Plume: Efficient and Complete Black-Box Checking of Weak Isolation Levels

SI LIU, ETH Zurich, Switzerland
LONG GU, State Key Laboratory for Novel Software Technology, Nanjing University, China
HENGFENG WEI*, State Key Laboratory for Novel Software Technology, Nanjing University, China
DAVID BASIN, ETH Zurich, Switzerland

Modern databases embrace weak isolation levels to cater for highly available transactions. However, weak isolation bugs have recently manifested in many production databases. This raises the concern of whether database implementations actually deliver their promised isolation guarantees in practice. In this paper we present Plume, the first efficient, complete, black-box checker for weak isolation levels. Plume builds on modular, fine-grained, transactional anomalous patterns, with which we establish sound and complete characterizations of representative weak isolation levels, including read committed, read atomicity, and transactional causal consistency. Plume leverages a novel combination of two techniques, vectors and tree clocks, to accelerate isolation checking. Our extensive assessment shows that Plume can reproduce all known violations in a large collection of anomalous database execution histories, detect new isolation bugs in three production databases along with informative counterexamples, find more weak isolation anomalies than the state-of-the-art checkers, and efficiently validate isolation guarantees under a wide variety of workloads.

CCS Concepts: • **Information systems → Database management system engines**; • **General and reference → Validation**; • **Software and its engineering → Consistency**; **Dynamic analysis**.

Additional Key Words and Phrases: weak isolation levels, formal specification, black-box testing

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

1 Introduction

Modern ACID and NewSQL databases often embrace *weak isolation levels*, catering for *highly available transactions* (HATs). These weak isolation levels are also referred to as HAT isolation levels (or guarantees) [Bailis et al. 2013a,b]. HATs ensure "always on" system operation even in the presence of network partitions, eschewing the performance penalties of stronger isolation levels such as snapshot isolation and serializability. Nevertheless, they still provide useful semantics like atomic visibility and causal consistency to database users and application programmers. The past decade has seen sustained, intensive efforts in designing highly available, weakly isolated, distributed transactions [Akkoorath et al. 2016; Bailis et al. 2016; Cheng et al. 2021; Didona et al. 2018;

*Corresponding author.

Authors' Contact Information: Si Liu, si.liu@inf.ethz.ch, ETH Zurich, Zurich, Switzerland; Long Gu, 502023320005@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Hengfeng Wei, hfwei@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; David Basin, basin@inf.ethz.ch, ETH Zurich, Zurich, Switzerland.

ACM 2475-1421/2024/10-ART302
https://doi.org/10.1145/3689742

Liu 2022; Liu et al. 2024b; Lloyd et al. 2013; Lu et al. 2020]. Recent years have also seen numerous applications of HAT semantics beyond databases, including serverless computing [Lykhenko et al. 2021; Wu et al. 2020] and microservices [Ferreira Loff et al. 2023; Pereira and Silva 2023].

HAT-compliant weak isolation guarantees include the SQL-92 standardized *read committed* [Berenson et al. 1995], which is the default level for most SQL databases [Bailis et al. 2013a; Crooks et al. 2017], and *read atomicity* (RA) [Bailis et al. 2016], which has seen many real-world applications, such as secondary indexing and foreign key constraints, and has recently been layered on Facebook's TAO to provide atomically visible transactions [Cheng et al. 2021]. Notably, *transactional causal consistency* (TCC) [Akkoorath et al. 2016; Lloyd et al. 2013] represents a successful marriage of the distributed computing and database communities by extending causal consistency [Ahamad et al. 1995; Perrin et al. 2016], the *strongest* consistency level achievable in an always-available system [Attiya et al. 2015], with the transactional guarantee. There has been a plethora of academic advances over the last decade in highly available and performant TCC database systems [Akkoorath et al. 2016; Didona et al. 2018; Du et al. 2014; Liu et al. 2024b; Lloyd et al. 2013; Lu et al. 2020; Mehdi et al. 2017]. Recent production adoptions of TCC and its variants include Neo4j [2023] (via "bookmarked" transactions), ElectricSQL [2023] (a successful transition from Cure [Akkoorath et al. 2016] to production), and Azure Cosmos DB [Microsoft 2023] (session guarantees and prefix consistency [Burckhardt et al. 2015] for its transactional batch).

HAT isolation bugs have however appeared in many production databases claiming to provide HAT or even stronger isolation levels [Biswas and Enea 2019; Huang et al. 2023; Jepsen 2024b].[1] This raises the concern of whether database implementations actually deliver their promised service-level agreements in practice. Many black-box testing and validation efforts [Biswas and Enea 2019; Huang et al. 2023; Kingsbury and Alvaro 2020; Tan et al. 2020; Zhang et al. 2023]—needed as database internals are often unavailable or impenetrable—have been devoted to finding isolation anomalies and validating isolation fulfilment. Two representative black-box checkers are the recent dbcop [Biswas and Enea 2019] and Elle [Kingsbury and Alvaro 2020], both of which are successful in detecting real-world HAT isolation bugs and have shown promising performance.

Research Gaps. Despite these recent advances, there are still challenges that existing work has not yet fully addressed.

G1 (**Completeness**): Most checkers miss bugs. Some of them [Tan et al. 2020; Zhang et al. 2023] focus on inter-transactional dependencies, while isolation bugs manifesting inside a transaction are also critical [Huang et al. 2023], e.g., non-repeatable reads detected in MySQL [Kingsbury 2023] and MariaDB (§5.2). Moreover, for more recent isolation levels such as RA and TCC, Elle searches for a mix of anomalies from different formalisms [Adya 1999; Cerone et al. 2015], which are not known to completely characterize the isolation level in question.

