On Mixing Database Isolation Levels

Qiuhuan Xiong
Nanjing University

Hengfeng Wei
Hunan University

Si Liu
Texas A&M University

Yuxing Chen
Renmin University of China

Jidong Ge
Nanjing University

ABSTRACT

Modern database systems widely support per-transaction isolation levels, yet the semantics of executing transactions under mixed isolation settings, as well as the guarantees such transactions provide, remain largely unspecified. This gap between industrial practice and theoretical foundations complicates correctness reasoning and can lead to severe data anomalies in practice.

To bridge this gap, we propose a formal semantic framework for mixing isolation levels. Our framework uniformly captures both classical isolation levels (e.g., serializability) and emerging ones (e.g., read atomicity), and generalizes existing formalizations that focus on single isolation settings. Beyond its formal foundations, the framework enables a range of practical applications. We demonstrate this through two applications: semantic conformance checking of concurrency control protocols and robustness checking for safely allocating isolation levels across transaction workloads. Together, this work represents an important step toward more reliable and performant databases supporting mixed isolation levels.

PVLDB Reference Format:
Qiuhuan Xiong, Hengfeng Wei, Si Liu, Yuxing Chen, and Jidong Ge. On Mixing Database Isolation Levels. 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/EagleBear2002/MixIso.

1 INTRODUCTION

Transactions and their isolation levels [49] are fundamental building blocks of modern database systems. Serializability [40], as the gold standard, ensures that a concurrent execution of transactions is equivalent to some serial execution of those transactions. While it effectively eliminates data anomalies such as lost updates and write skew, it incurs significant performance overheads, particularly under high contention and at scale. Consequently, many widely used databases, including traditional ones such as Microsoft SQL Server, Oracle, MySQL, and PostgreSQL, as well as emerging distributed databases like TiDB, YugabyteDB, and CockroachDB, all support assigning isolation levels on a per-transaction basis (e.g., via `SET TRANSACTION ISOLATION LEVEL`). In practice, database users can configure different isolation levels for their workloads, allowing some transactions to execute under relaxed isolation (e.g., *snapshot isolation* [10]) rather than enforcing serializability system-wide, thereby achieving performance gains.

However, a fundamental question remains largely unanswered: *what does it actually mean to run transactions under a heterogeneous or mixed isolation setting, and what guarantees do such transactions provide?* Despite widespread support for per-transaction isolation, the existing documentation, e.g., that of all the aforementioned databases, typically leaves the answer to this question unspecified or implementation-specific, let alone articulating a clear consensus (see [50, Appendix A] for a detailed discussion). As a result, users must rely on their own interpretations when using mixed isolation levels. Given the subtle semantics of different isolation levels and their complex interactions, such ambiguity can easily lead to unexpected data anomalies and even severe correctness violations.

Our Solution. We address this question by developing a formal semantic framework for mixing isolation levels, called MixIso, aimed at bridging the gap between long-standing industrial practice and the lack of a principled theoretical foundation. MixIso not only captures traditional isolation levels, such as serializability and snapshot isolation, but also accommodates a broad range of emerging isolation levels motivated by modern distributed applications, e.g., social media and microservices. These include *read atomicity* [6], *transactional causal consistency* [2, 36], *prefix consistency* [15], and *parallel snapshot isolation* [42]. In particular, some of them have already been integrated into real-world database systems and applications. For example, read atomicity has recently been layered on Facebook's TAO to provide atomically visible transactions [21]; ElectricSQL [23] offers causally consistent transactions; and Azure Cosmos DB [37] supports prefix consistency for transactional batches.

Compared to prior work [1, 12, 13, 19, 22, 51] formalizing isolation levels under the assumption that all transactions execute at the same isolation level, characterizing mixed isolation levels poses greater challenges. First, the problem *per se* is non-trivial to define: in a mixed setting, it is unclear what isolation guarantees should be expected both for individual transactions and for the overall execution. Second, different isolation levels exhibit subtle semantic corner cases, and their interactions in mixed executions further exacerbate this complexity.

To address these challenges, we propose a semantic characterization based on *per-transaction visibility*: the effects of other transactions that a given transaction is allowed to observe in a database execution. The overall execution is then considered consistent if, for each transaction, the effects it observes conform to its assigned isolation level. Our framework MixIso builds on the axioms of Cerone et al. [19], which provide a basis for specifying individual isolation levels declaratively, abstracting away database implementation details (e.g., timestamps as in Adya's formalism [1]). To validate the

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

correctness of our semantics, we establish an equivalence with the mixed isolation formalization of Bouajjani et al. [14] for the overlapping isolation levels, which is, however, specifically tailored for black-box isolation testing [13, 33]. Beyond covering additional isolation levels, including transactional causal consistency and parallel snapshot isolation, MixIso enables a broad range of applications, as we will see next (and in Section 9). Moreover, MixIso *generalizes* prior characterizations: when all transactions execute under the same isolation level, it coincides with existing semantics.

Application I: Semantic Conformance Checking. With a formal semantics in hand, our first application is to examine whether concurrency control mechanisms conform to their desired mixed isolation guarantees. Leveraging MixIso, we formally establish semantic conformance for two protocols. First, we show that the concurrency control mechanism combining snapshot isolation and strict two-phase locking (S2PL), as employed in Microsoft SQL Server and Oracle Berkeley DB [4, 24], correctly implements the intended mix of snapshot isolation and serializability. To our knowledge, this is the first formal conformance proof for practical database systems operating under mixed isolation levels. Second, we prove the correctness of our newly proposed concurrency control protocol that integrates three isolation levels, representing the first provably correct protocol that support more than two levels.

Beyond these case studies, we develop a general proof methodology for semantic conformance in mixed isolation settings, paving the way for automated verification [26, 35].

Application II: Robustness Checking. As a second application, we apply MixIso to address the *robustness* problem [20, 24]. Intuitively, robustness asks whether transactions executing under weaker isolation levels, such as snapshot isolation, can collectively guarantee serializability.[1] This problem is of practical importance, as weaker levels typically enable higher system performance [5, 34].

Based on MixIso, we propose a systematic approach for safely allocating, potentially many different, isolation levels across a given transaction workload (of program instances [11, 20]) so as to guarantee serializability.[2] At the core of this fine-grained allocation approach is a *static robustness criterion*, which determines whether *all* possible executions of the workload are serializable. This criterion is established by first introducing a *dynamic robustness criterion*, grounded in our axioms, which checks whether one given execution is serializable by analyzing its dependency graph [1].

We implement our approach as an isolation-level allocator, called MIA, and evaluate it against state-of-the-art approaches [11, 24] on three representative benchmarks: SmallBank, Courseware, and TPC-C. Experimental results show that MIA yields strictly weaker isolation allocations, while preserving serializability, for a large fraction of workloads generated from these benchmarks (up to 95%, 88%, and 100%, respectively). For example, on TPC-C [44], existing approaches can only lower the level to snapshot isolation, whereas MIA further assigns all program instances to even weaker isolation levels, including prefix consistency and read atomicity. In addition, our allocator is highly efficient and scales to large workloads, safely determining isolation level assignments for up to 10k program instances within one second. To our knowledge, this constitutes the first robustness results, spanning formal criteria, algorithmic techniques, and experimental validation, for a wide range of mixed isolation levels for distributed transactions.

[1]Equivalently, this problem can be viewed as safely allocating weaker isolation levels to transactions such that their executions remain equivalent to those under serializability.
[2]Intuitively, a transaction is a concrete execution of a program instance and consists of a sequence of read/write operations over specific keys and values. A workload is a set of program instances, whereas a database execution history consists of transactions.

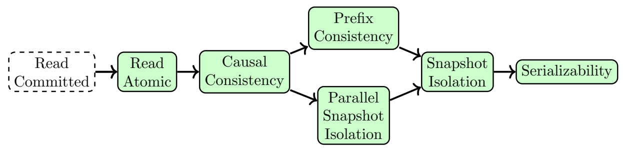


Figure 1: A hierarchy of prevalent isolation levels. $I \rightarrow I'$ means that I is strictly weaker than I'. Isolation levels supported by our semantic framework are shown in solid boxes.

These two applications are by no means exhaustive; instead, they are intended to showcase the potential of our formal semantic framework for advancing ongoing research directions in the database literature. Additional applications are sketched in Section 9.

To summarize, this work takes an important step toward bridging the gap between industrial practice and theory for mixing isolation levels. More broadly, it complements three lines of database research that have thus far largely focused on transactions executing under a single isolation level: (i) characterizing isolation levels; (ii) verifying database protocols against their claimed isolation guarantees; and (iii) robustness checking. See Section 8 for a discussion.

Due to space constraints, we defer the description of our protocol, along with all proofs and additional experimental results, to the appendix of the accompanying technical report [50].

2 PRELIMINARIES

Isolation Levels. To navigate the trade-offs between consistency, scalability, and availability, databases provide a spectrum of isolation levels. As shown in Figure 1, MixIso encompasses classical levels such as SER and SI, alongside more recent ones including RA, CC, PC, and PSI. We briefly explain some of these isolation levels.

Read Atomicity (RA) [6]. This isolation level guarantees that if a transaction observes any update from another transaction, it must observe all updates from that transaction. In particular, it rules out *fractured reads*, e.g., Joey sees that Ross has added Rachel as a friend, but fails to observe Rachel's corresponding addition of Ross, resulting in an inconsistent view of a bi-directional friendship.

Transactional Causal Consistency (CC) [2, 36]. This isolation level ensures that if a transaction T' causally depends on another transaction T (e.g., T' reads a value written by T), then any transaction that observes T' must also observe T. For example, under CC, Ross observing the comment written by Rachel on Joey's post without seeing the post itself is not allowed.

Prefix Consistency (PC) [15]. This level strengthens CC by ensuring that concurrent transactions are not observed in different orders, which is illustrated by the *long-fork* anomaly:

$$T_1 : \mathsf{W}(x, 1) \quad T_3 : \mathsf{R}(x, 1), \mathsf{R}(y, \bot)$$
$$T_2 : \mathsf{W}(y, 2) \quad T_4 : \mathsf{R}(x, \bot), \mathsf{R}(y, 2)$$

where T_3 and T_4 observe incompatible prefixes of the commit order. Equivalently, under PC, every transaction must observe a prefix of the global commit order.

Parallel Snapshot Isolation (PSI) [42]. This level also strengthens CC, but in a different way. In particular, it forbids *lost updates*, where concurrent transactions overwrite each other's updates, e.g., when two transactions deposit money into the same bank account at the same time, but one deposit is lost because it is overwritten by the other. However, unlike PC, PSI still allows long-fork anomalies.

Snapshot Isolation (SI) [10]. This level further enhances both PSI and PC by ruling out long forks and lost updates. In particular, in the earlier example, even if transactions T_3 and T_4 execute on different database replicas, they must observe a convergent snapshot of the database state. However, SI allows the *write skew* anomaly:

$$T_1 : \mathsf{R}(a1, 10), \mathsf{R}(a2, 10), \mathsf{W}(a1, -15)$$
$$T_2 : \mathsf{R}(a1, 10), \mathsf{R}(a2, 10), \mathsf{W}(a2, -15),$$

where both transactions check whether the total balance of the two bank accounts exceeds \$20 before withdrawing \$15 from one of them. When executed concurrently, both checks succeed, resulting in a negative total balance.

Serializability (SER) [40]. All of the aforementioned anomalies are forbidden by SER. It requires every concurrent transaction execution to be equivalent to some serial order of those transactions.

Relations. A binary relation R over two sets A and B is a subset of $A \times B$, i.e., $R \subseteq A \times B$. The domain of a relation $R \subseteq A \times B$ is defined as $\mathrm{dom}(R) \triangleq \{a \in A \mid \exists b \in B.\ (a, b) \in R\}$. In this paper, we mostly consider binary relations over a single set A, i.e., $R \subseteq A \times A$. We use R^+ and R^* to denote the transitive closure and the reflexive 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. For a total order $R \subseteq A \times A$, an element $e \in E \subseteq A$ is *maximal* (resp. *minimal*) in E if there is no $e' \in E$ such that $e \xrightarrow{R} e'$ (resp. $e' \xrightarrow{R} e$). In logical formulas, we write _ for irrelevant parts that are implicitly existentially quantified.

Transactions. We consider a transactional key-value store (KVS) managing a set of keys $\mathsf{K} = \{x, y, z, \dots\}$, which take on values from a set V. We denote by Op the set of possible read or write operations on keys: $\mathsf{Op} = \{\mathsf{R}_\iota(x, v), \mathsf{W}_\iota(x, v) \mid \iota \in \mathsf{OpId}, x \in \mathsf{K}, v \in \mathsf{V}\}$, where OpId is the set of operation identifiers. We omit operation identifiers when they are unimportant.

Definition 2.1. A ***transaction*** is a pair (O, po), where $O \subseteq \mathsf{Op}$ is a finite, non-empty set of operations and $\mathsf{po} \subseteq O \times O$ is a strict total order called the ***program order***.

Let $T = (O, \mathsf{po})$ be a transaction. For an operation $o \in O$ on key x, we denote by $\mathsf{po}_x^{-1}(o) \triangleq \{o' \in O \mid o' = \_(x, \_) \land o' \xrightarrow{\mathsf{po}} o\}$ the set of operations on x preceding o in po. We write $T \vdash \mathsf{W}(x, v)$ if T writes to x and the last value written is v, 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 use $\mathsf{WriteTx}_x \triangleq \{T \mid T \vdash \mathsf{W}(x, \_)\}$. Transactions T and T' *write-write conflict* with each other, denoted $T \bowtie T'$, if they write on the same key, i.e., $T, T' \in \mathsf{WriteTx}_x$ for some $x \in \mathsf{K}$. Transactions T *read-write conflict* with T', denoted $T \triangleleft T'$, if T reads from some key written by T'.

Table 1: Notation used in this paper.

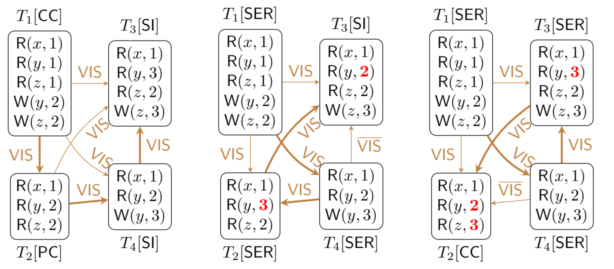
Notation	Meaning
$\mathsf{K}/\mathsf{V}/\mathsf{Op}$	set of keys/values/operations
$\mathsf{R}(x, v)/\mathsf{W}(x, v)$	read/write operation on key x with value v
$R \subseteq A \times B$	binary relation over A and B
$\mathrm{dom}(R)$	domain of relation R
R^+/R^*	transitive/reflexive transitive closure of R
$R\ ;\ S$	composition of R with S
$T = (O, \mathsf{po})$	transaction
po_x^{-1}	set of operations on x preceding o in po
$T \vdash \mathsf{R}(x, v)$	T reads from x before writing to it and v is the value returned by the first such read
$T \vdash \mathsf{W}(x, v)$	T writes to x and the last value written is v
$\mathsf{WriteTx}_x$	set of transactions writing to key x
$T \bowtie T'$	T write-write conflicts with T'
$T \triangleleft T'$	T read-write conflicts with T'
$T[\ell]$	transaction T with $\mathsf{level}(T) = \ell$
$\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$	history
$\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$	abstract execution
$\mathsf{VIS}^{-1}(T)$	set of transactions visible to T
$T_1 \xrightarrow{\overline{\mathsf{VIS}}} T_2$	$T_1 \xrightarrow{\mathsf{AR}} T_2 \land \neg(T_1 \xrightarrow{\mathsf{VIS}} T_2)$
$\mathsf{WR}, \mathsf{WW}, \mathsf{RW}$	dependency relations for $\mathcal{X}$
$T_1 \xrightarrow{\lambda} T_2$	any dependency relation between T_1 and T_2
$T_1 \xrightarrow{\lambda^*} T_2$	path of dependency relations from T_1 to T_2
$\mathsf{DDG}(\mathcal{X})$	dynamic dependency graph of $\mathcal{X}$
VIS-implicit/AR-implicit /Critical	edge types in $\mathsf{DDG}(\mathcal{X})$
$P = (O, \mathsf{po})$	program instance
$P[\ell]$	program instance P with $\mathsf{level}(P) = \ell$
$\mathsf{RSet}(P)/\mathsf{WSet}(P)$	read/write set of P
$P_1 \bowtie P_2$	P_1 write-write conflicts with P_2
$P_1 \triangleleft P_2$	P_1 read-write conflicts with P_2
$\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$	workload consisting of program instances
$\mathsf{WR}, \mathsf{WW}, \mathsf{RW}$	static dependency relations for $\mathcal{W}$
$\mathsf{SDG}(\mathcal{W})$	static dependency graph of $\mathcal{W}$

Histories. Clients interact with a transactional KVS by issuing transactions in *sessions*. A *history* records the client-visible outcomes of these interactions.

Definition 2.2. A ***history*** is a triple $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$, where $\mathcal{T}$ is a set of transactions with disjoint sets of operations, the *session order* $\mathsf{SO} \subseteq \mathcal{T} \times \mathcal{T}$ is the union of strict total orders on disjoint sets of $\mathcal{T}$, which correspond to transactions in different sessions, and **allocation** function $\mathsf{level} : \mathcal{T} \to \mathcal{L}$ that assigns an isolation level in $\mathcal{L}$ to each transaction in $\mathcal{T}$.

We consider $\mathcal{L} \triangleq \{\mathsf{RA}, \mathsf{CC}, \mathsf{PC}, \mathsf{PSI}, \mathsf{SI}, \mathsf{SER}\}$, ranged over by ℓ. We often write $T[\ell]$ to denote a transaction T such that $\mathsf{level}(T) = \ell$, and call it an ℓ-transaction. We write $T[\_]$ if the isolation level of T is irrelevant. The notation $T[\ell_1/\ell_2]$ means that T is an ℓ_1-transaction or an ℓ_2-transaction; $T[\ell_1/\cdots/\ell_i]$ is defined similarly.

We assume that every history contains a special transaction that writes the initial value $\mathsf{init} \in \mathsf{V}$ of all keys [13, 20]. This transaction precedes all the other transactions in SO.


T_1[CC]
R$(x,1)$ R$(y,1)$ R$(z,1)$ W$(y,2)$ W$(z,2)$
T_3[SI]
R$(x,1)$ R$(y,3)$ R$(z,2)$ W$(z,3)$
T_2[PC]
R$(x,1)$ R$(y,2)$ R$(z,2)$
T_4[SI]
R$(x,1)$ R$(y,2)$ W$(y,3)$
VIS
T_1[SER]
R$(x,1)$ R$(y,1)$ R$(z,1)$ W$(y,2)$ W$(z,2)$
T_3[SI]
R$(x,1)$ R$(y,$**2**$)$ R$(z,2)$ W$(z,3)$
T_2[SER]
R$(x,1)$ R$(y,$**3**$)$ R$(z,2)$
T_4[SER]
R$(x,1)$ R$(y,2)$ W$(y,3)$
VIS
$\overline{\text{VIS}}$
T_1[SER]
R$(x,1)$ R$(y,1)$ R$(z,1)$ W$(y,2)$ W$(z,2)$
T_3[SER]
R$(x,1)$ R$(y,$**3**$)$ R$(z,2)$ W$(z,3)$
T_2[CC]
R$(x,1)$ R$(y,$**2**$)$ R$(z,$**3**$)$
T_4[SER]
R$(x,1)$ R$(y,2)$ W$(y,3)$
VIS
$\overline{\text{VIS}}$

(a) $\mathcal{X}_1$ is consistent **(b) $\mathcal{X}_2$ is consistent** **(c) $\mathcal{X}_3$ is inconsistent**

Figure 2: Illustration of abstract executions. The AR relation can be inferred from the VIS relation in bold.

3 AN AXIOMATIC SEMANTIC FRAMEWORK FOR MIXING ISOLATION LEVELS

This section presents our axiomatic semantic framework, MixIso, for histories with mixed isolation levels. MixIso builds on the (VIS, AR) framework of Cerone et al. [19]. We first explain how to specify isolation levels for individual transactions using consistency axioms (Section 3.2), and then define what it means for an entire history to be consistent, i.e., to satisfy a mixed isolation level specification (Section 3.3).

3.1 Abstract Executions

The (VIS, AR) framework models concurrent transaction executions via two relations: visibility and arbitration [19]. Intuitively, the visibility relation defines the set of transactions whose effects are visible to each transaction, while the arbitration relation determines the execution order of transactions.

Definition 3.1. An ***abstract execution*** is a tuple $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR})$, where $(\mathcal{T}, \mathsf{SO}, \mathsf{level})$ is a history, *visibility* $\mathsf{VIS} \subseteq \mathcal{T} \times \mathcal{T}$ is an acyclic relation, and *arbitration* $\mathsf{AR} \subseteq \mathcal{T} \times \mathcal{T}$ is a strict total order such that $\mathsf{VIS} \subseteq \mathsf{AR}$.

For $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$, we often shorten $(\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR})$ to $(\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$. For an abstract execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$, we sometimes write $T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$ for $(T_1, T_2) \in \mathsf{VIS}$ to emphasize that the relation holds in $\mathcal{X}$. This notation also applies to AR. For a transaction T, the set $\mathsf{VIS}^{-1}(T) \triangleq \{S \mid S \xrightarrow{\mathsf{VIS}} T\}$ constitutes the transactions visible to T; we call it the *visible set* of T. We also write $T_1 \xrightarrow{\overline{\mathsf{VIS}}} T_2$ for $T_1 \xrightarrow{\mathsf{AR}} T_2 \land \neg(T_1 \xrightarrow{\mathsf{VIS}} T_2)$.

Example 3.2. Figure 2 presents three abstract executions over four transactions with identical access patterns (different values are highlighted in bold red), but different isolation level assignments and corresponding visibility and arbitration relations.

3.2 Consistency for Individual Transactions

An *isolation level* ℓ for an *individual* transaction T is a set of *consistency axioms* enforcing restrictions on the visibility relation involving T, i.e., on the set $\mathsf{VIS}^{-1}(T)$ of transactions visible to T.

$$\textsc{Int}(T) \equiv \forall r, x, v.\ (r = \mathsf{R}(x, v) \land \mathsf{po}_x^{-1}(r) \neq \emptyset) \implies \max_{\mathsf{po}}(\mathsf{po}_x^{-1}(r)) = \_(x, v).$$

$$\textsc{Ext}(T) \equiv \forall x, v.\ T \vdash \mathsf{R}(x, v) \implies \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x) \vdash \mathsf{W}(x, v).$$

$$\textsc{Session}(T) \equiv \forall T' \in \mathcal{T}.\ T' \xrightarrow{\mathsf{SO}} T \implies T' \xrightarrow{\mathsf{VIS}} T.$$

$$\textsc{TransVis}(T) \equiv \forall T', S \in \mathcal{T}.\ T' \xrightarrow{\mathsf{VIS}} S \xrightarrow{\mathsf{VIS}} T \implies T' \xrightarrow{\mathsf{VIS}} T.$$

$$\textsc{Prefix}(T) \equiv \forall T', S \in \mathcal{T}.\ T' \xrightarrow{\mathsf{AR}} S \xrightarrow{\mathsf{VIS}} T \implies T' \xrightarrow{\mathsf{VIS}} T$$

$$\textsc{NoConflict}(T) \equiv \forall T' \in \mathcal{T}.\ T' \bowtie T \land T' \xrightarrow{\mathsf{AR}} T \implies T' \xrightarrow{\mathsf{VIS}} T.$$

$$\textsc{TotalVis}(T) \equiv \forall T' \in \mathcal{T}.\ T' \xrightarrow{\mathsf{AR}} T \implies T' \xrightarrow{\mathsf{VIS}} T.$$

Figure 3: Consistency axioms for individual transactions.

Definition 3.3. Let $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR})$ be an abstract execution and $T = (O, \mathsf{po}) \in \mathcal{T}$ a transaction. The ***consistency axioms*** on T are defined as in Figure 3.

Let $r \triangleq \mathsf{R}(k, \_)$ be a read operation on k of transaction T. If r is the first operation on k in T, i.e., $\mathsf{po}_x^{-1}(r) = \emptyset$, then it is called an *external* read operation of T; otherwise, it is an *internal* read.

