Fast Verification of Strong Database Isolation

Zhiheng Cai
Tsinghua University

Si Liu
ETH Zurich

Hengfeng Wei
Nanjing University

Yuxing Chen
Tencent Inc.

Anqun Pan
Tencent Inc.

Abstract

Strong isolation guarantees, such as serializability and snapshot isolation, are essential for maintaining data consistency and integrity in modern databases. Verifying whether a database upholds its claimed guarantees is increasingly critical, as these guarantees form a contract between the vendor and its users. However, this task is challenging, particularly in black-box settings, where only observable system behavior is available and often involves uncertain dependencies between transactions.

In this paper, we present VeriStrong, a fast verifier for strong database isolation. At its core is a novel formalism called hyperpolygraphs, which compactly captures both certain and uncertain transactional dependencies in database executions. Leveraging this formalism, we develop sound and complete encodings for verifying both serializability and snapshot isolation. To achieve high efficiency, VeriStrong tailors SMT solving to the characteristics of database workloads, in contrast to prior general-purpose approaches. Our extensive evaluation across diverse benchmarks shows that VeriStrong not only significantly outperforms state-of-the-art verifiers on the workloads they support, but also scales to large, general workloads beyond their reach, while maintaining high accuracy in detecting isolation anomalies.

PVLDB Reference Format:
Zhiheng Cai, Si Liu, Hengfeng Wei, Yuxing Chen, and Anqun Pan. Fast Verification of Strong Database Isolation . PVLDB, 1(1): XXX-XXX, 2026.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CzxingcHen/VeriStrong.

1 Introduction

Strong isolation levels or guarantees—the I in ACID [38]—such as Serializability and Snapshot Isolation, are essential for ensuring data consistency and integrity in modern databases. These guarantees prevent subtle concurrency anomalies, such as lost updates, that can lead to incorrect application behavior or data corruption. Numerous production databases, including PostgreSQL and FoundationDB, support these strong isolation levels. Once a database is deployed, its promised isolation guarantees become a binding contract between the vendor and its users. This raises a critical question: *How can we be sure that the database upholds its promise?*

Answering this question is nontrivial. Modern databases have large codebases that are often inaccessible or too complex to reason about, even when available. This makes full verification of isolation guarantees practically infeasible. Black-box verification [5, 35] offers an effective alternative: a verifier collects execution histories of database transactions as an external observer and checks whether these histories satisfy the isolation level in question.

However, verifying database histories remains challenging. The verification problem for strong isolation levels has been proven to be NP-hard, even for verifying a single history [5]. This complexity arises from the fact that, as an external observer of a black-box database, a verifier must *infer* the internal execution order of transactions, e.g., determining which transaction wrote an earlier version and which one wrote a later one. This challenge is further compounded by practical constraints: given limited time and memory, only a finite number of "guesses" can be made by the verifier, and thus only a finite number of histories can be verified. Yet, examining more histories is highly desirable, either to increase the likelihood of uncovering anomalies or to strengthen confidence in their absence.

Recent years have seen significant progress in accelerating the verification of strong isolation guarantees [5, 16, 20, 35, 37, 40]. Representative verifiers include Cobra [35] for Serializability, PolySI [16] and Viper [40] for Snapshot Isolation, and Elle [20] for both (a detailed discussion is provided in Section 8). Despite these advances, verification efficiency remains a key limitation. Many verifiers have attempted to reduce verification overhead by leveraging advanced SMT (Satisfiability Modulo Theories) techniques [4]. However, general-purpose SMT solvers are prone to missing optimization opportunities specific to database verification.

Beyond this, two additional challenges make efficient verification even more difficult. First, all existing verifiers assume that transaction histories contain *unique write values*, meaning each read can be deterministically matched to a single write. This assumption greatly simplifies the task of inferring the internal execution order—reducing the verification problem from NP-hard to polynomial time—and serves as a key enabler of Elle's design in particular. However, this assumption does not hold in many real-world applications, where it is common for different transactions to write the same value. For instance, in an e-commerce platform, multiple users may submit identical orders (e.g., for the same quantity of an item). Likewise, standard database benchmarks such as Twitter [19] and RUBiS [34] include *duplicate write values* by default. In the black-box setting we consider, supporting such general workloads is highly desirable and essential for practical applicability.

Moreover, achieving both soundness (no false alarms) and completeness (no missed bugs) without compromising efficiency remains a challenging goal in verifying strong isolation guarantees. In particular, identifying all anomalies in a history is computationally expensive, yet critical: overlooking even a single anomaly can require significant effort to rediscover. This challenge is further exacerbated by recent findings [26, 31], which reveal that certain anomalies arise *only* in the presence of duplicate write values. In

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

such cases, applying existing verifiers risks both false negatives and false positives, undermining the reliability of verification results.

Our Solution. We present a novel black-box verifier, VeriStrong, that checks database histories against the two most widely used strong isolation guarantees, i.e., Serializability and Snapshot Isolation. VeriStrong tackles the aforementioned challenges through an end-to-end verification pipeline. This includes (i) representing and encoding general database histories, (ii) establishing a sound and complete correspondence between the encoding and the verification algorithm, and (iii) optimizing checking efficiency using SMT techniques tailored for strong isolation verification.

At the core of our solution is a new formalism called *hyper-polygraphs*, which captures both certain transactional dependencies (e.g., session order) and uncertain ones (e.g., version order). In black-box settings, uncertainties can also arise from ambiguous read-from relations caused by duplicate write values—an aspect that existing formalisms, such as polygraphs [32, 38] and their variants [16, 35, 40], cannot adequately express.

Hyper-polygraphs offer two key advantages. First, they provide a compact and expressive means of encoding transactional dependencies, making them particularly amenable to SMT-based verification. Second, they are generic, capable of representing a wide range of dependency-based isolation theories [1, 5, 8]. While our main focus is on formalizing Serializability and Snapshot Isolation under Adya's theory [1], we also show in the Appendix [7] how hyper-polygraphs naturally extend to capture the recent axiomatic model proposed by Biswas and Enea [5].

Building on this formalism, we develop sound and complete encodings for verifying both strong isolation guarantees, providing a rigorous theoretical foundation for reliable SMT-based verification. This enables our verifier to accurately (re)discover isolation anomalies in production databases (e.g., MariaDB), particularly those arising from duplicate write values that prior tools fail to handle.

To make verification fast, our key insight is that leveraging workload-specific knowledge can greatly boost SMT solving efficiency. To this end, we propose two domain-specific optimizations. First, we offload part of the solver's effort to a lightweight preprocessing phase by proactively resolving small conflict patterns. This reduces expensive backtracking during solving by eliminating many infeasible search paths early and strikes a practical balance between preprocessing and solving costs, improving overall performance.

Second, we design a polarity picking strategy—i.e., deciding whether to explore a dependency—guided by a dynamically maintained pseudo-topological order during SMT solving. Unlike general-purpose solvers that make such decisions "blindly", we leverage known partial orders, such as session order and certain read-from edges, to align these decisions with the likely transaction schedule enforced by the database. This alignment steers the solver away from conflict-prone search paths, thereby enhancing efficiency.

Contributions. Overall, we make the following contributions.

(1) We introduce hyper-polygraphs, a new formalism that compactly characterizes isolation levels, including Serializability and Snapshot Isolation, over general database histories.
(2) We formally establish the soundness and completeness of our new characterizations, grounded in Adya's theory, providing a theoretical foundation for reliable SMT-based verification.

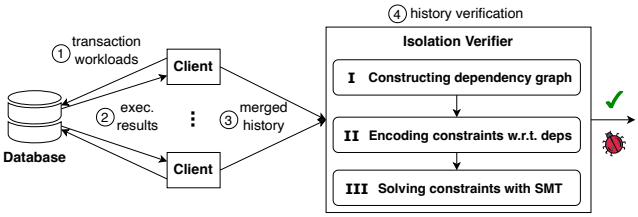


Figure 1: Workflow for verifying isolation guarantees in deployed databases. Steps II and III are specific to SMT-based approaches (see also Sections 4 and 6).

(3) We develop novel isolation verification algorithms by tailoring SMT solving to the characteristics of database workloads, incorporating optimizations that reduce solver overhead.
(4) We implement our approach in VeriStrong and assess it extensively on a variety of benchmarks. Results show that VeriStrong not only substantially outperforms state-of-the-art verifiers on the workloads they support, but also scales to large, general workloads beyond their reach, while maintaining high accuracy in detecting isolation anomalies.

This work also complements recent advances in isolation verification along two dimensions. First, it complements prior efforts [23, 29] on black-box verification of weaker levels like Read Committed, which are known to be less complex to verify [5]. Second, it targets deployed databases, complementing recent work on verifying isolation in their designs [13] and implementations [28].

We focus on Serializability as the running example throughout the paper, unless noted otherwise. We defer to the Appendix [7] how our approach extends to verifying Snapshot Isolation.

2 Background

Black-box verification of database isolation guarantees involves four key steps, as illustrated in Figure 1. In Step ①, clients issue transactional requests to the database. In Step ②, each client records the corresponding execution results, including the values read or written and the status of each transaction (either committed or aborted). Next, in Step ③, the logs from all clients are merged into a single history, which is then passed to an isolation verifier. Finally, in Step ④, the verifier analyzes the history to determine whether it satisfies the specified isolation guarantee.

How does the verifier make this decision? It constructs a dependency graph based on, e.g., Adya's theory [1] that is the *de facto* formalization of isolation guarantees. The verifier then searches the graph for cycles, which indicate violations of guarantees such as Serializability. Next, we recall the formal definitions underlying the history verification steps I–III from [16].

Relations. 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 the 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.

Transactions. We consider a distributed key-value store that manages a set of keys $\mathsf{Key} = \{x, y, z, \dots\}$; each key is associated with a value from a set Val. The set of operations, denoted by Op, consists of both read and write operations: $\mathsf{Op} = \{\mathsf{R}_\iota(x, v), \mathsf{W}_\iota(x, v) \mid \iota \in \mathsf{OpId}, x \in \mathsf{Key}, v \in \mathsf{Val}\}$, where OpId is the set of operation identifiers. For simplicity, operation identifiers may be omitted. We use _ to represent an irrelevant value in an operation, e.g., $\mathsf{R}(x, \_)$, which is implicitly existentially quantified.

Clients interact with the key-value store by issuing transactions.

Definition 2.1. A *transaction* is a pair (O, po), where $O \subseteq \mathsf{Op}$ is a finite, non-empty set of operations; $\mathsf{po} \subseteq O \times O$ is a strict total order, referred to as the *program order*, which indicates the execution order of operations within the transaction.

For a transaction T, we write $T \vdash \mathsf{W}(x, v)$ if T writes to the key x and v is the last written value, and $T \vdash \mathsf{R}(x, v)$ if T reads from x before writing to it, and v is the value returned by the first such read. We also define $\mathsf{WriteTx}_x = \{T \mid T \vdash \mathsf{W}(x, \_)\}$ to denote the set of transactions that write to x.

Histories. Transactions are grouped into client *sessions*, where each session consists of a sequence of transactions. We use *histories* to record the client-visible outcomes of these transactions.

Definition 2.2. A *history* is a pair $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$, where $\mathcal{T}$ is a set of transactions with disjoint sets of operations; $\mathsf{SO} \subseteq \mathcal{T} \times \mathcal{T}$ is the *session order*, which is a union of strict total orders over disjoint sets of $\mathcal{T}$, each corresponding to a distinct client session.

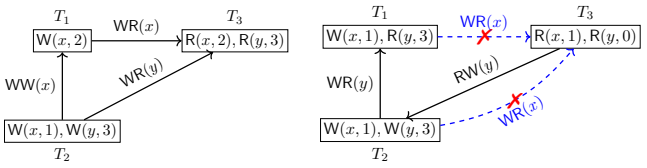
In line with existing formal models [1, 5, 8], we consider only committed transactions in a history; aborted transactions are handled separately. We also assume that every history contains a special transaction $T_\perp$ that writes the initial values of all keys.[1] This transaction precedes all the other transactions in SO across sessions. Note that existing work commonly assumes UniqueValue histories, where each write to the same key assigns a distinct value, whereas real-world scenarios often involve DuplicateValue writes.

Dependency Graphs. A dependency graph extends a history by introducing three types of relations (or edges), WR, WW, and RW, each capturing a different kind of dependency between transactions. These relations underlies the formalization of isolation guarantees in the style of Adya [1, 8]. The WR relation connects a transaction that reads a value to the transaction that wrote it. The WW relation defines a strict total order, also referred to as the *version order* [1], among transactions that write to the same key. The RW relation is derived from WR and WW, associating a transaction that reads a value to the one that overwrites it based on the version order.

Definition 2.3. A *dependency graph* is a tuple $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, where $(\mathcal{T}, \mathsf{SO})$ is a history and

- $\mathsf{WR} : \mathsf{Key} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that
  - $\forall x \in \mathsf{Key}.\ \forall S \in \mathcal{T}.\ S \vdash \mathsf{R}(x, \_) \implies \exists! T \in \mathcal{T}.\ T \xrightarrow{\mathsf{WR}(x)} S$
  - $\forall x \in \mathsf{Key}.\ \forall T, S \in \mathcal{T}.\ T \xrightarrow{\mathsf{WR}(x)} S \implies T \neq S \wedge \exists v \in \mathsf{Val}.\ T \vdash \mathsf{W}(x, v) \wedge S \vdash \mathsf{R}(x, v)$.
- $\mathsf{WW} : \mathsf{Key} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that $\forall x \in \mathsf{Key}$, $\mathsf{WW}(x)$ is a strict total order on $\mathsf{WriteTx}_x$.