G2 (**Efficiency**): Although weak isolation levels are polynomial time checkable [Biswas and Enea 2019] (with respect to a collected database execution history), graph traversals in existing tools [Biswas and Enea 2019; Kingsbury and Alvaro 2020] often require significant time in practice, searching for cycles (representing anomalies) under workloads of high concurrency with, e.g., skewed key accesses or a large number of clients. Such workloads are, however, desirable to stress test or validate databases. Despite the increasing trend of utilizing advanced solvers [Bayless et al. 2015; Potassco 2023] to check isolation levels, encoding and solving dependency constraints is still expensive in practice.

G3 (**Understandability**): Solver-based tools return hard-to-interpret counterexamples in general like unsatisfied clauses. Tools utilizing graph traversals may report large cycles, which are

[1]Violations of weak isolation guarantees, such as TCC, also violate stronger isolation guarantees, such as serializability. Databases offering stronger isolation guarantees would inherently be devoid of weak isolation bugs.

also uninformative with respect to the malfunctioning mechanisms. Moreover, depending on the granularity of the underlying characterizations of isolation levels, existing tools may not straightforwardly distinguish different anomalies, e.g., *fractured reads* and *non-monotonic reads*, which are, however, relevant to different concurrency control mechanisms [Bailis et al. 2013a, 2016]. All this makes understanding and debugging violations hard.

The Plume Checker. We present a novel black-box checker, called Plume, for checking weak HAT isolation guarantees. Plume builds on two *conceptual shifts* to fill all the above gaps.

Inspired by the semantic building blocks of HAT isolation levels, such as non-repeatable reads [Bailis et al. 2013a], atomic visibility [Bailis et al. 2016], causal consistency [Perrin et al. 2016], and data convergence [Akkoorath et al. 2016; Lloyd et al. 2011], we devise 14 *modular, fine-grained, transactional anomalous patterns* (TAPs) to characterize HAT isolation levels. We base our characterization on the axiomatic framework of Biswas and Enea [2019] for defining isolation levels. Nonetheless, we take a conceptually different perspective by defining them via their anomalies, for which we establish both soundness (no false positives) and completeness (missing no anomalies).

This *conceptual shift* allows us to effectively address both G1 and G3 as follows. First, our TAP-based characterization enables Plume to detect *all* possible HAT isolation violations in an execution history of transactions, as in black-box testing of database isolation guarantees (G1). This is highly desirable, especially given that the occurrence of a specific isolation bug is usually unpredictable with randomized test generation as commonly adopted by existing black-box checkers. Moreover, being fine-grained, our TAPs also correspond to specific defects of the building blocks underlying a database system, such as flaws in the atomic commitment of transactions or in maintaining causality. This enables Plume to report *informative* counterexamples with violating scenarios that aid developers in discovering their causes (G3).

In addition to individual anomalies, including one TAP for *cut isolation* [Bailis et al. 2013a], our modular TAPs enable soundly and completely checking other HAT isolation levels. Enabling 9 TAPs in Plume results in checking *read committed*. With 3 more TAPs, Plume checks *read atomicity*. All 14 TAPs are equivalent to checking *transactional causal consistency*.

To overcome G2, we first leverage our insight that SMT solvers specialize in resolving uncertain dependencies such as the order of concurrent writes, which heavily reside, however, in *stronger* isolation levels such as snapshot isolation and serializability [Huang et al. 2023; Tan et al. 2020; Zhang et al. 2023]. Hence, we adopt non-solver-based graph traversals in checking weak HAT isolation guarantees to mitigate the high overhead of encoding and solving uncertain dependencies.

Furthermore, inspired by the decades-long practice of using *vectors* [Fidge 1988] to *design* causally consistent databases, we utilize vectors to *black-box check* isolation levels, including the strongest achievable HAT isolation level to date, namely TCC. This *conceptual shift* enables Plume to accelerate graph traversals for both reachability checking and cycle detection by capturing and storing the transitive causal dependencies among transactions using vectors. More specifically, Plume maintains the following *invariant* during the construction of a transaction dependency graph: transaction t_2 is reachable from transaction t_1 if and only if the associated vector of t_2 is greater than or equal to that of t_1. Cycle detection then amounts to checking if there are two transactions having the same vector. Additionally, owing to this invariant, during bug search or vector updates, a traversal can backtrack early upon encountering a transaction with a larger vector. This is because the vectors of downstream transactions are at least as large as the current one.

To further speed up Plume, we also adapt the *tree clock* [Mathur et al. 2022] data structure, a recent advance in concurrent program analysis, in our utilization of vectors. This significantly reduces the time needed to compare and join vectors, which arises from the large number of client sessions and transactional dependencies.

Contributions. Overall, this paper makes the following contributions.

- We introduce a collection of modular, fine-grained TAPs capturing a wide range of isolation anomalies. Using these TAPs, we formally establish sound and complete characterizations of four prevalent HAT isolation levels (§3).
- We design an efficient TAP-based checking algorithm, leveraging a combination of vectors and tree clocks to accelerate the search for isolation anomalies (§4).
- We implement our algorithm in Plume, a tool for checking different isolation levels and individual anomalies. We also conduct an extensive assessment of Plume, along with the state-of-the-art black-box isolation checkers, on a wide variety of benchmarks (§5).

Our experiments show that Plume can (i) reproduce all known isolation bugs in over 3000 anomalous histories archived by the recent testing campaign, (ii) detect novel isolation bugs in three production database engines of different kinds with informative counterexamples, (iii) find more HAT isolation anomalies than the state-of-the-art checkers, (iv) substantially outperform the strong baselines on various benchmarks, and (v) scale to large workloads, with one million transactions and over 50 million operations successfully checked in under five minutes.

This work also complements two lines of research: (i) at the theoretical level, prior efforts [Adya 1999; Berenson et al. 1995; Bouajjani et al. 2017] on using anomalies to faithfully characterize data consistency properties, which, however, do not cover more recent HAT semantics such as *read atomicity* and *transactional causal consistency*; and (ii) at the practical level, recent advances in checking stronger isolation levels, including PolySI [Huang et al. 2023] and Viper [Zhang et al. 2023] for snapshot isolation as well as Cobra [Tan et al. 2020] for serializability.

