# Polygraph: From Transaction Isolation Guarantees to Isolation Verification

Jian Zhang
Northeastern University
Boston, Massachusetts, USA
zhang.jian3@northeastern.edu

Cheng Tan
Northeastern University
Boston, Massachusetts, USA
naizhengtan@gmail.com

## ABSTRACT

Isolation levels, such as serializability, define the correctness contract of database systems. As databases migrate to cloud and geo-distributed deployments and increasingly operate as black-box services, isolation can no longer be assumed—it must be verified. Existing formal definitions of isolation levels were not designed for efficient checking. Some depend on internal metadata unavailable to clients; others require expensive state exploration or yield disjoint checking procedures across isolation levels. No unified theoretical foundation exists for practical verification.

We present *Polygraph*, the first theoretical framework explicitly designed for efficient isolation checking. Polygraph models unknown dependencies in practice, supports white-box to black-box settings without assumptions on workloads, and unifies checking under a single algorithmic paradigm. We prove its equivalence to prior isolation level definitions and implement checkers that demonstrate practical and efficient checking across realistic workloads and configurations.

**VLDB Workshop Reference Format:**
Jian Zhang and Cheng Tan. Polygraph: From Transaction Isolation Guarantees to Isolation Verification. VLDB 2026 Workshop: 1st Symposium on Checking Consistency Principles.

## 1 INTRODUCTION

Isolation levels define the correctness contract of concurrent database systems. They form the foundation of database concurrency specifications: when a system violates its declared isolation level, applications may observe inconsistent states, resulting in ill-defined and potentially unsafe application behavior. These violations are not theoretical corner cases; they can invalidate safety checks, corrupt financial records, and undermine user trust [13, 20, 22].

Isolation levels inherently trade consistency for performance. Stronger guarantees typically incur higher coordination cost and limit scalability. In distributed settings, the CAP theorem [14] further constrains what is achievable under failures, forcing systems to adopt weaker isolation levels in exchange for availability or partition tolerance. As a result, modern deployments routinely operate under weaker isolation levels with relaxed guarantees [1, 15, 21, 25].


Proceedings of the VLDB Endowment. ISSN 2150-8097.

This shift makes isolation checking essential rather than optional [7, 12, 16, 28]. First, computation increasingly runs in outsourced and cloud environments, where databases operate as black boxes and users cannot directly observe execution. Second, modern databases rely on distributed replication protocols such as Paxos [17] and Raft [19] to tolerate failures. These protocols introduce complexity through sophisticated coordination logic, leader election, log replication, reconfiguration, and corner cases under partitions. Geo-replication further multiplies configuration parameters and failure modes. Third, today's database systems span multiple software layers—transaction managers, storage engines, replication modules, networking stacks, and underlying hypervisors. Cross-layer interactions may introduce subtle bugs that can violate isolation without obvious symptoms.

Given this complexity, isolation levels should be validated rather than trusted. This paper studies *isolation level checking*: given a database execution trace, a checker determines whether the trace satisfies a given isolation level (e.g., serializability). After examining existing literature, we identified four critical goals for an ideal isolation checking framework:

- *Abstraction*: Users should reason about logical transaction histories, independent of internal concurrency protocols (e.g., MVCC) [8, 11].
- *Interpretability*: Primitives must clearly explain illegal executions, mapping real-world anomalies to the formal model.
- *Adaptability*: The framework must support checking with minimal observable information (e.g., black-box environments) while gracefully leveraging any available internal scheduling metadata. It must also operate correctly under realistic configurations, including duplicated values [2].
- *Unification*: Both the formal properties and the checking algorithms should be uniform across major isolation levels.

However, the foundations of isolation levels—their formal definitions—do not fully support these needs. Table 1 summarizes how existing frameworks fall short of these combined goals. In brief, the ANSI definition [5] is incomplete. Adya's graphs [3] require internal database state (lacking adaptability). Cerone's axioms [8] use abstract relations that are unintuitive and split algorithms across levels (lacking interpretability and unification). Crooks's states [11] incur prohibitive search spaces. (A detailed comparison is in Appendix A.)

The inability of prior work to satisfy all four properties concurrently motivates our design of the *Polygraph Framework*, a unified foundation for efficiently checking isolation levels.[1] Polygraph is designed with isolation checking as a first-class goal. It enables the

[1]This paper is a part of Jian Zhang's doctoral dissertation [27].

**Table 1: Comparison of isolation level frameworks. B/G/W refer to black-/grey-/white-box respectively.**

| Framework | ANSI SQL-92 [5] | Adya's [3] | Cerone's [8] | Crooks's [11] | **Polygraph (ours)** |
|---|---|---|---|---|---|
| Setup | B | W | B/G/W | B | B/G/W |
| Abstraction | ✗ | ✓ | ✓ | ✓ | ✓ |
| Interpretability | ✓ | ✓ | ✗ | ✓ | ✓ |
| Adaptability | ✗ | ✗ | ✓ | ✗ | ✓ |
| Unification | ✗ | ✓ | ✗ | ✓ | ✓ |

implementation of efficient checkers through definitions that are operational, configurable, and adaptable to diverse system settings. Further, the framework exposes concrete structures that explain observed violations, enabling users to understand isolation failures rather than only detecting them. At the same time, it captures all major isolation levels within a unified formal model.

We make three contributions. First, we introduce the polygraph framework, the first theoretical framework explicitly designed for efficient isolation checking. Second, we prove its equivalence to prior formalizations of isolation levels. Third, we demonstrate, through implemented checkers, that polygraph enables practical verification across realistic workloads and configurations.

# 2 THE POLYGRAPH FRAMEWORK FOR CHARACTERIZING ISOLATION LEVELS

To verify isolation guarantees, we must first formalize the interactions between client applications and the database. We begin by defining the system model and then introduce the *Polygraph Framework* to characterize isolation levels by modeling the uncertainty inherent in observing histories from a black-box perspective.

## 2.1 System Model

We model a database system as a set of transactions operating on versioned objects.