- $\textsc{Int}(T)$: The *internal consistency axiom* for T ensures that an internal read of T from a key returns the same value as the last write to or read from this key in T.
- $\textsc{Ext}(T)$: The *external consistency axiom* for T ensures that an external read of T from a key returns the value written by the last transaction in AR among all the transactions that precede T in VIS and write this key.
- $\textsc{Session}(T)$: The *session axiom* for T ensures that all transactions preceding T in the session order are visible to T.
- $\textsc{TransVis}(T)$: The *transitive visibility axiom* for T enforces that if T observes S, then it also observes all VIS-predecessors of S. $\textsc{TransVis}(T)$ implies that VIS is transitive when restricted to the transactions visible to T.
- $\textsc{Prefix}(T)$: The *prefix axiom* for T enforces that if T observes S, then it also observes all AR-predecessors of S.
- $\textsc{NoConflict}(T)$: The *no-conflict axiom* for T requires T observe all AR-predecessors that *write-write conflict* with it.
- $\textsc{TotalVis}(T)$: The *total visibility axiom* for T requires T observe all of its AR-predecessors.

Note that we have not adopted the *no-conflict axiom* as in prior work [19] and defined it as $\forall T' \in \mathcal{T}.\ T' \bowtie T \implies (T \xrightarrow{\mathsf{VIS}} T' \lor T' \xrightarrow{\mathsf{VIS}} T)$. It is stronger than our version of $\textsc{NoConflict}(T)$ and unnecessarily requires $T \xrightarrow{\mathsf{AR}} T' \implies T \xrightarrow{\mathsf{VIS}} T'$ even if $\textsc{NoConflict}(T')$ does not hold.

Theorem 3.4. *Let* $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR})$ *be an abstract execution. For each transaction* $T \in \mathcal{T}$*, we have* $\mathsf{SER}(T) \implies \mathsf{SI}(T) \implies \mathsf{PSI}(T) \implies \mathsf{CC}(T) \implies \mathsf{RA}(T)$ *and* $\mathsf{SI}(T) \implies \mathsf{PC}(T) \implies \mathsf{CC}(T)$.

3.3 Formalizing Mixed Isolation Levels

Definition 3.5. An *abstract execution* $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR})$ is called ***consistent*** if for each transaction $T \in \mathcal{T}$, the consistency axioms (defined in Table 2) corresponding to its isolation level $\mathsf{level}(T)$ hold on T. A *history* $\mathcal{H}$ is called ***consistent*** if there exists a consistent abstract execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$.

Table 2: Isolation levels for individual transactions.

$\mathrm{RA}(T) \equiv \textsc{Int}(T) \wedge \textsc{Ext}(T) \wedge \textsc{Session}(T)$	$\mathrm{CC}(T) \equiv \mathrm{RA}(T) \wedge \textsc{TransVis}(T)$
$\mathrm{PC}(T) \equiv \mathrm{RA}(T) \wedge \textsc{Prefix}(T)$	$\mathrm{PSI}(T) \equiv \mathrm{RA}(T) \wedge \textsc{TransVis}(T) \wedge \textsc{NoConflict}(T)$
$\mathrm{SI}(T) \equiv \mathrm{RA}(T) \wedge \textsc{Prefix}(T) \wedge \textsc{NoConflict}(T)$	$\mathrm{SER}(T) \equiv \mathrm{RA}(T) \wedge \textsc{TotalVis}(T)$

Isolation levels widely used in distributed database systems, such as those described in Section 2, are special cases of mixed isolation levels, in which all transactions execute under the same isolation level. We prove that MixIso generalizes a prior formalization [19]; the proof is given in [50, Appendix C]. We explicitly state the definition of serializability here, as it will be used frequently in the remainder of the paper.

Definition 3.6. Let $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR})$ be an *abstract execution*, where $\forall T \in \mathcal{T}.\ \mathsf{level}(T) = \mathrm{SER}$. Then, $\mathcal{X}$ is ***serializable***, denoted $\mathcal{X} \models \mathrm{SER}$, if it is consistent. A history $\mathcal{H}$ is ***serializable*** if there exists a serializable abstract execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$.

Let $\mathcal{H}$ be a history consisting of SI and SER transactions. We also write $\mathcal{H} \models$ SI-SER to denote that $\mathcal{H}$ is consistent. Similar notations apply to other mixed isolation levels and abstract executions.

Example 3.7. In Figure 2, $\mathcal{X}_1$ and $\mathcal{X}_2$ are consistent; consequently, their corresponding histories are also consistent. In contrast, $\mathcal{X}_3$ is inconsistent, as it violates $\textsc{TransVis}(T_2)$: T_2 reads z from T_3, which in turn reads y from T_4, yet T_2 does not observe $\mathsf{W}(y, 3)$ of T_4. For the same reason, the history corresponding to $\mathcal{X}_3$ is also inconsistent.

To validate the correctness of our semantics, we establish an equivalence with the mixed isolation formalization recently proposed by Bouajjani et al. [14] for the overlapping isolation levels (i.e., RA, PC, SI, and SER), which is specifically tailored for black-box isolation testing [13, 33]. Our proof is given in [50, Appendix D].

4 APPLICATION I: CHECKING SEMANTIC CONFORMANCE

In this section, we study the conformance of concurrency control protocols to our formal semantics for mixed isolation levels. We first present a general proof methodology, and then show that, following this methodology, the SI–S2PL protocol used in Microsoft SQL Server and Oracle Berkeley DB [4, 24] conforms to the intended mix of SI-SER. In [50, Appendix F], we further introduce our newly proposed concurrency control protocol integrating three isolation levels, PC-SI-SER, and prove its semantic conformance.

4.1 Proof Methodology

To establish that a concurrency control protocol conforms to our semantics for mixed isolation levels, we show that every history $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$ generated by the protocol is consistent. That is, by Definition 3.5, there exists an abstract execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ corresponding to the history, such that for each transaction $T \in \mathcal{T}$, the consistency axioms associated with $\mathsf{level}(T)$ hold.

A semantic conformance proof has two main steps: ❶ we *construct* appropriate relations VIS and AR from the protocol's operational semantics; and ❷ we *verify* that the constructed relations satisfy the required consistency axioms for each transaction.

❶ **Construction.** In general, AR and VIS are derived from the protocol's commit rules, locking or validation mechanisms, and visibility constraints. For lock-based protocols, AR is induced by the commit order of conflicting transactions, and VIS consists of transactions whose writes are released before a transaction's reads. For timestamp-based protocols, AR follows transaction timestamps or serialization points, and VIS includes transactions preceding the snapshot or read timestamp. For optimistic concurrency control protocols, AR is determined by validation order, and VIS contains the transactions whose effects are taken into account during validation.

Under mixed isolation levels, VIS is typically constructed by a case analysis on isolation levels: different levels impose different visibility constraints, while all transactions share a common global arbitration order AR.

❷ **Verification.** When verifying the axioms imposed by $\mathsf{level}(T)$, $\textsc{Int}(T)$ is typically immediate, and $\textsc{Session}(T)$, $\textsc{Prefix}(T)$, and $\textsc{TotalVis}(T)$ follow from routine algebraic relational reasoning. The main proof obligations are $\textsc{Ext}(T)$ and $\textsc{NoConflict}(T)$, which require intricate, protocol-specific arguments. In particular, proving the $\textsc{Ext}(T)$ axiom requires showing that, for each external read of T, the protocol guarantees that the read returns the value written by the latest visible transaction according to the constructed VIS and AR relations. This involves a careful analysis of the protocol's read rules, visibility constraints, and commit orderings to ensure that no write from a visible transaction T' is missed by T's read. Such arguments can be carried out by contradiction, showing that T' would have committed before the read (and hence been observed) or cannot be visible to T due to, e.g., locking rules, or T' would cause T to abort due to, e.g., conflict detection mechanisms.

Finally, proving the $\textsc{NoConflict}(T)$ axiom generally relies on reasoning about the protocol's locking, validation, or conflict detection mechanisms, showing that two write-write conflicting transactions cannot both reach the commit state.

4.2 Conformance of the SI-S2PL Protocol

Algorithm 1 presents the pseudocode of the concurrency control protocol with SI and S2PL employed in Microsoft SQL Server and Oracle Berkeley DB. It supports clients choosing either SI or SER isolation level for each transaction. We assume a service now() that returns the current wall time. For simplicity, we assume that each procedure is executed atomically, and we omit the details such as deadlock handling, lock upgrade, and recovery.[3] We reference pseudocode lines using the format "algorithm#:line#".

SI transactions execute following the standard SI protocol [10]. Specifically, when an SI transaction T starts, it is assigned a start timestamp $T.sts$ (line 1:3). The writes of T are buffered in $T.buffer$ (line 1:7). Under SI, T reads data from its own write buffer for internal reads (line 1:10), and from the snapshot of committed transactions as of its start time $T.sts$ for external reads (line 1:12).

SER transactions execute following the "deferred updates" variant [27, 28] of strict two-phase locking (S2PL) protocol [49]. Specifically, when an SER transaction T writes to a key, it first acquires an exclusive lock on the key (line 1:6) before buffering the write in

[3]We discuss the multi-threading concurrency issues in [50, Appendix E]. Specifically, we show that if the procedure Commit(T) was not executed atomically and line 1:19 was moved before line 1:17, then Non-RepeatableRead anomaly may arise.

Algorithm 1 The SI-S2PL protocol [4]

```
    store : K → (T ⇀ V): versioned KV store, mapping each key to
    timestamped values
    $T.buffer$ : K ⇀ V: write buffer of transaction $T$
 1: procedure START($T$)
 2:     if level($T$) = SI                      ▷ no $sts$ for SER transactions
 3:         $T.sts \leftarrow$ now()
 4: procedure WRITE($T, k, v$)
 5:     if level($T$) = SER
 6:         $T.xlock(k)$                        ▷ acquire $xlock$ on $k$
 7:     $T.buffer[k] \leftarrow v$    ▷ buffer $v$, overwriting previous writes to $k$
 8: procedure READ($T, k$)
 9:     if $k \in$ dom($T.buffer$)        ▷ internal read: read my own write
10:         return $T.buffer[k]$
11:     if level($T$) = SI                ▷ external read: read from snapshot
12:         return value of store[$k$] at latest timestamp $< T.sts$
13:     if level($T$) = SER    ▷ external read: read latest committed value
14:         $T.slock(k)$                        ▷ acquire $slock$ on $k$
15:         return value of store[$k$] at latest timestamp
16: procedure COMMIT($T$)
17:     if level($T$) = SI
18:         $T.xlock(k), \forall k \in$ dom($T.buffer$)        ▷ acquire $xlocks$
19:     $T.cts \leftarrow$ now()                        ▷ level($T$) $\in$ {SI, SER}
20:     if level($T$) = SI                      ▷ level($T'$) $\in$ {SI, SER}
21:         if $\exists T'.\ T'.cts \in (T.sts, T.cts) \wedge T' \bowtie T$
22:             $T.unlocks()$
23:             return aborted
24:     store[$k$] $\leftarrow [T.cts \mapsto v], \forall [k \mapsto v] \in T.buffer$    ▷ install writes
25:     $T.unlocks()$                           ▷ level($T$) $\in$ {SI, SER}
26:     return committed
```

$T.buffer$ (line 1:7). Similar to SI transactions, an SER transaction T reads data from its own write buffer for internal reads (line 1:10) and from the latest committed transactions on requested keys for external reads (line 1:15). Note that T should first acquire shared locks on the keys for external reads (line 1:14).

When T is ready to commit, it is assigned a commit timestamp $T.cts \leftarrow$ now() (line 1:19), installs its buffered writes into store with timestamp $T.cts$ (line 1:24), and releases all its locks (line 1:25). An SI transaction T additionally (1) acquires exclusive locks on all keys it intends to write (line 1:18) *before* assigning $T.cts$, and (2) performs write-write conflict checking: T aborts if any other already committed *concurrent* transaction T' (i.e., $T'.cts \in (T.sts, T.cts)$) has written on keys that it intends to write (i.e., $T' \bowtie T$); see line 1:21.

THEOREM 4.1. *Algorithm 1 satisfies* SI-SER.

PROOF SKETCH. This protocol is timestamp-based: transactions are assigned start and commit timestamps, and reads are performed based on these timestamps. Hence, for any history $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$ produced by Algorithm 1, we define AR as $\forall T', T \in \mathcal{T}.\ T' \xrightarrow{\mathsf{AR}} T \iff T'.cts < T.cts$. The visibility relation VIS is defined via a case analysis following the read rules of SI and SER transactions (line 1:12 and line 1:15): $\forall T', T \in \mathcal{T}.\ T' \xrightarrow{\mathsf{VIS}} T \iff (\mathsf{level}(T) = \mathsf{SI} \wedge T'.cts < T.sts) \vee (\mathsf{level}(T) = \mathsf{SER} \wedge T'.cts < T.cts)$. The remaining axioms induced by level(T) are verified in [50, Appendix E]. □

5 APPLICATION II: ROBUSTNESS CHECKING

5.1 Transactional Programs and Workloads

Figure 4 shows the table schema and SQL code for the well-known SmallBank benchmark [3]. The schema consists of three *tables*: `Account(Name,CustomerID)` associates customer names with IDs; `Savings(CustomerID,Balance)` stores savings account balances of customers; and `Checking(CustomerID,Balance)` stores checking account balances. The application interacts with the database through five *transactional programs* (aka transaction templates [46]): `Balance(N)` checks the total balance of customer `N`'s savings and checking accounts; `WriteCheck(N,V)` conditionally debits amount `V` from customer `N`'s checking account based on the total balance across savings and checking, applying an overdraft penalty if insufficient, `DepositChecking(N,V)` deposits amount `V` into customer `N`'s checking account; `TransactSavings(N,V)` deposits amount `V` into customer `N`'s savings account; and `Amalgamate(N1,N2)` transfers the entire balance from customer `N1`'s savings account and checking account to customer `N2`'s checking account.

We assume that the read and write sets of transactional programs can be determined statically without executing them. For example, from the SQL code in Figure 4, we can infer that the `TransactSavings` program reads the `Account` table and writes the `Savings` table. This assumption is widely adopted and has been validated by the empirical study in [39].

When executing the SmallBank benchmark, a database system repeatedly invokes these transactional programs with different parameters, thereby generating a ***workload*** of *program instances*. Figure 5 shows an example workload comprising six program instances, including two instances of `Balance`, one instance of `DepositChecking`, and others. Each program instance is identified by its name and input arguments, which together determine its read and write sets.

We represent the read and write sets of each program instance at the granularity of individual *cells*. For example, the `TS(N1,V1)` instance in Figure 5 first reads the row of the `Account` table satisfying `Name:=N1` (corresponding to the `SELECT` statement of the `TS` program in Figure 4), which contains two cells: `N1` (the `Name` attribute) and `C1` (the `CustomerID` attribute), respectively. We denote these two cell read operations collectively as `R(Acc{N1,C1})`. Then, the instance performs `R(Sav{C1,B1})` when evaluating the `WHERE CustomerID:=C1` clause of the next `UPDATE` statement, and subsequently writes to the `B1` cell of the `Savings` table, i.e., the `Balance` attribute of the `C1` row, denoted `W(Sav{B1}, _)`. [4] Note that it does not write to the `C1` cell.

In this work, we assume that SQL statements contain only equality predicates on primary keys in their `WHERE` clauses, as in [3]. This allows the read and write sets of program instances to be represented as fixed sets of cells. This assumption also holds for many widely used benchmarks, including SmallBank [3], TPC-C [44], and Courseware [29] (see also Section 7).

5.2 The Static Robustness Problem

In this work, we study the static robustness problem at the *workload* level: Given a workload of program instances and an isolation level

[4]The underscore "_" indicates that the specific value is irrelevant for our analysis.

```
Schema (primary keys underlined)

Account(Name, CustomerID)                                short as Acc(N,C)
Savings(CustomerID, Balance)                             short as Sav(C,B)
Checking(CustomerID, Balance)                            short as Chk(C,B)
```

```
Balance(N)                                               short as Bal(N)

SELECT CustomerId INTO :X FROM Account WHERE Name=:N;
SELECT (S.Balance + C.Balance) AS TotalBalance
  FROM Savings S, Checking C WHERE S.CustomerId=:X AND C.CustomerId=:X;
```

```
WriteCheck(N,V)                                          short as WC(N,V)

SELECT CustomerId INTO :X FROM Account WHERE Name=:N;
SELECT (S.Balance + C.Balance) AS TotalBalance
  FROM Savings S, Checking C WHERE S.CustomerId=:X AND C.CustomerId=:X;
IF TotalBalance >= :V THEN
  UPDATE Checking SET Balance = Balance-:V WHERE CustomerId=:X;
ELSE
  UPDATE Checking SET Balance = Balance-(1+:V) WHERE CustomerId=:X;
```

```
DepositChecking(N,V)                                     short as DC(N,V)

SELECT CustomerId INTO :X FROM Account WHERE Name=:N;
UPDATE Checking SET Balance = Balance+:V WHERE CustomerId=:X;
```

```
TransactSavings(N,V)                                     short as TS(N,V)

SELECT CustomerId INTO :X FROM Account WHERE Name=:N;
UPDATE Savings SET Balance = Balance+:V WHERE CustomerId=:X;
```

```
Amalgamate(N1,N2)                                        short as Ama(N1,N2)

SELECT CustomerId INTO :X1 FROM Account WHERE Name=:N1;
SELECT CustomerId INTO :X2 FROM Account WHERE Name=:N2;
UPDATE Savings AS new SET Balance = 0 FROM Savings AS old
  WHERE new.CustomerId=:X1 AND old.CustomerId=new.CustomerId
  RETURNING old.Balance INTO :A;
UPDATE Checking AS new SET Balance = 0 FROM Checking AS old
  WHERE new.CustomerId=:X1 AND old.CustomerId=new.CustomerId
  RETURNING old.Balance INTO :B;
UPDATE Checking SET Balance = Balance + :A + :B WHERE CustomerId=:X2;
```

Figure 4: Table schema and SQL code for the SmallBank benchmark.

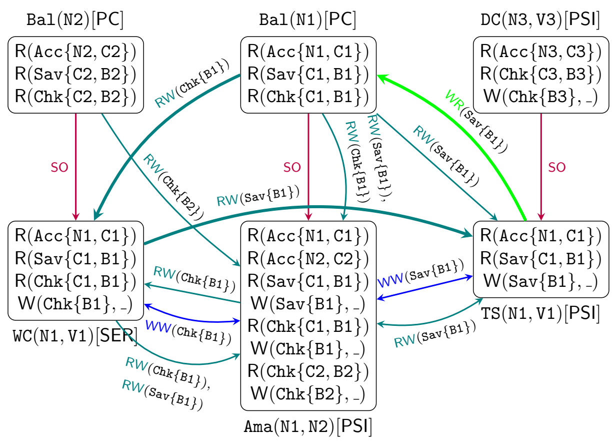


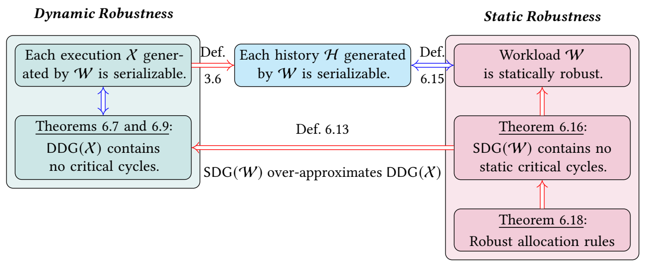
Figure 5: A workload $\mathcal{W}_{\text{rb}} = (\mathcal{P}_{\textbf{SmallBank}}, \textbf{SO}, \text{level}_{\text{rb}})$ consisting of six program instances of SmallBank programs and its static dependency graph SDG($\mathcal{W}_{\text{rb}}$). The notations for R and W operations are explained in Section 5.1. (To avoid clutter, not all WR edges are shown.)

allocation for them, the goal is to determine whether all histories generated by executing the workload are serializable when each program instance runs under its assigned isolation level.

Prior work has considered isolation level allocation for workloads based on the SmallBank benchmark. Beillahi et al. report that SmallBank is not robust when all transactions run under SI [7, Table 1], but do not characterize which mixed allocations would restore robustness. Cahill et al. [16] identify `WriteCheck` as a pivot, from which one may infer that it should run under SER while others may run under SI. Vandevoort et al. [45, Table 1] provide a robust allocation on SmallBank templates: `Bal`[SSI], `DC`[RC], `TS`[SSI], `Ama`[SSI], and `WC`[SSI], where SSI is an SER protocol [16, 17, 41].

5.3 Fine-grained Isolation Level Allocation

Our goal is to identify fine-grained robust (or safe) allocations that allow more program instances in a workload to run under a broader



Dynamic Robustness
Static Robustness
Each execution $\mathcal{X}$ generated by $\mathcal{W}$ is serializable.
Def. 3.6
Each history $\mathcal{H}$ generated by $\mathcal{W}$ is serializable.
Def. 6.15
Workload $\mathcal{W}$ is statically robust.
Theorems 6.7 and 6.9: DDG($\mathcal{X}$) contains no critical cycles.
Def. 6.13
SDG($\mathcal{W}$) over-approximates DDG($\mathcal{X}$)
Theorem 6.16: SDG($\mathcal{W}$) contains no static critical cycles.
Theorem 6.18: Robust allocation rules

Figure 6: An illustration of the dynamic-static interplay between robustness and dependency graphs.

range of weaker isolation levels. For example, Figure 5 illustrates a robust allocation for a SmallBank workload derived from the rules in Theorem 6.18: the two read-only `Bal` instances run under PC, the `TS`, `DC`, and `Ama` instances run under PSI, and the `WC` instance runs under SER.

5.4 Robustness and Dependency Graphs: A Dynamic-Static Interplay

The static robustness problem for workloads can be reduced to the dynamic robustness problem for abstract executions [11, 25, 46], which asks whether a given abstract execution is serializable; see Figure 6. Specifically, if all abstract executions generated by a workload under a given isolation level allocation are serializable, then by Definition 3.6 all histories generated by executing the workload are serializable; that is, the workload is statically robust under that allocation (Definition 6.15).

As we will show in Theorem 6.9, an abstract execution $\mathcal{X}$ generated under mixed isolation levels is serializable iff its *dynamic dependency graph* DDG($\mathcal{X}$) (Definition 6.1) contains no cycles of certain forms. It therefore suffices to ensure that all dynamic dependency graphs generated by the workload are acyclic in the specified manner. Since explicitly enumerating all such dynamic dependency graphs is generally infeasible, we instead construct an *over-approximation*, called the *static dependency graph* SDG($\mathcal{W}$) (Definition 6.13) of the workload $\mathcal{W}$.

By ensuring that the static dependency graph is acyclic in the same sense, we can guarantee that all dynamic dependency graphs are likewise acyclic, and hence that the workload is statically robust. The key challenges in this approach lie in (1) precisely characterizing serializable abstract executions under mixed isolation levels in terms of dynamic dependency graphs; and (2) defining a static dependency graph that soundly over-approximates all dynamic dependency graphs induced by the workload. We address these challenges in Sections 6.2 and 6.3, respectively.

The resulting static robustness criterion (Theorem 6.16) directly enables the isolation level allocation problem for workloads. In particular, Theorem 6.18 presents a set of simple yet effective rules for assigning isolation levels to program instances such that the resulting workload is statically robust.

6 ROBUSTNESS CRITERIA AND ALLOCATION

Following the roadmap outlined in Section 5.4, we first study the *dynamic robustness* problem for abstract executions, and then leverage the results to address the *static robustness* problem for workloads under mixed isolation levels.

6.1 Dynamic Dependency Graphs

The *dynamic dependency graph* [1, 11] of an abstract execution $\mathcal{X}$ captures three kinds of dependencies among transactions in $\mathcal{X}$. The WR dependency associates a transaction that reads some value with the one that writes this value. The WW dependency stipulates a strict total order among the transactions that write to the same key. The RW dependency relates a transaction that reads some value to the one that overwrites this value.