2 Background

2.1 Weak HAT Isolation Levels

Databases provide various isolation levels (or guarantees), as shown in Figure 1, accommodating different trade-offs between data consistency and performance. While highly available transactions (HATs) offer concurrency and performance benefits, they cannot achieve all possible isolation guarantees [Bailis et al. 2013a]. We briefly explain the informal definitions of the HAT-compliant isolation levels this work focuses on, and defer their formal definitions to §3.1.

Cut Isolation (CI). Each transaction should read from a non-changing snapshot over the data items [Bailis et al. 2013a]. It prevents *non-repeatable reads* [Berenson et al. 1995]. For instance, if a transaction reads the same data item more than once, it should always see the same value.

Read Committed (RC). This is the default isolation level of many databases, which guarantees that a transaction reads only committed changes made by other transactions. It avoids *dirty reads* (reading a value that was never committed) [Berenson et al. 1995] but allows anomalies like non-repeatable reads.

Read Atomicity (RA). This ensures that all or none of a transaction's updates are observed by other transactions. It prohibits *fractured reads* anomalies [Bailis et al. 2016], such as Rachel only observing one direction of a new (bidirectional) friendship between Joe and Ross in a social network.

Transactional Causal Consistency (TCC). Intuitively, this isolation guarantee combines non-transactional causal consistency (CC) and transactional RA, enhanced by an additional guarantee of data convergence [Akkoorath et al. 2016; Lloyd et al. 2013].

CC requires that two (non-transactional) read or write operations that are causally related must appear to all client sessions[2] in the same *causal order* [Ahamad et al. 1995; Perrin et al. 2016]. It

[2]A client session typically refers to a period of interaction between a client and a server, e.g., between logging into and logging out a database application.

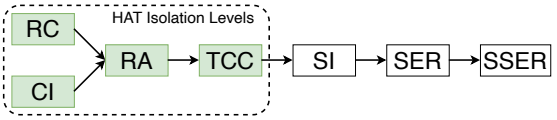


Fig. 1. A hierarchy of database isolation levels. The HAT-compliant ones are placed in the dashed box. $A \longrightarrow B$ means A is strictly weaker than B. CI: cut isolation [Bailis et al. 2013a]; RC: read committed [Berenson et al. 1995]; RA: read atomicity [Bailis et al. 2016]; TCC: transactional causal consistency [Akkoorath et al. 2016; Lloyd et al. 2013]; SI: snapshot isolation [Cerone and Gotsman 2018]; SER: serializability [Papadimitriou 1979]; SSER: strict serializability [Papadimitriou 1979].

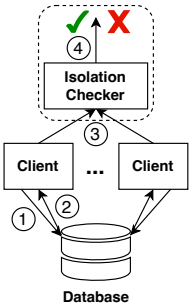


Fig. 2. Architecture of black-box database isolation checking.

prevents causality violations, such as Rachel observing Joe's response to Ross' message without seeing the message itself. However, causally unrelated transactions may be observed in different orders across sessions, which may result in the permanent divergence of these sessions' views under concurrent conflicting updates. For example, Joe and Ross independently updating the meeting place to "my place" in a road trip planner may cause confusion. The *convergence* property prevents this by requiring all views to eventually converge to the same state [Akkoorath et al. 2016; Lloyd et al. 2011], e.g., by applying the LWW (last-writer-wins) rule [Burckhardt et al. 2014]. Consequently, Rachel and Monica would see the same final place.

2.2 Black-box Database Isolation Checking

Black-box checking of database isolation guarantees typically proceeds as follows, also illustrated in Figure 2. Clients initiate transactional requests, typically produced by some randomized workload generator (see, e.g., §5.1), to a database that may be partitioned or replicated; nevertheless, the database is consistently treated as a black-box (①). Each client session logs the requests it sents, such as a read for key x, along with the corresponding results delivered by the database, e.g., the value of x (②). All the logs from each client are then combined into a unified history, that is subsequently provided to the isolation checker (③). Finally, the checker solves the *isolation checking problem*: determining whether the transaction history satisfies the isolation level in question. Upon detecting bugs, some checkers also return counterexamples (④).

The first three steps are the *de facto* procedures adopted by existing black-box checkers. However, they vary in their approaches for the final step, which falls mainly into two categories. In Category (i), isolation checkers, such as Elle [Kingsbury and Alvaro 2020], construct a kind of transactional dependency graph, such as Adya's directed serialization graph [Adya 1999], and search for cycles that represent isolation anomalies. In Category (ii), checkers like PolySI [Huang et al. 2023] encode the dependency graph and cycles into logic formulas, such as SAT formulas, and invoke an off-the-shelf solver like MonoSAT [Bayless et al. 2015]. Our approach belongs to Category (i), but utilizes transactional anomalous patterns to represent HAT isolation violations and a combination of vectors and tree clocks to accelerate violation searching.

Note that violations of weaker isolation guarantees, such as TCC, also violate stronger isolation guarantees, such as snapshot isolation. As we will see (§5.2), many isolation bugs found in production databases that claim to support snapshot isolation or even the stronger serializability are actually bugs for much weaker isolation levels like cut isolation. Note also that histories satisfying stronger isolation guarantees naturally satisfy weaker ones.

Assumptions. Following the common practice in black-box isolation checking [Biswas and Enea 2019; Huang et al. 2023; Kingsbury and Alvaro 2020; Zhang et al. 2023], we make two assumptions. First, every history contains a special transaction that writes the initial values of all keys. This transaction precedes all the other transactions across client sessions. In practice, we usually use multiple, short, write-only transactions to populate the database *before* testing starts. These transactions can be considered as a single, logical, write-only transaction.

Second, for each key, we assume that every write to the key assigns a unique value. Each read can therefore be uniquely associated with the transaction that issues the corresponding (dictating) write. The checking problem for, e.g., TCC, can then be solved in polynomial time [Biswas and Enea 2019]; otherwise, it is NP-hard in general. In practice, we can use the client identifier and local counter to ensure the uniqueness of written values.