*Definition 2.1 (Transaction).* A transaction $T = (M, PO)$, where $PO$ is a total order over a finite set of operations $M$. Operations include: $\text{read}(x) \rightarrow v$, $\text{write}(x, v)$, $\text{delete}(x)$, $\text{range}(k_{min}, k_{max})$, and $\text{iter}(k_{min}, k_{max})$. Unlike range which retrieves all keys in the left-closed, right-open interval $[k_{min}, k_{max})$, iter may halt early and retrieve only a partial sequence.

These five operations cover the standard APIs of modern key-value stores. Explicitly modeling range, delete and iter operations allows checkers to capture predicate dependencies (essential for phantoms) often omitted in simpler formalisms.

*Definition 2.2 (History & Execution).* A *history* $h$ is a sequence of committed transactions and their return values. An *execution* is a history where all dependencies, including write-write ($WW$), write-read ($WR$), predicate-write-read ($PWR$), read-write ($RW$), and predicate-read-write ($PRW$), are perfectly known.

In black-box environments, clients observe the history, but the underlying execution remains hidden.

## 2.2 Dependency Graphs

To reason about isolation violations, we visualize executions as graphs. We reformulate Adya's Direct Serialization Graph (DSG) [3] for convenience.

*Definition 2.3 (Direct Dependency Graph).* Given a history $h$ and version order, a DSG $G^l_{direct}(h) = (V, E)$ for isolation level $l$ consists of committed transactions $V$ and dependencies $E$. Edges include: $T_i \xrightarrow{\text{wr(x)}} T_j$ (write-read), $T_i \xrightarrow{\text{ww(x)}} T_j$ (write-write), $T_i \xrightarrow{\text{rw(x)}} T_j$ (read-write), $T_i \xrightarrow{\text{pwr(x)}} T_j$ (predicate-write-read), and $T_i \xrightarrow{\text{prw(x)}} T_j$ (predicate-read-write, caused by $T_j$ altering a key matching $T_i$'s range query). A complete direct dependency graph involves all the edge types above, but specific isolation levels may choose to exclude some types of edges.

While the DSG captures direct interactions, isolation checking requires reasoning about transitive and implied orderings. We therefore define the *Canonical Augmented Dependency Graph* (CADG), which extends the DSG with indirect write-dependencies, indirect anti-dependencies and indirect predicate-anti-dependencies.

*Definition 2.4 (Canonical Augmented Dependency Graph).* The CADG $G^l_{aug}(h)$ extends $G^l_{direct}(h)$ by inferring indirect dependencies. Specifically, it adds indirect write-dependencies (transitive writes to the same object), and for levels involving anti-dependencies, e.g., SER, it adds indirect anti-dependencies and predicate-anti-dependencies whenever a later version is installed after the version read by a transaction's read or predicate read.

## 2.3 Polygraph: Modeling Unknowns

The Polygraph framework bridges the gap between observable histories and hidden executions. It models ambiguity using: *known edges* (observable dependencies), *Superposition* (unknown orderings), and *Implications* (inferred dependencies).

*Definition 2.5 (Superposition).* A superposition is a tuple of alternative *possibilities*: $\langle possibility_1, \ldots, possibility_n \rangle$ where exactly one is true, capturing unknown dependencies.

(1) **Type 1 (Read Sources):** If a read $T_r$ may read from multiple potential writes $T_{w1}, \ldots, T_{wn}$, then $\langle T_{w1} \xrightarrow{\text{wr(x)}} T_r, \cdots, T_{wn} \xrightarrow{\text{wr(x)}} T_r \rangle$.
(2) **Type 2 (Write Orders):** Concurrent transactions $T_i, T_j$ writing the same key must be ordered, leading to $\langle T_i \xrightarrow{\text{ww(x)}} T_j, T_j \xrightarrow{\text{ww(x)}} T_i \rangle$.

*Definition 2.6 (Implication).* Implications are used to infer anti-dependencies from write-dependencies and read-dependencies.

(1) **Type 1:** $T_{wj} \xrightarrow{\text{ww(x)}} T_{pl} \wedge T_{wj} \xrightarrow{\text{wr(x)}} T_r \Rightarrow T_r \xrightarrow{\text{rw(x)}} T_{pl}$.

(2) **Type 2:** If $T_{pl}$ alters $x$'s matches in $T_r$, $T_{wj} \xrightarrow{\text{ww(x)}} T_{pl} \wedge T_{wj} \xrightarrow{\text{pwr(x)}} T_r \Rightarrow T_r \xrightarrow{\text{prw(x)}} T_{pl}$.

The polygraph abstraction was previously used by Cobra [23] for black-box serializability checking. We generalize it to multiple isolation levels.

*Definition 2.7 (Polygraph).* A Polygraph $\mathcal{G}^l(\mathcal{V}, \mathcal{E}, \mathcal{S}, \mathcal{I})$ for isolation level $l$ consists of committed transactions $\mathcal{V}$, known dependencies $\mathcal{E}$, superpositions $\mathcal{S}$, and implications $\mathcal{I}$.

(1) $\forall$ read of $x$ by $T_r$, if its write transaction $T_w$ is known, $T_w \xrightarrow{\text{wr(x)}} T_r \in \mathcal{E}$; otherwise, the superposition of all potential writes is in $\mathcal{S}$.

(2) $\forall x$, $\forall$ writers $T_{wi}, T_{wj}$ of the same data object $x$, add $\langle T_{wi} \xrightarrow{\text{ww(x)}} T_{wj}, T_{wj} \xrightarrow{\text{ww(x)}} T_{wi} \rangle$ to $\mathcal{S}$.

(3) $\forall T_w \xrightarrow{\text{wr(x)}} T_r \in \mathcal{E}$ (or potential read in $\mathcal{S}$) and writer $T_k$ of $x$, $T_k \neq T_w$, add a Type 1 implication to $\mathcal{I}$. Type 2 implications are added analogously for predicate reads.