Definition 6.1 ([11]). Let $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR})$ be an abstract execution. For $x \in \mathsf{K}$ and $T_1, T_2 \in \mathcal{T}$, we define the following ***dependency relations*** on $\mathcal{T}$:

- WR dependency:

$$T_1 \xrightarrow{\mathsf{WR}(x)} T_2 \iff T_2 \vdash \mathsf{R}(x, \_) \land T_1 = \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T_2) \cap \mathsf{WriteTx}_x).$$

- WW dependency:

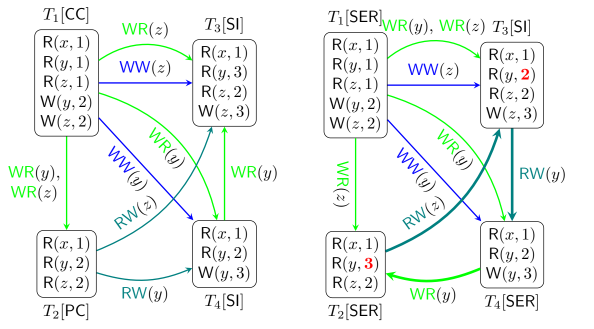
$$T_1 \xrightarrow{\mathsf{WW}(x)} T_2 \iff T_1 \xrightarrow{\mathsf{AR}} T_2 \land T_1, T_2 \in \mathsf{WriteTx}_x.$$

- RW dependency:

$$T_1 \xrightarrow{\mathsf{RW}(x)} T_2 \iff T_1 \neq T_2 \land \exists T_3.\ T_3 \xrightarrow{\mathsf{WR}(x)} T_1 \land T_3 \xrightarrow{\mathsf{WW}(x)} T_2.$$

The ***dynamic dependency graph*** $\mathcal{G}$ of $\mathcal{X}$ is a tuple $\mathrm{DDG}(\mathcal{X}) = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, where $\mathsf{WR} = \bigcup_{x \in \mathsf{K}} \mathsf{WR}(x)$, $\mathsf{WW} = \bigcup_{x \in \mathsf{K}} \mathsf{WW}(x)$, and $\mathsf{RW} = \bigcup_{x \in \mathsf{K}} \mathsf{RW}(x)$.

Example 6.2. Figure 7 illustrates the dynamic dependency graphs of $\mathcal{X}_1$ and $\mathcal{X}_2$ in Figure 2. For instance, in $\mathrm{DDG}(\mathcal{X}_2)$, there is an edge $T_4 \xrightarrow{\mathsf{WR}(y)} T_2$ because $T_4 = \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T_2) \cap \mathsf{WriteTx}_y)$; see Figure 2c for the VIS and AR relations of $\mathcal{X}_2$. Moreover, there is an edge $T_2 \xrightarrow{\mathsf{RW}(z)} T_3$, since $T_1 \xrightarrow{\mathsf{WR}(z)} T_2$ and $T_1 \xrightarrow{\mathsf{WW}(z)} T_3$. Similarly, we have $T_3 \xrightarrow{\mathsf{RW}(y)} T_4$. Hence, $\mathrm{DDG}(\mathcal{X}_2)$ contains a cycle $\pi = T_2 \xrightarrow{\mathsf{RW}(z)} T_3 \xrightarrow{\mathsf{RW}(y)} T_4 \xrightarrow{\mathsf{WR}(y)} T_2$.



T_1[CC]
R$(x,1)$ R$(y,1)$ R$(z,1)$ W$(y,2)$ W$(z,2)$
WR(z)
WW(z)
T_3[SI]
R$(x,1)$ R$(y,3)$ R$(z,2)$ W$(z,3)$
WR(y), WR(z)
WW(y)
WR(y)
RW(z)
WR(y)
R$(x,1)$ R$(y,2)$ R$(z,2)$
T_2[PC]
RW(y)
R$(x,1)$ R$(y,2)$ W$(y,3)$
T_4[SI]
T_1[SER]
WR(y), WR(z)
T_3[SI]
R$(x,1)$ R$(y,1)$ R$(z,1)$ W$(y,2)$ W$(z,2)$
WW(z)
R$(x,1)$ R$(y,2)$ R$(z,2)$ W$(z,3)$
WR(z)
WW(y)
WR(y)
RW(z)
RW(y)
R$(x,1)$ R$(y,3)$ R$(z,2)$
T_2[SER]
WR(y)
R$(x,1)$ R$(y,2)$ W$(y,3)$
T_4[SER]

(a) DDG($\mathcal{X}_1$) which is acyclic **(b) DDG($\mathcal{X}_2$) which is cyclic.**

Figure 7: Dynamic dependency graphs of abstract executions $\mathcal{X}_1$ and $\mathcal{X}_2$ in Figure 2. DDG($\mathcal{X}_2$) contains a cycle $\pi = T_2 \xrightarrow{\mathsf{RW}(z)} T_3 \xrightarrow{\mathsf{RW}(y)} T_4 \xrightarrow{\mathsf{WR}(y)} T_2$.

In the following, we write $T_1 \xrightarrow{\mathsf{WR}} T_2$ if $T_1 \xrightarrow{\mathsf{WR}(x)} T_2$ for some x when the key is irrelevant. We sometime write $T_1 \xrightarrow{\mathsf{WR}} T_2 \in \mathcal{G}$ to emphasize that the relation holds in $\mathcal{G}$. The two notations also apply to WW and RW. We write $T_1 \xrightarrow{\lambda} T_2$ if any of $T_1 \xrightarrow{\mathsf{WR}} T_2$, $T_1 \xrightarrow{\mathsf{WW}} T_2$, or $T_1 \xrightarrow{\mathsf{RW}} T_2$ holds, but the specific edge type is irrelevant. A path π in a graph $\mathcal{G}$, denoted $\pi \in \mathcal{G}$, is a non-empty finite sequence of edges $T_0 \xrightarrow{\lambda_0} T_1 \xrightarrow{\lambda_1} T_2 \cdots \xrightarrow{\lambda_{n-1}} T_n$. The path is a cycle if $T_0 = T_n$, and it is a ***simple*** cycle if no transactions are repeated except $T_0 = T_n$. We write $T \xrightarrow{\lambda^*} S$ for a (possible empty) path from T to S. A path in $\mathcal{G}$ is ***chord-free*** if there is no edge in $\mathcal{G}$ connecting two non-adjacent nodes in this path.

6.2 Dynamic Robustness Criterion

Definition 6.3. An abstract execution is called ***dynamically robust*** [11] iff it is serializable.

It is known that [1]:

Theorem 6.4. *An abstract execution $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR})$ is serializable iff for all $T \in \mathcal{T}$, $\textsc{Int}(T) \land \textsc{Ext}(T)$ holds and $\mathrm{DDG}(\mathcal{X})$ is acyclic.*

Therefore, it suffices to check whether $\mathrm{DDG}(\mathcal{X})$ is acyclic to determine whether $\mathcal{X}$ is dynamically robust. This requires us to understand the relation between edges in dynamic dependency graphs and relations in abstract executions. We state this relation in the following theorem, generalizing [11, Lemma 7] to mixed isolation levels. Note that Theorem 6.5 is *exhaustive* in covering all possible edge types (as well as SO) and isolation levels.

Theorem 6.5. *For an execution $\mathcal{X}$ and its dynamic dependency graph $\mathcal{G} = \mathrm{DDG}(\mathcal{X})$,*

(R1) $T_1[\_] \xrightarrow{\mathsf{SO}} T_2[\_] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$;
(R2) $T_1[\_] \xrightarrow{\mathsf{WR}} T_2[\_] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$;
(R3) $T_1[\_] \xrightarrow{\mathsf{WW}} T_2[\mathsf{SER/SI/PSI}] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$;

(R4) $T_1[\mathsf{SER}] \xrightarrow{\mathsf{RW}} T_2[\mathsf{SER}] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$;

(R5) $T_1[\_] \xrightarrow{\mathsf{WW}} T_2[\mathsf{PC/CC/RA}] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{AR}} T_2 \in \mathcal{X}$;

(R6) $T_1[\mathsf{SER}] \xrightarrow{\mathsf{RW}} T_2[\mathsf{SI/PSI/PC/CC/RA}] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{AR}} T_2 \in \mathcal{X}$;

(R7) $T_1[\mathsf{SI/PSI/PC/CC/RA}] \xrightarrow{\mathsf{RW}} T_2[\_] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{AR}} T_2 \in \mathcal{X} \lor T_2 \xrightarrow{\mathsf{VIS}} T_1 \in \mathcal{X}$.

In the following, we refer to the edge types on the left-hand sides of (R1)-(R4) as VIS-implicit edges (denoted $\xrightarrow{\mathsf{VIS}\text{-implicit}}$), those on (R5)-(R6) as AR-implicit edges (denoted $\xrightarrow{\mathsf{AR}\text{-implicit}}$), and that of (R7) as a *critical* edge (denoted $\xrightarrow{\text{Critical}}$).

Since VIS-implicit and AR-implicit edges by themselves cannot form cycles in $\mathsf{DDG}(\mathcal{X})$ (otherwise AR would be cyclic), every cycle in $\mathsf{DDG}(\mathcal{X})$ must contain at least one critical edge. Specifically, we have the following theorem.

THEOREM 6.6. *Let $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ be a consistent abstract execution such that $\mathcal{G} = \mathsf{DDG}(\mathcal{X})$ is cyclic. Then, every cycle in $\mathcal{G}$ contains at least one critical edge. Particularly, every cycle is of the form $T_1[\_] \xrightarrow{\lambda} T_2[\mathsf{SI/PSI/PC/CC/RA}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_]$, where T_3 is the AR-minimal transaction in the cycle, and T_2 is not a single-key read-only transaction.*

By distinguishing cases of $\mathsf{level}(T_2)$ in Theorem 6.6, we obtain a precise condition for $\mathcal{G}$ to be acyclic, generalizing [11, Definition 4 and Theorem 5] to mixed isolation levels.

THEOREM 6.7. *Let $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ be a consistent abstract execution. $\mathcal{G} = \mathsf{DDG}(\mathcal{X})$ is acyclic iff $\mathcal{G}$ does not contain a* ***critical cycle*** *π satisfying*

(C1) *π is simple and chord-free;*

(C2) *π has one of the following forms, where T_3 is the AR-minimal transaction in π and T_2 is not a single-key read-only transaction:*
- $\pi_1 = T_1[\_] \xrightarrow{\lambda} T_2[\mathsf{RA/CC}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_]$;
- $\pi_2 = T_1[\_] \xrightarrow{\lambda} T_2[\mathsf{PSI}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_]$, *where* $\neg(T_2 \bowtie T_3)$ *holds;*
- $\pi_3 = T_1[\_] \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} T_2[\mathsf{PC}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_]$, *where* $T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \notin \mathcal{G}$.
- $\pi_4 = T_1[\_] \xrightarrow{\mathsf{RW}(x)} T_2[\mathsf{SI}] \xrightarrow{\mathsf{RW}(y)} T_3[\_] \xrightarrow{\lambda^*} T_1[\_]$, *where* $x \neq y \land \neg(T_2 \bowtie T_3) \land T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \notin \mathcal{G}$.

(C3) $T_2 \xrightarrow{\mathsf{VIS}\text{-implicit} \,\cup\, \mathsf{AR}\text{-implicit}} T_3 \notin \mathcal{G}$.

Example 6.8. As shown in Figure 7, $\mathsf{DDG}(\mathcal{X}_1)$ is acyclic, which implies that $\mathcal{X}_1$ is serializable. In contrast, $\mathsf{DDG}(\mathcal{X}_2)$ contains the cycle $\pi = T_2[\mathsf{SER}] \xrightarrow{\mathsf{RW}(z)} T_3[\mathsf{SI}] \xrightarrow{\mathsf{RW}(y)} T_4[\mathsf{SER}] \xrightarrow{\mathsf{WR}(y)} T_2[\mathsf{SER}]$, which is a critical cycle of form π_4. Hence, $\mathcal{X}_2$ is not serializable.

By Theorems 6.4 and 6.7, we obtain our dynamic robustness criterion for abstract executions under mixed isolation levels.

THEOREM 6.9. *A consistent abstract execution $\mathcal{X}$ is dynamically robust iff $\mathsf{DDG}(\mathcal{X})$ contains no critical cycle.*

6.3 Static Dependency Graphs

We consider the setting where a database workload consists of a set of (transactional) program instances, extracted from transactional programs (e.g., Figure 4) by fixing their input parameters. A program instance describes the read and write operations of a transaction to be executed on the database. We denote by SOp the set of possible *static* operations on keys: $\mathsf{SOp} = \{\mathsf{R}_\iota(x), \mathsf{W}_\iota(x, \_) \mid \iota \in \mathsf{OpId}, x \in \mathsf{K}\}$,[5] where OpId is the set of operation identifiers.

Definition 6.10. A ***transactional program instance*** (or, program instance) P is a pair (O, po), where $O \subseteq \mathsf{SOp}$ is a finite, non-empty set of static operations and $\mathsf{po} \subseteq O \times O$ is a strict total order called the ***static program order***.

Let $P = (O, \mathsf{po})$ be a transactional program instance. We define the *read set* of P as $\mathsf{RSet}(P) \triangleq \{x \in \mathsf{K} \mid \exists o \in O.\ o = \mathsf{R}(x) \land \mathsf{po}_x^{-1}(o) = \emptyset\}$, and *write set* as $\mathsf{WSet}(P) \triangleq \{x \in \mathsf{K} \mid \exists o \in O.\ o = \mathsf{W}(x, \_)\}$. P is *read-only* if $\mathsf{WSet}(P) = \emptyset$. Otherwise, it is a *read-write* program instance. P is a *single-key* program instance if $|\mathsf{RSet}(P) \cup \mathsf{WSet}(P)| = 1$, and a *multi-key* program instance if. $|\mathsf{RSet}(P) \cup \mathsf{WSet}(P)| > 1$. Two program instances P_1 and P_2 *write-write conflict* with each other, denoted $P_1 \bowtie P_2$, if $\mathsf{WSet}(P_1) \cap \mathsf{WSet}(P_2) \neq \emptyset$. P_1 *read-write conflicts* with P_2, denoted $P_1 \triangleleft P_2$, if $\mathsf{RSet}(P_1) \cap \mathsf{WSet}(P_2) \neq \emptyset$.

When executed on a database, a program instance P performs a set of concrete read and write operations with specific values, resulting in a concrete transaction T. In this case, we call T an ***instantiation*** of P, denoted by the predicate $T[\![P]\!]$. Note that in our setting as described in Section 5.1, if $T[\![P]\!]$ holds, then T has the same read and write sets as P.

Definition 6.11. A ***workload*** is a pair $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$, where $\mathcal{P}$ is a set of program instances, the *static session order* $\mathsf{SO} \subseteq \mathcal{P} \times \mathcal{P}$ is the union of strict total orders on disjoint sets of $\mathcal{P}$, and $\mathsf{level} : \mathcal{P} \to \mathcal{L}$ is an ***allocation*** function that assigns an isolation level to each program instance in $\mathcal{P}$.

We refer readers to Figure 5 and Section 5.1 for an example workload based on the SmallBank benchmark.

We write $P[\ell]$ to denote a program instance P such that $\mathsf{level}(P) = \ell$, and call it an ℓ-program instance. We write $P[\_]$ if the isolation level of P is irrelevant. The notation $P[\ell_1/\ell_2]$ means that P is an ℓ_1- or ℓ_2-program instance; $P[\ell_1/\cdots/\ell_i]$ is defined similarly.

When executed on a database, a workload corresponds to a set of possible histories and abstract executions.

Definition 6.12. A *history* $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}_{\mathcal{H}})$ is ***generated*** by a workload $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level}_{\mathcal{W}})$ if there exists a bijection $g : \mathcal{T} \to \mathcal{P}$ satisfying $T[\![g(T)]\!] \land \mathsf{level}_{\mathcal{H}}(T) = \mathsf{level}_{\mathcal{W}}(g(T))$. An *abstract execution* $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ is ***generated*** by a workload $\mathcal{W}$ if $\mathcal{H}$ is generated by $\mathcal{W}$.

We use static dependency graphs of workloads to capture potential dependencies between program instances [11], which *over-approximate* the dependencies between concrete transactions in abstract executions generated by the workloads.

Definition 6.13. Let $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$ be a workload. For $x \in \mathsf{K}$ and $P_1, P_2 \in \mathcal{P}$ with $P_1 \neq P_2$, we define the following ***static dependency relations*** on $\mathcal{P}$:

[5]The value written by a static write operation is irrelevant for our analysis.

- Static WR dependency:
  $P_1 \xrightarrow{\mathsf{WR}(x)} P_2 \iff x \in \mathsf{WSet}(P_1) \cap \mathsf{RSet}(P_2)$.
- Static WW dependency:
  $P_1 \xrightarrow{\mathsf{WW}(x)} P_2 \iff x \in \mathsf{WSet}(P_1) \cap \mathsf{WSet}(P_2)$.
- Static RW dependency:
  $P_1 \xrightarrow{\mathsf{RW}(x)} P_2 \iff x \in \mathsf{RSet}(P_1) \cap \mathsf{WSet}(P_2)$.

The ***static dependency graph*** [11] of $\mathcal{W}$ is a tuple $\mathsf{SDG}(\mathcal{W}) = (\mathcal{P}, \mathsf{SO}, \mathsf{level}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, where $\mathsf{WR} = \bigcup_{x \in \mathsf{K}} \mathsf{WR}(x)$, $\mathsf{WW} = \bigcup_{x \in \mathsf{K}} \mathsf{WW}(x)$, and $\mathsf{RW} = \bigcup_{x \in \mathsf{K}} \mathsf{RW}(x)$.

Example 6.14. Figure 5 depicts the static dependency graph $\mathsf{SDG}(\mathcal{W}_{\mathrm{rb}})$ of the example workload $\mathcal{W}_{\mathrm{rb}}$ of SmallBank. For instance, there is an edge $\mathtt{Bal(N1)} \xrightarrow{\mathsf{RW}(\mathtt{Chk\{B1\}})} \mathtt{WC(N1,V1)}$ as $\mathtt{Bal(N1)}$ reads from $\mathtt{Chk\{B1\}}$ while $\mathtt{WC(N1,V1)}$ writes to it. Similarly, we have $\mathtt{WC(N1,V1)} \xrightarrow{\mathsf{RW}(\mathtt{Sav\{B1\}})} \mathtt{TS(N1,V1)}$. Moreover, there is an edge $\mathtt{TS(N1,V1)} \xrightarrow{\mathsf{WR}(\mathtt{Sav\{B1\}})} \mathtt{Bal(N1)}$ as $\mathtt{TS(N1,V1)}$ writes to $\mathtt{Sav\{B1\}}$ while $\mathtt{Bal(N1)}$ reads from it. As a result, there is a cycle in $\mathsf{SDG}(\mathcal{W}_{\mathrm{rb}})$: $\sigma = \mathtt{Bal(N1)[PC]} \xrightarrow{\mathsf{RW}(\mathtt{Chk\{B1\}})} \mathtt{WC(N1,V1)[SER]} \xrightarrow{\mathsf{RW}(\mathtt{Sav\{B1\}})} \mathtt{TS(N1,V1)[PSI]} \xrightarrow{\mathsf{WR}(\mathtt{Sav\{B1\}})} \mathtt{Bal(N1)[PC]}$.

6.4 Static Robustness Criterion

In this section, we establish the static robustness criterion for workloads under mixed isolation levels.

Definition 6.15. A workload $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$ is ***statically robust*** under level iff all the histories it generates are serializable.

Following the roadmap outlined in Section 5.4, to ensure that a workload $\mathcal{W}$ is statically robust, it suffices to ensure that its static dependency graph $\mathsf{SDG}(\mathcal{W})$ contains no cycles of certain forms, as characterized below.

Theorem 6.16. *A workload $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$ is statically robust under* level *if $\mathcal{G} = \mathsf{SDG}(\mathcal{W})$ contains no* ***static critical cycle*** *σ in one of the following forms, where P_2 is not a single-key read-only program instance, and $P_2 \xrightarrow{\mathsf{SO}} P_3 \notin \mathcal{G}$:*

(S1) $\sigma_1 = P_1[\_] \xrightarrow{\lambda} P_2[\mathsf{RA/CC}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_]$;

(S2) $\sigma_2 = P_1[\_] \xrightarrow{\lambda} P_2[\mathsf{PSI}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_]$, *where* $\neg(P_2 \bowtie P_3)$ *holds;*

(S3) $\sigma_3 = P_1[\_] \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} P_2[\mathsf{PC}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_]$, *where* $\neg(P_1 \xrightarrow{\mathsf{SO}} P_2)$ *holds;*

(S4) $\sigma_4 = P_1[\_] \xrightarrow{\mathsf{RW}(x)} P_2[\mathsf{SI}] \xrightarrow{\mathsf{RW}(y)} P_3[\_] \xrightarrow{\lambda^*} P_1[\_]$, *where* $x \neq y \land \neg(P_2 \bowtie P_3) \land \neg(P_1 \xrightarrow{\mathsf{SO}} P_2)$ *holds.*

Example 6.17. As shown in Figure 5, the static dependency graph $\mathsf{SDG}(\mathcal{W}_{\mathrm{rb}})$ of the example workload $\mathcal{W}_{\mathrm{rb}}$ contains no static critical cycles of any form. Therefore, by Theorem 6.16, the workload $\mathcal{W}_{\mathrm{rb}}$ is statically robust. In particular, the cycle σ presented in Example 6.14 is not a static critical cycle. However, if we modify the isolation level of $\mathtt{WC(N1,V1)}$ to SI, we obtain a static critical cycle of form σ_4: $\sigma = \mathtt{Bal(N1)[PC]} \xrightarrow{\mathsf{RW}(\mathtt{Chk\{B1\}})} \mathtt{WC(N1,V1)[SI]} \xrightarrow{\mathsf{RW}(\mathtt{Sav\{B1\}})} \mathtt{TS(N1,V1)[PSI]} \xrightarrow{\mathsf{WR}(\mathtt{Sav\{B1\}})} \mathtt{Bal(N1)[PC]}$.

6.5 Safe Allocation for Robustness

By Theorem 6.16, we can check whether a workload is statically robust. This theorem also facilitates the *safe allocation problem*: given a workload $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \_)$ that contains a set $\mathcal{P}$ of program instances but leaves the allocation function level unspecified, find an allocation function level such that the resulting workload $(\mathcal{P}, \mathsf{SO}, \mathsf{level})$ is statically robust. The following allocation rules provide a simple solution to this problem, which assign appropriate levels to program instances based on their read and write sets.

Theorem 6.18. *Let $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$ be a workload where the allocation function* level *is defined by the following rules (A1)–(A4). Then $\mathcal{W}$ is statically robust.*

(A1) $\mathsf{level}(P) = \mathsf{RA}$ *if P is a write-only or single-key read-only program instance;*

(A2) $\mathsf{level}(P) = \mathsf{PC}$ *if P is a multi-key read-only program instance;*

(A3) $\mathsf{level}(P) = \mathsf{PSI}$ *if P is a read-write program instance, and for each other program instance P', $\neg(P \lhd P') \lor (P \bowtie P')$ holds;*

(A4) $\mathsf{level}(P) = \mathsf{SER}$*, otherwise.*

Note that $\mathsf{level} : \mathcal{P} \to \mathcal{L}$ is well-defined by the allocation rules (A1)–(A4), as they form a complete partition of $\mathcal{P}$. Although simple, these allocation rules are effective in practice; see Section 7.

Example 6.19. The allocation function $\mathsf{level}_{\mathrm{rb}}$ in Figure 5 is derived from the rules in Theorem 6.18: the two `Bal` instances run under PC (A2), the `TS`, `DC`, and `Ama` instances run under PSI (A3), and the `WC` instance runs under SER (A4).

7 EVALUATION

We have implemented a tool, called MIA, to safely allocate mixed isolation levels to transaction workloads according to Theorem 6.18. Our allocator consists of approximately 900 lines of Java code. We defer to [50, Appendix M] its pseudocode and complexity analysis ($O(n^2)$, where n is the number of program instances).

We evaluate MIA, alongside state-of-the-art approaches [11, 24] (implemented in another roughly 900 lines of Java code), to answer the following two questions:

Q1. Effectiveness: Can MIA safely allocate *weaker* isolation levels than existing approaches while preserving serializability (Section 7.2)?

Q2. Efficiency: How efficient is MIA in computing isolation level allocations (Section 7.3)?

7.1 Benchmarks and Experimental Setup

Our experiments consider three representative benchmarks widely used in prior work [7, 16, 25]: SmallBank [3] (described in Section 5), TPC-C [44], and Courseware [29].