3 Characterizing HAT Isolation Levels via Anomalies

In this section we present our *transactional anomalous patterns* (TAPs). We also formally establish sound and complete characterizations of four prevalent HAT isolation levels using these TAPs, namely cut isolation (CI), read committed (RC), read atomicity (RA), and transactional causal consistency (TCC). Let us first describe the formal basis.

3.1 Context and Definitions

We consider a key-value data store managing a set of keys $\mathsf{K} = \{x, y, z, \dots\}$ associated with values from a set V. We use $\mathsf{R}(x, v)$ to denote a read operation that reads $v \in \mathsf{V}$ from $x \in \mathsf{K}$ and $\mathsf{W}(x, v)$ to denote a write operation that writes $v \in \mathsf{V}$ to $x \in \mathsf{K}$. In what follows, we write _ for a component that is irrelevant and implicitly existentially quantified. We write $\exists!$ to mean "unique existence."

A binary relation R over a set A is a subset of $A \times A$. For $a, b \in A$, we write $(a, b) \in R$ and $a \xrightarrow{R} b$ interchangeably. Let $I_A \triangleq \{(a, a) \mid a \in A\}$ be the identity relation on A and $R^{-1} \triangleq \{(b, a) \mid (a, b) \in R\}$ the inverse of R. We use R^+ to denote the transitive closure of R. A relation $R \subseteq A \times A$ is *acyclic* if $R^+ \cap I_A = \emptyset$. 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.

Clients interact with the data store by issuing transactions.

Definition 1 (Transaction [Biswas and Enea 2019]). A *transaction* is a pair (O, po), where O is a set of operations and $\mathrm{po} \subseteq O \times O$ is a strict total order over O called the *program order*.

We use $t, t_1, t_2, \dots$ to range over transactions, and use $O(t)$, $O_x(t)$, $R(t)$, $R_x(t)$, $W(t)$, and $W_x(t)$ to denote the sets of operations, operations on x, read operations, read operations on x, write operations, and write operations on x in a transaction t, respectively. The extension to sets of transactions is straightforward. For example, $R(T) = \bigcup_{t \in T} R(t)$ denotes the set of read operations in a set T of transactions. For a transaction t, 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 denote the program order of a transaction t as po_t.

Transactions are grouped into *sessions*, where a session is a sequence of transactions. We use *histories* to record the client-visible results of the interactions between clients and the data store.

Definition 2 (History [Biswas and Enea 2019]). A *history* is a tuple $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$, where T is a set of committed transactions, $\mathrm{SO} \subseteq T \times T$ is the (client) session order, and $\mathrm{WR} : \mathsf{K} \to 2^{T \times T}$ is the write-read relation, such that

- $\forall t, s \in T.\ \forall x \in \mathsf{K}.\ t \xrightarrow{\mathrm{WR}(x)} s \implies \exists v \in \mathsf{V}.\ t \neq s \wedge t \vdash \mathsf{W}(x, v) \wedge s \vdash \mathsf{R}(x, v)$ and $\forall s \in T.\ \forall x \in \mathsf{K}.\ s \vdash \mathsf{R}(x, \_) \implies \exists!\ t \in T.\ t \xrightarrow{\mathrm{WR}(x)} s$; and

- $(\mathrm{SO} \cup \bigcup_{x \in \mathsf{K}} \mathrm{WR}(x))$ is acyclic.

Note that a history records only committed transactions. We use $T_\otimes$ to denote the set of *aborted* transactions during a database execution. If $t_1 \xrightarrow{\mathrm{WR}(x)} t_2$, we say that t_2 reads a value on x written by t_1; we also write $t_1 \xrightarrow{\mathrm{WR}} t_2$ when x is irrelevant. For convenience, we abuse notation and use WR to also denote the relation $\bigcup_{x \in \mathsf{K}} \mathrm{WR}(x) \subseteq T \times T$. The *causal order* is then defined as $\mathrm{CO} \triangleq (\mathrm{SO} \cup \mathrm{WR})^+$, capturing the potential causality between transactions. We write $\mathsf{WT}_x = \{t \in T \mid \exists o \in O(t).\ o = \mathsf{W}(x, \_)\}$ (resp. $\mathsf{RT}_x = \{t \in T \mid \exists o \in O(t).\ o = \mathsf{R}(x, \_)\}$) to denote the set of transactions that write to (resp. read from) x. We further use $\mathrm{wr}(x)$ (or wr if x is irrelevant) to denote the write-read relation over the *operations* on x in a history. We allow a read operation to read the value written by a write operation within the same transaction.

We base our TAP-based characterization of HAT isolation levels on the axiomatic framework proposed by Biswas and Enea [2019]. A history satisfying an isolation level is defined as the existence of a session order SO, a write-read relation WR, and a *commit order* on its transactions which obey certain axioms (see below). The commit order $\mathrm{CM} \subseteq T \times T$ is a strict total order preserving the causal order of transactions, i.e., $\mathrm{CO} \subseteq \mathrm{CM}$. Intuitively, CM dictates the order in which transactions are committed to the "global" data store. Note that CM subsumes the well-known *version order* (per key) [Adya 1999]: Assuming that both transactions t and t' write to key x, $t \xrightarrow{\mathrm{CM}} t'$ if and only if the version of x written by t' is ordered after that written by t. We will use the commit order and version order interchangeably when the transactions involved write to *the same key*.

Biswas and Enea's framework is suitable for black-box isolation checking over database histories, since the SO, WR, and CO relations can be extracted or derived straightforwardly from a history, and the commit order can be effectively inferred from these three relations (see below). We unify the definitions of four representative HAT isolation levels within this formal framework, including our new axioms for CI and RC's *monotonic atomic view*. This underlies our subsequent design of fine-grained TAPs and proofs for their soundness and completeness.

