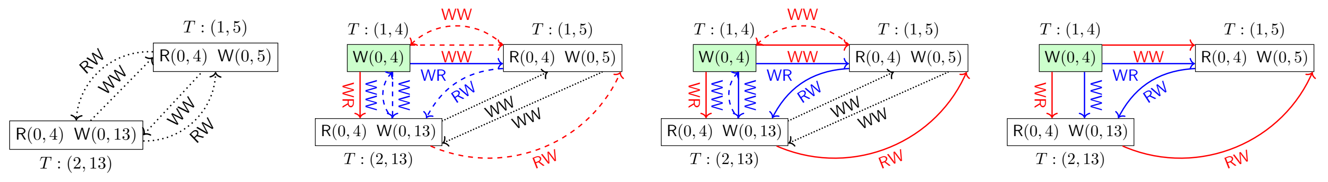
Given the original cycles returned by MonoSAT in Figure 5(a), PolySI first finds the (only) "missing" transaction T:(1,4) (colored in green) and the associated dependencies, as shown in Figure 5(b). Note that some of the dependencies are uncertain at this moment, e.g., the WW dependency between T:(1,4) and T:(1,5) (in red). PolySI then restores the violating scenario by resolving such uncertainties. For example, as depicted in Figure 5(c), PolySI determines that W(0,4) was actually installed first in the database, i.e., T:(1,4)$\xrightarrow{\text{WW}}$T:(1,5), because there would otherwise be an undesired cycle with the known dependencies, i.e., T:(1,5)$\xrightarrow{\text{WW}}$T:(1,4)$\xrightarrow{\text{WR}}$T:(1,5). The same reasoning applies to determine the WW dependency between T:(1,4) and T:(2,13) (in blue). Finally, PolySI finalizes the violating scenario by removing any remaining uncertainties including those dependencies not involved in the actual violation (the WW dependency between T:(1,5) and T:(2,13) in this case).

The violating scenario now becomes informative and explainable: transaction T:(1,4) writes value 4 on key 0, which is read by transactions T:(2,13) and T:(1,5). Both transactions subsequently commit their writes on key 0 by W(0,13) and W(0,5), respectively, which results in a *lost update* anomaly.

5.4 Performance Evaluation

In this section, we conduct an in-depth performance analysis of PolySI and compare it to the following black-box checkers:

- dbcop [7] is, to the best of our knowledge, the most efficient black-box SI checker that does not use an off-the-shelf solver. Note that, unlike our PolySI tool, dbcop does not check *aborted reads* or *intermediate reads* (see Section 4.5).
- Cobra [42] is the state-of-the-art SER checker utilizing both MonoSAT and GPUs to accelerate the checking procedure. Cobra serves as a baseline because (i) checking SI is more complicated than checking SER in general [7], and constraint pruning and the MonoSAT encoding for SI are more challenging in particular due to more complex cycle patterns in dependency graphs (Theorem 6, Section 2.2.3); and (ii) Cobra is the most performant SER checker to date.
- CobraSI: We implement the incremental algorithm [7, Section 4.3] for reducing checking SI to checking serializability (in polynomial time) to leverage Cobra. We consider two variants: (i) CobraSI without GPU for a fair comparison with PolySI and dbcop, which do not employ GPU or



(a) Original output **(b) Missing participants** **(c) Recovered scenario** **(d) Finalized scenario**

Figure 5: Lost update: the SI violation found in MariaDB-Galera. The original output dependencies are represented by dotted black arrows. The recovered dependencies are colored in red/blue with dashed and solid arrows representing uncertain and certain dependencies, respectively. The missing transaction is colored in green. We omit key 0, associated with all dependencies.

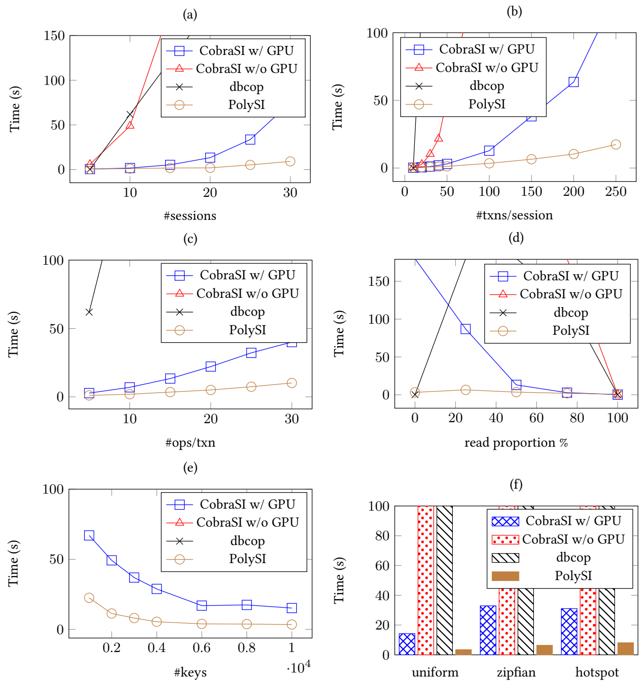


Figure 6: Performance comparison with the competing SI checkers under various workloads. Experiments time out at 180s; data points are not plotted for timed-out experiments.

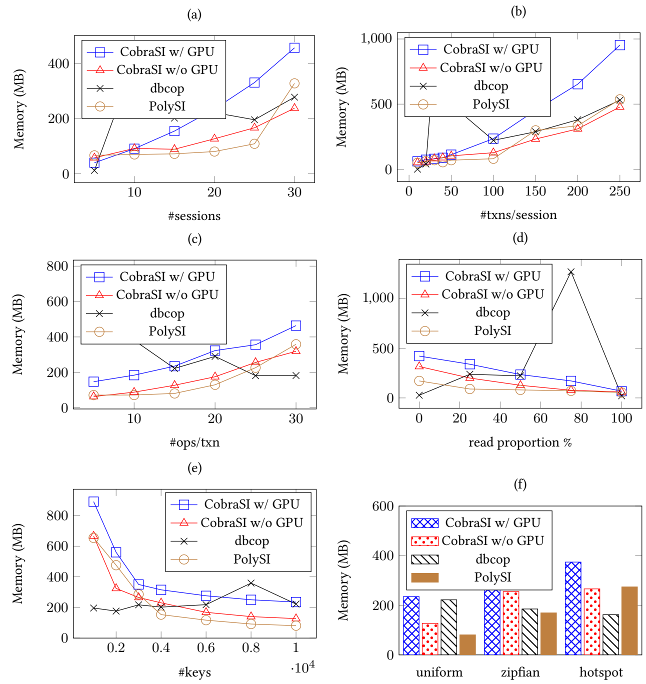


Figure 7: Comparison on memory overhead with competing SI checkers under various workloads.

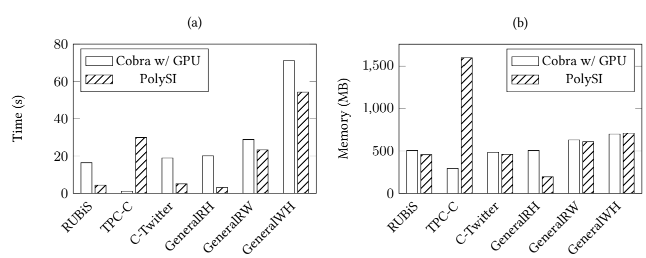


Figure 8: Comparison on time and memory overhead with Cobra with GPU acceleration under representative workloads.

multithreading; and (ii) CobraSI with GPU as a strong competitor.