While a complete Polygraph incorporates all components, specific isolation levels may selectively exclude certain edge types or implications (Section 3).

The notion of compatibility was introduced for black-box snapshot-isolation checking [16]. We extend this notion to the Polygraph Framework: a canonical augmented dependency graph is compatible with a Polygraph if it preserves known dependencies and satisfies all superpositions and implications.

*Definition 2.8 (Compatibility).* A CADG $G^l_{aug}$ is *compatible* with $\mathcal{G}^l$ if they share vertices $\mathcal{V}$, and $G^l_{aug}$ includes all edges in $\mathcal{E}$ while satisfying all superpositions in $\mathcal{S}$ and implications in $\mathcal{I}$.

Intuitively, if you pick one possibility for all the superpositions $S$ of the Polygraph, you will get a canonical augmented dependency graph that is compatible with the Polygraph.

*Definition 2.9 (Structural Properties of Polygraph).* A Polygraph $\mathcal{G}^l$ is *acyclic* (or contains a specific cycle type) if there exists at least one compatible $G^l_{aug}$ that is acyclic (or contains that cycle).

# 3 ISOLATION LEVELS

In this section, we will show how to use Polygraph to characterize various isolation levels simply by enabling specific types of dependency edges and implication rules. Table 2 summarizes the configuration for Serializability (SER) [4, §3.2.3], Repeatable Read (RR) [4, §3.2.4], Read Committed (RC) [4, §3.2.2], Snapshot Isolation (SI) [6, 9], and PL-2+ [4, §4.1]. We prove that our Polygraph definitions of SER, SI, RR, and RC are equivalent to their prior formal definitions; detailed proofs appear in Appendix C–F.

## 3.1 Serializability

Serializability (SER) is the strongest isolation level if without real-time ordering, requiring that the concurrent execution be equivalent to some serial ordering of transactions. We enable all dependency edges and enforce all implications in the Polygraph. Consequently, the absence of cycles guarantees a valid serial ordering. We defer the full proof of Theorem 3.1 to Appendix C.

Theorem 3.1 (Serializability). *For a history $h$, Polygraph $\mathcal{G}(h)$ is acyclic $\iff$ $h$ is serializable.*

## 3.2 Repeatable read

Repeatable read (RR) differs from serializability in that it does not prevent new items from appearing in a range scan (phantoms). Then, RR is characterized by disabling Type 2 Implications to exclude predicate-anti-dependencies. We prove our RR Polygraph $\mathcal{G}^{RR}(h)$ is equivalent to Adya's PL-2.99 [4] in Appendix E.

Theorem 3.2 (Repeatable read). *$\mathcal{G}^{RR}(h)$ is acyclic $\iff$ the history $h$ is repeatable read.*

## 3.3 Snapshot isolation

For snapshot isolation (SI), the node set $V$ maps each transaction $T_i$ to two event nodes [28, 29]: $\{B_i, C_i\}$. The dependency edges $E$ are instantiated between these events based on their logical temporal semantics (e.g., an anti-dependency dependency $T_i \xrightarrow{\text{rw(x)}} T_j$ becomes $B_i \xrightarrow{\text{rw(x)}} C_j$, representing that the read snapshot happened before the overwrite). Table 3 summarizes the complete edge mapping.

The SI Polygraph $\mathcal{G}^{SI}$ transforms $\mathcal{G}^{SER}$ by splitting transaction nodes into Begin/Commit pairs, adding intra-transaction edges $B_i \rightarrow C_i$, and mapping all known edges, superpositions, and implications using Table 3.

*Definition 3.3 (SI Polygraph).* Given a history $h$, the SI Polygraph $\mathcal{G}^{SI}(h)$ is defined as a transformation of the standard Polygraph $\mathcal{G}^{SER} = (V^{SER}, E^{SER}, S^{SER}, I^{SER})$. $\mathcal{G}^{SI}$ is the tuple $(V^{SI}, E^{SI}, S^{SI}, I^{SI})$ constructed as follows:

(1) If any transaction has aborted/intermediate reads(G1a/G1b), the $h$ is invalid for SI Polygraph and the construction stops.

(2) *Nodes ($V^{SI}$):* For every transaction $T_i \in V^{SER}$, create two nodes $B_i$ (Begin) and $C_i$ (Commit).

(3) *Edges ($E^{SI}$):* (1) Add the intra-transaction edge $B_i \rightarrow C_i$ for all $T_i$; (2) Map every dependency edge in $E^{SER}$ to its corresponding SI edge according to Table 3.

(4) *Superpositions ($S^{SI}$):* Apply the mapping from Table 3 to all edges within the superpositions in $S^{SER}$. For example, a Type 1 superposition transforms from $\langle T_{w1} \rightarrow T_r, \dots \rangle$ to: $\langle C_{w1} \rightarrow B_r, C_{w2} \rightarrow B_r, \cdots, C_{wn} \rightarrow B_r \rangle \in S_{SI}$.

(5) *Implications ($I^{SI}$):* Apply the mapping to all implications in $I^{SER}$. For example, the standard Type 1 implication: $T_w \xrightarrow{\text{wr(x)}} T_r \wedge T_w \xrightarrow{\text{ww(x)}} T_k \implies T_r \xrightarrow{\text{rw(x)}} T_k$ in $I^{SER}$ transforms into the SI implication: $C_w \xrightarrow{\text{wr(x)}} B_r \wedge C_w \xrightarrow{\text{ww(x)}} B_k \implies B_r \xrightarrow{\text{rw(x)}} C_k \in I_{SI}$

Snapshot isolation is characterized by the theorem below. Its proof is deferred to Appendix D.