TPC-C is an established industry standard for evaluating OLTP workloads. It models a wholesale distribution business with five core transaction types, whose default workload proportions are shown in parentheses: New-Order (45%), Payment (43%), Order-Status (4%), Delivery (4%), and Stock-Level (4%). Courseware models a university course enrollment workload based on student, course, and course-selection schemas. It comprises five transaction types: registration, course enrollment, course addition, course deletion, and selection-status queries, which mix reads and writes across

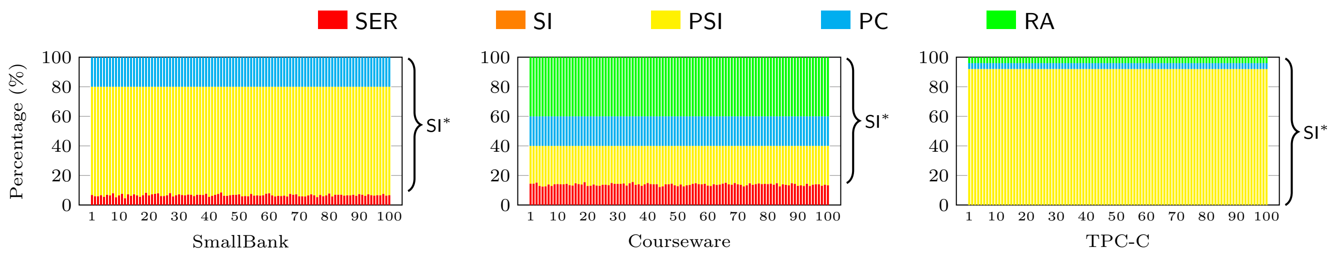


Figure 8: Allocationg isolation levels. SI* denotes the proportion of the workload assigned to SI by existing approaches [11, 24].

related entities. For both Courseware and SmallBank, the workload proportions are evenly distributed across all transaction types. Moreover, for all three benchmarks, each workload consists of 1000 program instances that uniformly access 500 keys.

To evaluate the scalability of MIA, we consider YCSB-like transactional workloads generated by a parametric workload generator. We configure it with several parameters, with default values shown in parentheses: the number of program instances (5000), the number of operations per instance (10), and the number of keys (300).

All experiments are performed on a local machine equipped with an Intel i7-13700k CPU and 64GB of RAM.

7.2 Q1: Allocating Weaker Isolation Levels

We first compare MIA with existing approaches across the three benchmarks. Figure 8 shows the results over 100 runs.

Overall, MIA consistently allocates weaker isolation levels than these approaches, owing to its fine-grained allocation strategy (Theorem 6.18). For approximately 95%, 88%, and 100% of workloads on SmallBank, Courseware, and TPC-C, respectively, existing approaches can only lower the levels to SI in order to preserve SER for the entire workload. In contrast, MIA is able to further lower the levels of individual program instances, depending on their program logic and dependencies. Note that the allocation results vary slightly across runs, as program instances of different types are generated randomly, leading to varying dependencies among them.

More specifically, for SmallBank, 20% of read-only program instances (`Balance`) are always allocated to PC, 60% (`Amalgamate`, `DepositChecking`, and `TransactSavings`) to PSI, and the remaining 20% of read-write instances (`WriteCheck`) to either PSI or SER. Regarding Courseware, 40% of program instances (`Register` and `AddCourse`) are allocated to RA, 20% (`Query`) to PC, and the remaining 40% (`Enroll` and `RemoveCourse`) to either PSI or SER. For TPC-C, 92% of program instances (`Delivery`, `NewOrder`, and `Payment`) are allocated to PSI, while RA and PC each account for 4%. Consequently, no instances are allocated to SER.

Detailed example allocations of workloads for each benchmark are deferred to [50, Appendix N].

7.3 Q2: Allocation Efficiency of MIA

Safely allocating weaker isolation levels to large workloads is highly desirable for achieving high system performance without sacrificing SER. Our second set of experiments evaluate MIA's scalability by varying several workload parameters. Figure 9 depicts the results.

Overall, MIA demonstrates strong performance, completing isolation level assignment for a workload of 10k program instances

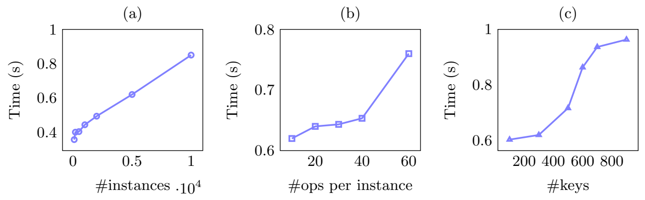


Figure 9: MIA's performance under varying workloads.

within one second, as shown in Figure 9(a). As expected, the allocation time increases with the number of program instances (a), operations per instance (b), and keys (c). Nevertheless, the observed overhead remains modest in practice, despite a worst-case time complexity of $O(n^2)$. This behavior can be attributed to characteristics of realistic workloads, in which the resulting dependency graphs are often sparse. For example, with more keys, write-write conflicts (i.e., WW dependencies) between instances become less frequent, as accesses are distributed over a larger key space.

Remark (Practical Impact). Lower isolation allocation can translate into practical performance improvements in database systems. We validate this using our proposed concurrency control protocol (Section 4). Compared to executing all transactions under SER, as well as to a mixed SI-SER allocation, our fine-grained PC-SI-SER allocation yields noticeable performance gains in both throughput and latency. For example, on Courseware, our allocation achieves a 300% (resp. 20%) throughput improvement over SER-only execution (resp. SI-SER allocation). See [50, Appendix O] for details.

8 RELATED WORK

This section discusses three lines of research on database transactions and isolation levels, which are closely related to our work.

Formalizing Isolation Levels. Substantial effort has been devoted to the formal characterization of isolation levels [1, 12, 13, 19, 22], alongside advances in reliable and performant database transaction systems. Collectively, these works have deepened our understanding of what it means for a database to satisfy a given isolation level. Yet, they do not address our mixed isolation settings, which raise a more general and practically relevant class of questions.

A recent work by Bouajjani et al. [14] takes a first step toward answering these questions. Our work differs from theirs primarily in the role played by the underlying formalizations. Their characterization builds on the framework in [13], which is specifically tailored for black-box validation of isolation guarantees (i.e., checking whether a collected execution history satisfies a prescribed mixed assignment of isolation levels). In contrast, MixIso is intended to serve as a more general semantic foundation. While it also can support black-box validation (see Section 9), MixIso enables a broader range of applications, including semantic conformance checking and robustness checking, as demonstrated in this paper.

Moreover, while the sets of isolation levels considered in the two works partially overlap, the specific levels targeted differ. We include CC and PSI, two isolation levels that have seen practical applications [23, 37, 38], while their framework includes RC that is particularly relevant to SQL databases. As mentioned in Section 3, the two characterizations are equivalent for the common isolation levels. Taken together, these two complementary works provide reference points for database designs and implementations.

Semantic Conformance Checking. Recent years have witnessed the development of effective black-box isolation checkers based on the aforementioned formalizations [13, 14, 18, 30, 32, 33, 43, 48]. All these tools target deployed database systems and validate semantic conformance to isolation guarantees by checking collected execution histories. With the exception of the recent work [14] discussed above, existing checkers focus on settings in which all transactions execute under the same isolation level.

Our work is closely related to prior efforts on verifying semantic conformance between database protocols (or designs) and their claimed isolation guarantees [26, 35], with the goal of ensuring that concurrency control mechanisms are correct from an early stage. Unlike these works, however, we are able to handle mixed isolation settings, as demonstrated by our proofs that two concurrency control protocols conform to their intended mixes of isolation levels. While these approaches are supported by automated or semi-automated tools, the development of tool support for our proof method remains an important direction for future work.

Robustness Checking. Early work by Fekete et al. [24, 25] introduces a dependency-graph-based criterion for checking robustness against SI. This approach is later extended by [11, 20] to isolation levels weaker than SI under the assumption of atomic visibility (i.e., RA). Vandevoort et al. [31, 46, 47] further study even weaker isolation levels, such as RC, which are commonly used in SQL databases. However, their work is restricted to centralized settings, where every read must observe the most recently committed version, an assumption that does not generally hold in distributed transactions. From a different technical perspective, Beillahi et al. [7–9] reduce robustness checking to reachability analysis in transactional programs running over, e.g., a sequentially consistent shared memory. This approach can be more precise in detecting robustness violations, albeit at the cost of higher checking overhead.

Our work distinguishes itself by establishing robustness criteria for distributed transactions under a wide range of mixed isolation levels. Unlike previous work, which focuses on robustness against individual isolation levels (with the exception of a specific mix of RC-SI-SSI [47]), we address a more general and practically relevant setting that naturally arises in databases supporting mixed isolation levels. This generalization requires an underlying formal semantics for such heterogeneity, as discussed earlier.

Similar to prior approaches [11, 20, 24, 25], our static analysis based on dependency graphs is over-approximated and thus conservative, in the sense that there may exist weaker isolation allocations that our analysis does not identify. Nevertheless, as shown in Section 7, our approach can already yield strictly lower isolation allocations than these approaches. Exploring more precise yet efficient allocation techniques, e.g., by leveraging reachability-based analysis as in [7], is interesting future work.

9 DISCUSSION AND CONCLUSION

MixIso provides a semantic foundation for mixing isolation levels. We have demonstrated its effectiveness through two applications: semantic conformance checking and robustness checking. Given the widespread support for per-transaction isolation levels in practice, together with the continuing emergence of new isolation levels for modern distributed applications, our work can pave the way to developing more reliable and performant transaction systems that support mixed isolation levels.

In the final part of the paper, we discuss the limitations of this work and sketch two additional applications that can be enabled by MixIso, pointing to directions for future exploration.

Limitations. MixIso is based on the KVS model. While it supports many commonly used SQL queries, such as those in SmallBank, TPC-C, and Courseware, it cannot handle range queries; supporting them would require extending our semantics to a relational model. In addition, MixIso builds on the axiomatic framework in [19], which assumes atomic visibility and thus does not cover RC. Beyond the future directions discussed in Section 8, our robustness analysis can be lifted from concrete workloads to transaction templates [46], enabling reasoning across all possible workloads.

Black-box Isolation Validation. This technique validates isolation guarantees by collecting execution histories of transactions as an external observer of the database and checking whether they satisfy the isolation level in question. Within MixIso, the key step is to derive a dependency-graph–based characterization for mixed isolation levels (akin to Adya's direct serialization graph for SER [1]), which is semantically equivalent to our axiomatic semantics.[6] This derivation can follow the technique in [20]. Under this characterization, violations of mixed isolation levels are represented as specific graph patterns, such as cycles, in the dependency graph. Validation then proceeds by searching for these patterns, and can be efficiently automated by encoding transactional dependencies as constraints and solving them using an SMT solver, as in [30, 43].

Adaptive Concurrency Control. MixIso enables the design of database systems supporting adaptive concurrency control [52], in which isolation levels are adjusted dynamically in response to application workload characteristics. This setting differs from robustness checking, which performs an offline analysis prior to workload execution. In an adaptive system, a semantics-guided adaptor built on MixIso operates alongside the concurrency control layer and assists in on-the-fly isolation selection. To improve performance, the system may consider lowering the isolation levels of certain transactions, potentially to different levels, while relying on MixIso to determine whether such adjustments are semantically safe in the presence of mixed isolation interactions. Based on this semantic check, the system can then select appropriate, independently implemented concurrency control mechanisms to execute transactions.

[6]Note that the dependency graphs used here are constructed from concrete execution histories, whereas those in Section 6 are defined over abstract executions.

REFERENCES

[1] A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph. D. Dissertation. Massachusetts Institute of Technology, USA.

[2] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *ICDCS 2016*. IEEE Computer Society, 405–414. doi:10.1109/ICDCS.2016.98

[3] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE 2008*. 576–585. doi:10.1109/ICDE.2008.4497466

[4] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Röhm. 2008. Serializable Executions with Snapshot Isolation: Modifying Application Code or Mixing Isolation Levels?. In *DASFAA 2008*, Vol. 4947. 267–281. doi:10.1007/978-3-540-78568-2_21

[5] 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.

[6] 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 (July 2016), 15:1–15:45. doi:10.1145/2909870

[7] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019. Checking Robustness Against Snapshot Isolation. In *CAV 2019*, Vol. 11562. 286–304. doi:10.1007/978-3-030-25543-5_17

[8] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019. Robustness Against Transactional Causal Consistency *(CONCUR, Vol. 140)*. 30:1–30:18. doi:10.4230/LIPICS.CONCUR.2019.30

[9] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2021. Checking Robustness Between Weak Transactional Consistency Models. In *ESOP 2021 (LNCS, Vol. 12648)*. Springer, 87–117. doi:10.1007/978-3-030-72019-3_4

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10. doi:10.1145/568271.223785

[11] Giovanni Bernardi and Alexey Gotsman. 2016. Robustness against Consistency Models with Atomic Visibility. In *CONCUR 2016*, Vol. 59. 7:1–7:15. doi:10.4230/LIPIcs.CONCUR.2016.7

[12] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.

[13] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 165 (Oct. 2019), 28 pages. doi:10.1145/3360591

[14] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. 2025. On the Complexity of Checking Mixed Isolation Levels for SQL Transactions. In *CAV 2025*, Vol. 15934. 315–337. doi:10.1007/978-3-031-98685-7_15

[15] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *ECOOP 2015*, Vol. 37. 568–590. doi:10.4230/LIPIcs.ECOOP.2015.568

[16] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable Isolation for Snapshot Databases. In *SIGMOD 2008*. ACM, 729–738. doi:10.1145/1376616.1376690

[17] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Transactions on Database Systems* 34, 4 (Dec. 2009), 1–42. doi:10.1145/1620585.1620587

[18] Zhiheng Cai, Si Liu, Hengfeng Wei, Yuxing Chen, and Anqun Pan. 2026. Fast Verification of Strong Database Isolation. *Proc. VLDB Endow.* 19, 4 (2026), 563–575.

[19] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *CONCUR 2015*, Vol. 42. 58–71. doi:10.4230/LIPIcs.CONCUR.2015.58

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

[21] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proc. VLDB Endow.* 14, 12 (2021), 3014–3027. doi:10.14778/3476311.3476379

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

[23] ElectricSQL. Accessed in January, 2026. https://electric-sql.com/.

[24] Alan Fekete. 2005. Allocating isolation levels to transactions. In *PODS 2005*. 206–215. doi:10.1145/1065167.1065193

[25] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems* 30, 2 (June 2005), 492–528. doi:10.1145/1071610.1071615

[26] 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. doi:10.14778/3718057.3718065

[27] Israel Gold and Haran Boral. 1986. The power of the private workspace model. *Information Systems* 11, 1 (1986), 1–7. doi:10.1016/0306-4379(86)90019-0

[28] Israel Gold, Oded Shmueli, and Micha Hofri. 1985. The private workspace model feasibility and applications to 2PL performance improvements. In *VLDB '85*. 192–208.

[29] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *POPL '16*. 371–384. doi:10.1145/2837614.2837625

[30] 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. doi:10.14778/3583140.3583145

[31] Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. 2022. Deciding Robustness for Lower SQL Isolation Levels. *ACM Transactions on Database Systems* 47, 4 (Dec. 2022), 1–41. doi:10.1145/3561049

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

[33] 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, OOPSLA (2024), 876–904. doi:10.1145/3689742

[34] 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. doi:10.1145/3639264

[35] 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. doi:10.1007/978-3-030-17465-1_3

[36] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI 2013*. USENIX Association, 313–328.

[37] Microsoft. Accessed in January, 2026. Azure CosmosDB DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[38] Neo4j. Accessed in January, 2026. https://neo4j.com/.

[39] Cuong D. T. Nguyen, Kevin Chen, Christopher DeCarolis, and Daniel J. Abadi. 2025. Are Database System Researchers Making Correct Assumptions about Transaction Workloads? *Proc. ACM Manag. Data* 3, 3, Article 131 (June 2025), 26 pages. doi:10.1145/3725268

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

[41] Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1850–1861. doi:10.14778/2367502.2367523

[42] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP '11*. 385–400. doi:10.1145/2043556.2043592

[43] 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.

[44] TPC-C. Accessed in December, 2025. TPC-C Homepage. https://www.tpc.org/tpcc/.

[45] Brecht Vandevoort, Alan D. Fekete, Bas Ketsman, Frank Neven, and Stijn Vansummeren. 2025. Using Read Promotion and Mixed Isolation Levels for Performant Yet Serializable Execution of Transaction Programs. In *Proc. VLDB Endow.* (2025) *(VLDB, Vol. 18)*. 2846–2858. doi:10.14778/3746405.3746412

[46] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2021. Robustness against Read Committed for Transaction Templates. In *Proceedings of the VLDB Endowment (VLDB, Vol. 14)*. 2141–2153. doi:10.14778/3476249.3476268

[47] Brecht Vandevoort, Bas Ketsman, and Frank Neven. 2025. Allocating Isolation Levels to Transactions in a Multiversion Setting. *ACM Transactions on Database Systems* 50, 2 (June 2025), 1–24. doi:10.1145/3716374

[48] 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. doi:10.1109/ICDE65448.2025.00298

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

[50] Qiuhuan Xiong, Hengfeng Wei, Si Liu, Yuxing Chen, and Jidong Ge. 2026. *On Mixing Database Isolation Levels.* Technical Report. https://github.com/EagleBear2002/MixIso/blob/main/tech-rpt.pdf.

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

[52] Qiyu Zhuang, Wei Lu, Shuang Liu, Yuxing Chen, Xinyue Shi, Zhanhao Zhao, Yipeng Sun, Anqun Pan, and Xiaoyong Du. 2025. TxnSails: Achieving Serializable Transaction Scheduling with Self-Adaptive Isolation Level Selection. *Proc. VLDB Endow.* 18, 11 (2025), 4227–4240. doi:10.14778/3749646.3749689

A SURVEY OF MIXED ISOLATION LEVEL SEMANTICS IN DATABASE SYSTEMS

In this section, we present a brief survey of how mixed isolation levels are supported in several mainstream database systems. Specifically, we examine (1) which isolation levels can be mixed within the same system, and (2) the semantics that these systems provide for such mixed-isolation executions. To ensure accuracy and authority, our discussion is based on official documentation, whitepapers, and research publications authored by the developers of these systems. We find that while many systems support mixed isolation levels, the semantics provided are often unclear, imprecise, implementation-specific, or informal.

The "General Note" for each database system below is based on our interpretation of the available sources. We recognize that some of these assessments may be subject to different interpretations, and our categorization is not intended to be definitive or exhaustive.

A.1 Oracle (AI) Database

A.1.1 General Note. The semantics for mixed isolation levels in Oracle AI Database are documented in terms of how isolation levels affect read consistency and concurrency control. It is quite general but lacks specific details about interactions between different isolation levels and are far from complete and formal.

A.1.2 Isolation Levels Supported. "Oracle AI Database provides the transaction isolation levels: Read Committed Isolation Level, Serializable Isolation Level, Read-Only Isolation Level, Read Committed Isolation Level, Serializable Isolation Level, and Read-Only Isolation Level. " [7]

A.1.3 Semantics for Mixed Isolation Levels. "Use the `SET TRANSACTION` statement to establish the current transaction as read-only or read-/write, establish its isolation level, assign it to a specified rollback segment, or assign a name to the transaction.

The operations performed by a `SET TRANSACTION` statement affect only your current transaction, not other users or other transactions. " [8]

"In the read committed isolation level, every query executed by a transaction sees only data committed before the query–not the transaction–began.

In the serializable isolation level, a transaction sees only changes committed at the time the transaction–not the query–began and changes made by the transaction itself.

The read-only isolation level is similar to the serializable isolation level, but read-only transactions do not permit data to be modified in the transaction unless the user is `SYS`.

Oracle AI Database permits a serializable transaction to modify a row only if changes to the row made by other transactions were already committed when the serializable transaction began." [9]

[7]https://docs.oracle.com/en/database/oracle/oracle-database/26/cncpt/data-concurrency-and-consistency.html#GUID-2A0FDFF0-5F72-4476-BFD2-060A20EA1685
[8]https://docs.oracle.com/en/database/oracle/oracle-database/26/sqlrf/SET-TRANSACTION.html#SQLRF55422
[9]https://docs.oracle.com/en/database/oracle/oracle-database/26/cncpt/data-concurrency-and-consistency.html#GUID-E6FB4581-CD65-4C18-AFDD-ACB3243238D3

A.2 PostgreSQL

A.2.1 General Note. The semantics for mixed isolation levels in PostgreSQL are documented in terms of how isolation levels affect read consistency and concurrency control. It is quite general but lacks specific details about interactions between different isolation levels and are far from complete and formal.

A.2.2 Isolation Levels Supported. "In PostgreSQL, you can request any of the four standard transaction isolation levels (i.e., `Read Uncommitted`, `Read Committed`, `Repeatable Read`, and `Serializable`), but internally only three distinct isolation levels are implemented, i.e., PostgreSQL's `Read Uncommitted` mode behaves like `Read Committed`. This is because it is the only sensible way to map the standard isolation levels to PostgreSQL's multiversion concurrency control architecture." [10]

A.2.3 Semantics for Mixed Isolation Levels. "The `SET TRANSACTION` command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. `SET SESSION CHARACTERISTICS` sets the default transaction characteristics for subsequent transactions of a session. These defaults can be overridden by `SET TRANSACTION` for an individual transaction.

The available transaction characteristics are the transaction isolation level, the transaction access mode (read/write or read-only), and the deferrable mode. In addition, a snapshot can be selected, though only for the current transaction, not as a session default.

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

`READ COMMITTED` A statement can only see rows committed before it began. This is the default.

`REPEATABLE READ` All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

`SERIALIZABLE` All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction. If a pattern of reads and writes among concurrent serializable transactions would create a situation which could not have occurred for any serial (one-at-a-time) execution of those transactions, one of them will be rolled back with a `serialization_failure` error." [11]

A.3 CockroachDB

A.3.1 General Note. The semantics for mixed isolation levels in CockroachDB are documented in terms of how isolation levels affect read consistency and concurrency control. It is quite general but lacks specific details about interactions between different isolation levels and are far from complete and formal.

A.3.2 Isolation Levels Supported. "Isolation is an element of ACID transactions that determines how concurrency is controlled, and ultimately guarantees consistency. CockroachDB offers two transaction isolation levels: `SERIALIZABLE` and `READ COMMITTED`. " [12]

[10]https://www.postgresql.org/docs/18/transaction-iso.html
[11]https://www.postgresql.org/docs/18/transaction-iso.html
[12]https://www.cockroachlabs.com/docs/v25.4/transactions#isolation-levels

A.3.3 Semantics for Mixed Isolation Levels. "Regardless of the isolation levels of other transactions, transactions behave according to their respective isolation levels: Statements in `SERIALIZABLE` transactions see data that committed before the transaction began, whereas statements in `READ COMMITTED` transactions see data that committed before each statement began. Therefore:

- If a `READ COMMITTED` transaction `R` commits before a `SERIALIZABLE` transaction `S`, every statement in `S` will observe all writes from `R`. Otherwise, `S` will not observe any writes from `R`.
- If a `SERIALIZABLE` transaction `S` commits before a `READ COMMITTED` transaction `R`, every subsequent statement in `R` will observe all writes from `S`. Otherwise, `R` will not observe any writes from `S`.

However, there is one difference in how `SERIALIZABLE` writes affect non-locking reads: While writes in a `SERIALIZABLE` transaction can block reads in concurrent `SERIALIZABLE` transactions, they will not block reads in concurrent `READ COMMITTED` transactions. Writes in a `READ COMMITTED` transaction will never block reads in concurrent transactions, regardless of their isolation levels. Therefore:

- If a `READ COMMITTED` transaction `R` writes but does not commit before a `SERIALIZABLE` transaction `S`, no statement in `S` will observe or be blocked by any uncommitted writes from `R`.
- If a `SERIALIZABLE` transaction `S` writes but does not commit before a `READ COMMITTED` transaction `R`, no statement in `R` will observe or be blocked by any uncommitted writes from `S`.
- If a `SERIALIZABLE` transaction `S1` writes but does not commit before a `SERIALIZABLE` transaction `S2`, the first statement in `S2` that would observe an unwritten row from `S1` will be blocked until `S1` commits or aborts." [13]

A.4 Microsoft SQL Server

A.4.1 General Note. The semantics for mixed isolation levels in Microsoft SQL Server is primarily documented in terms of how isolation levels affect locking behavior, and is thus implementation-specific.