5.4.1 Performance Comparison with State of the Art. Our first set of experiments compares PolySI with the competing SI checkers under a wide range of workloads. The input histories extracted from PostgreSQL (with the *repeatable read* isolation level) are all valid with respect to SI. The experimental results are shown in Figure 6: PolySI significantly surpasses not only the state-of-the-art SI checker dbcop but also CobraSI with GPU. In particular, with more concurrency, such as more sessions (a), transactions per session (b), and operations per transaction (c), CobraSI with GPU exhibits exponentially increasing checking time[5] while PolySI incurs only

[5] Two major reasons are: (i) Cobra has already been shown to exhibit exponential verification time under general workloads [42]; and (ii) the incremental algorithm for reducing checking SI to checking serializability typically doubles the number of transactions in a given history [7], rendering the checking even more expensive.

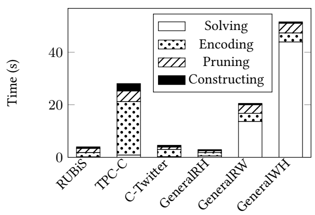


Figure 9: Decomposing PolySI's checking time into stages.

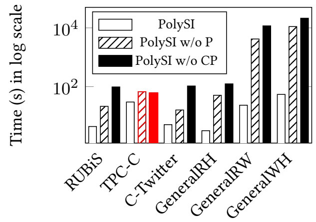


Figure 10: Diff. analysis of PolySI. Memory-exhausted runs are colored in red.

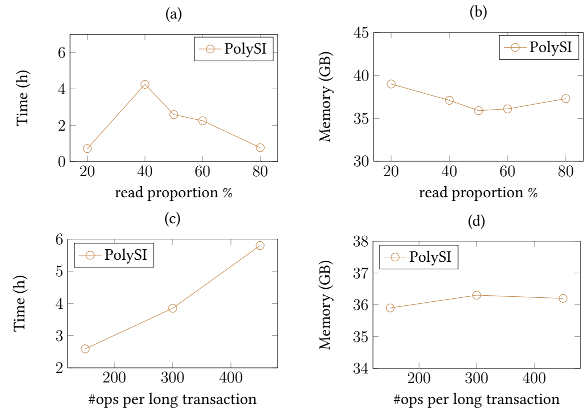


Figure 11: PolySI's overhead on large-sized workloads with one billion keys and one million transactions.

moderate overhead. The result depicted in Figure 6(f) is also consistent: with the skewed key accesses representing high concurrency as in the zipfian and hotspot distributions, both dbcop and CobraSI without GPU acceleration time out. Moreover, even with the GPU acceleration, CobraSI takes 6x more time than PolySI. Finally, unlike the other SI checkers, PolySI's performance is fairly stable with respect to varying read/write proportions (d) and keys (e).

In Figure 8(a) we compare PolySI with the baseline serializability checker Cobra. We present the checking time on various benchmarks. PolySI outperforms Cobra (with its GPU acceleration enabled) in five of the six benchmarks with up to 3x improvement (as for GeneralRH). The only exception is TPC-C, where most of the transactions have the read-modify-write pattern,[6] for which Cobra implements a specific optimization to efficiently infer dependencies before pruning and encoding.

We also measure the memory usage for all the checkers under the same settings as in Figure 6 and Figure 8(a). As shown in Figure 7, PolySI consumes less memory (for storing both generated graphs and constraints) than the competitors in general. Note that dbcop, the only checker that does not rely on solving and stores no constraints, is not competitive with PolySI for most of the cases. Regarding the comparison on specific benchmarks (Figure 8(b)), PolySI and Cobra with GPU acceleration have similar overheads, while PolySI (resp. Cobra) requires less memory for read-heavy workloads (resp. TPC-C).

Table 3: Number of constraints and unknown dependencies before and after pruning (P) in the six benchmarks.

Benchmark	#cons. before P	#cons. after P	#unk. dep. before P	#unk. dep. after P
TPC-C	386k	0	3628k	0
GeneralRH	4k	29	39k	77
RUBiS	14k	149	171k	839
C-Twitter	59k	277	307k	776
GeneralRW	90k	2565	401k	5435
GeneralWH	167k	6962	468k	14376

5.4.2 *Decomposition Analysis of PolySI.*

We measure PolySI's checking time in terms of stages: *constructing*, which builds up a generalized polygraph from a given history; *pruning*, which prunes constraints in the generalized polygraph; *encoding*, which encodes the graph and the remaining constraints; and *solving*, which runs the MonoSAT solver.

Figure 9 depicts the results on six different datasets. Constructing a generalized polygraph is relatively inexpensive. The overhead of pruning is fairly constant, regardless of the workloads; PolySI can effectively prune (resp. resolve) a huge number of constraints (resp. unknown dependencies) in this phase. See Table 3 for details. In particular, for TPC-C which contains only read-only and read-modify-write transactions, PolySI is able to resolve all uncertainties on WW relations and identify the unique version chain for each key. The encoding effort is moderate; TPC-C incurs more overhead as the number of operations in total is 5x more than the others. The solving time depends on the remaining constraints and unknown dependencies after pruning, e.g., the left four datasets incur negligible overhead (see Table 3).

5.4.3 *Differential Analysis of PolySI.*

To investigate the contributions of PolySI's two major optimizations, we experiment with three variants: (i) PolySI itself; (ii) PolySI without pruning (P) constraints; and (iii) PolySI without both compacting (C) and pruning the constraints. Figure 10 demonstrates the acceleration produced by each optimization. Note that the two variants without optimization exhibit (16GB) memory-exhausted runs on TPC-C, which contain considerably more uncertain dependencies (3628k) and constraints (386k) without pruning than the other datasets (see Table 3).

5.4.4 *Scalability.*

To assess PolySI's scalability, we generate transaction workloads with one billion keys and one million transactions with hundreds of millions of operations. We experiment with varying read proportions and long transaction sizes (up to 450 operations per transaction). As shown in Figure 11, PolySI consumes less than 40GB memory in all cases and at most 4 hours for checking one million transactions. We also observe that the time used increases linearly with larger-sized transactions while the memory overhead is fairly stable. To conclude, large-sized workloads are quite manageable for PolySI on modern hardware. Note that the competing checkers, as expected, fail to handle such workloads.

[6] In a read-modify-write transaction each read is followed by a write on the same key.

6 DISCUSSION

Fault Injection. We have found SI violations in three production databases without injecting faults, such as network partition and clock drift. Since PolySI is an off-the-shelf checker, it is straightforward to integrate it into existing testing frameworks with fault injection such as Jepsen [27] and CoFI [12]; both have been demonstrated to effectively trigger bugs in distributed systems.

Database Schema. In our testing of production databases, we used a simple, yet effective, database schema adopted by most of black-box checkers [7, 42, 51, 52]: a two-column table storing key-value pairs. Extending it to multi-columns or even the column-family data model could be done by: (i) representing each cell in a table as a *compound key*, i.e., "TableName:PrimaryKey:ColumnName", and a single value, i.e., the content of the cell [8, 32]; and (ii) utilizing the compiler in [8] to rewrite (more complex) SQL queries to key-value read/write operations.

Predicates. To the best of our knowledge, none of the state-of-the-art black-box checkers [7, 8, 30, 42, 51, 52] considers predicates nor can they detect predicate-specific anomalies. Given the non-predicate violations found by PolySI (as well as dbcop [7] and Elle [30]), we conjecture that more anomalies would arise with predicates. It is therefore interesting future work to extend our SI characterization to represent predicates and to explore optimizations with respect to encoding and pruning.

Unique Value. As demonstrated in our experiments, guaranteeing "unique value" is a pragmatic, purely black-box technique, and effective in detecting anomalies. When this assumption is broken, the complexity of the checking problem would become higher due to inferring uncertain WR dependencies (a single read may be related to multiple "false" writes). Accordingly, we could add the encoding in PolySI for unique existence of WR dependency among all uncertainties prior to SAT solving.