Theorem 3.4. *Given a history $h$, its SI Polygraph $G^{SI}(h)$ is acyclic $\iff$ $h$ is SI.*

**Table 2: Polygraph configuration for isolation levels**

| Isolation Level | Node Granularity ($V$) | Graph Edges Monitored ($E$) | Implications Active ($I$) | Proscribes |
|---|---|---|---|---|
| SER | $\{T_i\}$ | All ($wr$, $ww$, $rw$, $pwr$, $prw$) | All (Type 1 & 2) | Any cycle |
| RR | $\{T_i\}$ | Excludes $prw$ | Excludes Type 2 ($prw$) | Any cycle |
| RC | $\{T_i\}$ | Excludes $rw$, $prw$ | None ($I = \emptyset$) | Any cycle |
| SI | $\{B_i, C_i\}$ | All deps.<br>+ intra-txn dependencies | All (Type 1 & 2) | Any cycle |
| PL-2+ | $\{T_i\}$ | All ($wr$, $ww$, $rw$, $pwr$, $prw$) | All (Type 1 & 2) | Cycles with<br>0/1 anti edges |

**Table 3: Transformation mapping: standard Polygraph → SI Polygraph**

| Dependency | Standard Edge | SI Mapped Edge |
|---|---|---|
| Write-Read | $T_i \xrightarrow{\text{wr(x)}} T_j$ | $C_i \to B_j$ |
| Write-Write | $T_i \xrightarrow{\text{ww(x)}} T_j$ | $C_i \to B_j$ |
| Read-Write | $T_i \xrightarrow{\text{rw(x)}} T_j$ | $B_i \to C_j$ |
| Pred. Read | $T_i \xrightarrow{\text{pwr(x)}} T_j$ | $C_i \to B_j$ |
| Pred. Anti-Dep | $T_i \xrightarrow{\text{prw(x)}} T_j$ | $B_i \to C_j$ |

**Table 4: Lines of code (LOC) for each checker.**

| Checker | Total LOC |
|---|---|
| Strict serializability | 245 |
| Serializability | 241 |
| Repeatable read | 249 |
| Snapshot isolation | 285 |
| Read committed | 238 |
| PL-2+ | 265 |

**Table 5: Verification runtime across databases, isolation levels, and benchmarks.**

| Isolation | Database | #Txns | Benchmark | Runtime |
|---|---|---|---|---|
| SSER | Tapir [26] | 13.2K | BlindW-T | 1.12s |
| SER | PostgreSQL 15.2 [21] | 10.1K | BlindW-C | 4.5s |
| SI | TiKV 7.6.0 [24] | 10K | RandomBench | 365.47s |
| RR | MySQL 9.5.0 [1] | 10K | BlindW | 2.15s |
| PL-2+ | Port [18] | 0.9K | BlindW-E | 0.12s |
| RC | PostgreSQL 15.2 [21] | 10.1K | BlindW-C | 3.03s |

# 4 EVALUATION

In this section, we evaluate the Polygraph Framework along two dimensions: (1) its unification when implementing checkers of different isolation levels; (2) its adaptability of checking different configurations.

**Setup.** All the experiments are run on a desktop with a 12-core AMD Ryzen 5900, 64GB memory and Ubuntu 20.04.

## 4.1 Unified Checking Paradigm

We evaluate unification by building a verification tool for six isolation levels (RC [4], SER [4], SSER [26], RR [4], SI, and PL-2+ [4]) and measuring implementation complexity (Lines of Code, LOC).

**Shared Infrastructure vs. Specific Logic.** The tool totals 14K LOC. Over 98% is a one-time investment in shared infrastructure (trace parsing, graph construction, pruning strategies, and SMT encoding).

Specific checkers require just 238–285 LOC (under 2%), confirming that isolation levels only need modular constraint toggles rather than disparate algorithms:

- **RC, RR, SER:** Similar LOC (238–249) as weaker levels merely disable specific dependency edges.
- **SSER, SI:** Slightly larger (245, 285) to handle real-time order or node splitting (Section 3.3), keeping the core acyclicity logic identical.
- **PL-2+:** Needs 265 LOC to encode its specific cycle-prevention constraint.

In summary, Polygraph supports diverse isolation levels with negligible engineering effort by modularly configuring edges and constraints. Checking a new level simply requires updating the SMT encoding, disabling edge types or adjusting the node granularity.

## 4.2 Checking Diverse Workloads and Setups

We evaluate practicality by checking six isolation levels across five databases.

**Benchmarks.** We employed two benchmark categories. First, BlindW variants (BlindW-T [26], BlindW-C [10], BlindW-E [18] from prior work and our own) consist of blind writes with unique values. While read-dependencies are known, the massive unknown write-write order forces the solver to infer a valid version history.

Second, we developed RandomBench to include all the operations that we support, especially including range queries, true deletions, and iterators with non-unique values. This further maximizes Type 1 superpositions compared to BlindW variants, where read-dependencies are highly ambiguous. This benchmark represents the most challenging task that no prior black-box checker supports. Each transaction has 8 operations.

**Results.** Table 5 summarizes performance across five databases. Polygraph successfully verified black-box traces by resolving the resulting superpositions. Notably, it verified SI on TiKV using RandomBench, breaking prior checkers' impractical assumptions. While verification time increases due to high ambiguity in range queries and iterators, it confirms Polygraph's capability to verify complex anomalies strictly from client-observed values. When additional metadata (e.g., real-time ordering) is available, Polygraph automatically promotes superpositions into known edges, significantly accelerating verification (see Appendix B for details).

# REFERENCES


[1] [n.d.]. MySQL. https://www.mysql.com/.
[2] [n.d.]. Weird SELECT view when a record is modified to the same value by two transactions. https://jira.mariadb.org/browse/MDEV-26642.