[1] In practice, we use multiple short, write-only transactions to initially populate the database. These transactions can be viewed as a single, logical write-only transaction.



(a) A serializable UniqueValue history. **(b) An unserializable DuplicateValue history.**

Figure 2: Capturing Serializability via dependency graphs.

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

We use $\exists!$ to denote unique existence. For any component of a dependency graph $\mathcal{G}$, such as WW, we write it as $\mathsf{WW}_\mathcal{G}$. When the key x in $T \xrightarrow{R(x)} S$ is irrelevant or clear from context, we write $T \xrightarrow{R} S$, where $R \in \{\mathsf{WR}, \mathsf{WW}, \mathsf{RW}\}$.

Characterizing Serializability. Serializability is characterized by the *existence* of an acyclic dependency graph, along with the internal consistency of each transaction, as axiomatized by Int [8]. The emphasis on existence is crucial: as external observers of a black-box database, verifiers cannot directly observe the internal execution order, e.g., WW between writes. Instead, they must infer a possible internal schedule. The existence of an acyclic dependency graph provides a witness that explains how the database could have scheduled the transactions in a way that satisfies Serializability.

In addition, the Int axiom ensures that a read on a key returns the same value as the most recent proceeding access—either a write or a read—to that key within the same transaction.

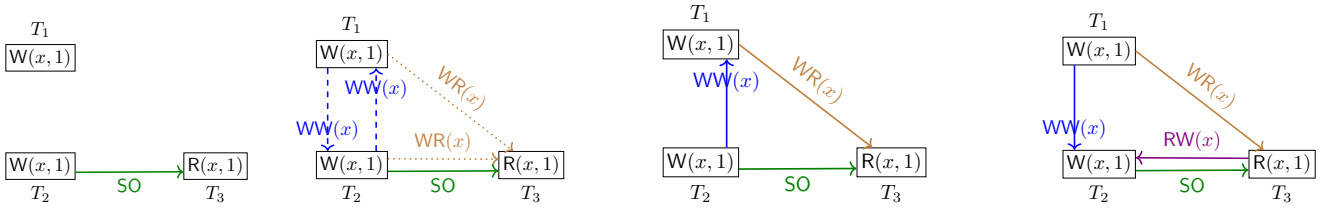
Theorem 2.4. *For a history* $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$,

$$\mathcal{H} \models \mathsf{SER} \iff \mathcal{H} \models \mathsf{Int} \wedge \exists \mathsf{WR}, \mathsf{WW}, \mathsf{RW}.\ \mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \wedge ((\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G} \cup \mathsf{RW}_\mathcal{G}) \text{ is acyclic}).$$

Example 2.5. Figure 2 illustrates the dependency-graph-based characterization of Serializability using two example histories. For simplicity, we omit $T_\perp$ in both cases. Figure 2a shows (the existence of) a serializable dependency graph with edges like $T_2 \xrightarrow{\mathsf{WW}(x)} T_1$, which are constructed from a UniqueValue history.

In contrast, it is impossible to build an acyclic dependency graph from the DuplicateValue history in Figure 2b. Specifically, for key y, we obtain the dependencies $T_2 \xrightarrow{\mathsf{WR}(y)} T_1$ and $T_3 \xrightarrow{\mathsf{RW}(y)} T_2$ (due to $T_\perp \xrightarrow{\mathsf{WR}(y)} T_3$ and $T_\perp \xrightarrow{\mathsf{WW}(y)} T_2$). Now consider the transaction from which $\mathsf{R}(x, 1)$ in T_3 reads its value. This leads to two possible WR edges (shown as dashed arrows): $T_1 \xrightarrow{\mathsf{WR}(x)} T_3$ and $T_2 \xrightarrow{\mathsf{WR}(x)} T_3$. However, in both cases, the resulting dependency graph contains a cycle.

Polygraphs. A dependency graph extending a history represents *one* possible interpretation of the dependencies among the involved transactions, which may or may not satisfy Serializability. To capture *all* possible dependency scenarios—allowing a verifier to



(a) General history $\mathcal{H}$. **(b) Hyper-polygraph $\mathcal{G}$ of $\mathcal{H}$.** **(c) An acyclic graph compatible with $\mathcal{G}$.** **(d) A cyclic graph compatible with $\mathcal{G}$.**

Figure 3: A general history $\mathcal{H}$, along with its hyper-polygraph $\mathcal{G}$ and two compatible graphs.

check whether any of them satisfies Serializability—state-of-the-art tools [16, 35, 40] utilize *polygraphs* [32, 38]. Intuitively, a polygraph can be seen as a compact representation of a family of dependency graphs, each corresponding to a different possible internal execution consistent with the observed history.

Definition 2.6. A polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ associated with a history $\mathcal{H} = (\mathcal{T}, \text{SO})$ is a directed labeled graph $(\mathcal{V}, \mathcal{E})$ called the *known graph*, together with a set C of *constraints*, such that

- $\mathcal{V}$ corresponds to all the transactions in $\mathcal{H}$;
- $\mathcal{E} = \{(T, S, \text{SO}) \mid T \xrightarrow{\text{SO}} S\} \cup \{(T, S, \text{WR}) \mid T \xrightarrow{\text{WR}} S\}$, where SO and WR, when used as the third component in a tuple, serve as edge labels (i.e., types of dependencies); and
- $C = \{(T_k, T_i, \text{WW}), (T_j, T_k, \text{RW}) \mid (T_i \xrightarrow{\text{WR}(x)} T_j) \wedge T_k \in \text{WriteTx}_x \wedge T_k \neq T_i \wedge T_k \neq T_j\}$.

Example 2.7. According to the above definition, the polygraph associated with the history in Figure 2a consists of a known graph with nodes $\{T_1, T_2, T_3\}$ and edges $\{T_1 \xrightarrow{\text{WR}} T_3, T_2 \xrightarrow{\text{WR}} T_3\}$, along with a set of two constraints $\{(T_2, T_1, \text{WW}), (T_3, T_2, \text{RW})\}$ capturing the uncertainty in version order between T_1 and T_2.

These constraints are then encoded as a SAT formula and solved using an SMT solver that supports theories like graph acyclicity, as we will see in Section 4. The acyclic dependency graph that satisfy Serializability in Figure 2a can be viewed as a solution produced by this solving process.

3 Hyper-Polygraphs

A dependency graph represents one possible execution of database transactions. A polygraph captures a family of such graphs by fixing the WR relation—an assumption that holds under UniqueValue, where each written value is unique and thus unambiguously matched by reads—while allowing uncertainty in the WW relation. However, this assumption, relied upon by all existing verifiers, does not always hold in practice, particularly under general workloads where the same value may be written multiple times, making read-from mappings ambiguous.

To address this representation problem, we introduce *hyper-polygraphs* as a generalization of polygraphs that additionally account for uncertainty in WR. Intuitively, a hyper-polygraph is a family of polygraphs, each corresponding to a distinct resolution of the WR relations, i.e., one polygraph per read-from mapping.

Definition 3.1. A hyper-polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ for a history $\mathcal{H} = (\mathcal{T}, \text{SO})$ is a directed labeled graph $(\mathcal{V}, \mathcal{E})$, referred to as the known graph, together with *a pair of constraint sets* $C = (C^{\text{WW}}, C^{\text{WR}})$, where

- $\mathcal{V}$ is the set of nodes, corresponding to the transactions in $\mathcal{H}$;
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \text{Type} \times \text{Key}$ is a set of edges, where each edge is labeled with a dependency type from $\text{Type} = \{\text{SO}, \text{WR}, \text{WW}, \text{RW}\}$ and a key from Key;[2]
- C^{WW} is a constraint set over uncertain version orders, defined as $C^{\text{WW}} = \left\{\{T \xrightarrow{\text{WW}(x)} S, S \xrightarrow{\text{WW}(x)} T\} \mid T \in \text{WriteTx}_x \wedge S \in \text{WriteTx}_x \wedge T \neq S\right\}$; and
- C^{WR} is a constraint set over uncertain read-from mappings, defined as $C^{\text{WR}} = \left\{ \bigcup_{T_i \vdash \text{W}(x,v)} \{T_i \xrightarrow{\text{WR}(x)} S\} \mid S \vdash \text{R}(x, v)\right\}$.

Given two transactions T and S, a type $\text{T} \in \text{Type}$, and a key $x \in \text{Key}$, we also write the edge (T, S, T, x) as $T \xrightarrow{\text{T}(x)} S$.

Example 3.2. Consider the general history $\mathcal{H}$ shown in Figure 3a. It consists of three transactions T_1, T_2, and T_3, with a session order edge $T_2 \xrightarrow{\text{SO}} T_3$. Both T_1 and T_2 write the same value 1 to the same key x. The hyper-polygraph $\mathcal{G}$ constructed from $\mathcal{H}$ includes the following two constraint sets:

- $C^{\text{WW}} = \left\{\{T_1 \xrightarrow{\text{WW}(x)} T_2, T_2 \xrightarrow{\text{WW}(x)} T_1\}\right\}$, representing the uncertainty in version order between T_1 and T_2 (shown as blue dashed edges in Figure 3b);
- $C^{\text{WR}} = \left\{(T_1 \xrightarrow{\text{WR}(x)} T_3,\ T_2 \xrightarrow{\text{WR}(x)} T_3)\right\}$, representing the uncertainty in read-from mappings for T_3 (shown as brown dotted edges in Figure 3b).

Characterizing Serializability using Hyper-Polygraphs. A hyper-polygraph for a DuplicateValue history can be viewed as a family of dependency graphs that are *compatible with* it, where each graph resolves the constraints by selecting exactly one WW or WR edge from the C^{WW} and C^{WR} constraint sets, respectively.

Definition 3.3. A directed labeled graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ is compatible with a hyper-polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ if

- $\mathcal{V}' = \mathcal{V}$;
- $\mathcal{E}' \supseteq \mathcal{E}$ such that $\forall x \in \text{Key}.\ \forall T, T', S \in \mathcal{V}'.\ (T', T, \text{WR}, x) \in \mathcal{E}' \wedge (T', S, \text{WW}, x) \in \mathcal{E}' \implies (T, S, \text{RW}, x) \in \mathcal{E}'$;
- $\forall C \in C^{\text{WW}}.\ |\mathcal{E}' \cap C| = 1$; and
- $\forall C \in C^{\text{WR}}.\ |\mathcal{E}' \cap C| = 1$.

[2] For edges of type SO, the key component is irrelevant.

Example 3.4. Figures 3c and 3d depict two graphs compatible with the hyper-polygraph $\mathcal{G}$ of the history $\mathcal{H}$. In Figure 3c, T_3 reads x from T_1, which overwrites the value written by T_2. In contrast, in Figure 3d, T_3 reads x from T_1, but the value is overwritten by T_2, resulting in $T_3 \xrightarrow{\mathrm{RW}(x)} T_2$.

Based on Theorem 2.4, we obtain the following hyper-polygraph-based characterization of Serializability; the proof is provided in the Appendix [7].

Theorem 3.5. *A history $\mathcal{H}$ satisfies* Serializability *if and only if $\mathcal{H} \models$ Int and there exists an acyclic graph compatible with the hyper-polygraph of $\mathcal{H}$.*

Example 3.6. The graph compatible with $\mathcal{G}$ in Figure 3d contains a cycle: $T_3 \xrightarrow{\mathrm{RW}(x)} T_2 \xrightarrow{\mathrm{SO}} T_3$. In contrast, the compatible graph in Figure 3c is acyclic. According to Theorem 3.5, we conclude that $\mathcal{H}_3$ satisfies Serializability.

4 Off-the-Shelf SMT Solving: A Baseline

To begin, we present a strong baseline approach for verifying Serializability using MonoSAT [4], an off-the-shelf SMT solver optimized for checking graph properties such as acyclicity. MonoSAT serves as the core engine for all state-of-the-art SMT-based isolation verifiers, including Cobra, PolySI, and Viper.

Given a history $\mathcal{H}$ and following the workflow in Figure 1, we first construct its hyper-polygraph. This involves extracting the transaction set $\mathcal{T}$, the session order SO, and unique WR dependencies for the known graph, along with possible WW and WR dependencies as constraints. We then focus on the two key steps: *encoding* the hyper-polygraph into SAT formulas (Section 4.1) and *solving* them with MonoSAT (Section 4.2). We illustrate each step using the example history $\mathcal{H}$ shown in Figure 3a.