Definition 3 (**Cut Isolation** [Bailis et al. 2013a, 2016])**.** A history $(T, \mathrm{SO}, \mathrm{WR})$ satisfies CI if and only if the following *CutIsolation* axiom holds.

(*CutIsolation* Axiom) *If a transaction reads the same key from other transactions more than once, then it reads the same value each time.* Formally, $\forall x \in \mathsf{K}.\ \forall v, v' \in \mathsf{V}.\ \forall t \in \mathsf{RT}_x.\ \forall t_1 \neq t, t_2 \neq t \in \mathsf{WT}_x.\ \forall r_1 \triangleq \mathsf{R}(x, v), r_2 \triangleq \mathsf{R}(x, v') \in R_x(t).\ \forall w_1 \in W_x(t_1).\ \forall w_2 \in W_x(t_2).\ (t_1 \neq t_2 \wedge r_1 \neq r_2 \wedge w_1 \xrightarrow{\mathrm{wr}(x)} r_1 \wedge w_2 \xrightarrow{\mathrm{wr}(x)} r_2 \implies v = v')$.

Note that in the case where $t_1 = t_2$, we have the two writes w_1 and w_2 in a single transaction, both writing to the same key but with different values. When another transaction reads from both writes, we categorize this scenario as an anomaly for reading an *intermediate* read (namely TAP-f, presented in §3.2).

Definition 4 (**Read Committed**)**.** A history $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$ satisfies RC if and only if there exists a commit order CM on T such that the following four requirements hold.

RC-(1) A read operation cannot read the value written by a later write in the same transaction. Formally, $\forall t \in T.\ \forall r \in R(t).\ \forall w \in W(t).\ w \xrightarrow{\mathrm{wr}} r \implies w \xrightarrow{\mathrm{po}_t} r$.

RC-(2) In a transaction t, if a read operation r on x is preceded by write operations to x in t, then it should return the value written by the lastest write before r to x in t. Formally, $\forall x \in \mathsf{K}.\ \forall t \in T.\ \forall r \in R_x(t).\ ((\exists w' \in W_x(t).\ w' \xrightarrow{\mathrm{po}_t} r) \implies (\exists w \in W_x(t).\ w \xrightarrow{\mathrm{po}_t} r \wedge w \xrightarrow{\mathrm{wr}(x)} r \wedge (\forall w'' \in W_x(t).\ w'' \xrightarrow{\mathrm{po}_t} w \vee w'' = w \ \vee\ r \xrightarrow{\mathrm{po}_t} w'')))$.

RC-(3) If a transaction writes to a key (possibly) multiple times, then only the last one should be visible to other transactions. Formally, $\forall x \in \mathsf{K}.\ \forall t \in T.\ \forall w, w' \in W_x(t).\ ((\exists t' \neq t \in \mathsf{RT}_x.\ \exists r \in R_x(t').\ w \xrightarrow{\mathrm{wr}(x)} r) \implies w' \xrightarrow{\mathrm{po}_t} w \vee w' = w)$.

RC-(4) *MonoAtomicView* Axiom [Adya 1999; Bailis et al. 2013a][3]: *If both transactions t_1 and t_2 write to x, and transaction t_3 reads $y \neq x$ from t_2 and then reads x from t_1, then $t_2 \xrightarrow{CM} t_1$.* Formally, $\forall x, y \neq x \in \mathsf{K}.\ \forall t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \forall t_3 \in \mathsf{RT}_y \setminus \{t_1, t_2\}.\ ((\exists w_x \in W(t_1).\ \exists w_y \in W(t_2).\ \exists r_x, r_y \in R(t_3).\ r_y \xrightarrow{\mathrm{po}_{t_3}} r_x \wedge w_y \xrightarrow{\mathrm{wr}(y)} r_y \wedge w_x \xrightarrow{\mathrm{wr}(x)} r_x) \implies t_2 \xrightarrow{\mathrm{CM}} t_1)$.

Definition 5 (Read Atomicity). A history $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$ satisfies RA if and only if it satisfies RC (more specifically, RC-(1–3)) and there exists a commit order CM on T such that the following *ReadAtomic* axiom holds.

(*ReadAtomic* Axiom) *If a transaction t_3 reads a key x from t_1 and there exists a transaction $t_2 \neq t_1$ that also writes to x such that $t_2 \xrightarrow{SO \cup WR} t_3$, then $t_2 \xrightarrow{CM} t_1$.* Formally, $\forall x \in \mathsf{K}.\ \forall t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \forall t_3 \in \mathsf{RT}_x \setminus \{t_1, t_2\}.\ (t_1 \xrightarrow{\mathrm{WR}(x)} t_3 \wedge t_2 \xrightarrow{\mathrm{SO} \cup \mathrm{WR}} t_3 \implies t_2 \xrightarrow{\mathrm{CM}} t_1)$.

Note that the *MonoAtomicView* axiom RC-(4) is implied by the *ReadAtomic* axiom.

Definition 6 (Transactional Causal Consistency). A history $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$ satisfies TCC if and only if it satisfies RC (more specifically, RC-(1–3)) and there exists a commit order CM on T such that the following *Causal* axiom holds.

(*Causal* Axiom) *If a transaction t_3 reads a key x from t_1 and there exists a transaction $t_2 \neq t_1$ that also writes to x such that $t_2 \xrightarrow{CO} t_3$, then $t_2 \xrightarrow{CM} t_1$.* Formally, $\forall x \in \mathsf{K}.\ \forall t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \forall t_3 \in \mathsf{RT}_x \setminus \{t_1, t_2\}.\ (t_1 \xrightarrow{\mathrm{WR}(x)} t_3 \wedge t_2 \xrightarrow{\mathrm{CO}} t_3 \implies t_2 \xrightarrow{\mathrm{CM}} t_1)$.

Note that RC-(4) is implied by the *Causal* axiom. Note also that TCC's *convergence* property requires that concurrent conflicting transactions are committed in the same order across all servers (or replicas). The existence of a commit order CM naturally reflects this property.