[3] Atul Adya. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology.
[4] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering*. IEEE Computer Society, San Diego, CA, USA, 67–78. https://doi.org/10.1109/ICDE.2000.839388
[5] American National Standards Institute. 1992. *American national standard for information systems - ANSI X3.135-1992 ; Database Language - SQL.* American National Standards Institute, New York, NY, USA.
[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, San Jose, CA, USA, 1–10. https://doi.org/10.1145/223784.223785
[7] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 165 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360591
[8] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 42. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58
[9] Andrea Cerone and Alexey Gotsman. 2016. Analysing Snapshot Isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. ACM, Chicago, IL, USA, 55–64. https://doi.org/10.1145/2933057.2933096
[10] cobrabench 2020. Cobra Bench. https://github.com/DBCobra/CobraBench.
[11] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is believing: a client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, Washington, DC, USA, 73–82. https://doi.org/10.1145/3087801.3087802
[12] DB-Cobra. 2020. Cobra Verifier. https://github.com/DBCobra/CobraVerifier.
[13] Flexcoin. 2014. Flexcoin. https://web.archive.org/web/20160408190656/http://www.flexcoin.com/.
[14] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601
[15] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[16] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-Box Checking of Snapshot Isolation in Databases. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1264–1276.
[17] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229
[18] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-optimal read-only transactions. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 19, 17 pages.
[19] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, Philadelphia, PA, USA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
[20] Nathaniel Popper. 2016. A hacking of more than $50 million dashes hopes in the world of virtual currency. New York Times DealBook. http://nyti.ms/1UdyDfx
[21] PostgreSQL Global Development Group. 2026. PostgreSQL Official Website. https://www.postgresql.org/. Accessed July 16, 2026.
[22] E. Sirer. 2014. Nosql meets bitcoin and brings down two exchanges: The story of flexcoin and poloniex. http://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/. Hacking Distributed.
[23] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, USA, 63–80.
[24] TiKV Authors. 2025. TiKV. https://tikv.org/. Accessed July 16, 2026.
[25] YugabyteDB. 2026. YugabyteDB Github Repo. https://github.com/yugabyte/yugabyte-db. Accessed July 16, 2026.
[26] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, Monterey, CA, USA, 263–278.
[27] Jian Zhang. 2026. *Validation of Transactional Isolation in Key-Value Stores.* Ph.D. Dissertation. Northeastern University.
[28] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the European Conference on Computer Systems*. ACM, Rome, Italy, 654–671. https://doi.org/10.1145/3552326.3567492
[29] Jian Zhang and Cheng Tan. 2024. Simplifying Snapshot Isolation: A New Definition, Equivalence, and Efficient Checking. In *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Athens, Greece, 23–29. https://doi.org/10.1145/3642976.3653032

# A DETAILED RELATED WORK

As discussed in Section 1, existing frameworks fall short of concurrently satisfying the abstraction, interpretability, adaptability, and unification goals for practical isolation checking.

ANSI SQL-92 [5] is fundamentally incomplete and lacks a unified abstraction. Adya's graph-based framework [3] is highly interpretable but relies on internal metadata (e.g., version orders), severely limiting its adaptability in black-box cloud settings [12, 28]. Cerone's axiomatic framework [8] (and its extensions [7]) offers adaptability but defines correctness existentially by searching for abstract visibility and arbitration relations. This makes it unintuitive to map forbidden patterns to concrete anomalies. Moreover, its reliance on disjoint algorithms for weak versus strong isolation models fails the unification goal. Finally, Crooks's state-based framework [11] abstracts away internal artifacts but incurs a computationally prohibitive search space involving both state transitions and transaction orderings, and lacks adaptability to leverage known internal metadata.

# B DETAILED EVALUATION: ADAPTABILITY TO ADDITIONAL INFORMATION

When richer information is available, Polygraph verifies more efficiently. For instance, Tapir's SSER implies a real-time ordering, acting as a proxy for a grey-box setup. Polygraph automatically promoted ambiguous superpositions into known edges, shrinking the search space and verifying 13.2K transactions in 1.12 seconds—over 4× faster than PostgreSQL's black-box serializability check (4.5s). This confirms the framework adaptively digests available metadata. It is also compatible with external techniques (e.g., CDC) that recover version orders. (Note: Port traces failing PL-2+ checking does not necessarily indicate bugs, as Port's exact guarantees relative to PL-2+ remain unverified).

# C SERIALIZABILITY

Proof of Theorem 3.1. ($\Rightarrow$) We prove if Polygraph $\mathcal{G}(h)$ is acyclic, then $h$ is serializable. By Definition 2.9, Polygraph is acyclic implies there exists an acyclic canonical augmented dependency graph $G^{l}_{aug}(h)$ that is compatible with $\mathcal{G}(h)$. By removing the indirect write-dependency and indirect anti-dependency edges, we can get a direct dependency graph. Removing edges will not introduce cycles and hence the induced direct dependency graph is still acyclic. By Adya's serializability definition [4], then $h$ is serializable.

($\Leftarrow$) We prove if $h$ is serializable, then the Polygraph $\mathcal{G}(h)$ is acyclic. If $h$ is serializable, then there exists an execution $s$ of transactions such that $s$'s direct dependency graph $DSG(h)$ is acyclic. The execution $s$ defines all the direct dependencies $WW/WR/RW/PWR/PRW$. Please note that if any dependency is known in $h$, then it must be included in $WW/WR/RW/PWR/PRW$. We can build its canonical augmented dependency graph $G^{l}_{aug}(h)$ by adding all the indirect write-dependencies, anti-dependencies

and predicate-anti-dependencies edges to $DSG(h)$ as defined in Definition 2.4. Please note the added edges do not affect the reachability of $DSG(h)$ because it just further connects indirectly-reachable nodes, which means $G^l_{aug}(h)$ is still acyclic. Below, we prove $G^l_{aug}(h)$ is compatible with $h$'s Polygraph $\mathcal{G}(h)$. Therefore, $\mathcal{G}(h)$ is acyclic.