Preliminaries. In propositional logic, a boolean variable v can take the value true or false. A *literal* l refers to a variable v or its negation $\neg v$. A *clause* C is a disjunction of one or more literals, e.g., $C = l_1 \vee l_2 \vee \cdots \vee l_n$. A *formula* $\mathcal{F}$ is constructed using literals and the logical connectives $\wedge$, $\vee$, $\neg$, or $\implies$. Typically, we represent formulas in conjunctive normal form (CNF), where $\mathcal{F}$ is a conjunction of multiple clauses: $\mathcal{F} = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. An *assignment* of a variable v is either true or false, represented as v or $\neg v$, respectively. A *model* $\mathcal{M}$ for a formula $\mathcal{F}$ is a set of assignments to the boolean vairables appearing in $\mathcal{F}$ that makes $\mathcal{F}$ true. If $\mathcal{M}$ does not specify assignments for all variables in $\mathcal{F}$, it is referred to as a *partial model*.

4.1 Encoding

We represent the existence of each edge in the hyper-polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ using a boolean variable. First, the known graph of $\mathcal{G}$ is encoded by the formula

$$\Phi_{KG} = \mathrm{SO}_{2,3}$$

where the variable $\mathrm{SO}_{2,3}$, set to true, indicates the presence of the SO edge $T_2 \xrightarrow{\mathrm{SO}} T_3$.

Second, the C^{WW} constraints of $\mathcal{G}$ are encoded by the formula

$$\Phi_{C^{\mathrm{WW}}} = (\mathrm{WW}^x_{1,2} \vee \mathrm{WW}^x_{2,1}) \wedge (\neg \mathrm{WW}^x_{1,2} \vee \neg \mathrm{WW}^x_{2,1})$$

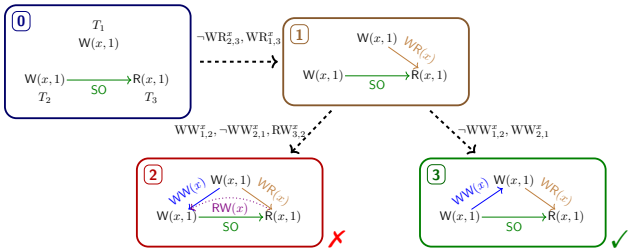


Figure 4: An example solving process for $\mathcal{H}$ in Figure 3.

where the variables $\mathrm{WW}^x_{1,2}$ and $\mathrm{WW}^x_{2,1}$ represent the existence of the edges $T_1 \xrightarrow{\mathrm{WW}(x)} T_2$ and $T_2 \xrightarrow{\mathrm{WW}(x)} T_1$, respectively. This formula enforces that exactly one of the two variables is assigned true, ensuring a total order between the two conflicting writes.

Moreover, the C^{WR} constraints of $\mathcal{G}$ are encoded by the formula

$$\Phi_{C^{\mathrm{WR}}} = (\mathrm{WR}^x_{1,3} \vee \mathrm{WR}^x_{2,3}) \wedge (\neg \mathrm{WR}^x_{1,3} \vee \neg \mathrm{WR}^x_{2,3})$$

where the variables $\mathrm{WR}^x_{1,3}$ and $\mathrm{WR}^x_{2,3}$ denote the presence of the edges $T_1 \xrightarrow{\mathrm{WR}(x)} T_3$ and $T_2 \xrightarrow{\mathrm{WR}(x)} T_3$, respectively. This formula enforces that exactly one of these edges holds, ensuring a unique read-from relation for T_3.

Finally, the encoding of the derivation rule for RW edges in $\mathcal{G}$ is captured by the formula

$$\begin{aligned}\Phi_{\mathrm{RW}} &= (\mathrm{WW}^x_{1,2} \wedge \mathrm{WR}^x_{1,3} \implies \mathrm{RW}^x_{3,2}) \wedge (\mathrm{WW}^x_{2,1} \wedge \mathrm{WR}^x_{2,3} \implies \mathrm{RW}^x_{3,1}) \\ &= (\neg \mathrm{WW}^x_{1,2} \vee \neg \mathrm{WR}^x_{1,3} \vee \mathrm{RW}^x_{3,2}) \wedge (\neg \mathrm{WW}^x_{2,1} \vee \neg \mathrm{WR}^x_{2,3} \vee \mathrm{RW}^x_{3,1})\end{aligned}$$

where $\mathrm{RW}^x_{3,1}$ and $\mathrm{RW}^x_{3,2}$ represent the existence of the $\mathrm{RW}(x)$ edges $T_3 \xrightarrow{\mathrm{RW}(x)} T_1$ and $T_3 \xrightarrow{\mathrm{RW}(x)} T_2$, respectively. This encoding ensures that an RW edge is derived only when both the corresponding WW and WR edges are present.

Overall, the complete encoding for $\mathcal{G}$ is given by

$$\Phi_{\mathcal{G}} = \Phi_{KG} \wedge \Phi_{C^{\mathrm{WW}}} \wedge \Phi_{C^{\mathrm{WR}}} \wedge \Phi_{\mathrm{RW}}.$$

4.2 Solving

Each satisfying assignment of $\Phi_{\mathcal{G}}$ corresponds to a graph $\mathcal{G}'$ that is compatible with $\mathcal{G}$. To ensure that $\mathcal{G}'$ is acyclic, we assert the predicate $\mathrm{Acyclic}(\mathcal{G}')$ in MonoSAT. Hence, the history $\mathcal{H}$ is serializable if and only if the formula $\Phi_{\mathcal{G}} \wedge \mathrm{Acyclic}(\mathcal{G}')$ is satisfiable.

MonoSAT implements the *Conflict-Driven Clause Learning with Theory* (CDCL(T)) framework [30] to decide the satisfiability of SAT formulas augmented with theory predicates. It combines a *SAT solver*, which searches for a satisfying assignment to the boolean formula (e.g., $\Phi_{\mathcal{G}}$), with a *theory solver*, which ensures that the asserted predicates (e.g., $\mathrm{Acyclic}(\mathcal{G}')$) hold under the current assignment. As the SAT solver assigns variables, the theory solver incrementally updates a graph G^* by adding edges for variables set to true. If a predicate is violated, a conflict clause is generated and learned by the SAT solver to prevent revisiting the same conflict.

The solving process may vary depending on the order in which variables are assigned. Figure 4 illustrates one such procedure for $\mathcal{G}$, with the graph G^* progressing through four stages.[3] Initially,

[3]We also illustrate in the Appendix [7] how the SAT solver and the theory solver collaborate in CDCL(T) for this example.

only $SO_{2,3}$ is assigned true, yielding ⓪. Suppose the SAT solver next assigns $\neg WR^x_{2,3}$. Then $WR^x_{1,3}$ must be true to satisfy the clause $WR^x_{1,3} \vee WR^x_{2,3}$. This is also known as *unit propagation* [10] that occurs when a clause becomes unit, i.e., only one literal remains unassigned ($WR^x_{1,3}$ in this case), forcing the literal to be assigned true to satisfy the clause. The theory solver adds the edge $T_1 \xrightarrow{WR(x)} T_3$ to G^*, leading to ①. No cycles are detected at this point.

Next, assigning $WW^x_{1,2}$ triggers $\neg WW^x_{2,1}$ due to mutual exclusion, and then $RW^x_{3,2}$ via the clause $\neg WW^x_{1,2} \vee \neg WR^x_{1,3} \vee RW^x_{3,2}$, which leads to the updated graph G^* in ②. At this point, the theory solver detects a cycle: $T_2 \xrightarrow{SO} T_3 \xrightarrow{RW(x)} T_2$ (see also Figure 3d). It then returns a conflict clause $\neg RW^x_{3,2} \vee \neg SO_{2,3}$ to the SAT solver, indicating that removing either edge would break the cycle. The SAT solver then backtracks from $WW^x_{1,2}$, assigns its negation $\neg WW^x_{1,2}$, and propagates $WW^x_{2,1}$. This results in ③, where all variables are assigned and no cycles are present. Hence, an acyclic compatible graph is found (see also Figure 3c), and the history $\mathcal{H}$ is serializable.

5 Workload Characteristics for SMT Solving

General-purpose SMT solvers like MonoSAT are agnostic to the characteristics of database transaction workloads, thereby missing opportunities for targeted optimization. In contrast, we hypothesize that leveraging workload-specific characteristics can significantly improve solving efficiency for isolation verification. This section presents two key observations that motivate the tailored SMT strategies introduced in the next section.

5.1 Prevalence of 2-width Cycles

A key bottleneck in solving arises from "bad" guesses that trigger cycles in a dependency graph, incurring costly backtracking overhead. To mitigate this, prior work [16, 35, 40] introduces *pruning*, a technique that identifies and precludes infeasible WW constraints by analyzing reachability. For example, consider a WW constraint $\{T_1 \xrightarrow{WW(x)} T_2, T_2 \xrightarrow{WW(x)} T_1\}$. If T_2 is already reachable from T_1, then choosing $T_2 \xrightarrow{WW(x)} T_1$ would result in a cycle. Hence, this choice is safely pruned. In the context of SAT solving, pruning corresponds to adding a conflict clause with a single WW variable, e.g., $\neg WW^x_{2,1}$ in this case, to the encoding for $\Phi_{\mathcal{G}}$ (Section 4.1).

Essentially, pruning shifts the cost of resolving search-time conflicts to a pre-solving analysis phase. This raises a natural question: can more of the solving overhead be offloaded to preprocessing? In other words, is there a sweet spot that balances the cost of precomputation with the benefit of reduced solving effort, thereby improving overall performance?

To explore this, we adapt the notion of *cycle width* [12, 15], which we define as the number of *unique* WW and WR variables appearing in a conflict clause. While conventional pruning relies on unit clauses (i.e., 1-width cycles), broader pruning can be achieved by encoding larger cycles using multi-variable clauses. To avoid redundancy, we count each variable only once and disregard SO variables, which are always true, and RW variables, which can be derived from WW and WR.

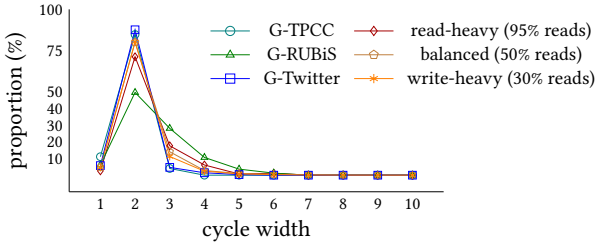


Figure 5: Distribution of minimal cycle widths in conflicts.

We empirically measure the widths of conflict cycles encountered during solving across six representative benchmarks, including TPC-C and YCSB-like transaction workloads with varying read proportions (see Section 7.1 for details). Each benchmark consists of 1000 transactions. For each conflict, we record the *minimal* cycle width, as it may correspond to multiple cycles depending on the search path. As shown in Figure 5, 2-width cycles dominate across all benchmarks. This suggests that many conflicts could be prevented by pre-encoding 2-variable clauses. We leverage this insight in Section 6.2, incorporating 2-width cycle encodings into the SAT formula $\Phi_{\mathcal{G}}$ to strike a balance between pre-solving effort and solving efficiency (further validated in Section 7.3.3).

5.2 Polarity as Schedule Reconstruction

Another major source of inefficiency in SMT solving stems from poor polarity choices when assigning decision variables. A key insight we introduce is that each polarity decision—whether to include or exclude a dependency edge—implicitly defines a transaction schedule. Suboptimal choices can conflict with the actual, though hidden (due to the black-box setting), schedule enforced by the database, leading to dependency cycles and costly backtracking.

General-purpose solvers such as MonoSAT treat all variables uniformly and assign polarity without regard to the execution context. In contrast, we observe that many dependencies, including session order SO and uniquely determined read-from relation WR, are already known from the observable history and can be leveraged to guide more informed polarity decisions. Ideally, if the internal transaction schedule were accessible, one could align polarity assignments perfectly to avoid conflicts. While this is infeasible in black-box settings, the visible dependencies still capture a significant portion of the execution order. We hypothesize that exploiting this partial order can substantially reduce conflict rates and backtracking overhead. The following example illustrates this intuition.

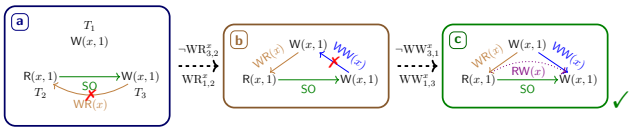


Figure 6: An example solving process: guiding polarity picking via known dependencies.

Example 5.1. In Figure 6, the solver must resolve two constraints: $\{T_1 \xrightarrow{WR(x)} T_2, T_3 \xrightarrow{WR(x)} T_2\}$ and $\{T_1 \xrightarrow{WW(x)} T_3, T_3 \xrightarrow{WW(x)} T_1\}$. Initially, in ⓐ, the solver must decide between two possible read-from candidates for T_2. By recognizing the known session order

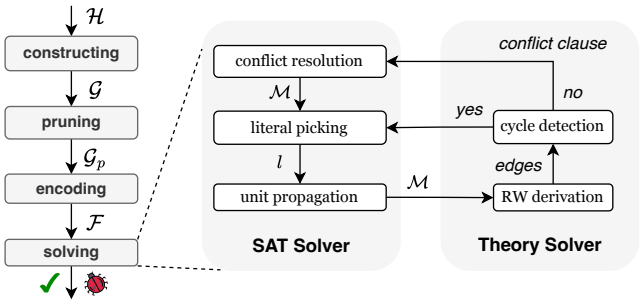


Figure 7: Workflow of VeriStrong.