Optimization for Long Histories. PolySI's overhead when checking one million transactions with 450 operations per long transaction is manageable for modern hardware. Still, optimizing PolySI for long transaction histories would help to reduce checking overhead, especially for *online* transactional processing workloads. We could consider periodically taking snapshots (via read-only transactions) across all sessions in a history using an additional client session. Such snapshots carry the summary of write dependencies thus far, which discards prior transactions in the history. As a result, at any point of time, one only needs to consider a segment of the history consisting of the latest snapshot and its subsequent transactions.

7 RELATED WORK

Characterizing Snapshot Isolation. Many frameworks and formalisms have been developed to characterize SI and its variants. Berenson et al. [5] considers SI as a multi-version concurrency control mechanism (described also in Section 2.1). Adya [1] presents the first formal definition of SI using dependency graphs, which, as pointed out by [11], still relies on low-level implementation choices such as how to order start and commit events in transactions. Cerone et al. [10] proposes an axiomatic framework to declaratively define SI with the dual notions of visibility (what transactions can observe) and arbitration (the order of installed versions/values). The follow-up work [11] characterizes SI solely in terms of Adya's dependency graphs, requiring no additional information about transactions. Crooks et al. [16] introduces an alternative implementation-agnostic formalization of SI and its variants based on client-observed values read/written.

Driven by black-box testing of SI, we base our GP-based characterization on Cerone and Gotsman's formal specification [11]. In particular, our new characterization:

(i) targets the prevalent *strong session* variant of SI [11, 19], where *sessions*, advocated by Terry et al. [43], have been adopted by many production databases in practice (e.g., DGraph [20], Galera [13], and CockroachDB [14]);

(ii) does not rely on implementation details such as concurrency control mechanism as in [5] and timestamps as in [1], and the operational semantics of the underlying database as in [49], which are usually invisible to the outsiders; and

(iii) naturally models uncertain dependencies inherent to black-box testing using generalized constraints (Section 3) and enables the acceleration of SMT solving by compacting constraints (Section 5.4).

Regarding the comparison with [16], despite its promising characterization of SI suitable for black-box testing, we are unaware of any checking algorithm based on it. A straightforward (suboptimal) implementation would require enumerating all permutations of the transactions in a history, e.g., 10k transactions in our experiment would require checking 10k-factorial permutations.

Dynamic Checking of SI. This technique determines whether a collected history from dynamically executing a database satisfies SI. We are unaware of any black-box SI checker that satisfies SIEGE+.

dbcop [7] is the most efficient black-box SI checker to date. The underlying checking algorithm runs in time $O(n^c)$, with n and c the number of transactions and clients involved in a single history, respectively. The authors devise both a polynomial-time algorithm for checking serializabilty (also with a fixed number of client sessions) and a polynomial-time algorithm for reducing checking SI to checking serializabilty. However, as demonstrated in Section 5.4, dbcop is practically not as efficient as our PolySI tool under various workloads. Moreover, dbcop is incomplete as it does not check non-cycle anomalies such as *aborted reads* and *intermediate reads* (Section 4.5). Finally, dbcop provides no details upon a violation; only a "false" answer is returned.

Elle [30] is a state-of-the-art checker for a variety of isolation levels, including strong session SI,[7] which is part of the Jepsen [27] testing framework. Elle requires specific data models like lists in workloads to infer the WW dependencies and specific APIs to perform list-specific operations such as "append". In contrast, PolySI is compatible with general and production workloads and uses standard key-value and SQL APIs. Elle builds upon Adya's formalization of SI [1], thus relying on the start and commit timestamps of transactions. Such information may not always be available, e.g., MongoDB [34] and TiDB [46] have no timestamps in their logs

[7] Despite the claim to support checking strong session SI, we have confirmed with the developer that Elle does not fulfill this functionality in its latest release [26].

for read-only transactions. Nonetheless, the underlying SI characterization for PolySI does not rely on any implementation details. Finally, despite the theoretical soundness and completeness (modulo determinate transactions) claim, Elle's actual implementation is unsound for efficiency reasons and there are also anomalies it cannot detect. We have confirmed this with the developer [26].

ConsAD [51] is a checker tailored to application servers as opposed to black-box databases in our setting. Its SI checking algorithm is also based on dependency graphs. To determine the WW dependencies, ConsAD enforces the commit order of update transactions using, e.g., artificial SQL queries, to acquire exclusive locks on the database records, resulting in additional overhead [40, 51]. Moreover, ConsAD is incapable of detecting non-cycle anomalies.

8 CONCLUSION

We have presented the design of PolySI, along with a novel characterization of SI using generalized polygraphs. We have established the soundness and completeness of our new characterization and PolySI's checking algorithm. Moreover, we have demonstrated PolySI's effectiveness by reproducing all of 2477 known anomalies and by finding new violations in three popular production databases, its efficiency by experimentally showing that it outperforms the state-of-the-art tools and can scale up to large-sized workloads, and its generality, operating over a wide range of workloads and databases of different kinds. Finally, we have leveraged PolySI's interpretation algorithm to identify the causes of the violations.

PolySI is the first black-box SI checker that satisfies the SIEGE+ principle. The obvious next step is to apply SMT solving to build SIEGE+ black-box checkers for other data consistency properties such as transactional causal consistency [21, 32] and the recently proposed regular sequential consistency [24]. Moreover, we will pursue the research directions discussed in Section 6.

REFERENCES

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. USA.

[2] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (nov 2013), 181–192. https://doi.org/10.14778/2732232.2732237

[3] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3, Article 15 (jul 2016), 45 pages. https://doi.org/10.1145/2909870

[4] Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. 2015. SAT modulo Monotonic Theories. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI'15).* AAAI Press, 3702–3709.

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD '95.* ACM, 1–10. https://doi.org/10.1145/223784.223785

[6] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., USA.

[7] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 165 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360591

[8] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 132 (oct 2021), 27 pages. https://doi.org/10.1145/3485546

[9] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *POPL'17.* ACM, 626–638.

[10] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *CONCUR'15 (LIPIcs)*, Vol. 42. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 58–71.

[11] Andrea Cerone and Alexey Gotsman. 2018. Analysing Snapshot Isolation. *J. ACM* 65, 2, Article 11 (Jan. 2018), 41 pages. https://doi.org/10.1145/3152396

[12] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *ASE 2020.* IEEE. https://doi.org/10.1145/3324884.3416548

[13] MariaDB Galera Cluster. Accessed August, 2022. https://mariadb.com/kb/en/what-is-mariadb-galera-cluster/.

[14] CockroachDB. Accessed August, 2022. https://www.cockroachlabs.com/.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[16] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *PODC '17.* ACM, 73–82. https://doi.org/10.1145/3087801.3087802

[17] Ben Darnell. Accessed August, 2022. Lessons Learned from 2+ Years of Nightly Jepsen Tests. https://www.cockroachlabs.com/blog/jepsen-tests-lessons/.

[18] Oracle Database. Accessed August, 2022. https://www.oracle.com/database/.

[19] Khuzaima Daudjee and Kenneth Salem. 2006. Lazy Database Replication with Snapshot Isolation. In *VLDB'06.* VLDB Endowment, 715–726.

[20] Dgraph. Accessed August, 2022. https://dgraph.io/.

[21] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal Consistency and Latency Optimality: Friend or Foe? *Proc. VLDB Endow.* 11, 11 (2018), 1618–1632.