A.4.2 Isolation Levels Supported. "Syntax for SQL Server, Azure SQL Database, and SQL database in Microsoft Fabric": [14]

```
SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
    }
```

A.4.3 Semantics for Mixed Isolation Levels. "When you use `sp_bindsession` to bind two sessions, each session retains its isolation level setting. Using `SET TRANSACTION ISOLATION LEVEL` to change the isolation level setting of one session doesn't affect the setting of any other sessions bound to it. " [15]

[13]https://www.cockroachlabs.com/docs/v25.4/transactions#mixed-isolation-levels
[14]https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql

"Transactions specify an isolation level that defines the degree to which one transaction must be isolated from the resource or data modifications made by other transactions. Isolation levels are described in terms of which concurrency side-effects, such as dirty reads or phantom reads, are allowed.

Transaction isolation levels control:

- Whether locks are acquired when data is read, and what type of locks are requested.
- How long the read locks are held.
- Whether a read operation referencing rows modified by another transaction:
  - Blocks until the exclusive lock on the row is freed.
  - Retrieves the committed version of the row that existed at the time the statement or transaction started.
  - Reads the uncommitted data modification." [16]

"The transaction isolation levels define the type of locks acquired on read operations. Shared locks acquired for `READ COMMITTED` or `REPEATABLE READ` are generally row locks, although the row locks can be escalated to page or table locks if a significant number of the rows in a page or table are referenced by the read. If the transaction modifies a row after it was read, the transaction acquires an exclusive lock to protect that row, and the exclusive lock is retained until the transaction completes.

Choosing a transaction isolation level doesn't affect the locks acquired to protect data modifications. A transaction always gets an exclusive lock on any data it modifies, and holds that lock until the transaction completes, regardless of the isolation level set for that transaction." [17]

A.5 MySQL (InnoDB)

A.5.1 General Note. The semantics for mixed isolation levels in MySQL (InnoDB) are primarily documented in terms of how isolation levels affect locking behavior, and is thus implementation-specific.

A.5.2 Isolation Levels Supported. "InnoDB offers all four transaction isolation levels described by the SQL:1992 standard: `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`. The default isolation level for InnoDB is `REPEATABLE READ`. " [18]

A.5.3 Semantics for Mixed Isolation Levels. "You can set transaction characteristics globally, for the current session, or for the next transaction only. " [19]

```
SET [GLOBAL | SESSION] TRANSACTION
  transaction_characteristic [,
      transaction_characteristic] ...

transaction_characteristic: {
```

[15]https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver17
[16]https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver17#isolation-levels-in-the-database-engine
[17]https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver17
[18]https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html
[19]https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html

```
    ISOLATION LEVEL level
  | access_mode
}

level: {
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
}

access_mode: {
     READ WRITE
   | READ ONLY
}
```

"InnoDB supports each of the transaction isolation levels described here using different locking strategies. You can enforce a high degree of consistency with the default REPEATABLE READ level, for operations on crucial data where ACID compliance is important. Or you can relax the consistency rules with READ COMMITTED or even READ UNCOMMITTED, in situations such as bulk reporting where precise consistency and repeatable results are less important than minimizing the amount of overhead for locking. SERIALIZABLE enforces even stricter rules than REPEATABLE READ, and is used mainly in specialized situations, such as with XA transactions and for troubleshooting issues with concurrency and deadlocks.

The following list describes how MySQL supports the different transaction levels. The list goes from the most commonly used level to the least used. " [20]

A.6 YugabyteDB

A.6.1 General Note. The semantics for mixed isolation levels in YugabyteDB are primarily documented in terms of how isolation levels affect locking behavior, and is thus implementation-specific.

A.6.2 Isolation Levels Supported. "Transaction isolation level support differs between the YSQL and YCQL APIs:

- YSQL supports Serializable, Snapshot, and Read Committed isolation levels.
- YCQL supports only Snapshot isolation using the BEGIN TRANSACTION syntax." [21]

A.6.3 Semantics for Mixed Isolation Levels. "Use the SET TRANSACTION statement to set the current transaction isolation level. " [22]

" In order to support the three isolation levels, the lock manager internally supports the following three types of locks:

- Serializable read lock is taken by serializable transactions on values that they read in order to guarantee they are not modified until the transaction commits.
- Serializable write lock is taken by serializable transactions on values they write.
- Snapshot isolation write lock is taken by a snapshot isolation (and also read committed) transaction on values that it modifies.

The following matrix shows conflicts between these types of locks at a high level: . . ." [23]

A.7 TiDB

A.7.1 General Note. The semantics for mixed isolation levels in TiDB seems missing from the official documentation. Instead, the official documentation describes the concurrency control mechanisms used to implement different isolation levels.

A.7.2 Isolation Levels Supported. "The isolation levels supported by TiDB are RC (Read Committed) and SI (Snapshot Isolation), where SI is basically equivalent to the RR (Repeatable Read) isolation level." [24]

A.7.3 Semantics for Mixed Isolation Levels. "The SET TRANSACTION statement can be used to change the current isolation level on a GLOBAL or SESSION basis. " [25]

B PROOF OF THEOREM 3.4

Proof of Theorem 3.4. Consider an abstract execution $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \text{level}, \mathsf{VIS}, \mathsf{AR})$ where $T \in \mathcal{T}$ is an arbitrary transaction. We first show that

- $\textsc{TotalVis}(T) \implies \textsc{Prefix}(T)$. Suppose that $\textsc{TotalVis}(T)$ holds. For any $T', S \in \mathcal{T}$, suppose that $T' \xrightarrow{\mathsf{AR}} S \xrightarrow{\mathsf{VIS}} T$. Since $\mathsf{VIS} \subseteq \mathsf{AR}$, we have
$$T' \xrightarrow{\mathsf{AR}} S \xrightarrow{\mathsf{AR}} T.$$
Since AR is transitive, we have
$$T' \xrightarrow{\mathsf{AR}} T.$$
By the definition of $\textsc{TotalVis}(T)$,
$$T' \xrightarrow{\mathsf{VIS}} T.$$
Therefore, $\textsc{Prefix}(T)$ holds.
- $\textsc{TotalVis}(T) \implies \textsc{NoConflict}(T)$. Suppose that $\textsc{TotalVis}(T)$ holds. For any $T' \in \mathcal{T}$, suppose that $T' \bowtie T \wedge T' \xrightarrow{\mathsf{AR}} T$. By the definition of $\textsc{TotalVis}(T)$, we have
$$T' \xrightarrow{\mathsf{VIS}} T.$$
Therefore, $\textsc{NoConflict}(T)$ holds.
- $\textsc{Prefix}(T) \implies \textsc{TransVis}(T)$. Suppose that $\textsc{Prefix}(T)$ holds. For any $T', S \in \mathcal{T}$, suppose that $T' \xrightarrow{\mathsf{VIS}} S \xrightarrow{\mathsf{VIS}} T$. Since $\mathsf{VIS} \subseteq \mathsf{AR}$, we have
$$T' \xrightarrow{\mathsf{AR}} S \xrightarrow{\mathsf{VIS}} T.$$
By the definition of $\textsc{Prefix}(T)$,
$$T' \xrightarrow{\mathsf{VIS}} T.$$
Therefore, $\textsc{TransVis}(T)$ holds.

Now we show the implications between isolation levels one by one.

[20] https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html
[21] https://docs.yugabyte.com/stable/architecture/transactions/isolation-levels/
[22] https://docs.yugabyte.com/stable/api/ysql/the-sql-language/statements/txn_set/
[23] https://docs.yugabyte.com/stable/architecture/transactions/isolation-levels/
[24] https://docs.pingcap.com/tidbcloud/dev-guide-transaction-restraints/#isolation-levels
[25] https://docs.pingcap.com/tidb/stable/sql-statement-set-transaction/#set-transaction

- CASE $\mathsf{SER}(T) \implies \mathsf{SI}(T)$. Follows from the definitions of $\mathsf{SER}(T)$ and $\mathsf{SI}(T)$, along with the fact that $\textsc{TotalVis}(T) \implies \textsc{Prefix}(T) \land \textsc{NoConflict}(T)$.
- CASE $\mathsf{SI}(T) \implies \mathsf{PSI}(T)$. Follows from the definitions of $\mathsf{SI}(T)$ and $\mathsf{PSI}(T)$, along with the fact that $\textsc{Prefix}(T) \implies \textsc{TransVis}(T)$.
- CASE $\mathsf{PSI}(T) \implies \mathsf{CC}(T)$. Directly follows from the definitions of $\mathsf{PSI}(T)$ and $\mathsf{CC}(T)$.
- CASE $\mathsf{CC}(T) \implies \mathsf{RA}(T)$. Directly follows from the definitions of $\mathsf{CC}(T)$ and $\mathsf{RA}(T)$.
- CASE $\mathsf{SI}(T) \implies \mathsf{PC}(T)$. Directly follows from the definitions of $\mathsf{SI}(T)$ and $\mathsf{PC}(T)$.
- CASE $\mathsf{PC}(T) \implies \mathsf{CC}(T)$. Follows from the definitions of $\mathsf{PC}(T)$ and $\mathsf{CC}(T)$, along with the fact that $\textsc{Prefix}(T) \implies \textsc{TransVis}(T)$.

□

C MIXED ISOLATION LEVELS AND TRADITIONAL ISOLATION LEVELS

Tables 3 and 4 summarize the traditional isolation levels and their corresponding consistency axioms. The following theorem states that the traditional isolation levels defined in Table 4 are special cases of mixed isolation levels defined in Table 2, where all transactions share the same isolation level.

THEOREM C.1. *For any history $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$ and traditional isolation level ℓ, $(\mathcal{T}, \mathsf{SO}) \models \ell$ iff $\mathcal{H}$ is consistent under mixed isolation level where $\forall T \in \mathcal{T}.\ \mathsf{level}(T) = \ell$.*

PROOF. We only need to show that for any abstract execution $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{VIS}, \mathsf{AR}), \mathcal{X}' \triangleq (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{AR}) \models \textsc{NoConflict}$ if and only if $\forall T \in \mathcal{T}.\ \mathcal{X} \models \textsc{NoConflict}(T)$ under the condition that $\forall T \in \mathcal{T}.\ \mathsf{level}(T) \in \{\mathsf{PSI}, \mathsf{SI}, \mathsf{SER}\}$. Other consistency axioms correspond directly between mixed isolation levels and traditional isolation levels and thus the proof is omitted here.

- The "$\implies$" direction. Suppose that $\mathcal{X}' \models \textsc{NoConflict}$. Consider any transactions $T, T' \in \mathcal{T}$ such that $T' \bowtie T$ and $\textsc{NoConflict}(T)$ holds for T. Now suppose that $T' \xrightarrow{\mathsf{AR}} T$. We need to show that $T' \xrightarrow{\mathsf{VIS}} T$ holds. Suppose by contradiction that
$$\neg(T' \xrightarrow{\mathsf{VIS}} T).$$
Since $\mathcal{X}' \models \textsc{NoConflict}$, we have
$$T' \xrightarrow{\mathsf{VIS}} T \lor T \xrightarrow{\mathsf{VIS}} T'.$$
Therefore, we have
$$T \xrightarrow{\mathsf{VIS}} T'.$$
Since $\mathsf{VIS} \subseteq \mathsf{AR}$, we have
$$T \xrightarrow{\mathsf{AR}} T'.$$
This contradicts the assumption that $T' \xrightarrow{\mathsf{AR}} T$. Hence, $T' \xrightarrow{\mathsf{VIS}} T$ holds, and thus $\mathcal{X} \models \textsc{NoConflict}(T)$.
- The "$\impliedby$" direction. Suppose that $\forall T \in \mathcal{T}.\ \mathcal{X} \models \textsc{NoConflict}(T)$ under the condition that $\forall T \in \mathcal{T}.\ \mathsf{level}(T) \in \{\mathsf{PSI}, \mathsf{SI}, \mathsf{SER}\}$. Consider any transactions $T, T' \in \mathcal{T}$ such that $T' \bowtie T$. We need to show that
$$T' \xrightarrow{\mathsf{VIS}} T \lor T \xrightarrow{\mathsf{VIS}} T'.$$
Suppose by contradiction that
$$\neg(T' \xrightarrow{\mathsf{VIS}} T) \land \neg(T \xrightarrow{\mathsf{VIS}} T').$$
Since $\forall T \in \mathcal{T}.\ \mathcal{X} \models \textsc{NoConflict}(T)$, we have that
$$\mathcal{X} \models \textsc{NoConflict}(T) \land \mathcal{X} \models \textsc{NoConflict}(T').$$
Suppose without loss of generality that $T' \xrightarrow{\mathsf{AR}} T$. Then, by $\mathcal{X} \models \textsc{NoConflict}(T)$, we have
$$T' \xrightarrow{\mathsf{VIS}} T,$$
which contradicts the assumption. Hence, $T' \xrightarrow{\mathsf{VIS}} T \lor T \xrightarrow{\mathsf{VIS}} T'$ holds, and thus $\mathcal{X}' \models \textsc{NoConflict}$.

□

D EQUIVALENCE BETWEEN BOUAJJANI ET AL.'S FRAMEWORK AND OUR FRAMEWORK: THEOREM AND PROOF

Recently Bouajjani et al. [14] also proposed an axiomatic semantics for mixed isolation levels, based on the work of Biswas et al. [13].

Definition D.1 (Histories in [14]). A ***history*** $h = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR})$ is a set $\mathcal{T}$ of transactions along with a strict partial session order $\mathsf{SO} \subseteq \mathcal{T} \times \mathcal{T}$, an allocation function $\mathsf{level} : \mathcal{T} \to \mathcal{L}$, and a write-read relation $\mathsf{WR} : \mathsf{K} \to 2^{\mathcal{T} \times \mathcal{T}}$ such that ($\exists!$ means "unique existence"):

- $\forall x \in \mathsf{K}, S \in \mathcal{T}.\ S \vdash \mathsf{R}(x, \_) \implies \exists!\ T \in \mathcal{T}.\ T \xrightarrow{\mathsf{WR}(x)} S.$
- $\forall x \in \mathsf{K}.\ \forall T, S \in \mathcal{T}.\ T \xrightarrow{\mathsf{WR}(x)} S \implies \exists v \in \mathsf{V}.\ T \neq S \land T \vdash \mathsf{W}(x, v) \land S \vdash \mathsf{W}(x, v).$
- $(\mathsf{SO} \cup \mathsf{WR})$ is acyclic.

For notational simplicity, Bouajjani et al.'s framework makes the following standard assumptions about histories.

- OneWrite: Each transaction contains at most one write operation per key.
- RYW: A read operation preceded by write operations on the same key returns the value written by the last preceding write to this key in the transaction.
- InitTran: Each history contains a special transaction that writes initial value init to all keys. This transaction precedes all the other transactions in SO.

Definition D.2 (Abstract Executions in [14]). An ***abstract execution*** $\xi = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR}, \mathsf{CO})$ is a history $h = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR})$ along with a strict total order $\mathsf{CO} \subseteq \mathcal{T} \times \mathcal{T}$ called commit order such that $(\mathsf{SO} \cup \mathsf{WR}) \subseteq \mathsf{CO}$.

Definition D.3 (Consistency Axioms for Individual Transactions in [14]). Let $\xi = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR}, \mathsf{CO})$ be an abstract execution and $T \in \mathcal{T}$ be a transaction. The ***consistency axioms*** for individual transactions are defined as in Figure 10.

Table 3: Consistency axioms for traditional isolation levels, constraining an abstract execution $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{AR})$ [19, 20].

$\forall T \in \mathcal{T}.\ \forall r, x, v.\ (r = \mathsf{R}(x, v) \land \mathsf{po}_x^{-1}(r) \neq \emptyset) \implies \max_{\mathsf{po}}(\mathsf{po}_x^{-1}(r)) = \_(x, v).$ (Int)		
$\forall T \in \mathcal{T}.\ \forall x, v.\ T \vdash \mathsf{R}(x, v) \implies \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x) \vdash \mathsf{W}(x, v)$ (Ext)		
$\mathsf{SO} \subseteq \mathsf{VIS}$ (Session)	$\mathsf{VIS}\ ;\ \mathsf{VIS} \subseteq \mathsf{VIS}$ (TransVis)	$\mathsf{AR}\ ;\ \mathsf{VIS} \subseteq \mathsf{VIS}$ (Prefix)
$\forall T, T' \in \mathcal{T}.\ T \bowtie T' \implies (T \xrightarrow{\mathsf{VIS}} T' \lor T' \xrightarrow{\mathsf{VIS}} T)$ (NoConflict)	$\mathsf{AR} \subseteq \mathsf{VIS}$ (TotalVis)	

$$\mathsf{ReadAtomic}(T) \equiv \forall x \in \mathsf{K}.\ \forall T_1, T_2 \in \mathcal{T}.\ T_1 \neq T_2 \land T_1 \xrightarrow{\mathsf{WR}(x)} T \land T_2 \in \mathsf{WriteTx}_x \land T_2 \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T \implies T_2 \xrightarrow{\mathsf{CO}} T_1.$$

$$\mathsf{Prefix}(T) \equiv \forall x \in \mathsf{K}.\ \forall T_1, T_2 \in \mathcal{T}.\ T_1 \neq T_2 \land T_1 \xrightarrow{\mathsf{WR}(x)} T \land T_2 \in \mathsf{WriteTx}_x \land T_2 \xrightarrow{\mathsf{CO}^*\ ;\ (\mathsf{SO} \cup \mathsf{WR})} T \implies T_2 \xrightarrow{\mathsf{CO}} T_1.$$

$$\mathsf{Conflict}(T) \equiv \forall x \in \mathsf{K}.\ \forall T_1, T_2, T_3 \in \mathcal{T}.\ T_1 \neq T_2 \land T_1 \xrightarrow{\mathsf{WR}(x)} T \land T_2 \in \mathsf{WriteTx}_x \land T_3 \bowtie T \land T_2 \xrightarrow{\mathsf{CO}^*} T_3 \xrightarrow{\mathsf{CO}} T \implies T_2 \xrightarrow{\mathsf{CO}} T_1.$$

$$\mathsf{SnapshotIsolation}(T) \equiv \mathsf{Prefix}(T) \land \mathsf{Conflict}(T).$$

$$\mathsf{Serializability}(T) \equiv \forall x \in \mathsf{K}.\ \forall T_1, T_2 \in \mathcal{T}.\ T_1 \neq T_2 \land T_1 \xrightarrow{\mathsf{WR}(x)} T \land T_2 \in \mathsf{WriteTx}_x \land T_2 \xrightarrow{\mathsf{CO}} T \implies T_2 \xrightarrow{\mathsf{CO}} T_1.$$

Figure 10: Consistency axioms for individual transactions in the framework of Bouajjani et al. [14].

Table 4: Traditional isolation levels defined by consistency axioms in Table 3.

$\mathsf{RA} \equiv$ Int $\land$ Ext $\land$ Session	$\mathsf{CC} \equiv \mathsf{RA} \land$ TransVis
$\mathsf{PC} \equiv \mathsf{RA} \land$ Prefix	$\mathsf{PSI} \equiv \mathsf{RA} \land$ TransVis $\land$ NoConflict
$\mathsf{SI} \equiv \mathsf{RA} \land$ Prefix $\land$ NoConflict	$\mathsf{SER} \equiv \mathsf{RA} \land$ TotalVis

Read Atomic ($\mathsf{ReadAtomic}(T)$), Prefix ($\mathsf{Prefix}(T)$), and Serializability ($\mathsf{Serializability}(T)$) axioms are defined using their homonymous axioms, and Snapshot Isolation ($\mathsf{SnapshotIsolation}(T)$) is defined as a conjunction of $\mathsf{Prefix}(T)$ and $\mathsf{Conflict}(T)$. Note that the framework of Bouajjani et al. does not include the $\mathsf{Causal}(T)$ axiom and, consequently, does not capture the CC or PSI isolation levels. Therefore, in the following equivalence theorem, we only consider the isolation levels in $\{\mathsf{RA}, \mathsf{PC}, \mathsf{SI}, \mathsf{SER}\}$.

The notion of "consistent abstract executions" and "consistent histories" are defined in the same way as in our framework; see Definition 3.5.

Definition D.4 (Consistent Histories in [14]). An *abstract execution* $\xi = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR}, \mathsf{CO})$ is called ***consistent*** if for each transaction $T \in \mathcal{T}$, the consistency axioms corresponding to its isolation level $\mathsf{level}(T)$ hold on T.

A *history* h is called ***consistent*** if there exists a consistent abstract execution $\xi = (h, \mathsf{WR}, \mathsf{CO})$ for h.

The equivalence between our framework and that of Bouajjani et al. [14] is established as follows.

Theorem D.5. *Let $h = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR})$ be a history in Bouajjani et al.'s framework and $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$ be the corresponding history in our framework. Then, h is consistent iff $\mathcal{H}$ is consistent.*

Proof of Theorem D.5. Suppose that $h = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR})$ is a history in Bouajjani et al.'s framework, and $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$ is the corresponding history in our framework.

On the one hand, we show that if h is consistent, then $\mathcal{H}$ is consistent. Since h is consistent, by Definition D.4, there exists an abstract execution $\xi = (\mathcal{T}, \mathsf{SO}, \mathsf{level}, \mathsf{WR}, \mathsf{CO})$ such that for every transaction $T \in \mathcal{T}$, the consistency axioms corresponding to its isolation level $\mathsf{level}(T)$ hold on T. By Definition 3.5, we need to construct an abstract execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ such that for every transaction $T \in \mathcal{T}$, the consistency axioms corresponding to its isolation level $\mathsf{level}(T)$ in our framework hold on T. To this end, we define $\mathsf{AR} = \mathsf{CO}$ and VIS as in Figure 11.

For $\mathcal{X}$ to be a valid abstract execution, we need to show that

- AR is a strict total order. This holds since $\mathsf{AR} = \mathsf{CO}$, which is a strict total order by Definition D.2.
- $\mathsf{VIS} \subseteq \mathsf{AR}$. We distinguish cases based on the isolation level of T and its construction of VIS, and show that $\mathsf{VIS} \subseteq \mathsf{AR}$ holds in each case.
  - Case $\mathsf{level}(T) = \mathsf{RA}$. By the construction of VIS for $\mathsf{level}(T) = \mathsf{RA}$ in Figure 11, we need to show that
    $$(\mathsf{SO} \cup \mathsf{WR}) \subseteq \mathsf{AR}.$$
    This holds due to $(\mathsf{SO} \cup \mathsf{WR}) \subseteq \mathsf{CO}$ (Definition D.2) and the fact that $\mathsf{AR} = \mathsf{CO}$.
  - Case $\mathsf{level}(T) = \mathsf{PC}$. By the construction of VIS for $\mathsf{level}(T) = \mathsf{PC}$ in Figure 11, we need to show that
    $$\mathsf{CO}^*\ ;\ (\mathsf{SO} \cup \mathsf{WR}) \subseteq \mathsf{AR}.$$
    This holds since $(\mathsf{SO} \cup \mathsf{WR}) \subseteq \mathsf{AR}$ by Case $\mathsf{level}(T) = \mathsf{RA}$ and $\mathsf{AR} = \mathsf{CO}$ is a strict total order.
  - Case $\mathsf{level}(T) = \mathsf{SI}$. By the construction of VIS for $\mathsf{level}(T) = \mathsf{SI}$ in Figure 11, we need to show that
    $$\mathsf{CO}^*\ ;\ (\mathsf{SO} \cup \mathsf{WR} \cup \{(T_1, T_2) \mid T_1 \xrightarrow{\mathsf{CO}} T_2 \land T_1 \bowtie T_2\}) \subseteq \mathsf{AR}.$$
    This holds since $(\mathsf{SO} \cup \mathsf{WR}) \subseteq \mathsf{AR}$ by Case $\mathsf{level}(T) = \mathsf{RA}$ and $\mathsf{AR} = \mathsf{CO}$ is a strict total order.
  - Case $\mathsf{level}(T) = \mathsf{SER}$. By the construction of VIS for $\mathsf{level}(T) = \mathsf{SER}$ in Figure 11, we need to show that
    $$\mathsf{CO} \subseteq \mathsf{AR}.$$

$$\forall S,T \in \mathcal{T}.\ S \xrightarrow{\mathsf{VIS}} T \iff$$

$$(\mathsf{level}(T) = \mathsf{RA} \land S \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T) \lor \tag{1}$$

$$(\mathsf{level}(T) = \mathsf{PC} \land S \xrightarrow{\mathsf{CO}^* \,;\, (\mathsf{SO} \cup \mathsf{WR})} T) \lor \tag{2}$$

$$(\mathsf{level}(T) = \mathsf{SI} \land S \xrightarrow{\mathsf{CO}^* \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \{(T_1,T_2) \mid T_1 \xrightarrow{\mathsf{CO}} T_2 \land T_1 \bowtie T_2\})} T) \lor \tag{3}$$

$$(\mathsf{level}(T) = \mathsf{SER} \land S \xrightarrow{\mathsf{CO}} T). \tag{4}$$

Figure 11: Definition of VIS in terms of SO, WR, and CO in the framework of Bouajjani et al. [14].