$T_2 \xrightarrow{\text{SO}} T_3$, the solver correctly assigns $\text{WR}^x_{3,2}$ = false, avoiding a potential cycle between T_2 and T_3. The solver then proceeds to (b), where it must resolve the WW constraint. Again, guided by the inferred dependency $T_1 \xrightarrow{\text{WR}(x)} T_3$, it avoids introducing a cycle, e.g., $T_1 \xrightarrow{\text{WR}(x)} T_2 \xrightarrow{\text{SO}} T_3 \xrightarrow{\text{WW}(x)} T_1$, by choosing $\text{WW}^x_{3,1}$ = false.

This example underscores the importance of polarity decisions: in large histories, a single incorrect polarity choice may introduce an edge that contradicts the actual schedule. Consequently, the solver is misled into exploring an exponentially larger search space, incurring substantial overhead due to cascading conflicts and backtracking. Motivated by this insight, the next section introduces a polarity picking strategy—integrated into our dedicated SMT solver—that exploits the observable partial order in the execution history. We evaluate its impact in Section 7.3.4.

6 VeriStrong with Tailored SMT Solving

6.1 Overview

VeriStrong builds upon the baseline algorithm described in Section 4. Motivated by our observations in Section 5, it incorporates two key optimizations: efficient handling of 2-width cycles (Section 6.2) and polarity picking for decision variables (Section 6.3).

Figure 7 shows the high-level workflow of VeriStrong, comprising four main stages: (i) *constructing* the hyper-polygraph $\mathcal{G}$ from the input history $\mathcal{H}$, (ii) *pruning* infeasible constraints in $\mathcal{G}$, (iii) *encoding* the pruned graph $\mathcal{G}_p$ into a SAT formula $\mathcal{F}$, and (iv) *solving* $\mathcal{F}$. The corresponding pseudo-code is given in Algorithm 1, which we refer to alongside the figure to explain the procedure.

The reminder of Algorithm 1 details the solving phase, specifically the interaction between the SAT frontend (SatSolve, Line 11) and the theory backend (TheorySolve, Line 26). Let G^* denote the current known graph maintained in the theory solver. Initially, G^* is set to the hyper-polygraph $\mathcal{G}$ constructed from $\mathcal{H}$. In each iteration, the SAT solver selects a decision variable v (Line 14) and its polarity, yielding a decision literal l (Line 15). This literal is then unit propagated, producing a partial assignment $\mathcal{M}$ (Line 16) that is passed to the theory solver for conflict checking (Line 17).

Specifically, the theory solver first derives RW edges from the current WW and WR edges (Line 27) and checks whether the updated graph G^* contains any cycles (Line 28). If a cycle is found, it generates a conflict clause and returns it to the SAT solver (Line 30). The SAT solver then resolves the conflict, either terminating with

Algorithm 1 The VeriStrong algorithm

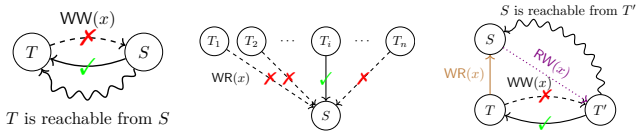
$\mathcal{H}$: the input history
$G^* = (\mathcal{V}, E^*)$: the current graph maintained in the theory solver; initialized as the hyper-polygraph $\mathcal{G}$ constructed from $\mathcal{H}$
Vars: a set of boolean variables managed by the SAT solver; initially $\emptyset$
Clauses: a set of clauses managed by the SAT solver; initially $\emptyset$
Both *Vars* and *Clauses* are constructed in Encode (Line 5).

```
1:  function VerifySER(H)
2:      G ← Construct(H)                          ▷ Alg. 4 in the Appendix [7]
3:      if ¬Prune(G)                              ▷ Alg. 5 in the Appendix; see also Section 6.2.1
4:          return false
5:      F ← Encode(G_p)                           ▷ Alg. 7 in the Appendix; see also Section 6.2.2
6:      return Solve(F, G_p)

7:  function Solve(F, G)
8:      for T --T(x)--> S ∈ E
9:          E* ← E* ∪ {T → S}
10:     return SatSolve(F, G)

11: function SatSolve(F, G)
12:     M_total ← ∅
13:     while |M_total| ≠ |Vars|
14:         v ← ChooseDecisionVar(M_total, Vars)
15:         l ← PickPolarity(v)                   ▷ Alg. 3 in Section 6.3
16:         M ← UnitPropagate(l, F)
17:         (ret, conflict_clause) ← TheorySolve(M)
18:         if ¬ret                               ▷ a conflict is detected
19:             if ¬ResolveConflict(M_total, M, conflict_clause)
20:                 return UNSAT
21:             else
22:                 M_total ← Backtrack(M_total, M, conflict_clause)
23:                 continue, goto Line 13
24:         M_total ← M_total ∪ M
25:     return SAT

26: function TheorySolve(M)
27:     edges ←DeriveRWEdges(M)
28:     if (E* ∪ edges) is cyclic                 ▷ use the PK algorithm [33]
29:         conflict_clause ← GenConflictClause(E*, edges)
30:         return (false, conflict_clause)
31:     else
32:         E* ← E* ∪ edges
33:         return (true, null)
```

UNSAT (Line 20) or backtracking and updating $\mathcal{M}$ (Line 22). If no cycle is detected, the solver continues by selecting the next decision variable. Once all variables have been assigned without encountering conflicts (Line 13), the solver reports SAT (Line 25), indicating that the history is serializable.

6.2 Small-Width Cycle Preprocessing

This optimization aims to shift the cost of resolving search-time conflicts into a pre-solving analysis focused on small-width cycles. It comprises two components: (i) an aggressive pruning phase that eliminates all 1-width cycles through reachability analysis, and (ii) a proactive encoding of 2-width cycles into the SAT formula, enabling early conflict detection via unit propagation during solving.



(a) Pruning WW (b) Pruning WR (c) Pruning via RW

Figure 8: Cases for pruning 1-width cycles.

6.2.1 Eliminating 1-width Cycles. We extend the pruning techniques from prior work [16, 35, 40], which focus on WW constraints, by also pruning WR constraints and their derived RW counterparts. The following examples illustrate the core ideas of our approach.

Case I: Pruning WW *Constraints.* Consider the two transactions T and S in Figure 8a, both writing to the same key x. If T is already reachable from S in the current known graph, then adding $T \xrightarrow{\mathrm{WW}(x)} S$ would introduce a 1-width cycle. This infeasible option can thus be pruned, and the valid edge $S \xrightarrow{\mathrm{WW}(x)} T$ is added to the known graph.

Case II: Pruning WR *Constraints.* We can similarly prune a WR edge if its inclusion would introduce a 1-width cycle in the known graph. Consider the scenario in Figure 8b, where a transaction S reads from key x and has n candidate writers $T_1, \ldots, T_n$. If $n-1$ of these candidates have been eliminated due to cycle formation, the remaining writer must be the only feasible source. In this case, we can safely add the edge $T_i \xrightarrow{\mathrm{WR}(x)} S$ to the known graph.

Case III: Pruning via Derived RW *Edges.* Derived edges may also lead to cycles. In Figure 8c, assume that adding $T \xrightarrow{\mathrm{WW}(x)} T'$ induces $S \xrightarrow{\mathrm{RW}(x)} T'$ (via $T \xrightarrow{\mathrm{WR}(x)} S$). If S is already reachable from T', this creates a cycle, so $T \xrightarrow{\mathrm{WW}(x)} T'$ is pruned.

VeriStrong iteratively applies this pruning process until the known graph stabilizes, i.e., all 1-width cycles have been eliminated. The complete pseudo-code is provided in the Appendix [7].

6.2.2 Encoding 2-width Cycles. To prevent conflicts caused by 2-width cycles during solving, VeriStrong encodes them into SAT formulas. The corresponding pseudo-code is provided in Algorithm 2. We consider two scenarios, as also illustrated in Figure 9.

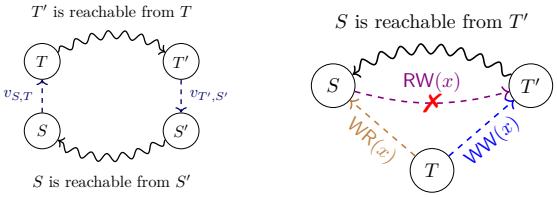
Case I: Canonical 2-Width Cycles. This case illustrates a canonical 2-width cycle formed by two known paths, as shown in Figure 9a. Let $T \rightsquigarrow T'$ and $S' \rightsquigarrow S$ be known dependencies in the graph, where $\rightsquigarrow$ denotes reachability between transactions. Adding both candidate edges $S \to T$ and $T' \to S'$ would complete a cycle. We prevent this by adding the clause $\neg v_{S,T} \vee \neg v_{T',S'}$ to the formula, where $v_{S,T}$ and $v_{T',S'}$ are the corresponding boolean variables.

Case II: Derived RW *Cycles.* In Figure 9b, assume $T' \rightsquigarrow S$ is known. Adding both $T \xrightarrow{\mathrm{WR}(x)} S$ and $T \xrightarrow{\mathrm{WW}(x)} T'$ would derive $S \xrightarrow{\mathrm{RW}(x)} T'$, forming a cycle. To avoid this, we encode the clause: $\neg \mathrm{WW}^x_{T,T'} \vee \neg \mathrm{WR}^x_{T,S}$.

While longer cycles could, in theory, also be encoded, enumerating clauses over k boolean variables incurs a time complexity of $O(|Vars|^k)$, making it impractical for large k. Our experiments show that 2-width clauses capture the majority of conflicts effectively (see Figure 15, Section 7.3.3). Therefore, VeriStrong focuses on encoding only 2-width cycles, striking a balance between preprocessing overhead and solving efficiency.

Algorithm 2 Encoding 2-width cycles

```
1: function EncodingCyclesOfWidth2(V, E)
2:     R ← Reachability(V, E)
3:     for v1, v2 ≠ v1 ∈ Vars
4:         if CanonicalCycle(v1, v2, R) ∨ RWCycle(v1, v2, R)
5:             Clauses ← Clauses ∪ {¬v1 ∨ ¬v2}
6: function CanonicalCycle(v1, v2, R)
7:     let (S, T) ← v1, (T', S') ← v2
8:     return (T, T') ∈ R ∧ (S, S') ∈ R
9: function RWCycle(v1, v2, R)
10:    return ∃x. ∃T, S, T'. T' ≠ T ∧ (T', S) ∈ R ∧ ((v1 ≜ WW^x_{T,T'} ∧
       v2 ≜ WR^x_{T,S}) ∨ (v2 ≜ WW^x_{T,T'} ∧ v1 ≜ WR^x_{T,S}))
```



(a) Canonical cycle (b) Derived RW cycle

Figure 9: Cases for encoding 2-width cycles.

6.3 Order-Guided Polarity Picking

VeriStrong leverages the current partial order of the dependency graph to guide polarity picking, as motivated in Section 5.2. To achieve this, we incorporate the *pseudo-topological order* maintained by the PK algorithm [33], which is originally developed for dynamic cycle detection and also adopted by MonoSAT. The full polarity picking procedure is given in Algorithm 3.

Algorithm 3 Pseudo-topological-order-guided polarity picking

```
1: function PickPolarity(v)
2:     let (from, to) ← v
3:     if level(from) < level(to)
4:         return v                          ▷ positive polarity
5:     else
6:         return ¬v                         ▷ negative polarity
```

The PK algorithm dynamically maintains a level $level(\cdot)$ for each vertex, which approximates its position in a topological ordering. When a candidate edge $(from, to)$ is considered, the algorithm first checks whether $level(from) < level(to)$. If so, the edge is guaranteed not to introduce a cycle and is immediately accepted. If not, the algorithm performs a reachability check to ensure that no cycle would be formed, and if the edge is added, it updates the levels of affected vertices to maintain consistency.

This pseudo-topological order reflects the known partial order in the current graph: every edge that has been included satisfies $level(from) < level(to)$. We exploit this order to guide polarity

selection. For a decision variable v representing a candidate edge $(from, to)$, we assign it a *positive polarity* (i.e., include the edge) when $level(from) < level(to)$, and a *negative polarity* (i.e., exclude the edge) otherwise.

This heuristic encourages decisions that respect the observed partial order and are thus less likely to introduce cycles. Although it cannot guarantee cycle-freedom in all cases, especially when the topological approximation is incomplete, it serves as a practical guide for reducing backtracking. If a polarity choice later results in a conflict, the solver will backtrack and try the opposite polarity.

7 Experiments

We have implemented our algorithm in a tool called VeriStrong, built on top of MiniSat [11], a minimalistic and open-source SAT solver. We selected MiniSat due to its simplicity and extensibility, which allowed us to integrate our optimization strategies effectively.

To support the optimization of small-width cycles described in Section 6.2, VeriStrong computes reachability in the known (acyclic) graph using dynamic programming. Specifically, it maintains a 0-1 matrix of size $|\mathcal{T}| \times |\mathcal{T}|$, where each entry records whether one transaction is reachable from another. By traversing nodes in reverse topological order, VeriStrong incrementally updates each node's reachable set: for an edge $(from, to)$, it unions the reachability set of *to* into that of *from*. We implement the matrix using bit vectors and bitwise operations, providing a lightweight alternative to GPU-based solutions [35]. Overall, VeriStrong comprises approximately 5,000 lines of C++ code.