[22] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *Proc. VLDB Endow.* 13, 12 (July 2020), 2773–2786. https://doi.org/10.14778/3407790.3407860

[23] Graphviz. Accessed December, 2022. Open source graph visualization software. https://graphviz.org/.

[24] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. 2021. Regular Sequential Serializability and Regular Sequential Consistency. In *SOSP'21.* ACM, 163–179.

[25] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, Haixiang Li, and Anqun Pan. 2022. *Efficient Black-box Checking of Snapshot Isolation in Databases.* Technical Report. https://github.com/anonymous-hippo/PolySI/blob/main/polysi-tr.pdf.

[26] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, Haixiang Li, and Anqun Pan. Accessed December, 2022. Issue #17. https://github.com/jepsen-io/elle/issues/17.

[27] Jepsen. Accessed August, 2022. https://jepsen.io.

[28] Jepsen. Accessed August, 2022. Issue #824. https://github.com/YugaByte/yugabyte-db/issues/824.

[29] Nick Kallen. Accessed August, 2022. Big Data in Real Time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter/.

[30] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 268–280.

[31] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[32] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *NSDI' 13.* USENIX Association, 313–328.

[33] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *OSDI 2020.* USENIX Association, 333–349.

[34] MongoDB. Accessed August, 2022. https://www.mongodb.com/.

[35] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (oct 1979), 631–653. https://doi.org/10.1145/322154.322158

[36] Daniel Peng and Frank Dabek. 2010. Large-Scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI'10.* USENIX Association, USA, 251–264.

[37] PostgreSQL. Accessed August, 2022. Transaction Isolation. https://www.postgresql.org/docs/current/transaction-iso.html.

[38] RUBiS. Accessed August, 2022. Auction Site for e-Commerce Technologies Benchmarking. https://projects.ow2.org/view/rubis/.

[39] Microsoft SQL Server. Accessed August, 2022. https://www.microsoft.com/en-us/sql-server/.

[40] Zechao Shang, Jeffrey Xu Yu, and Aaron J. Elmore. 2018. RushMon: Real-Time Isolation Anomalies Monitoring. In *SIGMOD '18.* ACM, 647–662. https://doi.org/10.1145/3183713.3196932

[41] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-Replicated Systems. In *SOSP '11.* ACM, 385–400. https://doi.org/10.1145/2043556.2043592

[42] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI'20.* Article 4, 18 pages.

[43] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *PDIS.* IEEE Computer Society, 140–149.

[44] Jepsen testing of MongoDB 4.2.6. Accessed August, 2022. http://jepsen.io/analyses/mongodb-4.2.6.

[45] Jepsen testing of TiDB 2.1.7. Accessed August, 2022. https://jepsen.io/analyses/tidb-2.1.7.

[46] TiDB. Accessed August, 2022. https://en.pingcap.com/tidb/.

[47] TPC. Accessed August, 2022. TPC-C: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

[48] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *SIGMOD 2017.* ACM, 5–20. https://doi.org/10.1145/3035918.3064037

[49] Shale Xiong, Andrea Cerone, Azalea Raad, and Philippa Gardner. 2020. Data Consistency in Transactional Storage Systems: A Centralised Semantics. In *ECOOP'20*, Vol. 166. 21:1–21:31. https://doi.org/10.4230/LIPIcs.ECOOP.2020.21

[50] YugabyteDB. Accessed August, 2022. https://www.yugabyte.com/.

[51] Kamal Zellag and Bettina Kemme. 2014. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *VLDB J.* 23, 1 (2014), 147–172. https://doi.org/10.1007/s00778-013-0318-x

[52] Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2019. Checking Causal Consistency of Distributed Databases. In *NETYS 2019 (LNCS)*, Vol. 11704. Springer, 35–51. https://doi.org/10.1007/978-3-030-31277-0_3

A THE CHECKING ALGORITHM FOR SI

The full version of the checking algorithm is given in Algorithm 2. In the following, we reference pseudocode lines using the format algorithm#:line#.

B PROOFS

B.1 Proof of Theorem 16

Proof. The proof proceeds in two directions.

(" $\Longrightarrow$ ") Suppose that $\mathcal{H}$ satisfies SI. By Theorem 6, $\mathcal{H}$ satisfies Int and there exist WR, WW, and RW relations with which $\mathcal{H}$ can be extended to a dependency graph G such that $(\mathrm{SO}_G \cup \mathrm{WR}_G \cup \mathrm{WW}_G) \ ; \ \mathrm{RW}_G?$ is acyclic. We show that the generalized polygraph G' of $\mathcal{H}$ is SI-acyclic by constructing a compatible graph G'' with G' such that $G''|_{\mathcal{R}}$ is acyclic when the edge types are ignored: Consider a constraint $\langle \mathrm{either}, \mathrm{or} \rangle$ in G' and any WW edge $T \xrightarrow{\mathrm{WW}} S$ in G. If $(T, S, \mathrm{WW}) \in \mathrm{either}$, add all the edges in either into G''. Otherwise, add all the edges in or into G''.

(" $\Longleftarrow$ ") Suppose that $\mathcal{H} \models$ Int and the generalized polygraph G' of $\mathcal{H}$ is SI-acyclic. By Definition 15, there exists a compatible graph G'' with G' such that $G''|_{\mathcal{R}}$ is acyclic when the edge types are ignored. We show that $\mathcal{H}$ satisfies SI by constructing suitable WR, WW, and RW relations with which $\mathcal{H}$ can be extended to a dependency graph G such that $(\mathrm{SO}_G \cup \mathrm{WR}_G \cup \mathrm{WW}_G) \ ; \ \mathrm{RW}_G?$ is acyclic: We simply take G to be G'' by defining, e.g., $\mathrm{WW} = \{(a, b) \mid (a, b, \mathrm{WW}) \in E_{G''}\}$. □

B.2 Proof of Theorem 17

Proof. We prove (1) by induction on the number of iterations of pruning. Consider an arbitrary iteration of pruning $\mathcal{P}$ (lines 2:36–2:68) and denote the generalized polygraphs just before and after $\mathcal{P}$ by G_1 and G_2, respectively. We should show that G_1 is SI-acyclic if and only if $\mathcal{P}$ does not return False from line 2:57 or line 2:65 and G_2 is SI-acyclic. The following proof proceeds in two directions.

(" $\Longrightarrow$ ") Suppose that G_1 is SI-acyclic. We then proceed by contradiction.

- Suppose that $\mathcal{P}$ returns False from line 2:57 or line 2:65. This happens when some constraint in G_1 constructed based on two write transactions, say T and S, on the same key cannot be resolved appropriately: every compatible graph with the induced SI graph of G_1 contains a cycle without adjacent RW edges, no matter whether $T \xrightarrow{\mathrm{WW}} S$ or $S \xrightarrow{\mathrm{WW}} T$ is in it. That is, G_1 is not SI-acyclic. Contradiction.
- Suppose that G_2 is not SI-acyclic. G_1 is not SI-acyclic because the set of constraints in G_2 is a subset of that in G_1 and $\mathcal{P}$ prunes a constraint only if one of its two possibilities cannot happen. Contradiction.

(" $\Longleftarrow$ ") Suppose that $\mathcal{P}$ does not return False from line 2:57 or line 2:65 and G_2 is SI-acyclic. Therefore, there exists an acyclic compatible graph with the induced SI graph of G_2. This is also an acyclic compatible graph with the induced SI graph of G_1. Hence, G_1 is SI-acyclic.

The second part of the theorem holds because any compatible graph with the induced SI graph of G_p is also a compatible graph with the induced SI graph of G. □

C THE INTERPRETATION ALGORITHM

In this section, we describe the interpretation algorithm, called Interpret, that helps locate the causes of violations found by PolySI; see Algorithm 3. The algorithm takes as input the undesired cycles constructed from the log generated by MonoSAT, and outputs a dependency graph demonstrating the cause of the violation.