This holds since $\mathsf{AR} = \mathsf{CO}$.

- VIS is acyclic. This holds since $\mathsf{VIS} \subseteq \mathsf{AR}$ and AR is acyclic.

In the following, we show that for every transaction $T \in \mathcal{T}$, the consistency axioms of our framework corresponding to its isolation level $\mathsf{level}(T)$ hold on T.

- Case $\mathsf{level}(T) = \mathsf{RA}$. Since ξ is consistent, $\mathsf{ReadAtomic}(T)$ holds.
  - Int(T). Consider any internal read $r = \mathsf{R}(x, v)$ in T for some x and v. If r does not exists, Int(T) holds trivially. Otherwise, let $o \triangleq \max_{\mathsf{po}}(\mathsf{po}_x^{-1}(r))$ be the last operation on x before r in T. We show that $o = \_(x, v)$, by distinguishing two cases depending on the type of o:
    - o is a write operation. By the RYW assumption about histories, r reads the value written by o. Thus, $o = \mathsf{W}(x, v)$.
    - o is a read operation. By the choise of o, there is no write operation on x between o and r in T. We then distinguish two sub-cases depending on whether there is a write operation on x in T *before* o. First, suppose there is such a write operation, then by OneWrite and RYW, both o and r read the value written by that write operation, Since $r = \mathsf{R}(x, v)$, we have $o = \mathsf{R}(x, v)$ as well. Second, suppose there is no such a write operation, we show that both o and r must read from the same transaction. Suppose not, i.e., there exists transactions $T_1 \neq T_2$ such that $T_1 \xrightarrow{\mathsf{WR}(x)} T$ and $T_2 \xrightarrow{\mathsf{WR}(x)} T$. By $\mathsf{ReadAtomic}(T)$, we have $T_1 \xrightarrow{\mathsf{CO}} T_2 \land T_2 \xrightarrow{\mathsf{CO}} T_1$. Therefore, CO is cyclic, contradicting the fact that CO is a strict total order. Thus, there exists a unique transaction T' such that both o and r read from T'. By OneWrite for T' and $r = \mathsf{R}(x, v)$, we have $o = \mathsf{R}(x, v)$.
  - Ext(T). Consider any external read $r = \mathsf{R}(x, v)$ in T for some x and v. If r does not exists, Ext(T) holds trivially. Otherwise, let T_1 be the unique transaction such that
  $$T_1 \vdash \mathsf{W}(x, v) \land T_1 \xrightarrow{\mathsf{WR}(x)} T.$$
  Hence, by the construction of VIS for $\mathsf{level}(T) = \mathsf{RA}$,
  $$T_1 \in (\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$
  Consider any $T_2 \in \mathcal{T}$ such that
  $$T_1 \neq T_2 \land T_2 \in (\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$
  That is,
  $$T_1 \neq T_2 \land T_2 \in \mathsf{WriteTx}_x \land T_2 \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T.$$
  By $\mathsf{ReadAtomic}(T)$, we have
  $$T_2 \xrightarrow{\mathsf{CO}} T_1.$$
  That is,
  $$T_2 \xrightarrow{\mathsf{AR}} T_1.$$
  Therefore,
  $$T_1 = \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$
  Hence, Ext(T) holds.
  - Session(T). Consider any $T' \in \mathcal{T}$ such that $T' \xrightarrow{\mathsf{SO}} T$. By construction of VIS for $\mathsf{level}(T) = \mathsf{RA}$, we have $T' \xrightarrow{\mathsf{VIS}} T$.
- Case $\mathsf{level}(T) = \mathsf{PC}$. Int$(T)$, Ext$(T)$, and Session$(T)$ hold by similar reasoning as in the case of $\mathsf{level}(T) = \mathsf{RA}$. In the following, we show that Prefix(T) holds. Consider any $T_1, T_2 \in \mathcal{T}$ such that
$$T_1 \xrightarrow{\mathsf{AR}} T_2 \xrightarrow{\mathsf{VIS}} T.$$
By the construction of VIS and AR for $\mathsf{level}(T) = \mathsf{PC}$,
$$T_1 \xrightarrow{\mathsf{CO}} T_2 \xrightarrow{\mathsf{CO}^* \,;\, (\mathsf{SO} \cup \mathsf{WR})} T.$$
That is,
$$\exists T_3 \in \mathcal{T}.\ T_1 \xrightarrow{\mathsf{CO}} T_2 \xrightarrow{\mathsf{CO}^*} T_3 \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T.$$
Since CO is transitive, we have
$$T_1 \xrightarrow{\mathsf{CO}} T_3 \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T.$$
Hence, by the construction of VIS for $\mathsf{level}(T) = \mathsf{PC}$,
$$T_1 \xrightarrow{\mathsf{VIS}} T.$$
Therefore, Prefix(T) holds.

- CASE level(T) = SI. INT(T), EXT(T), SESSION(T), and PREFIX(T) hold by similar reasoning as in the case of level(T) = PC. In the following, we show that NOCONFLICT(T) holds. Consider any $T' \in \mathcal{T}$ such that
$$T' \bowtie T \wedge T' \xrightarrow{\mathsf{AR}} T.$$
By the construction of AR for level(T) = SI,
$$T' \bowtie T \wedge T' \xrightarrow{\mathsf{CO}} T.$$
That is,
$$T' \xrightarrow{\{(T_1,T_2)\ \mid\ T_1 \xrightarrow{\mathsf{CO}} T_2 \wedge T_1 \bowtie T_2\}} T.$$
By the construction of VIS for level(T) = SI,
$$T' \xrightarrow{\mathsf{VIS}} T.$$
Therefore, NOCONFLICT(T) holds.
- CASE level(T) = SER. INT(T), EXT(T), and SESSION(T) hold by similar reasoning as in the case of level(T) = RA. In the following, we show that TOTALVIS(T) holds. Consider any $T' \in \mathcal{T}$ such that
$$T' \xrightarrow{\mathsf{AR}} T.$$
By the construction of AR for level(T) = SER,
$$T' \xrightarrow{\mathsf{CO}} T.$$
By the construction of VIS for level(T) = SER,
$$T' \xrightarrow{\mathsf{VIS}} T.$$
Therefore, TOTALVIS(T) holds.

On the other hand, we show that if $\mathcal{H}$ is consistent, then h is consistent. Since $\mathcal{H}$ is consistent, by Definition 3.5, there exists an abstract execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ such that for every transaction $T \in \mathcal{T}$, the consistency axioms corresponding to its isolation level level(T) in our framework hold on T. By Definition D.4, we need to construct an abstract execution $\xi = (\mathcal{T}, \mathsf{SO}, \text{level}, \mathsf{WR}, \mathsf{CO})$ such that for every transaction $T \in \mathcal{T}$, the consistency axioms corresponding to its isolation level level(T) in Bouajjani et al.'s framework hold on T. To this end, we define CO = AR, which is a strict total order.

By EXT(T), we define WR(x) like:
$$T' \xrightarrow{\mathsf{WR}(x)} T \iff T' = \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$
and then $\mathsf{WR} = \bigcup_{x \in \mathsf{K}} \mathsf{WR}(x)$.

For ξ to be a valid abstract execution, we need to show that
$$(\mathsf{SO} \cup \mathsf{WR}) \subseteq \mathsf{CO}.$$
First, by the definition of WR(x) above, we have
$$\mathsf{WR} \subseteq \mathsf{VIS}.$$
Since $\mathsf{VIS} \subseteq \mathsf{AR}$ and $\mathsf{CO} = \mathsf{AR}$, we have
$$\mathsf{WR} \subseteq \mathsf{CO}.$$
Since is an abstract execution and all consistency levels we consider enforces SO(T), we have
$$\mathsf{SO} \subseteq \mathsf{AR} = \mathsf{CO}.$$
Therefore, we have
$$(\mathsf{SO} \cup \mathsf{WR}) \subseteq \mathsf{CO}.$$
In the following, we show that for every transaction $T \in \mathcal{T}$, the consistency axioms of the framework of Bouajjani et al. corresponding to its isolation level level(T) hold on T.

- CASE level(T) = RA. Since $\mathcal{H}$ is consistent, INT(T), EXT(T), and SESSION(T) hold. We need to show that ReadAtomic(T) holds. Consider any $T_1, T_2 \in \mathcal{T}$ such that for some x,
$$T_1 \neq T_2 \wedge T_1 \xrightarrow{\mathsf{WR}(x)} T \wedge T_2 \in \mathsf{WriteTx}_x \wedge T_2 \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T.$$
By EXT(T) and $T_1 \xrightarrow{\mathsf{WR}(x)} T$, we have
$$T_1 = \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$
By SESSION(T), EXT(T), and $T_2 \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T$, we have
$$T_2 \xrightarrow{\mathsf{VIS}} T.$$
Since $T_2 \in \mathsf{WriteTx}_x$,
$$T_2 \in (\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$
Thus,
$$T_2 \xrightarrow{\mathsf{AR}} T_1.$$
Since CO = AR,
$$T_2 \xrightarrow{\mathsf{CO}} T_1.$$
Thus, ReadAtomic(T) holds.
- CASE level(T) = PC. Since $\mathcal{H}$ is consistent, INT(T), EXT(T), SESSION(T), and PREFIX(T) hold. We need to show that Prefix(T) holds. Consider any $T_1, T_2 \in \mathcal{T}$ such that for some x,
$$T_1 \neq T_2 \wedge T_1 \xrightarrow{\mathsf{WR}(x)} T \wedge T_2 \in \mathsf{WriteTx}_x \wedge \exists T_3.\ T_2 \xrightarrow{\mathsf{CO}^*} T_3 \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T.$$
By EXT(T) and $T_1 \xrightarrow{\mathsf{WR}(x)} T$, we have
$$T_1 = \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$
By SESSION(T), EXT(T), and $T_3 \xrightarrow{\mathsf{SO} \cup \mathsf{WR}} T$,
$$T_3 \xrightarrow{\mathsf{VIS}} T.$$
By PREFIX(T) and CO = AR,
$$T_2 \xrightarrow{\mathsf{CO}} T_3 \xrightarrow{\mathsf{VIS}} T \implies T_2 \xrightarrow{\mathsf{VIS}} T.$$
Therefore,
$$T_2 \xrightarrow{\mathsf{CO}^*} T_3 \xrightarrow{\mathsf{VIS}} T \implies T_2 \xrightarrow{\mathsf{VIS}} T.$$
Since $T_2 \in \mathsf{WriteTx}_x$,
$$T_2 \in (\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$
Therefore,
$$T_2 \xrightarrow{\mathsf{AR}} T_1.$$
Since CO = AR,
$$T_2 \xrightarrow{\mathsf{CO}} T_1.$$
Thus, Prefix(T) holds.

- CASE level(T) = SI. Since $\mathcal{H}$ is consistent, INT(T), EXT(T), SESSION(T), PREFIX(T), and NOCONFLICT(T) hold. We need to show that both Prefix(T) and Conflict(T) hold. Since SI$(T) \equiv$ PC$(T) \land$ NOCONFLICT(T) and we have shown that Prefix(T) holds if PC(T) holds, it suffices to show that Conflict(T) holds. Consider any $T_1, T_2, T_3 \in \mathcal{T}$ such that for some x,

$$T_1 \neq T_2 \land T_1 \xrightarrow{\mathsf{WR}(x)} T \land T_2 \in \mathsf{WriteTx}_x \land T_3 \bowtie T \land T_2 \xrightarrow{\mathsf{CO}^*} T_3 \xrightarrow{\mathsf{CO}} T.$$

  By EXT(T) and $T_1 \xrightarrow{\mathsf{WR}(x)} T$, we have

$$T_1 = \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$

  By NOCONFLICT, $T_3 \bowtie T$, $T_3 \xrightarrow{\mathsf{CO}} T$, and $\mathsf{CO} = \mathsf{AR}$,

$$T_3 \xrightarrow{\mathsf{VIS}} T.$$

  By PREFIX(T) and $\mathsf{CO} = \mathsf{AR}$,

$$T_2 \xrightarrow{\mathsf{CO}} T_3 \xrightarrow{\mathsf{VIS}} T \implies T_2 \xrightarrow{\mathsf{VIS}} T.$$

  Therefore,

$$T_2 \xrightarrow{\mathsf{CO}^*} T_3 \xrightarrow{\mathsf{VIS}} T \implies T_2 \xrightarrow{\mathsf{VIS}} T.$$

  Since $T_2 \in \mathsf{WriteTx}_x$,

$$T_2 \in (\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$

  Therefore,

$$T_2 \xrightarrow{\mathsf{AR}} T_1.$$

  Since $\mathsf{CO} = \mathsf{AR}$,

$$T_2 \xrightarrow{\mathsf{CO}} T_1.$$

  Thus, Conflict(T) holds.
- CASE level(T) = SER. Since $\mathcal{H}$ is consistent, INT(T), EXT(T), SESSION(T), and TOTALVIS(T) hold. We need to show that Serializability(T) holds. Consider any $T_1, T_2 \in \mathcal{T}$ such that for some x,

$$T_1 \neq T_2 \land T_1 \xrightarrow{\mathsf{WR}(x)} T \land T_2 \in \mathsf{WriteTx}_x \land T_2 \xrightarrow{\mathsf{CO}} T.$$

  By EXT(T) and $T_1 \xrightarrow{\mathsf{WR}(x)} T$, we have

$$T_1 = \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$

  By TOTALVIS(T), $T_2 \xrightarrow{\mathsf{CO}} T$, and $\mathsf{CO} = \mathsf{AR}$,

$$T_2 \xrightarrow{\mathsf{VIS}} T.$$

  Since $T_2 \in \mathsf{WriteTx}_x$,

$$T_2 \in (\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x).$$

  Therefore,

$$T_2 \xrightarrow{\mathsf{AR}} T_1.$$

  Since $\mathsf{CO} = \mathsf{AR}$,

$$T_2 \xrightarrow{\mathsf{CO}} T_1.$$

  Thus, Serializability(T) holds.

□

E PROOF OF THEOREM 4.1

PROOF OF THEOREM 4.1. For any history $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$ produced by Algorithm 1, we define AR as follows:

$$\forall T', T \in \mathcal{T}.\ T' \xrightarrow{\mathsf{AR}} T \iff T'.cts < T.cts.$$

Clearly, AR is a strict total order. The visibility relation VIS is defined as

$$\forall T', T \in \mathcal{T}.\ T' \xrightarrow{\mathsf{VIS}} T \iff$$
$$(\mathsf{level}(T) = \mathsf{SI} \land T'.cts < T.sts) \lor$$
$$(\mathsf{level}(T) = \mathsf{SER} \land T'.cts < T.cts).$$

Note that we choose $T'.cts < T.cts$ for SER transactions rather than $T'.cts < T.sts$ (as for PC and SI transactions; see line 2:8) to ensure that SER transactions observe all its AR-predecessors, required by the TOTALVIS axiom.

It is easy to verify that VIS is irreflexive and $\mathsf{VIS} \subseteq \mathsf{AR}$:

- VIS is irreflexive: For any transaction $T \in \mathcal{T}$, by definition of VIS, $T \xrightarrow{\mathsf{VIS}} T$ implies either

$$\mathsf{level}(T) = \mathsf{SI} \land T.cts < T.sts,$$

  or

$$\mathsf{level}(T) = \mathsf{SER} \land T.cts < T.cts,$$

  which is impossible in both cases.
- $\mathsf{VIS} \subseteq \mathsf{AR}$: Consider any transactions $T', T \in \mathcal{T}$ such that $T' \xrightarrow{\mathsf{VIS}} T$. By definition of VIS, we have either

$$\mathsf{level}(T) = \mathsf{SI} \land T'.cts < T.sts,$$

  or

$$\mathsf{level}(T) = \mathsf{SER} \land T'.cts < T.cts.$$

  In both cases, we have (due to $T.sts < T.cts$)

$$T'.cts < T.cts.$$

  Therefore, by definition of AR,

$$T' \xrightarrow{\mathsf{AR}} T.$$

In the following, we show that the corresponding abstract execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ for $\mathcal{H}$ is consistent: Fix a transaction $T \in \mathcal{T}$. We show that the consistency axioms corresponding to level(T) hold on T.

- INT(T): If there is no *internal* read in T, then INT(T) holds vacuously. Otherwise, INT(T) holds because T, whether at SI or SER level, buffers writes in its private *buffer* (line 1:7) and reads from *buffer* for internal reads (line 1:10).
- EXT(T): If there is no *external* read in T, then EXT(T) holds vacuously. Otherwise, consider any external read $r \triangleq \mathsf{R}(x, v)$ in T for some $x \in \mathsf{K}$ and $v \in \mathsf{V}$. Let $S \triangleq \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x)$. We distinguish two cases depending on whether T is at SI or SER level, and show that $S \vdash \mathsf{W}(x, v)$.

- Case I: level(T) = SI. By definition of VIS for SI transactions,

  $$\text{VIS}^{-1}(T) = \{T' \in \mathcal{T} \mid T'.cts < T.sts\}.$$

  By line 1:12, v is the value written to x by the last transaction, according to AR, in

  $$V(r) \triangleq \{T' \in \mathcal{T} \mid T'.cts < T.sts\},$$

  Since $V(r) = \text{VIS}^{-1}(T)$, by definition of S,

  $$S = \max_{\text{AR}}(V(r) \cap \text{WriteTx}_x).$$

  That is, S is the last transaction in $V(r)$ that writes to x according to AR. Therefore, $S \vdash \text{W}(x, v)$ holds.
- Case II: level(T) = SER. By definition of VIS for SER transactions,

  $$\text{VIS}^{-1}(T) = \{T' \in \mathcal{T} \mid T'.cts < T.cts\}.$$

  By line 1:15, v is the value written to x by the last transaction, according to AR, in

  $$V(r) \triangleq \{T' \in \mathcal{T} \mid T'.cts < r.\text{now}()\},$$

  where r.now() is the time when r is performed (line 1:15). In the following, we show that

  $$V(r) \cap \text{WriteTx}_x = \text{VIS}^{-1}(T) \cap \text{WriteTx}_x.$$

  On the one hand, since $r.\text{now}() < T.cts$,

  $$V(r) \subseteq \text{VIS}^{-1}(T).$$

  Hence,

  $$V(r) \cap \text{WriteTx}_x \subseteq \text{VIS}^{-1}(T) \cap \text{WriteTx}_x.$$

  On the other hand, suppose by contradiction that

  $$\text{VIS}^{-1}(T) \cap \text{WriteTx}_x \nsubseteq V(r) \cap \text{WriteTx}_x.$$

  Then there exists a transaction $T' \in \mathcal{T}$ such that

  $$T' \in \text{WriteTx}_x \land T'.cts \in (r.\text{now}(), T.cts).$$

  By line 1:14, T acquires a shared lock on x before r.now(). By line 1:18 for SI transactions or line 1:6 for SER transactions, T' acquires an exclusive lock on x. Furthermore, both T and T' hold these locks until they commits (line 1:25), particularly after they are assigned commit timestamps (line 1:19). Since shared and exclusive locks on the same key are incompatible, it cannot be that $T'.cts \in (r.\text{now}(), T.cts)$. Therefore,

  $$\text{VIS}^{-1}(T) \cap \text{WriteTx}_x \subseteq V(r) \cap \text{WriteTx}_x.$$

- Session(T): Consider any transaction $T' \in \mathcal{T}$ such that $T' \xrightarrow{\text{SO}} T$. By definition of SO, T starts after T' commits.
  - Case I: level(T) = SI. Since T starts after T' commits and the wall time always increases, we have

    $$T'.cts < T.sts.$$

    By definition of VIS for SI transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

  - Case II: level(T) = SER. Since T starts after T' commits, T commits after it starts, and the wall time always increases, we have

    $$T'.cts < T.cts.$$

    By definition of VIS for SER transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

- Prefix(T): We need to show that for any transactions $T', S \in \mathcal{T}$, the following holds:

  $$T' \xrightarrow{\text{AR}} S \xrightarrow{\text{VIS}} T \implies T' \xrightarrow{\text{VIS}} T.$$

  Since $T' \xrightarrow{\text{AR}} S$, by definition of AR,

  $$T'.cts < S.cts.$$

  In the following, we consider two cases depending on whether T is at SI or SER level.
  - Case I: level(T) = SI. Since $S \xrightarrow{\text{VIS}} T$, by definition of VIS for SI transactions,

    $$S.cts < T.sts.$$

    Thus,

    $$T'.cts < S.cts < T.sts.$$

    Therefore, by definition of VIS for SI transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

  - Case II: level(T) = SER. Since $S \xrightarrow{\text{VIS}} T$, by definition of VIS for SER transactions,

    $$S.cts < T.cts.$$

    Thus,

    $$T'.cts < S.cts < T.cts.$$

    Therefore, by definition of VIS for SER transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

- NoConflict(T): Consider a transaction $T' \neq T$ such that $T \bowtie T'$. Suppose that $T' \xrightarrow{\text{AR}} T$. By definition of AR,

  $$T'.cts < T.cts.$$

  In the following, we consider two cases depending on whether T is at SI or SER level, and show that $T' \xrightarrow{\text{VIS}} T$.
  - Case I: level(T) = SI. If $T.sts < T'.cts$, then we have

    $$T'.cts \in (T.sts, T.cts) \land T' \bowtie T.$$

    By line 1:21, T should be aborted. Therefore,

    $$T'.cts < T.sts.$$

    By definition of VIS for SI transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

  - Case II: level(T) = SER. By definition of VIS for SER transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

Algorithm 2 The PC-SI-SER protocol

```
    store and $T.buffer$ as in Algorithm 1
 1: procedure START($T$)
 2:     $T.sts \leftarrow$ now()                              ▷ also for SER transactions
 3: procedure WRITE($T, k, v$)
 4:     $T.buffer[k] \leftarrow v$
 5: procedure READ($T, k$)
 6:     if $k \in$ dom($T.buffer$)
 7:         return $T.buffer[k]$
 8:     return value of store[$k$] at latest timestamp $< T.sts$
 9: procedure COMMIT($T$)
10:     $T.cts \leftarrow$ now()
11:     if level($T$) = SI $\wedge \exists T'.\ T'.cts \in (T.sts, T.cts) \wedge T' \bowtie T$
12:         return aborted
13:     if level($T$) = SER $\wedge \exists T'.\ T'.cts \in (T.sts, T.cts) \wedge (T \bowtie T' \vee T \triangleleft T')$
14:         return aborted
15:     store[$k$] $\leftarrow [T.cts \mapsto v], \forall [k \mapsto v] \in T.buffer$
16:     return committed
```

- TOTALVIS(T): Note that in this case, level(T) = SER. Consider a transaction $T' \in \mathcal{T}$ such that
$$T' \xrightarrow{\mathsf{AR}} T.$$
By definition of AR,
$$T'.cts < T.cts.$$
By definition of VIS for SER transactions,
$$T' \xrightarrow{\mathsf{VIS}} T.$$

□

Example E.1. We show that if the procedure COMMIT(T) was not executed atomically and line 1:19 was moved before line 1:17, then Non-RepeatableRead anomaly may arise.

Consider the scenario where two transactions T and T' with level(T) = level(T') = SI operate on the same key k with initial value v_0. Transaction T is a writer and T' is a reader.

(1) T executes COMMIT. Because the commit procedure assigns $T.cts$ before acquiring locks (in the modified algorithm), T immediately sets its commit timestamp to t. The write to k has not been installed into store yet.
(2) T' begins after t. T' receives a start timestamp s' with $s' > t$ and performs its first read of k. Since T's write has not yet been installed, T' reads the old version (value v_0).
(3) T acquires $xlock(k)$ in COMMIT. T continues its commit procedure, acquiring an exclusive lock on k and installing the new value v_t into store with its commit time t.
(4) T' reads k again. T' now reads the current version of k and sees v_t. Thus, T' has read two different values (v_0 and v_t) for the same key in the same transaction, violating the repeatable read guarantee.