We conduct an extensive evaluation of VeriStrong, focusing on its checking efficiency, while also assessing its bug-finding effectiveness. Specifically, we aim to answer the following questions:

- **Efficiency:** How efficiently does VeriStrong perform across various general workloads (Section 7.2.1)? Can it outperform existing tools (Section 7.2.2)? Does it scale to large histories (Section 7.2.3)? How do its individual components contribute to the overall performance (Section 7.3)?
- **Effectiveness:** How effective is VeriStrong at uncovering isolation bugs in production database systems, particularly those related to DuplicateValue (Section 7.4)?

7.1 Benchmarks and Experimental Setup

We evaluate VeriStrong and competing verifiers using two categories of benchmarks. The first category comprises YCSB-like transactional workloads, produced by a parametric workload generator built on top of PolySI [16]. This generator supports several configurable parameters, with default values shown in parentheses: the number of client sessions (20), the number of transactions per session (100), the number of read/write operations per transaction (20), the overall read proportion (50%), the total number of keys (5k), and the proportion of keys with duplicate write values (50%).

We characterize duplicate write values using a Zipfian distribution, where the parameter θ (0.5) controls the degree of duplication, and N (100) defines the size of value space. Based on this configuration, we generate a set of representative workloads, including RH (read-heavy, 95% reads), WH (write-heavy, 70% writes), and BL (balanced, 50% reads). These workloads are aligned with UniqueValue workloads commonly used in prior work. In addition, we include HD, a variant of BL with a high degree of duplication ($\theta = 1.5$). Each workload is executed on a PostgreSQL v15 instance to produce transactional histories, each containing at least 10k transactions and 80k operations. These histories are sufficient to highlight VeriStrong's advantages over competing tools.

The second category comprises application-level benchmarks: TPC-C [36], a standard OLTP benchmark configured with 1 warehouse, 10 districts, 30k customers, and 5 transaction types; RUBiS [34], an auction-based system with 20k users and 200k items; and TwitterClone [19], a blogging application with Zipfian key access patterns. We use a PostgreSQL instance to generate general transactional histories from these applications, which we refer to as G-TPCC, G-RUBiS, and G-Twitter, respectively. Notably, both RUBiS and TwitterClone naturally generate duplicate write values. However, prior work [16, 35, 40] disabled this behavior to enforce the UniqueValue assumption. In contrast, we preserve these duplicate-value semantics to more accurately reflect real-world behavior.

Unless otherwise stated, all experiments are performed on a local machine equipped with an Intel 13th Gen i5 CPU and 32GB RAM.

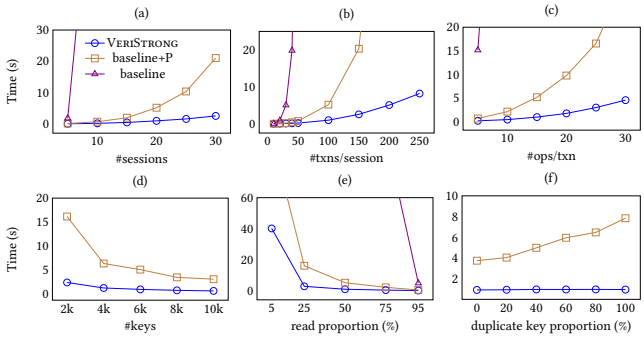


Figure 10: Verification time comparison across a range of general workloads. Timeout (60s) data points are not plotted.

7.2 Performance Evaluation

7.2.1 Comparison with Baselines. Our first set of experiments compares VeriStrong with the baseline algorithm (Section 4) across various general workloads. We also include a stronger variant, baseline+P, which incorporates the pruning technique for WW constraints (Section 5.1).

The experimental results are shown in Figure 10, omitting data points that exceed the 60s timeout. VeriStrong consistently outperforms both the baseline and its stronger variant. Specifically, under increased concurrency, such as more sessions (a), more transactions per session (b), more operations per transaction (c), and smaller key space (d), the two baselines suffer from exponential verification time, while VeriStrong incurs only moderate overhead. Furthermore, in scenarios with increased uncertainty in WW and WR dependencies, such as write-heavy workloads (e) and higher proportions of duplicate keys (f), VeriStrong remains the fastest while maintaining fairly stable verification efficiency.

In addition, we measure the memory usage of all three tools under the same settings as in Figure 10. The trends are similar: VeriStrong uses less memory than both baselines across a range of workloads. Detailed memory plots are provided in the Appendix [7].

7.2.2 Comparison with State-of-the-Art. Our second set of experiments evaluate VeriStrong against four state-of-the-art verifiers: Cobra [35] for Serializability, PolySI [16] and Viper [40] for Snapshot Isolation, and dbcop [5] for both. Since Cobra supports GPU acceleration for pruning, we also include its GPU-enabled version in our evaluation.[4] Note that dbcop is included as a representative of non-SMT-based verifiers, which rely on graph traversal algorithms such as depth-first search to detect cycles.

For a fair comparison, we use the BlindW-RW dataset adopted in prior work [35, 40]. This dataset exclusively consists of UniqueValue histories, as all the above verifiers are restricted to such input. Each history contains an even mix of read-only and write-only transactions, with each transaction comprising eight operations.

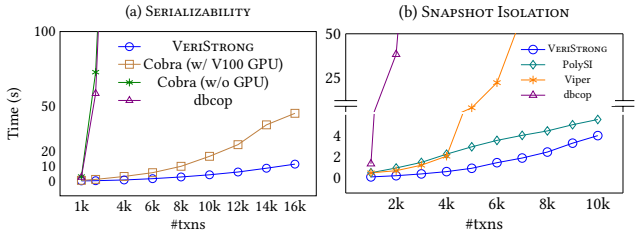


Figure 11: Comparison with existing verifiers under UniqueValue histories. Both experiments are conducted using the BlindW-RW datasets, which differ only in key space: 10k keys in (a) and 2k keys in (b).

As shown in Figure 11, VeriStrong outperforms all competitors in verifying both isolation levels. In particular, with just 16k transactions, it achieves up to 4.4x speedup, even compared to Cobra with GPU acceleration. For Snapshot Isolation, VeriStrong delivers up to 4x speedup over PolySI and substantially outperforms Viper and dbcop. These results highlight the benefits of our dedicated SMT solving, which leverages workload-specific characteristics to enhance verification efficiency.

7.2.3 Scalability. Verifying large histories is highly desirable for increasing confidence in a database's fulfilment of promised isolation guarantee. Following the previous experiments, we evaluate VeriStrong's scalability on large histories, each containing up to 100k transactions and 2 million operations with a total of 50k keys.[5]

As shown in Figure 12, verifying large histories is manageable for VeriStrong on modern hardware, e.g., Serializability verification completes in under 5 hours using less than 64 GB of memory. In addition, two expected trends are observed. First, both runtime and memory usage increase as the proportion of read operations decreases (from RH to BL to WH workloads). This trend is primarily due to the increased uncertainty in WW dependencies, which expands the set of potential read-from candidates and ultimately amplifies the overall uncertainty.

Second, verifying Snapshot Isolation incurs higher overhead than Serializability. This is expected: while Serializability requires only an acyclic dependency graph, Snapshot Isolation permits certain cycles, making the analysis inherently more complex.

[4]Due to resource constraints, we run Cobra on a server equipped with an Intel Xeon Platinum 8163 CPU, an NVIDIA V100 GPU, and 32GB of RAM.
[5]The experiments were conducted on a server with an AMD EPYC 7H12 64-Core processor and 1TB of memory.

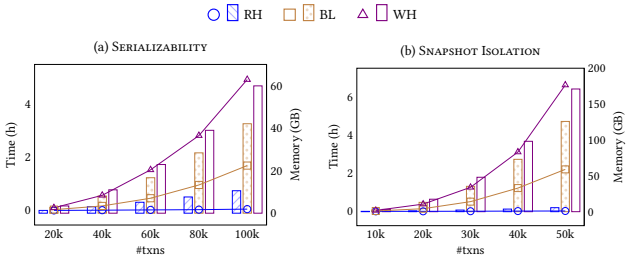


Figure 12: VeriStrong's performance under large workloads. Time and memory usage are plotted in lines and bars.

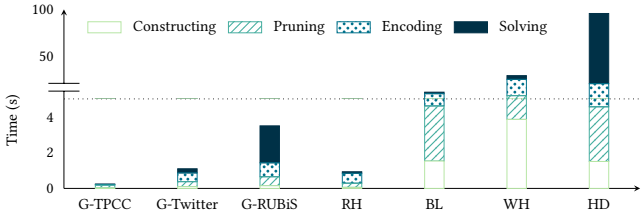


Figure 13: Breakdown of checking time across stages.

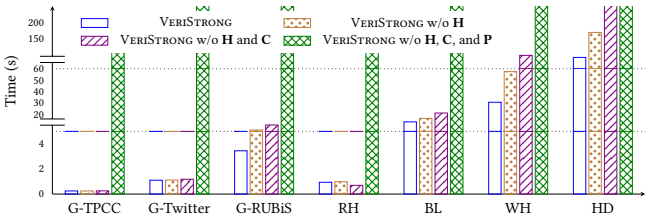


Figure 14: Ablation analysis of VeriStrong's optimizations.

In practice, VeriStrong maintains an additional induced graph for Snapshot Isolation, which is more expensive to construct than deriving RW edges alone as for Serializability. This gap is also evident in prior evaluations on UniqueValue histories: Cobra verifies a history with 10k transactions against Serializability in 15s [35], whereas Viper requires over 400s to verify Snapshot Isolation for the same amount of transactions [40]. This gap becomes even more pronounced in the presence of duplicate values.

7.3 Dissecting VeriStrong

In this section, we take a closer look at VeriStrong, examining how its individual components and optimization strategies contribute to the overall checking efficiency.

7.3.1 Decomposition Analysis. We decompose VeriStrong's checking time into four stages: constructing, pruning, encoding, and solving. Figure 13 presents the breakdown across seven benchmarks (see Section 7.1). Overall, the constructing and pruning stages are relatively inexpensive. The G-TPCC workload is a special case where all constraints are pruned prior to encoding and solving, resulting in negligible cost for the latter stages. In most benchmarks—except for G-RUBiS and HD—the solving time remains relatively low, whereas the encoding stage dominates. This behavior is expected: our small-width cycles optimization shifts some of the burden from solving to pruning and encoding, while keeping these two stages efficient. As a result, the overall checking time is reduced.

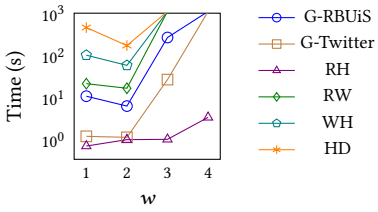


Figure 15: Runtime of Variant (i) under varying encoded cycle widths (w). Timeout (600s) data points are not plotted.

Table 1: The number of conflicts encountered during solving with or without Optimization H.

Benchmark	VeriStrong	w/o **H**
G-TPCC	0	0
G-Twitter	79	99
G-RUBiS	89	246
RH	0	0
BL	50	497
WH	86	1427
HD	936	2047

Table 2: Average checking time on anomalous histories (excluding those that timed out beyond 900s).

Verifier	SER	SI
Cobra	1293ms	–
PolySI	–	359ms
Viper	–	6559ms
dbcop	551ms	108ms
VeriStrong	143ms	18ms
#histories	2073	434

7.3.2 Ablation Analysis. To assess the contributions of the major optimizations in VeriStrong, we consider three variants: (i) VeriStrong without the heuristic for polarity picking (**H**); (ii) VeriStrong without **H** and without the encoding of small-width cycles (**C**); and (iii) VeriStrong without **H**, **C**, and pruning (**P**).

Figure 14 presents our ablation analysis results. Overall, all optimizations proves effective, though their impact varies across different workloads. Optimization **H** is particularly beneficial for G-RUBiS, BL, WH, and HD, but shows little effect on G-Twitter and RH. This is because the latter two are read-heavy workloads, where cycles are rarely encountered during solving. Consequently, the potential for **H** to resolve conflicts through polarity picking is limited.

We also observe that the workloads where Optimization **C** is most effective largely overlap with those where **H** is effective. Interestingly, for RH, Optimization **C** slightly increases the overall checking time. This is expected, as the overhead stems from the enumeration of small-width cycles, which might be unnecessary in cycle-sparse workloads like RH.

As noted earlier, all constraints in G-TPCC are pruned, allowing the solving phase to be bypassed entirely. This explains why Optimization **P** has a particularly pronounced impact in this case.

7.3.3 Impact of Cycle Width. Recall that in Optimization **C**, we encode 2-width cycles (see Section 6.2.2). A natural question arises: what is the performance impact of encoding cycles of larger widths? To answer this, we conduct a comparative experiment using variants of Variant (i), varying the width of encoded cycles. We plot the results in Figure 15, where $w = 1$ corresponds to a baseline with pruning only (i.e., no cycle encoding), while $w \geqslant 2$ indicates that cycles of width w are explicitly encoded.