It is often difficult to locate the cause of a violation based solely on the original cycles because the cycles may miss some crucial information such as the core participating transactions and the dependencies between transactions. Therefore, Interpret first restores such information from the generalized polygraph of the history to reproduce the whole violating scenario (line 3:2). This may bring uncertain dependencies to the resulting polygraph, which are then resolved via pruning (line 3:3). Finally, Interpret finalizes the violating scenario by removing any remaining uncertain dependencies, as they are the "effect" of the violation instead of the "cause" (line 3:4).

Restore Transactions and Dependencies. The procedure Restore restores dependencies and the transactions involved in these dependencies in two steps (line 3:6). First, it checks each RW dependency in the input undesired cycles and restores its associated WW and WR dependencies if they are missing (lines 3:8 – 3:11). Then, for each WW dependency in an undesired cycle, it restores the other direction of this WW dependency (if missing) which may be involved in another undesired cycle (line 3:17 - line 3:34). Now it has recovered the minimal counter example. To help understand the violation more clearly, it checks each RW dependency in the input undesired cycles and restore its associated WW and WR dependencies if they are missing ((line 3:8 – 3:11).

Resolve Uncertain Dependencies. The restored violating scenario may contain WW and RW dependencies that are still uncertain at this moment. For an uncertain dependency to be part of the cause of a violation, it must be resolved. The procedure Resolve resolves as many uncertain dependencies as possible using the same idea of pruning (Section 4.3). Specifically, if an uncertain dependency from either (resp. or) in a constraint would create an undesired cycle with other certain dependencies, it, along with its associated dependencies in either (resp. or) is removed and the dependencies in or (resp. either) become certain (lines 3:42 - 3:48).

Finalize the Violating Scenario. The procedure Finalize finalizes the violating scenario by removing all the remaining uncertain dependencies, as they are the "effect" of the violation instead of the "cause" (line 3:49).

D CAUSALITY VIOLATIONS FOUND

D.1 A Causality Violation Found in Dgraph

There might be multiple "missing" transactions that together contribute to a violation. As depicted in Figure 12(b), PolySI restores the potentially involved five transactions (in green) and the associated dependencies from the original output cycle in Figure 12(a). In particular, two sub-scenarios are involved: the left subgraph concerns the $\mathrm{RW}(656)$ dependency on key 656 (in blue) while the right subgraph concerns the dependency $\mathrm{WW}(402)$ on key 402 (in red).

Algorithm 2 The PolySI algorithm for checking SI (the full version)

```
procedure CheckSI(H)
    if H ⊭ Int ∨ AbortedReads ∨ IntermediateReads
        return false
    CreateKnownGraph(H)
    GenerateConstraints(H)
    if ¬PruneConstraints()
        return false
    SAT-Encode()
    return MonoSAT-Solve()

procedure CreateKnownGraph(H)
    for all T, S ∈ 𝒯 such that T --SO--> S
        E_G ← E_G ∪ (T, S, SO)
    for all T, S ∈ 𝒯 such that T --WR--> S
        E_G ← E_G ∪ (T, S, WR)

procedure GenerateConstraints(H)
    for all x ∈ Key
        for all T, S ∈ WriteTx_x such that T ≠ S
            either ← {(T, S, WW)} ∪ ⋃_{T' ∈ WR(x)(T)} {(T', S, RW)}
            or ← {(S, T, WW)} ∪ ⋃_{S' ∈ WR(x)(S)} {(S', T, RW)}
            C_G ← C_G ∪ {⟨either, or⟩}

procedure SAT-Encode()
    for all v_i, v_j ∈ V_G such that i ≠ j
        Create two Boolean variables BV_{i,j} and BV^I_{i,j}
        BV ← BV ∪ {BV_{i,j}, BV^I_{i,j}}
    for all (v_i, v_j) ∈ E_G                        ▷ encode the known graph G
        CL ← CL ∪ {BV_{i,j} = True}
    for all ⟨either, or⟩ ∈ C_G                      ▷ encode the constraints G
        CL ← CL ∪ { ( ⋀_{(v_i,v_j,_) ∈ either} BV_{i,j} ∧ ⋀_{(v_i,v_j,_) ∈ or} ¬BV_{i,j} ) ∨
              ( ⋀_{(v_i,v_j,_) ∈ or} BV_{i,j} ∧ ⋀_{(v_i,v_j,_) ∈ either} ¬BV_{i,j} ) }
    Dep ← G|_{SO_G ∪ WR_G ∪ WW_G}
    E_Dep ← E_Dep ∪ {(_, _, WW) ∈ either ∪ or | ⟨either, or⟩ ∈ C_G}
    AntiDep ← G|_{RW_G}
    E_AntiDep ← E_AntiDep ∪ {(_, _, RW) ∈ either ∪ or | ⟨either, or⟩ ∈ C_G}
    CL ← CL ∪ { BV^I_{i,j} = (BV_{i,j} ∧ (v_i, v_j, _) ∈ E_Dep) ∨
          ( ⋁_{(v_i,v_k,_) ∈ E_Dep, (v_k,v_j,_) ∈ E_AntiDep} BV_{i,k} ∧ BV_{k,j} ) | v_i, v_j ∈ V_G }   ▷ encode the induced SI graph I of G

procedure PruneConstraints()
    repeat
        Dep ← G|_{SO_G ∪ WR_G ∪ WW_G}
        AntiDep ← G|_{RW_G}
        KI ← Dep ∪ (Dep ; AntiDep)
        reachability ← Reachability(KI)
        for all cons ← ⟨either, or⟩ ∈ C_G
            for all (from, to, type) ∈ either        ▷ for the "either" possibility
                if type = WW
                    if (to, from) ∈ reachability
                        C_G ← C_G \ {cons}
                        E_G ← E_G ∪ or
                        break the "for all (from, to, type) ∈ either" loop
                else                                 ▷ type = RW
                    for all prec ∈ V_Dep such that (prec, from, _) ∈ E_Dep
                        if (to, prec) ∈ reachability
                            C_G ← C_G \ {cons}
                            E_G ← E_G ∪ or
                            break the "for all (from, to, type) ∈ either" loop
            for all (from, to, type) ∈ or            ▷ for the "or" possibility
                if type = WW
                    if (to, from) ∈ reachability
                        if cons ∉ C_G                ▷ neither "either" nor "or" is possible
                            return False
                        C_G ← C_G \ {cons}
                        E_G ← E_G ∪ either
                        break the "for all (from, to, type) ∈ or" loop
                else                                 ▷ type = RW
                    for all prec ∈ V_Dep such that (prec, from, _) ∈ E_Dep
                        if (to, prec) ∈ reachability
                            if cons ∉ C_G            ▷ neither "either" nor "or" is possible
                                return False
                            C_G ← C_G \ {cons}
                            E_G ← E_G ∪ either
                            break the "for all (from, to, type) ∈ or" loop
    until C_G remains unchanged
    return True

procedure MonoSAT-Solve()
    solver ← MonoSAT-Solver(BV, CL)
    return Solve(solver, I is acyclic)
```

Following the same procedure as in the MariaDB-Galera example, PolySI resolves the outstanding dependencies if no undesired cycles arise; see Figure 12(c). For example, we have T:(4,172)$\xrightarrow{\text{WW}}$T:(10,471) (in blue) as there would otherwise be a cycle T:(10,471)$\xrightarrow{\text{WW}}$T:(4,172)$\xrightarrow{\text{WR}}$T:(10,467)$\xrightarrow{\text{SO}}$T:(10,471). The final violating scenario is shown in Figure 12(d) where two impossible (dashed) dependencies in Figure 12(c) have been eliminated.