The proof that $G^l_{aug}$ is compatible with $h$'s Polygraph $\mathcal{G}$:

(1) $G^l_{aug}, DSG, \mathcal{G}$ share the same vertex set;
(2) $\forall$ known dependency edge $edge \in \mathcal{G}$, $edge$ is included in the execution $e$ and $DSG$ by Definition 2.2. The process of contructing $G^l_{aug}$from $DSG$ does not remove any edge in $DSG$ and hence $edge$ still exists in $G^l_{aug}$;
(3) $\forall$ superposition of type 1 (read dependencies) $\langle T_{w1} \xrightarrow{\text{wr(x)}} T_r, T_{w2} \xrightarrow{\text{wr(x)}} T_r, \cdots, T_{wn} \xrightarrow{\text{wr(x)}} T_r\rangle \in \mathcal{S}$, $\exists 1 \le i \le n$ such that $T_{wi} \xrightarrow{\text{wr(x)}} T_r \in DSG$ and $\forall 1 \le j \le n, j \ne i, T_{wj} \xrightarrow{\text{wr(x)}} T_r \notin DSG$ because (a) By Definition 2.3, $T_r$ is known to read from its unique write transaction, say $T_w$; (b) $T_w$ must be out of $T_{w1}, T_{w2}, \cdots, T_{wn}$ because $T_{w1}, T_{w2}, \cdots, T_{wn}$ contains all the possible candidate write transactions of the read $T_r$. That's to say, this superposition holds in $DSG$. Recall that we do not add any read-dependency edge to $G^l_{aug}$ when we construct it from $DSG$, so the superposition still holds in $G^l_{aug}$;
(4) $\forall$ superposition of type 2 (write dependencies) $\langle T_i \xrightarrow{\text{ww(x)}} T_j,\ T_j \xrightarrow{\text{ww(x)}} T_i\rangle \in \mathcal{S}$,
   (a) case (a) $T_i$ and $T_j$ are consecutive in the write order of $x$. Then either $T_i \xrightarrow{\text{ww(x)}} T_j \in DSG$ or $T_j \xrightarrow{\text{ww(x)}} T_i \in DSG$, say $T_i \xrightarrow{\text{ww(x)}} T_j \in DSG$ without loss of generality. The construction process of $G^l_{aug}$ will keep $T_i \xrightarrow{\text{ww(x)}} T_j$ in $G^l_{aug}$. Recall that $G^l_{aug}$ is acyclic, then $T_j \xrightarrow{\text{ww(x)}} T_i \notin G^l_{aug}$. Therefore, the superposition $\langle T_i \xrightarrow{\text{ww(x)}} T_j,\ T_j \xrightarrow{\text{ww(x)}} T_i\rangle$ holds in $G^l_{aug}$;
   (b) case (b) $T_i$ and $T_j$ are not adjacent in the write order of $x$. But either $T_i$ is before $T_j$ or $T_j$ is before $T_i$ in the write order of $x$. Assume $T_i$ is before $T_j$ in the write order of $x$. Even though both $T_i \xrightarrow{\text{ww(x)}} T_j$ and $T_j \xrightarrow{\text{ww(x)}} T_i$ are not in $DSG$, the indirect write-dependency $T_i \xrightarrow{\text{ww(x)}} T_j$ will be added to $G^l_{aug}$ when we construct $G^l_{aug}$ from the $DSG$. Recall that $G^l_{aug}$ is acyclic, then $G^l_{aug}$ does not have the edge $T_j \xrightarrow{\text{ww(x)}} T_i$. Therefore, the superposition $\langle T_i \xrightarrow{\text{ww(x)}} T_j,\ T_j \xrightarrow{\text{ww(x)}} T_i\rangle$ holds in $G^l_{aug}$;
(5) $\forall$ implication $T_{wj} \xrightarrow{\text{ww(x)}} T_{wi} \wedge T_{wj} \xrightarrow{\text{wr(x)}} T_r \Rightarrow T_r \xrightarrow{\text{rw(x)}} T_{wi} \in \mathcal{I}$,
   (a) if $T_{wi}$ is the immediate next version of $x$ after $T_{wj}$ and $T_{wj} \xrightarrow{\text{wr(x)}} T_r$ in $DSG$, then the edge $T_r \xrightarrow{\text{rw(x)}} T_{wi}$ exists in $DSG$ by Definition 2.3. $T_r \xrightarrow{\text{rw(x)}} T_{wi}$ also exists in $G^l_{aug}$ by construction, then the implication holds;
   (b) if $T_{wj}$ is the immediate next version of $x$ after $T_{wi}$ and $T_{wj} \xrightarrow{\text{wr(x)}} T_r$ in $DSG$, then the condition $T_{wj} \xrightarrow{\text{ww(x)}} T_{wi} \wedge T_{wj} \xrightarrow{\text{wr(x)}} T_r$ is not met in both $DSG$ and $G^l_{aug}$, and hence the implication always holds in $G^l_{aug}$;
   (c) $T_{wj}$ is before $T_{wi}$ in the write order of $x$ but they are not consecutive, and $T_{wj} \xrightarrow{\text{wr(x)}} T_r$ in $DSG$. Since we added indirect anti-dependencies to $G^l_{aug}$ as defined by the rule 3 in Definition 2.4, then $T_r \xrightarrow{\text{rw(x)}} T_{wi} \in G^l_{aug}$, which means the implication holds in $G^l_{aug}$.
   (d) $T_{wi}$ is before $T_{wj}$ in the write order of $x$ but $T_{wj}$ is not the immediate next write of $x$ after $T_{wi}$, and $T_{wj} \xrightarrow{\text{wr(x)}} T_r$ in $DSG$. Then the condition $T_{wj} \xrightarrow{\text{ww(x)}} T_{wi} \wedge T_{wj} \xrightarrow{\text{wr(x)}} T_r$ is not met in both $DSG$ and $G^l_{aug}$, and the implication always holds ;

The proof for predicate-anti-dependencies implications is similar and hence is omitted.
Therefore, $\mathcal{G}$ is acyclic.

□

# D SNAPSHOT ISOLATION

In this section, we show that the Polygraph Framework characterizes Adya's snapshot isolation definition [3, 4].

Zhang et al. [29, Definition 6] introduced BC-graph. We therefore define SI canonical augmented dependency graph as its augmentation and establish the following lemma before proving Theorem 3.4.