Across all six benchmarks (excluding G-TPCC, where solving is bypassed), encoding 2-width cycles ($w = 2$) consistently delivers the best performance, striking a balance between pruning effectiveness and solving overhead. The only exception is RH, where solving time is already low, suggesting that in cycle-sparse workloads, the modest overhead of encoding small-width cycles has limited impact on overall performance.

7.3.4 Impact of Polarity Picking. To better understand how Optimization **H** improves VeriStrong's performance, we examine its effect on reducing the number of encountered conflicts (i.e., the number of backtracks) during solving. We conduct a comparative experiment between VeriStrong and Variant (i) across seven benchmarks; the results are shown in Table 1. Overall, **H** significantly reduces the number of conflicts. This effect becomes more pronounced as the write proportion and the duplication rate of write values increase, which is consistent with the performance differences observed in Figure 14. The only two exceptions are G-TPCC and RH, both of which exhibit no conflicts. In G-TPCC, all constraints are pruned prior to solving. In RH, although not all constraints are pruned, its read-heavy nature results in conflict-free solving even without **H**.

7.4 Effectiveness

Beyond checking performance, an essential criterion for a verifier is its ability to accurately detect isolation anomalies. To this end, we validate VeriStrong against an extensive set of known anomallies.

Reproducing UniqueValue Anomalies. VeriStrong reproduces all known anomalies from a substantial set of 2507 UniqueValue histories: 2073 for Serializability and 434 for Snapshot Isolation. These histories were originally collected by prior work [5, 16, 35] from earlier versions of widely-used databases—including CockroachDB, YugabyteDB, and Dgraph—and were generated under Serializability and Snapshot Isolation, respectively. In addition, we report the runtime of each tool: as shown in Table 2, VeriStrong achieves significantly lower average checking times than the state-of-the-art verifiers.

(Re)discovering DuplicateValue Anomalies. Motivated by recent bug reports [26, 31], we apply VeriStrong to test both MySQL and MariaDB. VeriStrong replicates the reported anomalies in earlier versions of both databases and, notably, rediscovers the same bug in the latest stable version of MariaDB (v 11.5.2). We reported this issue to the developers, who confirmed it as a valid bug. Interestingly, while a fix had been introduced in an earlier version, it was not enabled by default due to potential compatibility issues with existing applications [27]. Note that all these DuplicateValue anomalies lie beyond the capabilities of existing verifiers, which are limited to verifying UniqueValue histories.

8 Related Work

Recent years have seen significant progress in verifying database isolation guarantees. These advances fall into two main categories.

Black-Box Verification. A recent line of work applies SMT solving to verify strong isolation guarantees. Tools such as Viper [40] and PolySI [16] target Snapshot Isolation, while Cobra [35] focuses on Serializability. They all encode execution histories as polygraphs or their variants and rely on the off-the-shelf solver MonoSAT to detect cycles that represent anomalies.

VeriStrong follows this approach but differs in three key ways. First, rather than relying solely on general-purpose SMT solvers, it incorporates workload-specific optimizations to enhance performance. Second, existing polygraph-based approaches are limited to UniqueValue histories. VeriStrong introduces an expressive hyper-polygraph representation that also supports DuplicateValue, thereby broadening applicability. Third, VeriStrong provides both soundness and completeness, ensuring reliable isolation verification. In contrast, existing verifiers risk false positives and false negatives, particularly in the presence of duplicate write values.

Non-solver tools [5, 20, 23, 29, 37] employ graph traversal algorithms, such as DFS or optimized variants, to verify isolation guarantees. The dbcop tool [5] applies a polynomial-time algorithm to verify Serializability over UniqueValue histories (with a bounded number of sessions), along with a polynomial-time reduction from verifying Snapshot Isolation to verifying Serializability. However, dbcop has been reported to be less efficient than Cobra [35] and PolySI [16], which in turn are outperformed by VeriStrong.

Elle [20], the isolation checker used in the Jepsen framework [17], infers version orders (i.e., WW dependencies) from list-append workloads. However, it relies on the UniqueValue assumption, which limits its effectiveness when applied to general workloads containing duplicate write values (see Section 9). MTC [37] exploits a class of simplified database workloads, called mini-transactions (e.g., read-modify-write transactions) to achieves quadratic-time verification under the same UniqueValue assumption. While MTC strikes a practical balance between performance and bug-finding effectiveness, its applicability is limited, as real-world workloads often diverge from such simplified transaction patterns.

Other tools, such as Plume [23] and AWDIT [29], focus on verifying weaker isolation levels [2, 24], such as Read Committed and Transactional Causal Consistency. These guarantees are known to be verifiable in polynomial time [5] and are computationally less complex than the stronger ones we target.

White-Box Approaches. Emme [9] and Chronos [21] are white-box verifiers for strong isolation guarantees. In contrast to black-box approaches, they utilize additional transaction metadata, such as start and commit timestamps, to infer execution order and reduce verification overhead. However, this metadata is often database-specific, requiring code changes for each system, and may be inaccessible in black-box settings.

CAT [25] is an isolation verifier based on linear temporal logic model checking, targeting database designs specified in the Maude formal language. It requires access to internal transaction details, such as timestamps, during execution. VerIso [13] formally verifies a database design's isolation guarantees across all possible behaviors via theorem proving. Mathiasen et al. [28] propose separation logic specifications for weak isolation guarantees and apply them to verify the correctness of a single-node key-value store implementation. Our work complements these efforts by operating in a black-box setting without requiring system internals.

9 Discussion and Conclusion

Supporting List-Append Workloads. Elle has proven highly effective for verifying strong isolation guarantees. Its key innovation lies in leveraging Jepsen's list-append operations to efficiently infer uncertainties in WW dependencies. A list-append history consists of two types of operations: append (A), which installs a value (analogous to a write), and read (R), which returns a list of installed values. The order of elements in the returned list reflects the version order among the corresponding writes, thereby resolving WW ambiguities. For example, reading the list `[2,1]` on key x indicates that $A(x, 2)$ precedes $A(x, 1)$ in the version order. In contrast, such ordering remains ambiguous in the read-write register setting, which is the primary focus of most existing verifiers, including VeriStrong.

However, Elle becomes ineffective when DuplicateValue is introduced. For instance, reading a list `[1,1]` provides no information about the ordering between two append operations with the same value. This exposes a fundamental limitation of Elle in handling general histories, which remains an open challenge.

Suppoting Other Formalisms. Our hyper-polygraphs are not restricted to Adya's formalism; rather, they are expressive enough to support other dependency-based characterizations of isolation guarantees, such as the axiomatic framework proposed by Biswas and Enea [5]. These frameworks were introduced to capture more recent isolation levels, including Read Atomic [3] and Prefix Consistency [6]. Our approach supports them by enabling precise modeling of both certain and uncertain dependencies, e.g., commit order [5], which, like the WW relation, models the version order between writes. As a demonstration, we have encoded Biswas and Enea's framework using hyper-polygraphs; see the Appendix [7].

Suppoting Weak Isolation Levels. While our dedicated SMT solving is highly efficient for verifying strong isolation levels, it may be less suitable for weaker ones such as Read Committed, where SMT may be unnecessarily heavyweight [23]. In practice, users can combine VeriStrong with existing weak isolation verifiers [23, 29] to achieve broader coverage. This integration is straightforward in the black-box setting, as all tools share a unified input format. We have realized such an integration with the weak isolation verifier Plume in the IsoVista system [14].

Limitations. While our work lifts the strong UniqueValue assumption made by prior verifiers, it currently targets key-value databases. Extending our approach to relational models—particularly to support predicate reads—is an interesting direction for future work, potentially by building on techniques from [18, 39].

In addition, our current implementation is designed for offline verification. While it can, in principle, be applied in online settings, doing so efficiently would require addressing runtime-specific challenges, such as garbage collection of historical transactions.

Conclusion. We have presented a novel approach for verifying strong database isolation, centered around hyper-polygraphs—a general formalism for modeling dependency-based isolation semantics. We have focused on Adya's theory and established sound and complete characterizations for both Serializability and Snapshot Isolation. Our verifier achieves high efficiency by tailoring SMT solving to database workload characteristics, in contrast to the state-of-the-art that relies solely on general-purpose solvers.

As future work, beyond addressing the aforementioned challenges, we plan to explore the synergy between our history verification approach and recent advances in workload generation [22], an orthogonal line of research that has shown strong potential for uncovering isolation anomalies.

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.

[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 (July 2016).

[4] Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. 2015. SAT Modulo Monotonic Theories. In *AAAI 2015*. AAAI Press, 3702–3709.

[5] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 165 (2019).

[6] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *ECOOP 2015 (LIPIcs, Vol. 37)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 568–590.

[7] Zhiheng Cai, Si Liu, Hengfeng Wei, Yuxing Chen, and Anqun Pan. 2025. *Fast Verification of Strong Database Isolation.* Technical Report. https://github.com/CzxingcHen/VeriStrong.

[8] Andrea Cerone and Alexey Gotsman. 2018. Analysing Snapshot Isolation. *J. ACM* 65, 2, Article 11 (Jan. 2018).

[9] Jack Clark, Alastair F. Donaldson, John Wickerson, and Manuel Rigger. 2024. Validating Database System Isolation Level Implementations with Version Certificate Recovery. In *EuroSys '24*. ACM, 754–768.

[10] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397.

[11] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003 (LNCS, Vol. 2919)*. Springer, 502–518.

[12] Hongyu Fan, Zhihang Sun, and Fei He. 2023. Satisfiability Modulo Ordering Consistency Theory for SC, TSO, and PSO Memory Models. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 6 (March 2023), 37 pages.

[13] Shabnam Ghasemirad, Si Liu, Christoph Sprenger, Luca Multazzu, and David Basin. 2025. VerIso: Verifiable Isolation Guarantees for Database Transactions. *Proc. VLDB Endow.* 18, 5 (2025), 1362–1375.

[14] Long Gu, Si Liu, Tiancheng Xing, Hengfeng Wei, Yuxing Chen, and David Basin. 2024. IsoVista: Black-Box Checking Database Isolation Guarantees. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 4325–4328.

[15] Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In *PLDI '21*. ACM, 1264–1279.

[16] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-box Checking of Snapshot Isolation in Databases. *Proc. VLDB Endow.* 16, 6 (2023), 1264–1276.

[17] Jepsen. Accessed in May, 2025. https://jepsen.io.

[18] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *OSDI 23*. USENIX Association, 397–417.

[19] Nick Kallen. Accessed in May, 2025. Big Data in Real Time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter/.

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

[21] Hexu Li, Hengfeng Wei, Hongrong Ouyang, Yuxing Chen, Na Yang, Ruohao Zhang, and Anqun Pan. 2025. Online Timestamp-based Transactional Isolation Checking of Database Systems. In *ICDE 2025*. IEEE Computer Society, 3738–3750.

[22] Keqiang Li, Siyang Weng, Lyu Ni, Chengcheng Yang, Rong Zhang, Xuan Zhou, and Aoying Zhou. 2024. DBStorm: Generating Various Effective Workloads for Testing Isolation Levels. In *ISSTA 2024*. ACM, 755–767.

[23] Si Liu, Long Gu, Hengfeng Wei, and David Basin. 2024. Plume: Efficient and Complete Black-Box Checking of Weak Isolation Levels. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 302 (Oct. 2024), 29 pages. doi:10.1145/3689742

[24] Si Liu, Luca Multazzu, Hengfeng Wei, and David A. Basin. 2024. NOC-NOC: Towards Performance-optimal Distributed Transactions. *Proc. ACM Manag. Data* 2, 1, Article 9 (March 2024), 25 pages.

[25] Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. 2019. Automatic Analysis of Consistency Properties of Distributed Transaction Systems in Maude. In *TACAS 2019 (LNCS, Vol. 11428)*. Springer, 40–57.

[26] MariaDB-#26642. 2022. Weird SELECT view when a record is modified to the same value by two transactions. https://jira.mariadb.org/browse/MDEV-26642.

[27] MariaDB-#35262. 2024. INT violation when two transactions modify a record to the same value concurrently. https://jira.mariadb.org/browse/MDEV-35262.

[28] Anders Alnor Mathiasen, Léon Gondelman, Léon Ducruet, Amin Timany, and Lars Birkedal. 2025. Reasoning about Weak Isolation Levels in Separation Logic. *Proc. ACM Program. Lang.* ICFP (2025).

[29] Lasse Møldrup and Andreas Pavlogiannis. 2025. AWDIT: An Optimal Weak Database Isolation Tester. *Proc. ACM Program. Lang.* 9, PLDI, Article 209 (June 2025), 25 pages.

[30] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: engineering an efficient SAT solver. In *DAC '01*. ACM, 530–535.

[31] MySQL-#100328. 2020. Inconsistent behavior with isolation levels when binlog enabled. https://bugs.mysql.com/bug.php?id=100328.

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

[33] David J. Pearce and Paul H. J. Kelly. 2006. A dynamic topological sort algorithm for directed acyclic graphs. *ACM J. Exp. Algorithmics* 11 (2006).

[34] RUBiS. Accessed in May, 2025. Auction Site for e-Commerce Technologies Benchmarking. https://projects.ow2.org/view/rubis/.

[35] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI 2020*. USENIX Association, 63–80.