This violation occurs as the causality order is violated, which is a happens-before relationship between any two transactions in a given history [31, 32].[8] More specifically, transaction T:(9,428) causally depends on transaction T:(10,471) (via the counterclockwise path in the right subgraph) and transaction T:(10,471) causally depends on transaction T:(4,172) (via the clockwise path in the left subgraph). Hence, transaction T:(9,428) should have fetched the value 7 of key 656 written by transaction T:(10,471), instead of the value 3 of transaction T:(4,172), to respect the causality order.

[8]Intuitively, transaction T causally depends on transaction S if any of the following conditions holds: (i) T and S are issued in the same session and S is executed before T; (ii) T reads the value written by S; and (iii) there exists another transaction R such that T causally depends on R which in turn causally depends on S.

D.2 A Causality Violation Found in YugabyteDB

Figure 13 shows how Algorithm 3 interprets a causality violation found in YugabyteDB. MonoSAT reports an undesired cycle $T : (0,7) \xrightarrow{\text{WW}(10)} T : (1,15) \xrightarrow{\text{WR}(13)} T : (0,6) \xrightarrow{\text{SO}} T : (0,7)$; see Figure 13a. In this scenario, no transactions observe values that have been causally overwritten; consider the only transaction $T : (0,6)$ in Figure 13a that reads. To locate the cause of

Algorithm 3 The interpretation algorithm.

$\mathcal{H} = (\mathcal{T}, \mathrm{SO})$: the original history
$G = (\mathrm{V\_1}, \mathrm{E\_1}, \mathrm{C\_1})$: the polygraph
$C = (\mathrm{V\_2}, \mathrm{E\_2})$: The cycle found by PolySI

procedure INTERPRET($\mathcal{H}, G, C$)
  $\mathrm{Graph_{recovered}} \leftarrow$ RESTORE($\mathcal{H}, G, C$)
  $\mathrm{Graph_{tagged}} \leftarrow$ RESOLVE($\mathrm{Graph_{recovered}}, \mathcal{H}, G, C$)
  $\mathrm{Graph_{finalized}} \leftarrow$ FINALIZE($\mathrm{Graph_{tagged}}$)
  return $\mathrm{Graph_{recovered}}, \mathrm{Graph_{tagged}}, \mathrm{Graph_{finalized}}$

procedure RESTORE($\mathcal{H}, G, C$)
  $\mathrm{Graph_{recovered}} \leftarrow$ FIND_ACS(C)
  for all $(u \xrightarrow{\mathrm{RW}} v) \in \mathrm{E\_2}$
    for all $c \in \mathrm{C\_1}$
      if $((u \xrightarrow{\mathrm{RW}} v) \in \mathrm{c}) \wedge (\exists w, (w \xrightarrow{\mathrm{WW}} v) \in \mathrm{c})$
        $\mathrm{Graph_{recovered}} \leftarrow \mathrm{Graph_{recovered}} \cup \{w \xrightarrow{\mathrm{WW}} v\} \cup \{w \xrightarrow{\mathrm{WR}} u\}$
  return $\mathrm{Graph_{recovered}}$

procedure FIND_ACS($\mathrm{Graph_{acs}}, G$)
  graph_size $\leftarrow$ number of edges in $\mathrm{Graph_{acs}}$
  minimal_extend_deps $\leftarrow +\infty$
  minimal_acs $\leftarrow$ empty
  for all dep $\in (\mathrm{E\_2} \cap \mathrm{c}.either)$, where $\mathrm{c} \in \mathrm{C\_1}$
    if $(\exists \mathrm{dep}' \in \mathrm{c}.or, \mathrm{dep}' \in \mathrm{Graph_{recovered}})$ continue
    for all $\mathrm{dep}' \in \mathrm{c}.or$
      for all (cycle $(\mathrm{V}', \mathrm{E}') \subseteq G) \wedge (\mathrm{dep}' \in \mathrm{E}')$
        $\mathrm{Graph_{tested}} \leftarrow \mathrm{Graph_{recovered}} \cup (\mathrm{V}', \mathrm{E}')$
        (extendDeps, extendDeps) $\leftarrow$ FIND_ACS($\mathrm{Graph_{tested}}, G$)
        if extendDeps < minimal_extend_deps
          minimal_extend_deps $\leftarrow$ extendDeps
          minimal_acs $\leftarrow \mathrm{graph_{extended}}$
  for all dep $\in (\mathrm{E\_2} \cap \mathrm{c}.or)$, where $\mathrm{c} \in \mathrm{C\_1}$
    if $(\exists \mathrm{dep}' \in \mathrm{c}.either, \mathrm{dep}' \in \mathrm{Graph_{recovered}})$ continue
    for all $\mathrm{dep}' \in \mathrm{c}.either$
      for all (cycle $(\mathrm{V}', \mathrm{E}') \subseteq G) \wedge (\mathrm{dep}' \in \mathrm{E}')$
        $\mathrm{Graph_{tested}} \leftarrow \mathrm{Graph_{recovered}} \cup (\mathrm{V}', \mathrm{E}')$
        (extendDeps, extendDeps) $\leftarrow$ FIND_ACS($\mathrm{Graph_{tested}}, G$)
        if extendDeps < minimal_extend_deps
          minimal_extend_deps $\leftarrow$ extendDeps
          minimal_acs $\leftarrow \mathrm{graph_{extended}}$
  return (minimal_extend_deps, minimal_acs + graph_size)

procedure RESOLVE($\mathrm{Graph_{recovered}}, \mathcal{H}, G, C$)
  $\mathrm{Graph_{tagged}} \leftarrow \mathrm{Graph_{recovered}}$
  for all dep $\in \mathrm{Graph_{tagged}}$
    if dep $\in \mathrm{E\_1}$
      dep.$tag \leftarrow$ 'certain'
    else
      dep.$tag \leftarrow$ 'uncertain'
  while $\mathrm{Graph_{tagged}}$ was changed in the last loop
    for all dep $\in \mathrm{Graph_{tagged}}$
      if (dep.tag = 'uncertain') $\wedge$ (dep in a cycle $(\mathrm{V}', \mathrm{E}')$)
        if $\forall \mathrm{dep}' \in \mathrm{E}' \wedge \mathrm{dep}' \neq \mathrm{dep}$, $\mathrm{dep}'.tag$ = certain
          dep.$tag \leftarrow$ 'uncertain'
          for all $\mathrm{dep}_{opposite}$, where $\{\mathrm{dep}/\mathrm{dep}_{opposite}\} \in \mathrm{C\_1}$
            $\mathrm{dep}_{opposite}.tag \leftarrow$ 'certain'
  return $\mathrm{Graph_{tagged}}$

procedure FINALIZE($\mathrm{Graph_{tagged}}$)
  $\mathrm{Graph_{finalized}} \leftarrow \mathrm{Graph_{tagged}}$
  for all dep $\in \mathrm{Graph_{finalized}}$
    if dep.tag = 'uncertain'
      $\mathrm{Graph_{finalized}} \leftarrow \mathrm{Graph_{finalized}} \setminus \{\mathrm{dep}\}$
  return $\mathrm{Graph_{finalized}}$

the causality violation, PolySI first finds the (only) "missing" transaction $T : (0, 9)$ (colored in green) and the associated dependencies, as shown in Figure 13b. The WW and RW dependencies are uncertain at this moment (represented by dashed arrows), while the WR dependency is certain (represented by solid arrows, colored in blue). PolySI then restores the violating scenario by resolving such uncertainties. Specifically, as shown in Figure 13c, PolySI determines that $W(10, 3)$ of transaction $T : (0, 7)$ was actually installed before $W(10, 26)$ of transaction $T : (1, 15)$, i.e., $T : (0, 7) \xrightarrow{\mathrm{WW}(10)} T : (1, 15)$. Otherwise, the other direction of dependency $T : (1, 15) \xrightarrow{\mathrm{WW}(10)} T : (0, 7)$ would enforce the dependency $T : (0, 9) \xrightarrow{\mathrm{RW}(10)} T : (0, 7)$, which would create an undesired cycle with the known dependency $T : (0, 7) \xrightarrow{\mathrm{SO}} T : (0, 9)$. Finally, PolySI finalizes the violating scenario by removing the remaining uncertainties from Figure 13c to obtain Figure 13d.