F OUR PC-SI-SER PROTOCOL

Algorithm 2 presents our newly proposed concurrency control protocol that supports clients choosing PC, SI, or SER isolation level for each transaction. It is an adaptation of the well-known centralized SI protocol [1, 10].

When a transaction T starts, it is assigned a start timestamp $T.sts$ (line 2:2). The writes are buffered in $T.buffer$ (line 2:4). The transaction T reads data from its own write buffer for internal reads (line 2:7), and from the snapshot of committed transactions as of its start time $T.sts$ for external reads (line 2:8). When T is ready to commit, it is assigned a commit timestamp $T.cts$ (line 2:10). If T is an SI or SER transaction, and any other already committed concurrent transaction T' has written on keys that it intends to write (line 2:11 and line 2:13), T will be aborted. This prevents the "lost updates" anomaly disallowed by both SI and SER, but allowed by PC. Additionally, if T is an SER transaction, and any other already committed concurrent transaction T' has written on keys that it intends to read (line 2:13), it will also be aborted.

THEOREM F.1. *Algorithm 2 satisfies* PC-SI-SER.

PROOF OF THEOREM F.1. For any history $\mathcal{H} = (\mathcal{T}, \mathsf{SO}, \mathsf{level})$ produced by Algorithm 2, we define AR as follows:
$$\forall T', T \in \mathcal{T}.\ T' \xrightarrow{\mathsf{AR}} T \iff T'.cts < T.cts.$$
Clearly, AR is a strict total order. The visibility relation VIS is defined as
$$\forall T', T \in \mathcal{T}.\ T' \xrightarrow{\mathsf{VIS}} T \iff$$
$$(\mathsf{level}(T) = \mathsf{PC} \wedge T'.cts < T.sts) \vee$$
$$(\mathsf{level}(T) = \mathsf{SI} \wedge T'.cts < T.sts) \vee$$
$$(\mathsf{level}(T) = \mathsf{SER} \wedge T'.cts < T.cts).$$
It is easy to verify that VIS is irreflexive and VIS $\subseteq$ AR. [26] In the following, we show that the corresponding abstract execution $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{AR})$ for $\mathcal{H}$ is consistent: Fix a transaction $T \in \mathcal{T}$. We show that the consistency axioms corresponding to level(T) hold on T.

- INT(T): If there is no *internal* read in T, then INT(T) holds vacuously. Otherwise, INT(T) holds because T buffers writes in its private *buffer* (line 2:4) and reads from *buffer* for internal reads (line 2:7).
- EXT(T): If there is no *external* read in T, then EXT(T) holds vacuously. Otherwise, consider any external read $r \triangleq \mathsf{R}(x, v)$ in T for some $x \in \mathsf{K}$ and $v \in \mathsf{V}$. By line 2:8, v is the value written to x by the last transaction, according to AR, in
$$V(r) \triangleq \{T' \in \mathcal{T} \mid T'.cts < T.sts\}.$$
Let $S \triangleq \max_{\mathsf{AR}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x)$. We distinguish three cases depending on whether T is at PC, SI, or SER level, and show that $S \vdash \mathsf{W}(x, v)$.
  - CASE I: level(T) = PC. By definition of VIS for PC transactions,
  $$\mathsf{VIS}^{-1}(T) = \{T' \in \mathcal{T} \mid T'.cts < T.sts\}.$$
  Since $V(r) = \mathsf{VIS}^{-1}(T)$, by definition of S,
  $$S = \max_{\mathsf{AR}}(V(r) \cap \mathsf{WriteTx}_x).$$
  That is, S is the last transaction in $V(r)$ that writes to x according to AR. Therefore, $S \vdash \mathsf{W}(x, v)$ holds.

[26] The proof is similar to that in the proof of Theorem 4.1. We omit the details for brevity.

- Case II: $\text{level}(T) = \text{SI}$. The proof is the same as that of Case I for PC transactions.
- Case III: $\text{level}(T) = \text{SER}$. By definition of VIS for SER transactions,

  $$\text{VIS}^{-1}(T) = \{T' \in \mathcal{T} \mid T'.cts < T.cts\}.$$

  In the following, we show that

  $$V(r) \cap \text{WriteTx}_x = \text{VIS}^{-1}(T) \cap \text{WriteTx}_x.$$

  On the one hand, since $T.sts < T.cts$,

  $$V(r) \subseteq \text{VIS}^{-1}(T).$$

  Hence,

  $$V(r) \cap \text{WriteTx}_x \subseteq \text{VIS}^{-1}(T) \cap \text{WriteTx}_x.$$

  On the other hand, suppose by contradiction that

  $$\text{VIS}^{-1}(T) \cap \text{WriteTx}_x \nsubseteq V(r) \cap \text{WriteTx}_x.$$

  Then there exists a transaction $T' \in \mathcal{T}$ such that

  $$T' \in \text{WriteTx}_x \land T'.cts \in (T.sts, T.cts).$$

  Since $T \vdash \text{R}(x, v)$, we have $T \triangleleft T'$. By line 2:13, T would abort, which is a contradiction. Therefore,

  $$\text{VIS}^{-1}(T) \cap \text{WriteTx}_x \subseteq V(r) \cap \text{WriteTx}_x.$$

- Session(T): Consider any transaction $T' \in \mathcal{T}$ such that $T' \xrightarrow{\text{SO}} T$. By definition of SO, T starts after T' commits.
  - Case I: $\text{level}(T) = \text{PC}$. Since T starts after T' commits and the wall time always increases, we have

    $$T'.cts < T.sts.$$

    By definition of VIS for PC transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

  - Case II: $\text{level}(T) = \text{SI}$. The proof is the same as that of Case I for PC transactions.
  - Case III: $\text{level}(T) = \text{SER}$. Since T starts after T' commits, T commits after it starts, and the wall time always increases, we have

    $$T'.cts < T.cts.$$

    By definition of VIS for SER transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

- Prefix(T): We need to show that for any transactions $T', S \in \mathcal{T}$, the following holds:

  $$T' \xrightarrow{\text{AR}} S \xrightarrow{\text{VIS}} T \implies T' \xrightarrow{\text{VIS}} T.$$

  Since $T' \xrightarrow{\text{AR}} S$, by definition of AR,

  $$T'.cts < S.cts.$$

  In the following, we consider three cases depending on whether T is at PC, SI, or SER level.
  - Case I: $\text{level}(T) = \text{PC}$. Since $S \xrightarrow{\text{VIS}} T$, by definition of VIS for PC transactions,

    $$S.cts < T.sts.$$

    Thus,

    $$T'.cts < S.cts < T.sts.$$

    Therefore, by definition of VIS for PC transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

  - Case II: $\text{level}(T) = \text{SI}$. The proof is the same as that of Case I for PC transactions.
  - Case III: $\text{level}(T) = \text{SER}$. Since $S \xrightarrow{\text{VIS}} T$, by definition of VIS for SER transactions,

    $$S.cts < T.cts.$$

    Thus,

    $$T'.cts < S.cts < T.cts.$$

    Therefore, by definition of VIS for SER transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

- NoConflict(T): Note that in this case, $\text{level}(T) = \text{SI}$ or $\text{level}(T) = \text{SER}$. Consider a transaction $T' \neq T$ such that $T \bowtie T'$. Suppose that $T' \xrightarrow{\text{AR}} T$. By definition of AR,

  $$T'.cts < T.cts.$$

  In the following, we consider two cases depending on whether T is at SI or SER level, and show that $T' \xrightarrow{\text{VIS}} T$.
  - Case I: $\text{level}(T) = \text{SI}$. If $T.sts < T'.cts$, then we have

    $$T'.cts \in (T.sts, T.cts) \land T' \bowtie T.$$

    By line 2:11, T should be aborted. Therefore,

    $$T'.cts < T.sts.$$

    By definition of VIS for SI transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

  - Case II: $\text{level}(T) = \text{SER}$. By definition of VIS for SER transactions,

    $$T' \xrightarrow{\text{VIS}} T.$$

- TotalVis: Note that in this case, $\text{level}(T) = \text{SER}$. Consider a transaction $T' \in \mathcal{T}$ such that

  $$T' \xrightarrow{\text{AR}} T.$$

  By definition of AR,

  $$T'.cts < T.cts.$$

  By definition of VIS for SER transactions,

  $$T' \xrightarrow{\text{VIS}} T.$$

□

G PROOF OF THEOREM 6.5

Proof of Theorem 6.5. We prove each relation separately:

(R1) The goal is $T_1[\_] \xrightarrow{\mathsf{SO}} T_2[\_] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$. According to Table 2, Session(T_2) holds for T_2, regardless of its isolation level. By the definition of Session(T) and $T_1 \xrightarrow{\mathsf{SO}} T_2$, we have $T_1 \xrightarrow{\mathsf{VIS}} T_2$.

(R2) The goal is $T_1[\_] \xrightarrow{\mathsf{WR}(x)} T_2[\_] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$. By the definition of WR (Definition 6.1).

(R3) The goal is $T_1[\_] \xrightarrow{\mathsf{WW}(x)} T_2[\mathsf{SER/SI/PSI}] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$.
Suppose that $T_1[\_] \xrightarrow{\mathsf{WW}(x)} T_2[\mathsf{SER/SI/PSI}] \in \mathcal{G}$. By the definition of WW (Definition 6.1), we have

$$T_1 \xrightarrow{\mathsf{AR}} T_2 \wedge T_1, T_2 \in \mathsf{WriteTx}_x.$$

Hence,

$$T_1 \xrightarrow{\mathsf{AR}} T_2 \wedge T_1 \bowtie T_2.$$

By NoConflict(T_2) and level(T_2) $\in \{\mathsf{SER}, \mathsf{SI}, \mathsf{PSI}\}$, we have $T_1 \xrightarrow{\mathsf{VIS}} T_2$.

(R4) The goal is $T_1[\mathsf{SER}] \xrightarrow{\mathsf{RW}(x)} T_2[\mathsf{SER}] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{VIS}} T_2 \in \mathcal{X}$.
Suppose that $T_1[\mathsf{SER}] \xrightarrow{\mathsf{RW}(x)} T_2[\mathsf{SER}] \in \mathcal{G}$. We distinguish two cases:

- Case I: $T_1 \xrightarrow{\mathsf{AR}} T_2$. By TotalVis($T_2$) and level($T_2$) = SER, we have $T_1 \xrightarrow{\mathsf{VIS}} T_2$.
- Case II: $T_2 \xrightarrow{\mathsf{AR}} T_1$. By TotalVis($T_1$) and level($T_2$) = SER, we have

$$T_2 \xrightarrow{\mathsf{VIS}} T_1.$$

By the definition of RW (Definition 6.1), there exists $T_3 \in \mathsf{WriteTx}_x$ for some x, such that

$$T_3 \xrightarrow{\mathsf{WR}(x)} T_1 \wedge T_3 \xrightarrow{\mathsf{WW}(x)} T_2.$$

By ((R2)) and the definition of WW (Definition 6.1),

$$T_3 \xrightarrow{\mathsf{VIS}} T_1 \wedge T_3 \xrightarrow{\mathsf{AR}} T_2.$$

Combining with $T_2 \xrightarrow{\mathsf{VIS}} T_1$, we have

$$\{T_2, T_3\} \subseteq \mathsf{VIS}^{-1}(T_1) \wedge T_3 \xrightarrow{\mathsf{AR}} T_2.$$

However, by the definition of WR (Definition 6.1), $T_3 \xrightarrow{\mathsf{WR}(x)} T_1$ cannot hold. Therefore, this case is impossible.

(R5) The goal is $T_1[\_] \xrightarrow{\mathsf{WW}(x)} T_2[\mathsf{PC/CC/RA}] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{AR}} T_2 \in \mathcal{X}$.
By the definition of WW.

(R6) The goal is $T_1[\mathsf{SER}] \xrightarrow{\mathsf{RW}(x)} T_2[\mathsf{SI/PSI/PC/CC/RA}] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{AR}} T_2 \in \mathcal{X}$.
Suppose that $T_1[\mathsf{SER}] \xrightarrow{\mathsf{RW}(x)} T_2[\mathsf{SI/PSI/PC/CC/RA}] \in \mathcal{G}$. By the same argument as in Case II of (R4), the case $T_2 \xrightarrow{\mathsf{AR}} T_1$ is impossible. Since AR is total, we have $T_1 \xrightarrow{\mathsf{AR}} T_2$.

(R7) The goal is $T_1[\mathsf{SI/PSI/PC/CC/RA}] \xrightarrow{\mathsf{RW}(x)} T_2[\_] \in \mathcal{G} \implies T_1 \xrightarrow{\mathsf{AR}} T_2 \in \mathcal{X} \vee T_2 \xrightarrow{\mathsf{VIS}} T_1 \in \mathcal{X}$.
Suppose that $T_1[\mathsf{SI/PSI/PC/CC/RA}] \xrightarrow{\mathsf{RW}(x)} T_2[\_] \in \mathcal{G}$. Furthermore, suppose that $T_1 \xrightarrow{\mathsf{AR}} T_2$ does not hold. We need to show that $T_2 \xrightarrow{\mathsf{VIS}} T_1$ holds.
Since AR is total, we have

$$T_2 \xrightarrow{\mathsf{AR}} T_1.$$

Suppose by contradiction that

$$T_2 \xrightarrow{\mathsf{VIS}} T_1.$$

By the same argument as in Case II of (R4), we have a contradiction. Therefore,

$$\neg(T_2 \xrightarrow{\mathsf{VIS}} T_1).$$

Combining with $T_2 \xrightarrow{\mathsf{AR}} T_1$, we have

$$T_2 \xrightarrow{\mathsf{VIS}} T_1.$$

□

H PROOF OF THEOREM 6.6

Proof of Theorem 6.6. Suppose that $\mathcal{G}$ contains a cycle

$$\pi = T_1 \xrightarrow{\lambda^*} T_n = T_1.$$

Let T_3 be the AR-minimal transaction among $T_1, \ldots, T_n$; such a transaction must exist since AR is a total order. Let T_2 be the predecessor of T_3 in π. That is (note that T_3 can be T_1),

$$\pi = T_1 \xrightarrow{\lambda} T_2 \xrightarrow{\lambda} T_3 \xrightarrow{\lambda^*} T_1.$$

Since T_3 is AR-minimal,

$$\neg(T_2 \xrightarrow{\mathsf{AR}} T_3).$$

By Theorem 6.5,

$$T_2 \xrightarrow{\mathsf{VIS}\text{-implicit} \,\cup\, \mathsf{AR}\text{-implicit}} T_3 \notin \mathcal{G}.$$

Since Theorem 6.4 is exhaustive, $T_2 \xrightarrow{\lambda} T_3 \in \pi$ must be a critical edge. Therefore, π is of the form

$$\pi = T_1[\_] \xrightarrow{\lambda} T_2[\mathsf{SI/PSI/PC/CC/RA}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_],$$

with T_3 as the AR-minimal transaction in the cycle.

Now we show that T_2 is not a single-key read-only transaction. Suppose by contradiction that T_2 is a single-key (say, on k) read-only transaction. Then, π must be (note that in this case T_3 cannot be T_1)

$$\pi = T_1[\_] \xrightarrow{\mathsf{WR}(k)} T_2[\mathsf{SI/PSI/PC/CC/RA}] \xrightarrow{\mathsf{RW}(k)} T_3[\_] \xrightarrow{\lambda^*} T_1[\_].$$

By definition of RW, we have

$$T_1 \xrightarrow{\mathsf{WW}(k)} T_3.$$

By Theorem 6.5 (R5),

$$T_1 \xrightarrow{\mathsf{AR}} T_3.$$

Then, T_3 cannot the AR-minimal transaction in π, contradicting the choice of T_3. Therefore, T_2 is not a single-key read-only transaction. □

I PROOF OF THEOREM 6.7

Proof of Theorem 6.7. The " $\Longrightarrow$ " direction is obvious. For the " $\Longleftarrow$ " direction, we prove by contraposition. Suppose that $\mathcal{G}$ contains a cycle π. If π does not satisfy (C1), we can easily build one that does. Therefore, in the following, we assume that π is simple and chord-free, satisfying (C1).

By Theorem 6.6, π is of the form

$$\pi = T_1[\_] \xrightarrow{\lambda} T_2[\mathsf{SI}/\mathsf{PSI}/\mathsf{PC}/\mathsf{CC}/\mathsf{RA}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_],$$

with T_3 as the AR-minimal transaction in π. Particularly,

$$T_3 \xrightarrow{\mathsf{AR}} T_1 \wedge T_3 \xrightarrow{\mathsf{AR}} T_2.$$

Furthermore, by Theorem 6.5 (R4), we have

$$T_3 \xrightarrow{\overline{\mathsf{VIS}}} T_2.$$

In the following, we show that π satisfies both (C2) and (C3).

(C3) Suppose by contradiction that $T_2 \xrightarrow{\mathsf{VIS}\text{-implicit} \cup \mathsf{AR}\text{-implicit}} T_3 \in \mathcal{G}$ holds. By Theorem 6.5, $T_2 \xrightarrow{\mathsf{AR}} T_3$, contradicting $T_3 \xrightarrow{\mathsf{AR}} T_2$. Therefore, (C3) holds.

(C2) We distinguish cases according to level(T_2), and show that π must be one of π_1, π_2, π_3, and π_4.

- Case I: level$(T_2) \in \{\mathsf{RA}, \mathsf{CC}\}$. This is the case of π_1.
- Case II: level$(T_2) = \mathsf{PSI}$. Suppose by contradiction that $T_2 \bowtie T_3$. By NoConflict(T_2) and $T_3 \xrightarrow{\mathsf{AR}} T_2$, we have $T_3 \xrightarrow{\mathsf{VIS}} T_2$, contradicting $T_3 \xrightarrow{\overline{\mathsf{VIS}}} T_2$. Therefore, $\neg(T_2 \bowtie T_3)$ holds. This is the case of π_2.
- Case III: level$(T_2) = \mathsf{PC}$. Suppose by contradiction that $T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \in \mathcal{G}$. By Theorem 6.5, we have $T_1 \xrightarrow{\mathsf{VIS}} T_2$. By Prefix$(T_2)$ and $T_3 \xrightarrow{\mathsf{AR}} T_1$, we have $T_3 \xrightarrow{\mathsf{VIS}} T_2$. This contradicts $T_3 \xrightarrow{\overline{\mathsf{VIS}}} T_2$. Thus, $T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \notin \mathcal{G}$. By Theorem 6.5, $T_1 \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} T_2 \in \pi$. This is the case of π_3.
- Case IV: level$(T_2) = \mathsf{SI}$. Suppose that π is of the form

$$\pi = T_1[\_] \xrightarrow{\lambda} T_2[\mathsf{SI}] \xrightarrow{\mathsf{RW}(y)} T_3 \xrightarrow{\lambda^*} T_1,$$

for some key y. As in the case of level$(T_2) = \mathsf{PSI}$, it holds that

$$\neg(T_2 \bowtie T_3).$$

As in the case of level$(T_2) = \mathsf{PC}$,

$$T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \notin \mathcal{G}.$$

By Theorem 6.5 and level$(T_2) = \mathsf{SI}$,

$$T_1 \xrightarrow{\mathsf{RW}(x)} T_2 \in \pi,$$

for some key x. If $x = y$, then

$$T_2, T_3 \in \mathsf{WriteTx}_x.$$

Therefore, we have

$$T_2 \bowtie T_3,$$

contradicting $\neg(T_2 \bowtie T_3)$. Therefore, $x \neq y$. This is the case of π_4.

□

J PROOF OF THEOREM 6.16

Proof of Theorem 6.16. Consider a workload $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$ such that SDG$(\mathcal{W})$ contains no static critical cycles in the forms (S1)–(S4). We show that $\mathcal{W}$ is statically robust by proving that any abstract execution $\mathcal{X}$ generated by $\mathcal{W}$ is dynamically robust. By Theorem 6.9, it suffices to show that DDG$(\mathcal{X})$ contains no critical cycle satisfying (C1)–(C2).

Suppose by contradiction that DDG$(\mathcal{X})$ contains a critical cycle

$$\pi = T_1[\![P_1]\!] \xrightarrow{\lambda} T_2[\![P_2]\!] \xrightarrow{\mathsf{RW}} T_3[\![P_3]\!] \xrightarrow{\lambda^*} T_1[\![P_1]\!],$$

satisfying (C1)–(C2) of Theorem 6.7. By (C2) of Theorem 6.7, T_2 is not a single-key read-only transaction. Since T_2 is an instantiation of P_2, P_2 is not a single-key read-only program instance.

Suppose by contradiction that $P_2 \xrightarrow{\mathsf{SO}} P_3 \in$ SDG$(\mathcal{W})$ holds. Since T_2 and T_3 are instantiations of P_2 and P_3 respectively, they have the same program order relation. That is, we have $T_2 \xrightarrow{\mathsf{SO}} T_3 \in$ DDG$(\mathcal{X})$. By (R1) of Theorem 6.5, we have $T_2 \xrightarrow{\mathsf{VIS}} T_3$, contradicting (C3) of Theorem 6.7. Therefore, $P_2 \xrightarrow{\mathsf{SO}} P_3 \notin$ SDG$(\mathcal{W})$ holds.

In the following, we consider each form of π in turn, and show that each case leads to a static critical cycle in SDG$(\mathcal{W})$, contradicting the assumption.

- Case I: π_1. In this case, π is of the form

$$\pi_1 = T_1[\_] \xrightarrow{\lambda} T_2[\mathsf{RA}/\mathsf{CC}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_].$$

Since SDG$(\mathcal{W})$ is an over-approximation of DDG$(\mathcal{X})$, we have that SDG$(\mathcal{W})$ contains a static critical cycle of the form (see (S1) of Theorem 6.16)

$$\sigma_1 = P_1[\_] \xrightarrow{\lambda} P_2[\mathsf{RA}/\mathsf{CC}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_].$$

- Case II: π_2. In this case, π is of the form

$$\pi_2 = T_1[\_] \xrightarrow{\lambda} T_2[\mathsf{PSI}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_],$$

where $\neg(T_2 \bowtie T_3)$ holds.
Since SDG$(\mathcal{W})$ is an over-approximation of DDG$(\mathcal{X})$, we have that SDG$(\mathcal{W})$ contains a static critical cycle of the form (see (S2) of Theorem 6.16)

$$\sigma_2 = P_1[\_] \xrightarrow{\lambda} P_2[\mathsf{PSI}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_].$$

Since T_2 and T_3 are instantiations of P_2 and P_3 respectively, T_2 and T_3 have the same read and write sets as P_2 and P_3 respectively. Since $\neg(T_2 \bowtie T_3)$ holds, we have $\neg(P_2 \bowtie P_3)$.
- Case III: π_3. In this case, π is of the form

$$\pi_3 = T_1[\_] \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} T_2[\mathsf{PC}] \xrightarrow{\mathsf{RW}} T_3[\_] \xrightarrow{\lambda^*} T_1[\_],$$

where $T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \notin$ DDG$(\mathcal{X})$.
Since SDG$(\mathcal{W})$ is an over-approximation of DDG$(\mathcal{X})$, we have that SDG$(\mathcal{W})$ contains a static critical cycle of the form (see (S3) of Theorem 6.16)

$$\sigma_3 = P_1[\_] \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} P_2[\mathsf{PC}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_].$$

Suppose by contradiction that $P_1 \xrightarrow{\mathsf{SO}} P_2$ holds. Then, for the instantiations $T_1[\![P_1]\!]$ and $T_2[\![P_2]\!]$, we have $T_1 \xrightarrow{\mathsf{SO}} T_2$. By (R1) of Theorem 6.5, we have $T_1 \xrightarrow{\mathsf{VIS}} T_2$. This contradicts $T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \notin \mathrm{DDG}(\mathcal{X})$. Hence, $\neg(P_1 \xrightarrow{\mathsf{SO}} P_2)$ holds. [27]

- Case IV: π_4. In this case, π is of the form

$$\pi_4 = T_1[\_] \xrightarrow{\mathsf{RW}(x)} T_2[\mathsf{SI}] \xrightarrow{\mathsf{RW}(y)} T_3[\_] \xrightarrow{\lambda^*} T_1[\_],$$

where $x \neq y \land \neg(T_2 \bowtie T_3) \land T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \notin \mathrm{DDG}(\mathcal{X})$. Since $\mathrm{SDG}(\mathcal{W})$ is an over-approximation of $\mathrm{DDG}(\mathcal{X})$, we have that $\mathrm{SDG}(\mathcal{W})$ contains a static critical cycle of the form (see (S4) of Theorem 6.16)

$$\sigma_4 = P_1[\_] \xrightarrow{\mathsf{RW}(x)} P_2[\mathsf{SI}] \xrightarrow{\mathsf{RW}(y)} P_3[\_] \xrightarrow{\lambda^*} P_1[\_],$$

where $x \neq y$ holds.
Since T_2 and T_3 are instantiations of P_2 and P_3 respectively, T_2 and T_3 have the same read and write sets as P_2 and P_3 respectively. Since $\neg(T_2 \bowtie T_3)$ holds, we have $\neg(P_2 \bowtie P_3)$.
Suppose by contradiction that $P_1 \xrightarrow{\mathsf{SO}} P_2$ holds. Then, for the instantiations $T_1[\![P_1]\!]$ and $T_2[\![P_2]\!]$, we have $T_1 \xrightarrow{\mathsf{SO}} T_2$. By (R1) of Theorem 6.5, we have $T_1 \xrightarrow{\mathsf{VIS}} T_2$. This contradicts $T_1 \xrightarrow{\mathsf{VIS}\text{-implicit}} T_2 \notin \mathrm{DDG}(\mathcal{X})$. Hence, $\neg(P_1 \xrightarrow{\mathsf{SO}} P_2)$ holds.