[36] TPC. Accessed in October, 2023. TPC-C: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

[37] Hengfeng Wei, Jiang Xiao, Na Yang, Si Liu, Zijing Yin, Yuxing Chen, and Anqun Pan. 2025. Boosting End-to-End Database Isolation Checking via Mini-Transactions. In *ICDE 2025*. IEEE Computer Society, 3998–4010.

[38] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann.

[39] Rui Yang, Ziyu Cui, Wensheng Dou, Yu Gao, Jiansen Song, Xudong Xie, and Jun Wei. 2025. Detecting Isolation Anomalies in Relational DBMSs. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA076 (June 2025), 23 pages.

[40] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *EuroSys 2023*. ACM, 654–671.

A Verifying Snapshot Isolation

To demonstrate the generality of our proposed techniques, this section summarizes how hyper-polygraphs can be used to characterize Snapshot Isolation (SI), a widely used strong isolation guarantee in practice, and how our optimizations extend naturally to verifying histories against SI.

Unlike verifying Serializability, where detecting any cycle is sufficient to determine a violation, verifying Snapshot Isolation is more complex, requiring identifying cycles that do not contain two adjacent RW edges [8, Theorem 4.1]. Inspired by prior work [16], we define the *induced SI graph* based on a directed labeled graph that is compatible with the hyper-polygraph of a history (as specified in Definition 3.3).

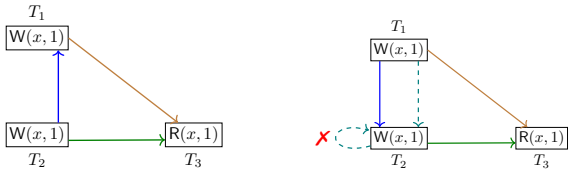
Definition A.1. Let $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ be a graph compatible with a hyper-polygraph $\mathcal{G}$. Let $\mathcal{E}_{\mathsf{T}}$ ($\mathsf{T} \in \mathsf{Type}$) denote the subset of $\mathcal{E}$ that consists of edges labeled with T. The *induced SI graph* of $\mathcal{G}'$ is defined as

$$\mathcal{G}'_{\mathrm{I}} = (\mathcal{V}', (\mathcal{E}'_{\mathsf{SO}} \cup \mathcal{E}'_{\mathsf{WR}} \cup \mathcal{E}'_{\mathsf{WW}}) \ ; \ \mathcal{E}'_{\mathsf{RW}}?).^{6}$$

We can then characterize Snapshot Isolation using hyper-polygraphs as follows. Illustrative examples are given below, and the proof is provided in Section B.2.

Theorem A.2. *A history $\mathcal{H}$ satisfies* Snapshot Isolation *if and only if $\mathcal{H} \models$ Int and there exists a graph compatible with the hyper-polygraph of $\mathcal{H}$ whose induced SI graph is acyclic.*

The following example illustrates the induced SI graph, reusing the examples in Figure 3.



(a) The induced SI graph of the graph in Figure 3c. **(b) The induced SI graph of the graph in Figure 3d.**

Figure 16: Induced SI graphs of compatible graphs in Figure 3.

Example A.3. We reuse the history in Example 3.2 (in Figure 3) to illustrate induced SI graphs. Figure 16 shows induced SI graphs of the compatible graphs in Figure 3. The edges of induced SI graph of the graph in Figure 3c (shown in Figure 16a) are the same as the compatible graph, since there is no RW edge. In constrast, the induced SI graph of the graph in Figure 3d (shown in Figure 16b) consists of two new induced edges (shown as dashed arrows): $T_1 \to T_2$ (induced from $T_1 \xrightarrow{\mathsf{WR}(x)} T_3$ and $T_3 \xrightarrow{\mathsf{RW}(x)} T_2$) and $T_2 \to T_2$ (induced from $T_2 \xrightarrow{\mathsf{SO}} T_3$ and $T_3 \xrightarrow{\mathsf{RW}(x)} T_2$). Note that $T_2 \to T_2$ is actually a self-loop, indicating that the induced SI graph in Figure 16b is cyclic.

[6] Here, we slightly abuse the notation of ; and ?, where $\mathcal{E}_{\mathsf{T}}$ is treated as a subset of $\mathcal{V} \times \mathcal{V}$ by selecting edges labeled T in $\mathcal{E}$ and ignoring keys .

The high-level workflow for verifying Snapshot Isolation is aligned with that for Serializability, as illustrated in Figure 7. The key difference lies in the theory solver: for Snapshot Isolation, the verification procedure maintains the induced SI graph constructed from the known graph. Both cycle detection and conflict clause generation are performed on this induced graph. Additionally, the small-width cycle optimization is adapted to prune and encode cycles in the induced SI graph, based on the composition rule (SO∪WR∪WW) ; RW?. The polarity picking heuristic applies straightforwardly in this setting.

B Proofs

B.1 Proof of Theorem 3.5

Proof. The proof proceeds in two directions.

("⇒") Suppose that $\mathcal{H}$ satisfies Serializability. By Theorem 2.4, $\mathcal{H}$ satisfies Int, and there exists WR, WW and RW relations with which $\mathcal{H}$ can be extended to a dependency graph $\mathcal{G}$ such that $\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}} \cup \mathsf{RW}_{\mathcal{G}}$ is acyclic. Denote the hyper-polygraph of $\mathcal{H}$ as $\mathcal{G}' \triangleq (\mathcal{V}', \mathcal{E}', \mathcal{C}')$. We construct an acyclic compatible graph $\mathcal{G}'' \triangleq (\mathcal{V}'', \mathcal{E}'')$ of $\mathcal{G}'$ based on $\mathcal{G}$, where $\mathcal{V}'' = \mathcal{V}'$ corresponds to transactions of $\mathcal{H}$ by definition, and $\mathcal{E}'' = \mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}} \cup \mathsf{RW}_{\mathcal{G}}$ with each edge labeled by its relation name. $\mathcal{G}''$ is acyclic and satisfies Definition 3.3.

("⇐") Suppose that $\mathcal{H}$ satisfies Int and their exists an acyclic compatible graph $\mathcal{G}''$ of the hyper-polygraph $\mathcal{G}'$ of $\mathcal{H}$. We construct a dependency graph $\mathcal{G}$ based on $\mathcal{G}''$, where $\mathsf{SO}_{\mathcal{G}}, \mathsf{WR}_{\mathcal{G}}, \mathsf{WW}_{\mathcal{G}}$ and $\mathsf{RW}_{\mathcal{G}}$ consist a partition of $\mathcal{E}''$ by their labels. For example, $\mathsf{SO}_{\mathcal{G}} = \bigcup_{T \xrightarrow{\mathsf{SO}} S} (T, S)$, and $\forall x \in \mathsf{Key}.\ \mathsf{WR}_{\mathcal{G}}(x) = \bigcup_{T \xrightarrow{\mathsf{WR}(x)} S} (T, S)$ (the same for WW and RW). $\mathcal{G}$ is a valid dependency graph by Definition 2.3. Since $\mathcal{G}''$ is acyclic, $\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}} \cup \mathsf{RW}_{\mathcal{G}}$ is also acyclic. $\mathcal{G}$ satisfies Serializability by Theorem 2.4. □

B.2 Proof of Theorem A.2

The proof of the hyper-polygraph-based characterization for Snapshot Isolation closely follows the same strategy as that of Serializability. Both directions of the construction – deriving a valid dependency graph from a compatible graph and vice versa – are analogous and can be reused verbatim.

Proof. This proof proceeds in two directions.

("⇒") Suppose that $\mathcal{H}$ satisfies Snapshot Isolation. By [8, Theorem 4.1], $\mathcal{H}$ satisfies Int, and there exists WR, WW and RW relations with which $\mathcal{H}$ can be extended to a dependency graph $\mathcal{G}$ such that $(\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}}) \ ; \ \mathsf{RW}_{\mathcal{G}}?$ is acyclic. Denote the hyper-polygraph of $\mathcal{H}$ as $\mathcal{G}' \triangleq (\mathcal{V}', \mathcal{E}', \mathcal{C}')$. We construct an acyclic compatible graph $\mathcal{G}'' \triangleq (\mathcal{V}'', \mathcal{E}'')$ of $\mathcal{G}'$ based on $\mathcal{G}$, where $\mathcal{V}'' = \mathcal{V}'$ corresponds to transactions of $\mathcal{H}$ by definition, and $\mathcal{E}'' = \mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}} \cup \mathsf{RW}_{\mathcal{G}}$ with each edge labeled by its relation name. The induced SI graph of $\mathcal{G}''$ is acyclic, and $\mathcal{G}''$ satisfies Definition 3.3.

("⇐") Suppose that $\mathcal{H}$ satisfies Int and their exists a compatible graph $\mathcal{G}''$ of the hyper-polygraph $\mathcal{G}'$ of $\mathcal{H}$ whose induced SI graph $\mathcal{G}''_{\mathrm{I}}$ is acyclic. We construct a dependency graph $\mathcal{G}$ based on $\mathcal{G}''$, where $\mathsf{SO}_{\mathcal{G}}, \mathsf{WR}_{\mathcal{G}}, \mathsf{WW}_{\mathcal{G}}$ and $\mathsf{RW}_{\mathcal{G}}$ consist a partition of

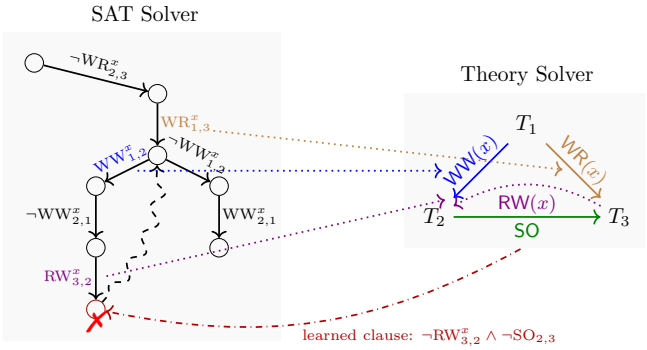


Figure 17: An illstration of how the SAT solver and the theory solver collaborate in CDCL(T) for the history $\mathcal{H}$ in Figure 3.

$\mathcal{E}''$ by their labels. For example, $\mathrm{SO}_{\mathcal{G}} = \bigcup_{T \xrightarrow{\mathrm{SO}} S} (T, S)$, and $\forall x \in$ Key. $\mathrm{WR}_{\mathcal{G}}(x) = \bigcup_{T \xrightarrow{\mathrm{WR}(x)} S} (T, S)$ (the same for WW and RW). $\mathcal{G}$ is a valid dependency graph by Definition 2.3. Since $\mathcal{G}_1''$ is acyclic, $(\mathrm{SO}_{\mathcal{G}} \cup \mathrm{WR}_{\mathcal{G}} \cup \mathrm{WW}_{\mathcal{G}})$; $\mathrm{RW}_{\mathcal{G}}?$ is also acyclic. $\mathcal{G}$ satisfies Snapshot Isolation by [8, Theorem 4.1]. □

C An Illustration of CDCL(T)

Figure 17 illustrates the interaction of the SAT solver and the theory solver when solving $\mathcal{H}$ in Figure 3. The solving procedure is identical to that in Figure 4; see Section 4.2.

D Other Algorithms

Algorithm 4 shows the process of constructing the hyper-polygraph $\mathcal{G}$ of a given history $\mathcal{H}$. It is called at Line 2 in Algorithm 1; see Section 6.1.

Algorithm 4 Construct the hyper-polygraph for a history $\mathcal{H}$

$\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$: the known graph; initially $(\mathcal{T}, \emptyset, (\emptyset, \emptyset))$

1: **function** Construct($\mathcal{H}$)
2: **for** $T, S \in \mathcal{T}$ such that $T \xrightarrow{\mathrm{SO}} S$
3: $\mathcal{E} \leftarrow \mathcal{E} \cup \{T \xrightarrow{\mathrm{SO}} S\}$
4: **for** $S \in \mathcal{T}$ such that $S \vdash \mathrm{R}(x, v) \wedge |\{T \in \mathcal{T} \mid T \vdash \mathrm{W}(x, v)\}| = 1$
5: $\mathcal{E} \leftarrow \mathcal{E} \cup \{T \xrightarrow{\mathrm{WR}(x)} S\}$
6: $C^{\mathrm{WW}} = \left\{\{T \xrightarrow{\mathrm{WW}(x)} S, S \xrightarrow{\mathrm{WW}(x)} T\} \mid T \in \mathrm{WriteTx}_x \wedge S \in \mathrm{WriteTx}_x \wedge T \neq S\right\}$
7: $C^{\mathrm{WR}} = \left\{\{\bigcup_{T_i \vdash \mathrm{W}(x,v)} \{T_i \xrightarrow{\mathrm{WR}(x)} S\}\} \mid S \vdash \mathrm{R}(x, v)\right\}$

Algorithm 5 shows the process of pruning the hyper-polygraph $\mathcal{G}$. It is called at Line 3 in Algorithm 1; see Section 6.1.