Thanks to the participation of transaction $T : (0, 9)$, the cause of the causality violation becomes clear: Transaction $T : (0, 7)$ causally depends on transaction $T : (1, 15)$, via $T : (1, 15) \xrightarrow{\mathrm{WR}(13)} T : (0, 6) \xrightarrow{\mathrm{SO}} T : (0, 7)$. However, transaction $T : (0, 9)$ following transaction $T : (0, 7)$ on the same session reads the value 26 of key 10 from transaction $T : (1, 15)$, which should have been overwritten by transaction $T : (0, 7)$.

E MINIMALITY OF COUNTEREXAMPLES

In this section we show that our interpretation algorithm actually returns a *minimal* counterexample which contains the cycle found by MonoSAT and is just informative to help understand the violation.

E.1 Definitions

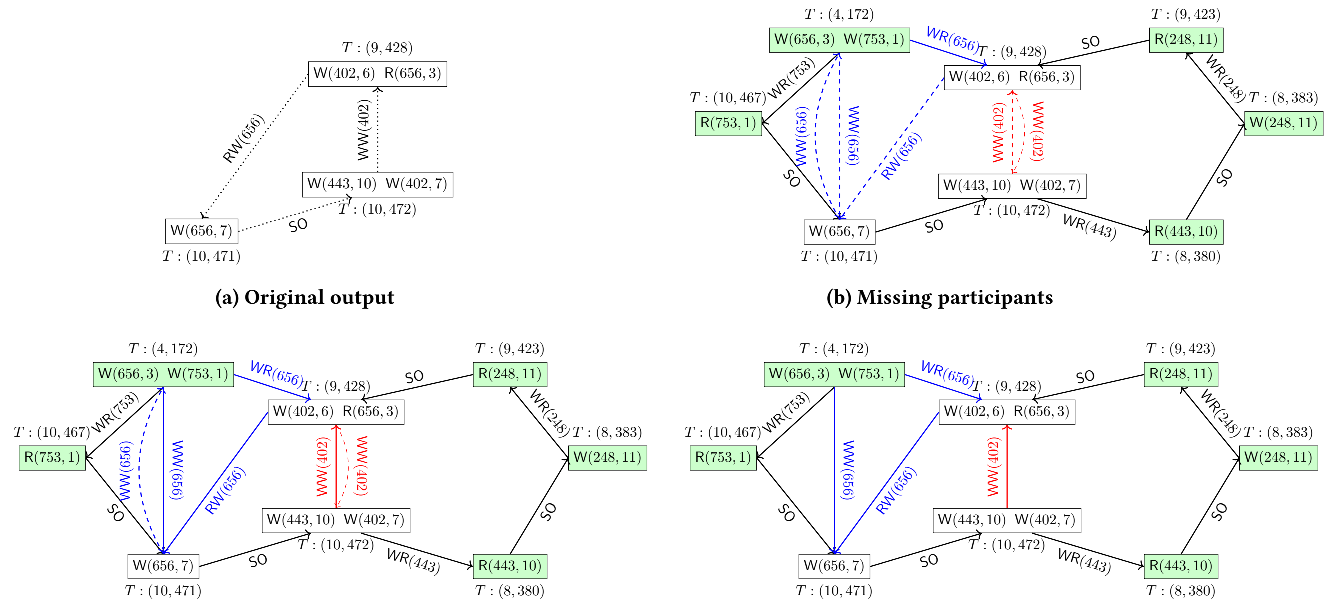
Definition 21. A ***violation*** is a polygraph that fails the checking of PolySI.

Let C be the cycle found by MonoSAT in polygraph $G = \{V, E, Cons\}$. Consider the violation $vio = \{V', E', Cons'\}$ returned by the interpretation algorithm. It satisfies the following properties:

- $vio \subseteq G$, which means $V' \subseteq V, E' \subseteq E, Cons' \subseteq Cons$
- $C \subseteq vio$.
- vio fails the checking of PolySI.

We call vio a violation with respect to $\{G, C\}$.

Definition 22. A violation vio is ***minimal***, if removing any dependency edge from vio makes it no longer a violation.



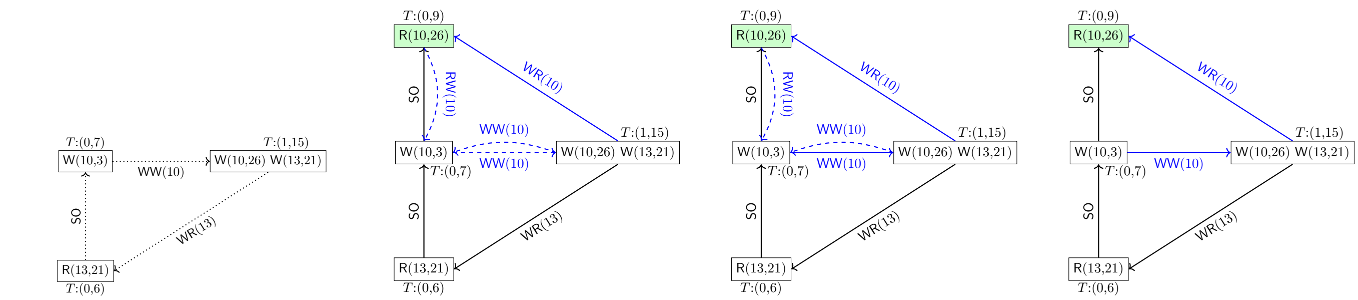
(a) Original output

(b) Missing participants

(c) Recovered scenario

(d) Finalized scenario

Figure 12: Causality violation: the SI anomaly found in Dgraph. Dashed and solid arrows represent uncertain and certain dependencies, respectively. Recovered transactions are colored in green. The core dependencies involved in the two sub-scenarios are colored in red and blue, respectively.



(a) Original output

(b) Missing participants

(c) Recovered scenario

(d) Finalized scenario

Figure 13: Causality violation: the SI anomaly found in YugabyteDB. The recovered dependencies are colored in blue with dashed and solid arrows indicating uncertain and certain dependencies, respectively. The (only) missing transaction is colored in green.

There may be more than one minimal violation with respect to $\{G, C\}$. And we name the one who has the least number of dependencies ***the minimal counter example with respect to*** $\{G, C\}$, which we expect to get.

Definition 23. Given a polygraph G and a cycle C, there may be more than one minimal violations based on $\{G, C\}$. We name the one which has the least number of dependencies ***the minimal counterexample with respect to*** $\{G, C\}$.

E.2 Patterns of Minimal Counterexamples

A minimal counter example is a special polygraph, which consists of several cycles. We define a data structure called "adjoining cycle set", and prove that a minimal counterexample is exactly a minimal complete adjoining cycle set.

Definition 24. A set of cycles acs is called ***adjoining cycle set***, if $\forall$ cycle $C_1 \in acs$, $\exists$ a constraint $cons = \{dep_1, .../dep_2, ...\}$ and a cycle $C_2 \in acs$, s.t. $dep_1 \in C_1$ and $dep_2 \in C_2$.