We now formally relate these isolation levels (cf. Figure 1).

Theorem 1. TCC is stronger than RA, and RA is stronger than CI.

Proof. Since $\mathrm{SO} \cup \mathrm{WR} \subseteq \mathrm{CO}$, the *Causal* axiom implies the *ReadAtomic* axiom. Hence, TCC is stronger than RA.

Now we show that RA is stronger than CI. Let $\mathcal{H}$ be a history satisfying RA. Suppose by contradiction that $\mathcal{H}$ violates the *CutIsolation* axiom. Then $\mathcal{H}$ contains a transaction t that reads from a key from other transactions more than once, but with different values. Formally,

$$\exists x \in \mathsf{K}.\ \exists v, v' \neq v \in \mathsf{V}.\ \exists t \in \mathsf{RT}_x.\ \exists t_1 \neq t, t_2 \neq t \in \mathsf{WT}_x.\ \exists r_1 \triangleq \mathsf{R}(x, v), r_2 \triangleq \mathsf{R}(x, v') \in R_x(t).$$
$$\exists w_1 \in W_x(t_1).\ \exists w_2 \in W_x(t_2).\ (t_1 \neq t_2 \wedge w_1 \xrightarrow{\mathrm{wr}(x)} r_1 \wedge w_2 \xrightarrow{\mathrm{wr}(x)} r_2).$$

By the *ReadAtomic* axiom of RA, $t_1 \xrightarrow{\mathrm{CM}} t_2$ and $t_2 \xrightarrow{\mathrm{CM}} t_1$, contradicting that CM is a strict total order. Therefore, $\mathcal{H}$ also satisfies CI. □

[3]This axiom is also called the *Monotonic Reads* property in [Adya 1999].

Table 1. The description and formalization of 14 TAPs (also visualized in Figure 3).

TAP	Description	TAP	Description
(a)	A transaction reads a value out of thin air. $\exists r \in R(T).\ \forall w \in W(T \cup T_{\otimes}).\ \neg(w \xrightarrow{\text{wr}} r)$.	(b)	A transaction reads a value written by an aborted transaction. $\exists r \in R(T).\ \exists w \in W(T_{\otimes}).\ w \xrightarrow{\text{wr}} r$.
(c)	A transaction reads from a future write within the same transaction. $\exists t \in T.\ \exists w \in W(t).\ \exists r \in R(t).$ $(w \xrightarrow{\text{wr}} r \wedge r \xrightarrow{\text{po}_t} w)$.	(d)	Transaction t reads key x from transaction $t' \neq t$, but t has written to x before this read. $\exists x \in \mathsf{K}.\ \exists t, t' \neq t \in W_x(t).\ \exists r \in R_x(t).\ \exists w \in W_x(t).$ $\exists w' \in W_x(t').\ (w' \xrightarrow{\text{wr}(x)} r \wedge w \xrightarrow{\text{po}_t} r)$.
(e)	Transaction t reads key x from a write w in itself, but w is not the last write on x in t before this read. $\exists x \in \mathsf{K}.\ \exists t \in T.\ \exists w, w' \neq w \in W_x(t).\ \exists r \in R_x(t).$ $(w \xrightarrow{\text{po}_t} w' \xrightarrow{\text{po}_t} r \wedge w \xrightarrow{\text{wr}(x)} r)$	(f)	Transaction t reads key x from a write w in transaction $t' \neq t$ which writes x more than once, but w is not the last write on x in t'. $\exists x \in \mathsf{K}.\ \exists t \in \mathsf{RT}_x.\ \exists t' \neq t \in \mathsf{WT}_x.\ \exists r \in R_x(t).$ $\exists w, w' \neq w \in W_x(t').\ (w \xrightarrow{\text{wr}(x)} r \wedge w \xrightarrow{\text{po}_{t'}} w')$.
(g)	The relation $\text{SO} \cup \text{WR}$ is cyclic. $(\text{SO} \cup \text{WR})^+ \cap I_T \neq \emptyset$.	(h)	Transaction t_3 reads y from t_2 and then reads $x \neq y$ from t_1. Transaction t_2 also writes to x but $t_1 \xrightarrow{\text{CO}} t_2$. $\exists x, y \neq x \in \mathsf{K}.\ \exists t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \exists t_3 \in (\mathsf{RT}_x \cap \mathsf{RT}_y) \setminus \{t_1, t_2\}.$ $\exists w_x \in W_x(t_1).\ \exists w_y \in W_y(t_2).\ \exists r_x \in R_x(t_3).\ \exists r_y \in R_y(t_3).$ $(w_x \xrightarrow{\text{wr}(x)} r_x \wedge w_y \xrightarrow{\text{wr}(y)} r_y \wedge r_y \xrightarrow{\text{po}_{t_3}} r_x \wedge t_1 \xrightarrow{\text{CO}} t_2)$.