Algorithm 5 Prune

1: **function** Prune($\mathcal{G}$)
2: **repeat**
3: **for** $cons \leftarrow \{either \triangleq T \xrightarrow{\mathrm{WW}(x)} S, or \triangleq S \xrightarrow{\mathrm{WW}(x)} T\} \in C^{\mathrm{WW}}$
4: $either\_edges \leftarrow \{either\} \cup \{S' \xrightarrow{\mathrm{RW}(x)} S \mid T \xrightarrow{\mathrm{WR}(x)} S' \in \mathcal{E}\}$
5: $or\_edges \leftarrow \{or\} \cup \{T' \xrightarrow{\mathrm{RW}(x)} T \mid S \xrightarrow{\mathrm{WR}(x)} T' \in \mathcal{E}\}$
6: **if** $\mathcal{E} \cup either\_edges$ is cyclic
7: $cons \leftarrow C \setminus \{either\}$
8: $C^{\mathrm{WW}} \leftarrow C^{\mathrm{WW}} \setminus \{cons\}$
9: **if** $\mathcal{E} \cup or\_edges$ is cyclic
10: $cons \leftarrow C \setminus \{or\}$
11: $C^{\mathrm{WW}} \leftarrow C^{\mathrm{WW}} \setminus \{cons\}$
12: **if** $cons = \emptyset$
13: **return** false
14: **if** $cons = \{either\}$
15: $\mathcal{E} \leftarrow \mathcal{E} \cup or\_edges$
16: **if** $cons = \{or\}$
17: $\mathcal{E} \leftarrow \mathcal{E} \cup either\_edges$
18: **for** $cons \in C^{\mathrm{WR}}$
19: **for** $wr \leftarrow T \xrightarrow{\mathrm{WR}(x)} S \in cons$
20: $wr\_edges \leftarrow \{wr\} \cup \{S \xrightarrow{\mathrm{RW}(x)} T' \mid T \xrightarrow{\mathrm{WW}(x)} T'\}$
21: **if** $\mathcal{E} \cup wr\_edges$ is cyclic
22: $cons \leftarrow cons \setminus \{wr\}$
23: $C^{\mathrm{WR}} \leftarrow C^{\mathrm{WR}} \setminus \{cons\}$
24: **if** $cons = \emptyset$
25: **return** false
26: **if** $|cons| = 1 \wedge |cons| = \{wr \triangleq T \xrightarrow{\mathrm{WR}(x)} S\}$
27: $wr\_edges \leftarrow \{wr\} \cup \{S \xrightarrow{\mathrm{RW}(x)} T' \mid T \xrightarrow{\mathrm{WW}(x)} T'\}$
28: $\mathcal{E} \leftarrow \mathcal{E} \cup wr\_edges$
29: **until** $\mathcal{G}$ remains unchanged
30: **return** true

Algorithm 6 shows the process of computing reachability of a given known graph. It is introduced in the setup part of Section 7, as an auxiliary procedure in Algorithm 5 (Prune) and Algorithm 2 (EncodeCyclesOfWidth2) in acyclicity testing.

Algorithm 6 Compute reachability of the known graph

1: **function** Reachability($\mathcal{V}, \mathcal{E}$)
2: $\mathcal{R} \leftarrow 0^{|\mathcal{V}| \times |\mathcal{V}|}$
3: $reversed\_topo\_order \leftarrow$ TopologicalSort($\mathcal{V}, \mathcal{E}$)
4: **for** $x \in reversed\_topo\_order$
5: $\mathcal{R}_{xx} \leftarrow 1$
6: **for** $(x, y, \_, \_) \in \mathcal{E}$
7: $\mathcal{R}_x \leftarrow \mathcal{R}_x \cup \mathcal{R}_y$

Algorithm 7 shows the process of encoding. It is called at Line 5 in Algorithm 1; see Section 6.1.

Algorithm 7 Encode

1: **function** EncodeCyclesOfWidth2($\mathcal{V}, \mathcal{E}, C$)
2: **for** $\{T \xrightarrow{\mathrm{WW}(x)} S, S \xrightarrow{\mathrm{WW}(x)} T\} \in C^{\mathrm{WW}}$ ▷ *encode* WW *constraints*
3: $Vars \leftarrow Vars \cup \{\mathrm{WW}^x_{T,S}, \mathrm{WW}^x_{S,T}\}$
4: $Clauses \leftarrow Clauses \cup \{\mathrm{WW}^x_{T,S} \vee \mathrm{WW}^x_{S,T}, \neg\mathrm{WW}^x_{T,S} \vee \neg\mathrm{WW}^x_{S,T}\}$
5: **for** $cons \in C^{\mathrm{WR}}$ ▷ *encode* WR *constraints*
6: $Vars \leftarrow Vars \cup \{\mathrm{WR}^x_{T,S} \mid T \xrightarrow{\mathrm{WR}(x)} S \in cons\}$
7: $Clauses \leftarrow Clauses \cup \{ \bigvee_{T \xrightarrow{\mathrm{WR}(x)} S \in cons} \mathrm{WR}^x_{T,S} \}$
8: EncodeCyclesOfWidth2($\mathcal{V}, \mathcal{E}$) ▷ *Algorithm 2*

To generate conflict clauses, we introduce a boolean expression *reason* for each edge in G^*. Let $\mathbb{F}$ denote the set of all boolean formulas. Formally, we define $G^* \triangleq (\mathcal{V}, E^*)$, where $E^* \subseteq \mathcal{V} \times \mathcal{V} \times \mathbb{F}$. We write $T \xrightarrow{reason} S$ to denote an edge $(T, S, reason)$ in G^*, where $reason \in \mathbb{F}$ is the reason for the edge from T to S. Edges in the initial known graph are assigned the reason true (Line 9 in Algorithm 1).

Algorithm 8 (DeriveRWEdges) is called at Line 27 in Algorithm 1. It shows the procedure of generating *edges*, which includes the WW (or WR) edge corresponding to the assigned boolean variable and the derived RW edges. The reason of a WW or WR edge is the corresponding boolean variable. The reason of an RW edge is the conjunction of the reasons of the WW edge and the WR edge from where it is derived (see Line 4 and Line 6).

Algorithm 9 (GenConflictClause) is called at Line 29 in Algorithm 1. Given a detected cycle, the conflict clause is the negation of the conjunction of the reasons of the edges involved in the cycle (see Line 3).

Algorithm 8 Derive RW edges

1: **function** DeriveRWEdges($\mathcal{M}$)
2: $edges \leftarrow \emptyset$
3: **for** $\mathrm{WW}^x_{T,T'} \in \mathcal{M}$ ▷ WW *variables that are assigned true*
4: $edges \leftarrow edges \cup \{T \xrightarrow{\mathrm{WW}^x_{T,T'}} T'\} \cup \{S \xrightarrow{\mathrm{WW}^x_{T,T'}} T' \mid T \xrightarrow{\mathrm{WR}(x)} S \in \mathcal{E}\} \cup \{S \xrightarrow{\mathrm{WW}^x_{T,T'} \wedge \mathrm{WR}^x_{T,S}} T' \mid T \xrightarrow{\mathrm{WR}^x_{T,S}} S \in E^*\}$
5: **for** $\mathrm{WR}^x_{T,S} \in \mathcal{M}$ ▷ WR *variables that are assigned true*
6: $edges \leftarrow edges \cup \{T \xrightarrow{\mathrm{WR}^x_{T,S}} S\} \cup \{S \xrightarrow{\mathrm{WR}^x_{T,S}} T' \mid T \xrightarrow{\mathrm{WW}(x)} T' \in \mathcal{E}\} \cup \{S \xrightarrow{\mathrm{WW}^x_{T,T'} \wedge \mathrm{WR}^x_{T,S}} T' \mid T \xrightarrow{\mathrm{WW}^x_{T,T'}} T' \in E^*\}$
7: **return** *edges*

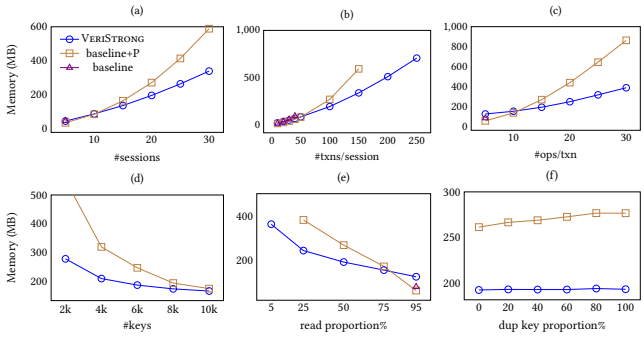


Figure 18: Peak memory usage comparison of VeriStrong and the baseline (Section 4) on various general workloads. Timeout data points are not plotted.

Algorithm 9 Generate conflict clause

1: **function** GenConflictClause(E^*, *edges*)
2: $cycle \triangleq \{T_1 \xrightarrow{reason_1} T_2, T_2 \xrightarrow{reason_2} T_3, \cdots, T_n \xrightarrow{reason_n} T_1\} \leftarrow$ Cycle($E^* \cup edges$)
3: $conflict\_clause \leftarrow \bigvee_{T_i \xrightarrow{reason_i} T_{i+1} \in cycle} (\neg reason_i)$
4: **return** *conflict_clause*

E Memory Usage Comparison

Figure 18 shows the peak memory usage of VeriStrong and the baseline on various general workloads, complementing Figure 10; see Section 7.2.1.

F Characterizing Biswas et al.'s Theory

In the theory of Biswas et al. [5], a history satisfies a given isolation level if there exists a strict total order CO (called *commit order*) on its transactions that extends both the WR and SO relations and satisfies certain additional properties. In this section, we show how to characterize isolation levels under this formalism using hyper-polygraphs, taking Serializability as a representative example.

Serializability requires that for any transaction T_1 writing to a key x that is read by a transaction T_3, every commit predecessor of T_3 that also writes to x must precede T_1 in commit order. That is,

Theorem F.1. *For a history* $\mathcal{H} = (\mathcal{T}, \mathrm{SO})$,

$$\mathcal{H} \models \mathrm{SER} \iff \mathcal{H} \models \mathrm{Int} \wedge \exists \mathrm{CO}.\, \forall x \in \mathrm{Key}.\, \forall T_1, T_2, T_3 \in \mathcal{T}.$$
$$T_1 \neq T_2 \wedge T_1 \xrightarrow{\mathrm{WR}(x)} T_3 \wedge T_2 \vdash \mathrm{W}(x, \_) \wedge$$
$$(T_2, T_3) \in \mathrm{CO} \implies (T_2, T_1) \in \mathrm{CO}.$$

The definition of hyper-polygraph under Biswas et al.'s formalism is a straightforward adaptation of Definition 3.1, with the constraint set C^{WW} replaced by C^{CO}.

Definition F.2. A hyper-polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ for a history $\mathcal{H} = (\mathcal{T}, \mathrm{SO})$ is a directed labeled graph $(\mathcal{V}, \mathcal{E})$, referred to as the known graph, together with *a pair of constraint sets* $C = (C^{\mathrm{CO}}, C^{\mathrm{WR}})$, where

- $\mathcal{V}$ is the set of nodes, corresponding to the transactions in $\mathcal{H}$;

- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathsf{Type} \times \mathsf{Key}$ is a set of edges, where each edge is labeled with a dependency type from $\mathsf{Type} = \{\mathsf{SO}, \mathsf{WR}, \mathsf{CO}\}$ and a key from Key;
- C^{CO} is a constraint set over uncertain version orders, defined as $C^{\mathsf{CO}} = \left\{\{T \xrightarrow{\mathsf{CO}} S, S \xrightarrow{\mathsf{CO}} T\} \mid T, S \in \mathcal{T} \wedge T \neq S\right\}$; and
- C^{WR} is a constraint set over uncertain read-from mappings, defined as $C^{\mathsf{WR}} = \left\{ \bigcup_{T_i \vdash \mathsf{W}(x,v)} \{T_i \xrightarrow{\mathsf{WR}(x)} S\} \mid S \vdash \mathsf{R}(x,v) \right\}$.

A compatible graph of such a hyper-polygraph resolves the uncertainty in CO and WR, while enforcing the required properties of CO.

Definition F.3. A directed labeled graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ is said to be *SER-compatible* with a hyper-polygraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, C)$ if

- $\mathcal{V}' = \mathcal{V}$;
- $\mathcal{E}' \supseteq \mathcal{E}$ such that $\forall T_1, T_2, T_3 \in \mathcal{V}.\ T_1 \neq T_2 \wedge T_1 \xrightarrow{\mathsf{WR}(x)} T_3 \wedge T_2 \in \mathsf{WriteTx}_x \wedge T_2 \xrightarrow{\mathsf{CO}} T_3 \implies T_2 \xrightarrow{\mathsf{CO}} T_1$;
- $\forall C \in C^{\mathsf{CO}}.\ |\mathcal{E}' \cap C| = 1$; and
- $\forall C \in C^{\mathsf{WR}}.\ |\mathcal{E}' \cap C| = 1$.

Consequently, the hyper-polygraph-based characterization of Serializability in Biswas et al.'s formalism follows directly from Theorem F.1.

Theorem F.4. *A history $\mathcal{H}$ satisfies* Serializability *if and only if $\mathcal{H} \models$ Int and there exists an acyclic graph that is SER-compatible with the hyper-polygraph of $\mathcal{H}$.*

Similarly, other isolation guarantees defined in Biswas et al.'s framework can also be characterized using hyper-polygraphs. Note that Read Committed requires a finer-grained modeling of the program order (po).