*Definition D.1 (SI canonical augmented dependency graph).* Given a history $h$ together with its dependencies and a time-precedes order $\prec_t$ over begin/commit events, an SI canonical augmented dependency graph $G$ is constructed as follows.

$T_i, T_j$ are any two transactions and $x$ is any data object or key in the database.

(1) Construct the BC-graph as specified in [29, Definition 6] first;
(2) Augment the BC-graph with additional edges, provided they do not already exist, as follows:
   (a) $T_i, T_j$ write $x$, and $T_j$'s version is after $T_i$'s version in the version order, add $C_i \xrightarrow{\text{ww(x)}} B_j$.
   (b) $T_i$ reads a version of $x$ and $T_j$ writes a later version of $x$ add $B_i \xrightarrow{\text{rw(x)}} C_j$.
   (c) If $T_r$ has a predicate read of $x$ and $T_w$ installs a later version of $x$ that changes the matches of that predicate read, add $B_r \xrightarrow{\text{prw(x)}} C_w$.

Lemma D.2. *Given a history $h$, its SI canonical augmented dependency graph shares the same reachability as its BC-graph, i.e., for any nodes $v_i, v_j$, if there is a path from $v_i$ to $v_j$ in the SI canonical augmented dependency graph, there is also a path from $v_i$ to $v_j$ in BC-graph, and vice versa.*

Proof. Let's use $BCG^{SI}_{direct}(h)$ and $BCG^{SI}_{aug}(h)$ to denote the BC-graph and SI canonical augmented dependency graph respectively. Recall that $BCG^{SI}_{aug}(h)$ is constructed based on $BCG^{SI}_{direct}(h)$ by adding extra edges, so it is trivial that for any nodes $v_i, v_j$, if there is a path from $v_i$ to $v_j$ in $BCG^{SI}_{direct}(h)$, there is also a path from $v_i$ to $v_j$ in $BCG^{SI}_{aug}(h)$ .

Next, we establish the direction from SI canonical augmented dependency graph to BC-graph by demonstrating that the newly added edges of $BCG^{SI}_{aug}(h)$ strictly connect nodes that are already reachable in the BC-graph $BCG^{SI}_{direct}(h)$.

(1) Consider an added edge $C_i \xrightarrow{\text{ww(x)}} B_j$. It indicates that both $T_i$ and $T_j$ write $x$, and that $T_j$ installs a newer version by construction. Let $T_1 \xrightarrow{\text{ww(x)}} T_2 \xrightarrow{\text{ww(x)}} \cdots \xrightarrow{\text{ww(x)}} T_n$ denote the write total order of $x$ in the direct dependency graph $G^{SER}_{direct}(h)$. Then, by construction, $BCG^{SI}_{direct}(h)$ contains edges $C_k \xrightarrow{\text{ww(x)}} B_{k+1}$ for every $1 \le k \le n-1$. Combined with the intra-transaction edges, these edges form a path $C_i \xrightarrow{\text{ww(x)}} B_{i+1} \to C_{i+1} \xrightarrow{\text{ww(x)}} \cdots \xrightarrow{\text{ww(x)}} B_j$ in $BCG^{SI}_{direct}(h)$.

(2) Consider an added edge $B_i \xrightarrow{\text{rw(x)}} C_j$. It indicates that $T_i$ reads a version of $x$ and $T_j$ writes a later version by construction. Let $T_k$ denote the transaction that installs the immediate next version of $x$ after the version read by $T_i$. Then, by construction, $BCG^{SI}_{direct}(h)$ contains an anti-dependency edge $B_i \xrightarrow{\text{rw(x)}} C_k$. Similar to reasoning in Case 1, there is also a path with only write-dependency and intra-txn edges from $C_k$ to $B_j$ in the BC-graph. Combine the edge $B_i \xrightarrow{\text{rw(x)}} C_k$, the path from $C_k$ to $B_j$ and the intra-txn edge $B_j \to C_j$, then we have found a path $B_i \rightsquigarrow C_j$ in $BCG^{SI}_{direct}(h)$.

(3) Consider an added edge $B_i \xrightarrow{\text{prw(x)}} C_j$. It indicates that $T_i$ has a predicate read of $x$ and that $T_j$ installs a later version that changes the matches of that predicate read. Let $T_k$ denote the first transaction after the version observed by $T_i$ that installs a version changing the matches of that predicate read. Such a transaction exists because $T_j$ changes the matches, and $T_k$ precedes or equals $T_j$ in the version order. By construction of the direct graph, $BCG^{SI}_{direct}(h)$ contains the predicate-anti-dependency edge $B_i \xrightarrow{\text{prw(x)}} C_k$. Similar to Case 1, there is a path with only write-dependency and intra-transaction edges from $C_k$ to $B_j$. Combining these edges with the intra-transaction edge $B_j \to C_j$ yields a path from $B_i$ to $C_j$ in $BCG^{SI}_{direct}(h)$.

□

Finally, we prove Theorem 3.4.

Proof. "⇒" The SI Polygraph exists, so the required validity conditions are enforced by construction and the history $h$ is valid for constructing SI canonical augmented dependency graph and BC-graph. On the other hand, The SI Polygraph $\mathcal{G}^{SI}(h)$ is acyclic ⇒ there exists an acyclic SI canonical augmented dependency graph $BCG^{SI}_{aug}(h)$ that is compatible with SI Polygraph $\mathcal{G}^{SI}(h)$. By removing the indirect write-dependency, anti-dependency, and predicate-anti-dependency edges, we can get a partial BC-graph $BCG^{SI}_{direct}(h)$. By Lemma D.2, the induced BC-graph is still acyclic. Please note that we call it a partial BC-graph because, even though it contains all direct write-dependencies, read-dependencies, predicate-read-dependencies, anti-dependencies, and predicate-anti-dependencies, it does not have start-dependency edges. Then we topologically sort $BCG^{SI}_{direct}(h)$ and get a total order of the begin/commit events, say $\hat{s}$. Let $\hat{s}$ be the time-precedes order, then we can add start-dependency edges to $BCG^{SI}_{direct}(h)$ according to $\hat{s}$ and obtain a complete BC-graph $BCG^{SI}_{direct}(h)$. $BCG^{SI}_{direct}(h)$ already defines an execution, say $e$, of $h$ because it contains all the dependencies now. All the added edges respect the order $\hat{s}$ and hence will not introduce cycles to $BCG^{SI}_{direct}(h)$. Hence, the BC-graph $BCG^{SI}_{direct}(h)$ is still acyclic. By [29, Theorem 8], the history $h$ is Adya SI.