(i)	Transaction t_3 reads y from t_2 and then reads $x \neq y$ from t_1. Transaction t_2 also writes to x but $t_1 \xrightarrow{\text{CM}} t_2$. This is a general case of (h). $\exists x, y \neq x \in \mathsf{K}.\ \exists t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \exists t_3 \in (\mathsf{RT}_x \cap \mathsf{RT}_y) \setminus \{t_1, t_2\}.$ $\exists w_x \in W_x(t_1).\ \exists w_y \in W_y(t_2).\ \exists r_x \in R_x(t_3).\ \exists r_y \in R_y(t_3).$ $(w_x \xrightarrow{\text{wr}(x)} r_x \wedge w_y \xrightarrow{\text{wr}(y)} r_y \wedge r_y \xrightarrow{\text{po}_{t_3}} r_x \wedge t_1 \xrightarrow{\text{CM}} t_2)$.	(j)	A transaction reads from a key from other transactions more than once, but with different values. $\exists x \in \mathsf{K}.\ \exists v, v' \neq v \in \mathsf{V}.\ \exists t \in \mathsf{RT}_x.\ \exists t_1 \neq t, t_2 \neq t \in \mathsf{WT}_x.$ $\exists r_1 \triangleq \mathsf{R}(x, v), r_2 \triangleq \mathsf{R}(x, v') \in R_x(t).$ $\exists w_1 \in W_x(t_1).\ \exists w_2 \in W_x(t_2).$ $(t_1 \neq t_2 \wedge w_1 \xrightarrow{\text{wr}(x)} r_1 \wedge w_2 \xrightarrow{\text{wr}(x)} r_2)$.
(k)	Transaction t_3 reads x from t_1 and $y \neq x$ from t_2. Transaction t_2 also writes to x but $t_1 \xrightarrow{\text{CO}} t_2$. $\exists x, y \neq x \in \mathsf{K}.\ \exists t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \exists t_3 \in (\mathsf{RT}_x \cap \mathsf{RT}_y) \setminus \{t_1, t_2\}.$ $\exists w_x \in W_x(t_1).\ \exists w_y \in W_y(t_2).\ \exists r_x \in R_x(t_3).\ \exists r_y \in R_y(t_3).$ $(w_x \xrightarrow{\text{wr}(x)} r_x \wedge w_y \xrightarrow{\text{wr}(y)} r_y \wedge t_1 \xrightarrow{\text{CO}} t_2)$.	(l)	Transaction t_3 reads x from t_1 and $y \neq x$ from t_2. Transaction t_2 also writes to x but $t_1 \xrightarrow{\text{CM}} t_2$. This is a general case of (i) and (k). $\exists x, y \neq x \in \mathsf{K}.\ \exists t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \exists t_3 \in (\mathsf{RT}_x \cap \mathsf{RT}_y) \setminus \{t_1, t_2\}.$ $\exists w_x \in W_x(t_1).\ \exists w_y \in W_y(t_2).\ \exists r_x \in R_x(t_3).\ \exists r_y \in R_y(t_3).$ $(w_x \xrightarrow{\text{wr}(x)} r_x \wedge w_y \xrightarrow{\text{wr}(y)} r_y \wedge t_1 \xrightarrow{\text{CM}} t_2)$.
(m)	Transaction t_3 reads x from t_1. There is a transaction t_2 that also writes to x such that $t_1 \xrightarrow{\text{CO}} t_2 \xrightarrow{\text{CO}} t_3$. $\exists x \in \mathsf{K}.\ \exists t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \exists t_3 \in \mathsf{RT}_x \setminus \{t_1, t_2\}.$ $(t_1 \xrightarrow{\text{WR}(x)} t_3 \wedge t_1 \xrightarrow{\text{CO}} t_2 \xrightarrow{\text{CO}} t_3)$.	(n)	Transaction t_3 reads x from t_1. There is a transaction t_2 that also writes to x such that $t_1 \xrightarrow{\text{CM}} t_2 \xrightarrow{\text{CO}} t_3$. This is a general case of (l) and (m). $\exists x \in \mathsf{K}.\ \exists t_1, t_2 \neq t_1 \in \mathsf{WT}_x.\ \exists t_3 \in \mathsf{RT}_x \setminus \{t_1, t_2\}.$ $(t_1 \xrightarrow{\text{WR}(x)} t_3 \wedge t_1 \xrightarrow{\text{CM}} t_2 \xrightarrow{\text{CO}} t_3)$.