□

K PROOF OF THEOREM 6.18

Proof of Theorem 6.18. Consider a workload $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$ where level is defined by the allocation rules (A1)–(A3). By Theorem 6.16, we prove that $\mathcal{W}$ is statically robust by showing that its static dependency graph $\mathcal{G} = \mathrm{SDG}(\mathcal{W})$ contains no static critical cycles of the forms σ_1 to σ_4.

- Case I: σ_1. Suppose by contradiction that $\mathcal{G}$ contains a static critical cycle

$$\sigma_1 = P_1[\_] \xrightarrow{\lambda} P_2[\mathsf{RA/CC}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_].$$

Since $\mathsf{level}(P_2) = \mathsf{RA}$, by the allocation rule (A1), P_2 is a write-only or single-key read-only program instance. Since $P_2 \xrightarrow{\mathsf{RW}} P_3$, P_2 cannot be write-only. On the other hand, P_2 cannot be single-key read-only either, as required by Theorem 6.16. Therefore, we have a contradiction, and $\mathcal{G}$ contains no such static critical cycle σ_1.

- Case II: σ_2. Suppose by contradiction that $\mathcal{G}$ contains a static critical cycle

$$\sigma_2 = P_1[\_] \xrightarrow{\lambda} P_2[\mathsf{PSI}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_],$$

where $\neg(P_2 \bowtie P_3)$ holds.
Since $\mathsf{level}(P_2) = \mathsf{PSI}$, by the allocation rule (A3), P_2 is a read-write program instance, and for each other program instance $P' \in \mathcal{P}$, we have

$$\neg(P_2 \triangleleft P') \lor (P_2 \bowtie P').$$

Particularly, for $P' = P_3$, we have

$$\neg(P_2 \triangleleft P_3) \lor (P_2 \bowtie P_3).$$

Since $P_2 \xrightarrow{\mathsf{RW}} P_3$, we have

$$P_2 \triangleleft P_3.$$

Therefore, it must be

$$P_2 \bowtie P_3.$$

However, this contradicts the condition of σ_2 that $\neg(P_2 \bowtie P_3)$ holds. Therefore, $\mathcal{G}$ contains no such static critical cycle σ_2.

- Case III: σ_3. Suppose by contradiction that $\mathcal{G}$ contains a static critical cycle

$$\sigma_3 = P_1[\_] \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} P_2[\mathsf{PC}] \xrightarrow{\mathsf{RW}} P_3[\_] \xrightarrow{\lambda^*} P_1[\_],$$

where $\neg(P_1 \xrightarrow{\mathsf{SO}} P_2)$ holds.
Since $\mathsf{level}(P_2) = \mathsf{PC}$, by the allocation rule (A2), P_2 is a multi-key read-only program instance. However, since $P_1 \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} P_2$, P_2 cannot be a read-only program instance. Therefore, we have a contradiction, and $\mathcal{G}$ contains no such static critical cycle σ_3.

- Case IV: σ_4. Suppose by contradiction that $\mathcal{G}$ contains a static critical cycle

$$\sigma_4 = P_1[\_] \xrightarrow{\mathsf{RW}(x)} P_2[\mathsf{SI}] \xrightarrow{\mathsf{RW}(y)} P_3[\_] \xrightarrow{\lambda^*} P_1[\_],$$

where $x \neq y \land \neg(P_2 \bowtie P_3) \land \neg(P_1 \xrightarrow{\mathsf{SO}} P_2)$ holds.
However, according to the allocation rules (A1)–(A3), no program instance P is allocated with $\mathsf{level}(P) = \mathsf{SI}$. Therefore, we have a contradiction, and $\mathcal{G}$ contains no such static critical cycle σ_4.

□

L ALGORITHM FOR STATIC ROBUSTNESS VALIDATION

The static robustness validator (Algorithm 3) takes as input a workload $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$, constructs the static dependency graph $\mathcal{G} = \mathrm{SDG}(\mathcal{W})$ (line 3:4), and checks for static critical cycles according to Theorem 6.16 (line 3:10). Specifically, the algorithm iterates over each RW edge $P_2 \xrightarrow{\mathsf{RW}} P_3$ in $\mathcal{G}$ (line 3:6) and checks whether it participates in a static critical cycle of the forms σ_1 (line 3:13), σ_2 (line 3:15), σ_3 (line 3:17), or σ_4 (line 3:21). Any such edge indicates that $\mathcal{W}$ is not statically robust (line 3:8).

The overall time complexity of the algorithm is $O(n^3)$:

$$\underbrace{O(n^2)}_{\textsc{BuildSDG}} + \underbrace{O(n^3)}_{\textsc{TransitiveClosure}} + \underbrace{O(m)}_{\mathsf{RW}\text{ edges iteration}} \cdot \max\Big(\underbrace{O(1)}_{\sigma_1}, \underbrace{O(1)}_{\sigma_2}, \underbrace{O(n)}_{\sigma_3}, \underbrace{O(n)}_{\sigma_4}\Big) = O(n^3),$$

where $n = |\mathcal{P}|$ is the number of program instances and $m = O(n^2)$ is the number of edges in $\mathcal{G}$.

[27]Note that since $P_1 \xrightarrow{\mathsf{WR}} P_2$ does *not* necessarily imply $T_1 \xrightarrow{\mathsf{WR}} T_2$, we cannot enforce $\neg(P_1 \xrightarrow{\mathsf{WR}} P_2)$ in this case.

Algorithm 3 Static robustness validation

```
1: $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$: a workload to verify
2: TC: boolean matrix for reachability on SDG($\mathcal{W}$) among program instances in $\mathcal{P}$
3: procedure VerifyRobustness($\mathcal{W}$)
4:     $\mathcal{G} \leftarrow$ BuildSDG($\mathcal{W}$)
5:     TC $\leftarrow$ TransitiveClosure($\mathcal{G}$)                    ▷ $O(n^3)$ using Floyd-Warshall
6:     for all edge $P_2 \xrightarrow{\mathsf{RW}} P_3$ in $\mathcal{G}$ do
7:         if IsInStaticCriticalCycle($P_2, P_3, \mathcal{G}, \mathsf{TC}, \mathcal{W}$)
8:             return false
9:     return true
10: procedure IsInStaticCriticalCycle($P_2, P_3, \mathcal{G}, \mathsf{TC}, \mathcal{W}$)
11:     if IsSingleVariableReadOnly($P_2$) or $P_2 \xrightarrow{\mathsf{SO}} P_3$
12:         return false                    ▷ $P_2$ is write-only or single-key read-only
13:     if $\mathsf{level}(P_2) \in \{\mathsf{RA}, \mathsf{CC}\}$     ▷ $\sigma_1 : P_1[\_] \xrightarrow{\lambda} P_2[\mathsf{RA/CC}] \xrightarrow{\mathsf{RW}} P_3[\_]$
14:         return $\mathsf{TC}[P_3, P_1]$
15:     else if $\mathsf{level}(P_2) = \mathsf{PSI} \land \neg(P_2 \bowtie P_3)$     ▷ $\sigma_2 : P_1[\_] \xrightarrow{\lambda} P_2[\mathsf{PSI}] \xrightarrow{\mathsf{RW}} P_3[\_]$
16:         return $\mathsf{TC}[P_3, P_1]$
17:     else if $\mathsf{level}(P_2) = \mathsf{PC}$     ▷ $\sigma_3 : P_1[\_] \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} P_2[\mathsf{PC}] \xrightarrow{\mathsf{RW}} P_3[\_]$
18:         for all $P_1 \xrightarrow{\mathsf{WW} \cup \mathsf{RW}} P_2 \land \neg(P_1 \xrightarrow{\mathsf{SO}} P_2)$ do
19:             if $\mathsf{TC}[P_3, P_1]$
20:                 return true
21:     else if $\mathsf{level}(P_2) = \mathsf{SI}$     ▷ $\sigma_4 : P_1[\_] \xrightarrow{\mathsf{RW}} P_2[\mathsf{SI}] \xrightarrow{\mathsf{RW}} P_3[\_]$
22:         for all $P_1 \xrightarrow{\mathsf{RW}(x)} P_2 \xrightarrow{\mathsf{RW}(y)} P_3 \land x \neq y \land \land\neg(P_2 \bowtie P_3) \land \neg(P_1 \xrightarrow{\mathsf{SO}} P_2)$ do
23:             if $\mathsf{TC}[P_3, P_1]$
24:                 return true
25:     return false
```

Algorithm 4 The robust allocation algorithm

Require: $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \_)$: a workload with unspecified isolation levels
Ensure: $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$ with allocated isolation levels level is statically robust

```
1: procedure Allocate($\mathcal{P}$)
2:     $\mathcal{G} \leftarrow$ BuildSDG($\mathcal{P}$)
3:     for all $P \in \mathcal{P}$ do
4:         if IsWriteOnly($P$) $\lor$ IsSingleVariableReadOnly($P$)
5:             $\mathsf{level}(P) \leftarrow \mathsf{RA}$                ▷ Allocation rule (A1)
6:         else if IsReadOnly($P$)
7:             $\mathsf{level}(P) \leftarrow \mathsf{PC}$                ▷ Allocation rule (A2)
8:         else if HasConflictingReadWrite($P, \mathcal{P}$)
9:             $\mathsf{level}(P) \leftarrow \mathsf{SER}$               ▷ Allocation rule (A4)
10:        else
11:            $\mathsf{level}(P) \leftarrow \mathsf{PSI}$               ▷ Allocation rule (A3)
12:    return level
13: procedure IsWriteOnly($P$)
14:    return $\mathsf{WSet}(P) \neq \emptyset \land \mathsf{RSet}(P) = \emptyset$
15: procedure IsReadOnly($P$)
16:    return $\mathsf{RSet}(P) \neq \emptyset \land \mathsf{WSet}(P) = \emptyset$
17: procedure IsSingleVariableReadOnly($P$)
18:    return $\mathsf{WSet}(P) = \emptyset \land |\mathsf{RSet}(P)| = 1$
19: procedure HasConflictingReadWrite($P, \mathcal{P}$)
20:    for all $P \xrightarrow{\mathsf{RW}} P'$ do
21:        if $\neg(P \bowtie P')$
22:            return true
23:    return false
```

M ALGORITHM FOR ROBUST ALLOCATION

The allocation algorithm (Algorithm 4) takes as input a workload $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \_)$ with unspecified isolation levels, and assigns isolation levels to $\mathcal{P}$ according to Theorem 6.18, so that the resulting $\mathcal{W} = (\mathcal{P}, \mathsf{SO}, \mathsf{level})$ is statically robust. Specifically, it first constructs the static dependency graph $\mathcal{G} = \mathrm{SDG}(\mathcal{W})$ (line 4:2). Then, for each program instance $P \in \mathcal{P}$ (line 4:3), it assigns the appropriate isolation level based on the read and write sets of P and its dependencies with other program instances in $\mathcal{P}$ (line 4:4–line 4:11).

The overall time complexity of the algorithm is $O(n^2)$:

$$\underbrace{O(n^2)}_{\textsc{BuildSDG}} + \underbrace{O(n)}_{P \in \mathcal{P}} \cdot \max\Big(\underbrace{O(1)}_{(A1)}, \underbrace{O(1)}_{(A2)}, \underbrace{O(n)}_{(A4)}, \underbrace{O(1)}_{(A3)}\Big) = O(n^2),$$

where $n = |\mathcal{P}|$ is the number of program instances.

N ROBUSTNESS ALLOCATION FOR BENCHMARK WORKLOADS

SmallBank. Figure 12 illustrates the isolation levels allocated to the SmallBank workload. By Theorem 6.18, all `Balance` program instances are allocated to PC since they are read-only. All `Amalgamate`, `DepositChecking`, and `TransactSavings` program

		RA (400 instances)			
addCourse_1	...	addCourse_200	register_1	...	register_200
		PSI (255 instances)			
enrol_1	enrol_71	enrol_105	enrol_132	enrol_159	enrol_189
enrol_10	enrol_77	enrol_107	enrol_133	enrol_162	enrol_199
enrol_112	enrol_8	enrol_112	enrol_135	enrol_164	enrol_20
enrol_114	enrol_82	enrol_114	enrol_137	enrol_168	enrol_200
enrol_116	enrol_86	enrol_116	enrol_139	enrol_17	enrol_21
enrol_117	enrol_87	enrol_117	enrol_142	enrol_173	enrol_22
enrol_125	enrol_94	enrol_125	enrol_144	enrol_179	enrol_25
enrol_130		enrol_130	enrol_148	enrol_18	enrol_30
remCourse_1	...	remCourse_67	remCourse_134	remCourse_200	
		PC (200 instances)			
query_1	...	query_200			
		SER (145 instances)			
enrol_1	enrol_27	enrol_54	enrol_82	enrol_109	enrol_138
enrol_10	enrol_28	enrol_55	enrol_83	enrol_11	enrol_140
enrol_100	enrol_29	enrol_56	enrol_84	enrol_110	enrol_141
enrol_101	enrol_3	enrol_57	enrol_85	enrol_111	enrol_143
enrol_102	enrol_31	enrol_58	enrol_88	enrol_113	enrol_145
enrol_103	enrol_32	enrol_59	enrol_89	enrol_115	enrol_146
enrol_104	enrol_34	enrol_60	enrol_9	enrol_118	enrol_147
enrol_106	enrol_35	enrol_61	enrol_90	enrol_119	enrol_149
enrol_108	enrol_36	enrol_62	enrol_91	enrol_12	enrol_150
enrol_109	enrol_37	enrol_63	enrol_92	enrol_120	enrol_152
enrol_11	enrol_39	enrol_64	enrol_93	enrol_121	enrol_154
enrol_113	enrol_40	enrol_65	enrol_95	enrol_122	enrol_156
enrol_115	enrol_41	enrol_66	enrol_96	enrol_123	enrol_158
enrol_118	enrol_43	enrol_67	enrol_97	enrol_124	enrol_16
enrol_119	enrol_44	enrol_68	enrol_98	enrol_126	enrol_160
enrol_12	enrol_45	enrol_69	enrol_99	enrol_127	enrol_161
enrol_120	enrol_47	enrol_70	enrol_128	enrol_163	
enrol_121	enrol_48	enrol_72	enrol_129	enrol_165	
enrol_122	enrol_49	enrol_73	enrol_13	enrol_166	
enrol_123	enrol_5	enrol_74	enrol_131	enrol_167	
enrol_124	enrol_50	enrol_75	enrol_134	enrol_169	
enrol_126	enrol_51	enrol_76	enrol_136	enrol_170	
enrol_127	enrol_52	enrol_78	enrol_138	enrol_171	
enrol_128	enrol_53	enrol_79	enrol_14	enrol_172	
enrol_129	enrol_54	enrol_80	enrol_140	enrol_174	
enrol_13	enrol_55	enrol_81	enrol_141	enrol_175	

Figure 13: Isolation levels allocated to the Courseware workload (with 1000 instances).

		RA (40 instances)			
StockLevel_1	...	StockLevel_40			
		PSI (920 instances)			
Delivery_1	...	Delivery_40	NewOrder_1	...	NewOrder_450
Payment_1	...	Payment_430			
		PC (40 instances)			
OrderStatus_1	...	OrderStatus_40			

Figure 14: Isolation levels allocated to the TPC-C workload (with 1000 instances).

		PSI (732 instances)			
Amalgamate_1	...	Amalgamate_200	DepositChecking_1	...	DepositChecking
TransactSavings_1	...	TransactSavings_200			
WriteCheck_1	WriteCheck_3	WriteCheck_4	WriteCheck_5	WriteCheck_6	WriteCheck_8
WriteCheck_9	WriteCheck_10	WriteCheck_11	WriteCheck_12	WriteCheck_13	WriteCheck_16
WriteCheck_18	WriteCheck_20	WriteCheck_21	WriteCheck_22	WriteCheck_23	WriteCheck_24
WriteCheck_25	WriteCheck_26	WriteCheck_28	WriteCheck_31	WriteCheck_34	WriteCheck_36
WriteCheck_38	WriteCheck_39	WriteCheck_41	WriteCheck_42	WriteCheck_43	WriteCheck_44
WriteCheck_46	WriteCheck_47	WriteCheck_48	WriteCheck_49	WriteCheck_51	WriteCheck_54
WriteCheck_58	WriteCheck_59	WriteCheck_60	WriteCheck_62	WriteCheck_65	WriteCheck_67
WriteCheck_68	WriteCheck_69	WriteCheck_70	WriteCheck_71	WriteCheck_72	WriteCheck_74
WriteCheck_75	WriteCheck_76	WriteCheck_77	WriteCheck_79	WriteCheck_82	WriteCheck_83
WriteCheck_84	WriteCheck_85	WriteCheck_86	WriteCheck_88	WriteCheck_89	WriteCheck_90
WriteCheck_93	WriteCheck_95	WriteCheck_96	WriteCheck_98	WriteCheck_99	WriteCheck_100
WriteCheck_101	WriteCheck_106	WriteCheck_107	WriteCheck_108	WriteCheck_109	WriteCheck_110
WriteCheck_112	WriteCheck_113	WriteCheck_115	WriteCheck_117	WriteCheck_119	WriteCheck_120
WriteCheck_121	WriteCheck_122	WriteCheck_123	WriteCheck_125	WriteCheck_126	WriteCheck_128
WriteCheck_130	WriteCheck_131	WriteCheck_132	WriteCheck_134	WriteCheck_135	WriteCheck_137
WriteCheck_138	WriteCheck_141	WriteCheck_142	WriteCheck_143	WriteCheck_144	WriteCheck_145
WriteCheck_147	WriteCheck_148	WriteCheck_149	WriteCheck_150	WriteCheck_151	WriteCheck_152
WriteCheck_154	WriteCheck_158	WriteCheck_159	WriteCheck_161	WriteCheck_164	WriteCheck_166
WriteCheck_167	WriteCheck_169	WriteCheck_170	WriteCheck_172	WriteCheck_173	WriteCheck_174
WriteCheck_176	WriteCheck_177	WriteCheck_178	WriteCheck_182	WriteCheck_183	WriteCheck_184
WriteCheck_187	WriteCheck_188	WriteCheck_189	WriteCheck_190	WriteCheck_191	WriteCheck_192
WriteCheck_194	WriteCheck_195	WriteCheck_197	WriteCheck_198	WriteCheck_199	WriteCheck_200
		PC (200 instances)			
Balance_1	...	Balance_200			
		SER (68 instances)			
WriteCheck_2	WriteCheck_7	WriteCheck_14	WriteCheck_15	WriteCheck_17	WriteCheck_19
WriteCheck_27	WriteCheck_29	WriteCheck_30	WriteCheck_32	WriteCheck_33	WriteCheck_35
WriteCheck_37	WriteCheck_40	WriteCheck_45	WriteCheck_50	WriteCheck_52	WriteCheck_53
WriteCheck_55	WriteCheck_56	WriteCheck_57	WriteCheck_61	WriteCheck_63	WriteCheck_64
WriteCheck_66	WriteCheck_73	WriteCheck_78	WriteCheck_80	WriteCheck_81	WriteCheck_87
WriteCheck_91	WriteCheck_92	WriteCheck_94	WriteCheck_97	WriteCheck_102	WriteCheck_103
WriteCheck_104	WriteCheck_105	WriteCheck_111	WriteCheck_114	WriteCheck_116	WriteCheck_118
WriteCheck_124	WriteCheck_127	WriteCheck_129	WriteCheck_133	WriteCheck_136	WriteCheck_139
WriteCheck_140	WriteCheck_146	WriteCheck_153	WriteCheck_155	WriteCheck_156	WriteCheck_157
WriteCheck_160	WriteCheck_162	WriteCheck_163	WriteCheck_165	WriteCheck_168	WriteCheck_171
WriteCheck_175	WriteCheck_179	WriteCheck_180	WriteCheck_181	WriteCheck_185	WriteCheck_186
WriteCheck_193	WriteCheck_196				

Figure 12: Isolation levels allocated to the SmallBank workload (with 1000 instances).

instances are allocated to PSI for they all satisfy the condition (A3). The WriteCheck program instances incur more complex conflicts, and are allocated to either PSI or SER, depending on their specific conflicts with other program instances in the workload.

Courseware. Figure 13 illustrates the allocation result for the Courseware workload. By Theorem 6.18, all Query program instances are allocated to PC since they are read-only. All AddCourse program instances are allocated to RA for they are write-only. All Register program instances are allocated to RA for they are single-key read-only. Enroll and RemoveCourse program instances incur more complex conflicts, and are allocated to either PSI or SER, depending on their specific conflicts with other program instances in the workload.

TPC-C. Figure 14 illustrates the isolation levels allocated to the TPC-C workload. By Theorem 6.18, all StockLevel program instances are allocated to PC since they are single-key read-only. All Delivery, NewOrder, and Payment program instances are allocated to PSI for they all satisfy the condition (A3). Particularly, no program instances are allocated to SER in this workload.

O PERFORMANCE EVALUATION ON PC-SI-SER

We have implemented our PC-SI-SER protocol in Algorithm 2. In this section, we evaluate the performance gains of PC-SI-SER when a workload is allocated isolation levels from {PC, SI, SER}, rather than from {SI, SER} or only SER.

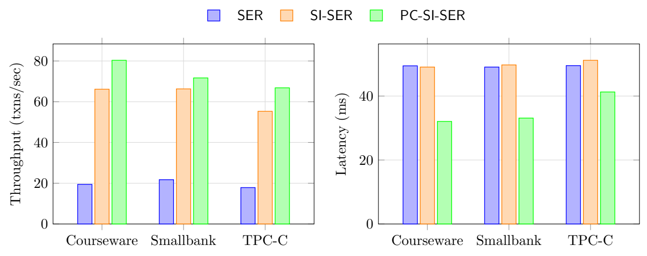


Figure 15: Performance comparison of three isolation allocations across three benchmarks.

O.1 Experimental Setup

We conduct these experiments on a local machine, with the same setup as described in Section 7. The implementation of the PC-SI-SER protocol is multi-threaded, where each thread simulates a separate database node. Inter-node communication incurs random latency ranging from 30ms to 50ms, simulating wide-area network conditions in distributed databases. For each benchmark, we generate five workloads, each consisting of 10 sessions with 20 program instances per session.

O.2 Performance Evaluation

Figure 15 presents the throughput and latency comparison under different configurations across three benchmarks. Our allocation over {PC, SI, SER} demonstrates significant performance improvements over the alternatives over {SI, SER} and only SER. Specifically, the allocation over {PC, SI, SER} achieves approximately 3-4x improvement in throughput across all benchmarks, compared to running the whole workload under SER, and about 8-22% improvement compared to running under {SI, SER} (for Courseware, it is 80.34 vs 66.14 txns/sec; for SmallBank, it is 71.69 vs 66.30 txns/sec; for TPC-C, it is 66.82 vs 55.30 txns/sec).

Our fine-grained isolation level allocation also substantially reduces the average transaction latency. Specifically, it reduces latency by 33-35% on Courseware and SmallBank and by about 19% on TPC-C compared to running under SI-SER.