- C_1 is the adjoining cycle of C_2, while C_2 is the adjoining cycle of C_1.
- If $\forall$ constraint $cons$, the two choice of $cons$ both have dependencies in acs or neither have dependencies in acs, then acs is ***complete***.

Definition 24 describes the data structure called ***adjoining cycle set***. Figure 14 gives an example of adjoining cycle set. When we say a adjoining cycle set is complete, it means no more cycles can be added to the set and each constraint included in the set has two choices. $\{C1, C3\}$ is an adjoining cycle set, but it is incomplete because there exists a constraint (the green one) that has only one choice in $\{C1, C3\}$. $\{C1, C2, C3\}$ is a complete adjoining cycle set, since each constraint has two choices in $\{C1, C2.C3\}$.

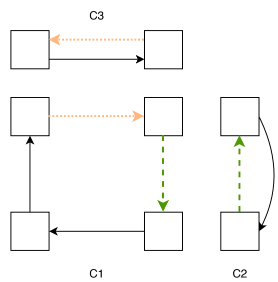


Figure 14: Complete and incomplete adjoining cycle sets. $\{C1, C2, C3\}$ is a complete adjoining cycle set. $\{C1, C2\}$ and $\{C1, C3\}$ are incomplete cycle sets.

Lemma 25. *A minimal violation is exactly a complete adjoining cycle set.*

Proof. If there exists a violation vio, which do not contain any complete adjoining cycle set. It contain at least one incomplete cycle set acs_1, because a violation must have at least one cycle. So there exist a cycle $C_1 \in acs_1$ and a constraint $cons = \{dep_1..., /...\}$, where $dep_1 \in acs_1$ and all dependencies in the other choice of $cons$ do not contained by acs_1. Then we let $cons$ choose the other choice, and then dep_1 will disappear and C_1 will not be a cycle any more. Next we focus on $acs_1 - \{C_1\}$, (if acs_1 has more than one cycle), we will get another incomplete cycle group $acs_2 = acs_1 - \{C_1\}$. Keep doing this recurisely, all cycles in acs_1 will be broken. And this polygraph will not be a violation.

If there exists two complete adjoining cycle group in vio, then we can delete one complete adjoining cycle group and the left polygraph is still a violation. So it is not a minimal violation.

In conclusion, a minimal violation vio contains exactly one complete adjoining cycle set acs. Since both choices of each constraint are in one cycle, acs is a violation that cannot pass the verification of PolySI. vio is a minimal violation, $acs \subseteq vio$ and acs is a violation, then it is trivial that $vio = acs$. □

Definition 26. A complete adjoining cycle set acs is called ***minimal complete adjoining cycle set containing cycle*** C if

- $C \in acs$
- $\forall$ complete adjoining cycle set acs' containing C, the number of dependencies in $acs' \geq$ that in acs

Theorem 27 (Minimal counterexample). *A minimal complete adjoining cycle set containing cycle C is exactly the minimal counter example with respect to $\{G, C\}$.*

Proof. A minimal complete adjoining cycle set containing C is one of complete adjoining cycle sets containing C which has the least number of dependencies. By lemma 25, it is the smallest minimal violation containing cycle C. So it is one of minimal violations with respect to $\{G, C\}$, and has the least number of dependencies. By definition 23, it is exactly the minimal counter example with respect to $\{G, C\}$. □

E.3 Restoring Minimal Patterns

The core idea of our interpretation algorithm is to restore the ***minimal complete adjoining cycle set containing*** C. Function *Find_ACS* is designed to do this task. (line 3:13)

For each WW or RW dependency in an undesired cycle, it checks each dependency in the other direction (if missing), and restores the undesired cycles containing these dependencies (line 3:17 - line 3:34). Doing this recursively (line 3:22 and line 3:31), the function will detect all of the adjoining cycle sets that contain $\mathrm{Graph}_{\mathrm{acs}}$. The minimal one will be returned.

The algorithm uses the brute force and sounds inefficient. However, it works well in practical experiments. From our experience of verification, when it tries to detect a cycle, there usually exists a small cycle with only one WW or RW dependency. For example, in the violation we found in figure 12 a, we first need to find a adjoining cycle from the dependency $T : (9, 428) \rightarrow T : 10, 471$. There exists a simple cycle $T : (10, 471) \rightarrow T : (4, 172) \rightarrow T : (10, 467) \rightarrow T : (10, 471)$. This cycle has a small size and only has one WW dependency, which means the function does not need to detect more cycles recursively in order to restore a complete adjoining cycle set, and can pay no attention to other possible adjoining cycle sets that contain more than 3 dependencies.

However, there still exist the worst cases that the interpretation algorithm cannot return the minimal counter example quickly. But in this case, we can be more patient to wait for it. Since we have already known the result that there exists a violation, and we just need to wait for the minimal counter example. And if it still costs too much time, we can interrupt it and output a minimal violation we have found instead of the smallest one, after the interpretation algorithm has run enough time.

Theorem 28 (Minimality). *PolySI always returns a minimal counterexample with respect to G and C, with G the polygraph built from a history and C the cycle output by MonoSAT.*

Proof. Our interpretation algorithm will return a 'minimal complete adjoining cycle set' containing cycle C, which is exactly the minimal counter example. □

Now we have restored the minimal counter example. However, to help understand the violation, we still restore some more dependencies. For each rw dependency appears in violations, it is caused by a wr dependency and a ww dependency. If we delete the wr dependency and ww dependency, it is difficult to understand the rw dependency. So for each rw dependency in a minimal counterexample, we recover the related wr and ww dependencies to the violation (Alg3:line8-11). This is very useful in our experiments. For example, in the violation we found in figure 5, the transaction $T : (1, 4)$ is recovered from two rw dependencies. Without the recovered transaction, it is difficult to understand the violation.

F PERFORMANCE EVALUATION OF POLYSI-LIST

Figure 15 shows the performance evaluation of PolySI-List, the extension to PolySI for handling Elle-like logs/histories with the "list" data structure.

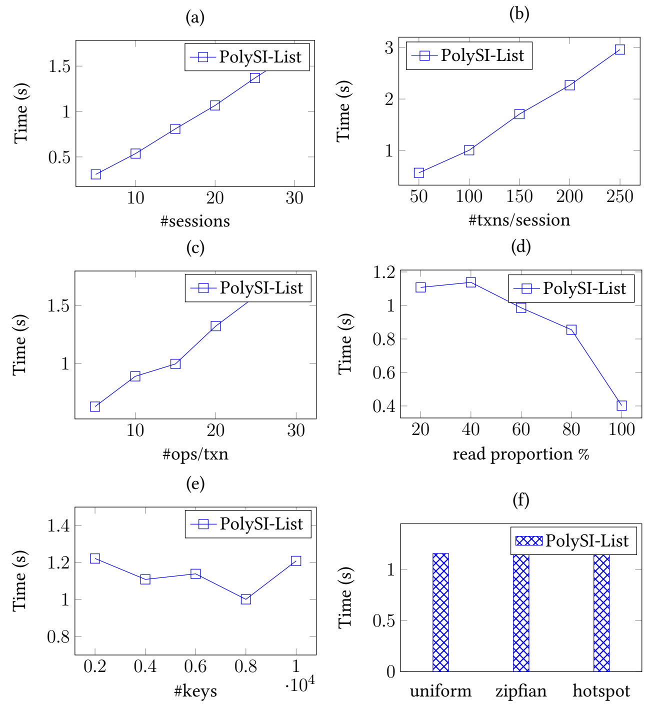


Figure 15: Performance evaluation of PolySI-List.