"⇐" If $h$ is SI, then G1a/G1b is proscribed naturally by Adya's definition and hence the SI Polygraph $\mathcal{G}^{SI}(h)$ exists. On the other hand, $h$ is SI ⇒ there exists a time-precedes order $\prec_t$, and all the dependencies such that the BC-graph $BCG^{SI}_{direct}(h)$ exists and is acyclic by [29, Theorem 8]. Then we can build its SI canonical augmented dependency graph $BCG^{SI}_{aug}(h)$ as defined in Definition D.1. By Lemma D.2, $BCG^{SI}_{aug}(h)$ is still acyclic. The $BCG^{SI}_{aug}(h)$ is compatible with $h$'s SI Polygraph $\mathcal{G}^{SI}(h)$ (The compatibility proof is similar to that of serializability and is omitted here). Therefore, the SI Polygraph $\mathcal{G}^{SI}(h)$ exists and is acyclic. □

# E REPEATABLE READ

*Definition E.1 (direct RR graph).* A direct RR graph is the same as the direct dependency graph (Definition 2.3) except that it excludes predicate-anti-dependency edges.

*Definition E.2 (RR canonical augmented dependency graph).* A RR canonical augmented dependency graph $G^{RR}_{aug}$ is the same as the canonical augmented dependency graph $G^{SER}_{aug}$ (Definition 2.4) except that it excludes predicate-anti-dependency edges.

Proof of Theorem 3.2:

Proof. (⇒) We show that if RR Polygraph $\mathcal{G}(h)$ is acyclic, then $h$ satisfies repeatable read. By Definition 2.9, RR Polygraph is acyclic ⇒ there exists an acyclic RR canonical augmented dependency graph $G$ that is compatible with RR Polygraph $\mathcal{G}(h)$. By removing the indirect write-dependency and anti-dependency edges, we can get a direct dependency graph. Removing edges will not introduce cycles and hence the induced direct dependency graph is still acyclic. By Adya's definition of repeatable read, $h$ satisfies repeatable read.

(⇐) We show that if $h$ satisfies repeatable read, then the RR Polygraph $\mathcal{G}(h)$ is acyclic. If $h$ satisfies repeatable read, then there exists an execution whose $WW$, $WR$, $RW$, and $PWR$ dependencies form an acyclic direct RR graph $DRRG(h)$. Then we can build its RR canonical augmented dependency graph $BRRG(h)$ by adding all the possible indirect write-dependencies and anti-dependencies edges to $DRRG(h)$ as defined in Definition E.2. Please note the added edges do not affect the reachability of $DRRG(h)$ because they only further connect indirectly reachable nodes, which means $BRRG(h)$ is still acyclic. The $BRRG(h)$ is compatible with $h$'s RR Polygraph

$\mathcal{G}(h)$. Its proof is similar to the compatibility proof in Appendix C and is omitted. Therefore, $\mathcal{G}(h)$ is acyclic. □

# F READ COMMITTED

Read committed (RC) is a weaker model that allows fuzzy reads and allows phantoms. We model RC by further relaxing the constraints by excluding both anti-dependencies and predicate-anti-dependencies. Consequently, the set of Implications is empty ($I = \emptyset$). We prove the theorem in Appendix F.

Theorem F.1 (Read committed). *$\mathcal{G}^{RC}(h)$ is acyclic $\iff$ $h$ is read committed.*

*Definition F.2 (direct RC graph).* A direct RC graph is the same as the complete direct dependency graph (Definition 2.3) except direct RC graph excludes all anti-dependency and predicate-anti-dependency edges.

*Definition F.3 (RC canonical augmented dependency graph).* A RC canonical augmented dependency graph $G^{RC}_{aug}$ is the same as the canonical augmented dependency graph $G^{SER}_{aug}$ (Definition 2.4) except RC canonical augmented dependency graph excludes all anti-dependencies and predicate-anti-dependencies.

Proof of Theorem F.1:

Proof. ($\Rightarrow$) We will show if RC Polygraph $\mathcal{G}(h)$ is acyclic, then $h$ is read committed. By Definition 2.9, RC Polygraph is acyclic $\Rightarrow$ there exists an acyclic RC canonical augmented dependency graph $G$ that is compatible with RC Polygraph $\mathcal{G}(h)$. By removing the indirect write-dependency edges, we can get a direct dependency graph. Removing edges will not introduce cycles and hence the induced direct dependency graph is still acyclic. By Adya's read committed definition, then $h$ is read committed.

($\Leftarrow$) We will show if $h$ is read committed, then the RC Polygraph $\mathcal{G}(h)$ is acyclic. $h$ is read committed $\Rightarrow$ there exists $WW$, $WR$ and $PWR$ such that the direct dependency graph $DRCG(h)$ is acyclic. Then we can build its RC canonical augmented dependency graph $BRCG(h)$ by adding all the possible indirect write-dependencies edges to $DRCG(h)$ as defined in Definition F.3. Please note the added edges do not affect the reachability of $DRCG(h)$ because they only further connect indirectly reachable nodes, which means $BRCG(h)$ is still acyclic. The $BRCG(h)$ is compatible with $h$'s RC Polygraph $\mathcal{G}(h)$. Its proof is similar to the compatibility proof in Appendix C and is omitted. (Recall $\mathcal{I}$ is empty for RC Polygraph.) Therefore, $\mathcal{G}(h)$ is acyclic. □