3.2 Transactional Anomalous Patterns

Arriving at sound and complete characterizations of HAT isolation levels using fine-grained TAPs is highly non-trivial. While it is relatively straightforward to characterize some anomalies (e.g., *aborted reads* (TAP-b) and *cyclic causal orders* (TAP-g)) omitted in existing definitions (e.g., Definition 2), the challenging part lies in characterizing *all* anomalies to achieve completeness. This task is further complicated by isolation levels' subtle semantic corner cases as well as their divergent interpretations in the literature and in practice. For example, Biswas and Enea [2019] and Bailis et al. [2016] have differing definitions of RA; Elle [Kingsbury and Alvaro 2020] checks Adya's PL-2 [Adya 1999] for RC, which, unlike Biswas and Enea's framework that dbcop [Biswas and Enea 2019] builds on, does not cover monotonic reads (as we will see in Table 5).

Our design of TAPs is an iterative process. We incrementally devise TAPs according to the semantic building blocks for HATs. These building blocks range from committed reads, monotonic reads, atomic visibility, to transactional causality, and also cover both intra-transactional (e.g., repeatable reads) and inter-transactional (e.g., convergence) dependencies. In particular, as we will

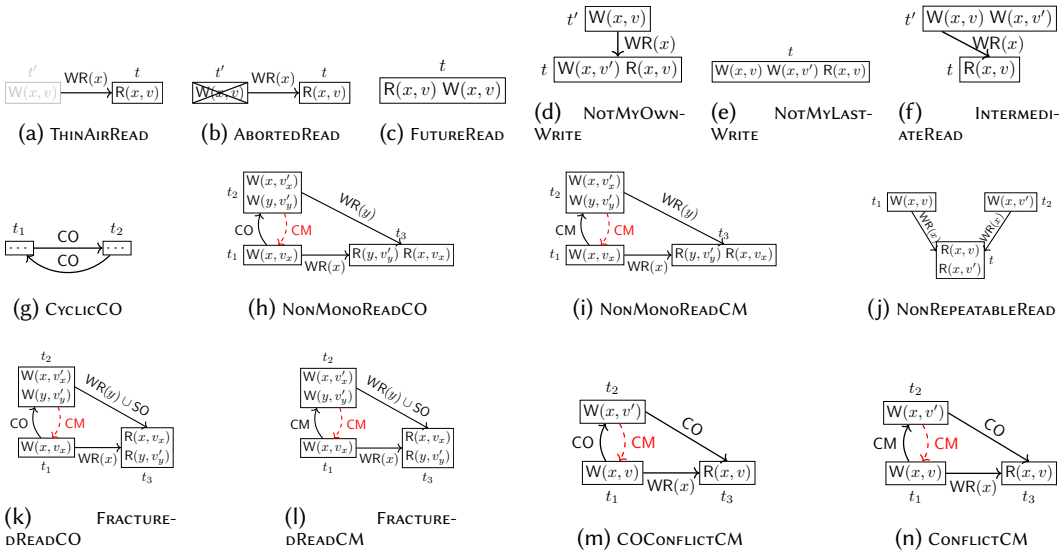


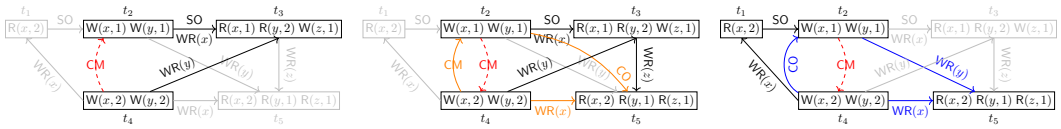
Fig. 3. Visualizing the 14 TAPs. Program order is enforced between horizontally aligned operations within a transaction, e.g., in TAP-c $\mathsf{R}(x, v)$ is executed before $\mathsf{W}(x, v)$. Dashed arrows indicate the CM orders inferred by the axioms in §3.1, which are not part of the TAPs. Irrelevant operations within transactions are omitted.

see (§5.3), many state-of-the-art isolation checkers are built by mainly capturing inter-transactional dependencies, and miss intra-transactional anomalies like non-repeatable reads (labeled "critical" by MariaDB). Moreover, we gradually refine these TAPs using existing formalizations of isolation levels and consistency properties in the literature [Adya 1999; Bailis et al. 2013a; Berenson et al. 1995; Biswas and Enea 2019; Bouajjani et al. 2017; Cerone et al. 2015; Crooks et al. 2017; Xiong et al. 2020] as well as isolation anomalies found in the wild [Biswas and Enea 2019; Huang et al. 2023; Jepsen 2024b]. This iterative design process terminates with our successful formal proofs of soundness and completeness (§3.3); when our attempted proofs failed, we repeated this process.

Table 1 describes our 14 TAPs, along with their formalization. The accompanying visualization is shown in Figure 3. It is worth mentioning that our idea of using TAPs to capture HAT semantics is inspired by (i) the "bad patterns" for formalizing non-transactional causal consistency [Bouajjani et al. 2017] and (ii) the prohibited phenomena for characterizing isolation levels [Adya 1999; Berenson et al. 1995]. Some of our TAPs semantically "overlap" with their anomalies, e.g., cyclic causal orders as in [Bouajjani et al. 2017], and aborted or intermediate reads as in [Adya 1999]. Nonetheless, these prior works do not establish sound and complete characterizations for more recent HAT isolation guarantees such as read atomicity and transactional causal consistency. In contrast, we unify the formalization of various HAT isolation levels using our devised TAPs.

Note also that we base our TAPs on the framework of Biswas and Enea [2019], and some of them can be seen as the "negation" of positive patterns (cf. [Biswas and Enea 2019, Figure 2]). For example, our TAP-m and TAP-n correspond to their causal pattern. For such cases, we refine the negation by distinguishing the version order conflicts, e.g., via causal transitive dependencies (TAP-m) or convergence (TAP-n). This provides users with greater insight into the violations because these properties are often developed with different mechanisms, e.g., logical timestamping for causality in constrast to conflict resolution for data convergence [Liu et al. 2024b; Lloyd et al. 2013].

Nonetheless, simple negation is insufficient to achieve our design goal of fine-grained, sound, and complete characterizations of various isolation levels. For instance, to characterize RC, we decouple



(a) Version order from t_4 to t_2 (b) ConflictCM (TAP-n) (c) FracturedReadCO (TAP-k)

Fig. 4. TAP-n and TAP-k coexist in a single history. We shade the irrelevant parts in each case.

Table 2. Characterizing four HAT isolation levels via TAPs.

Isolation Level	Prohibited TAPs
Transactional Causal Consistency	All 14 TAPs
Read Atomicity	TAP-a to TAP-l (12 TAPs)
Read Committed	TAP-a to TAP-i (9 TAPs)
Cut Isolation	TAP-j

non-monotonic reads from Biswas and Enea's semantics of RC and augment them with several intra-transactional anomalies defined over program orders of operations within transactions. We also decouple non-repeatable reads from their RA for characterizing cut isolation. These decompositions allow users to check isolation guarantees by simply combining their desired TAPs; this is, in particular, motivated by the aforementioned divergence of interpreting isolation levels.

Our fine-grained TAPs are designed to facilitate understanding and debugging the violations found (which are therefore not necessarily minimal for defining a specific isolation level). However, solver-based tools return incomprehensible unsatisfied clause in general. Non-solver tools may return uninformative cycles. Note that dbcop, which bases its cycle detection also on the axioms in [Biswas and Enea 2019], cannot straightforwardly distinguish non-monotonic reads (TAP-h) and fractured reads (TAP-k). However, these two kinds of anomalies are typically relevant to different concurrency control protocols [Bailis et al. 2016].

Below we showcase two kinds of TAPs using a single history. As we will see (§5.3), many histories contain multiple TAPs. Our checker can identify *all* of them, while most of the existing checkers miss certain anomalies.

Example 1. Figure 4 illustrates both TAP-n and TAP-k in a single history. In Figure 4a, as $t_4 \xrightarrow{\mathrm{WR}(y)} t_3$, $t_2 \xrightarrow{\mathrm{WR}(x)} t_3$, and both t_2 and t_4 write to x, we have $t_4 \xrightarrow{\mathrm{CM}} t_2$ (according to the *ReadAtomic* Axiom). In Figure 4b, $t_2 \xrightarrow{\mathrm{CO}} t_5$ (due to $t_2 \xrightarrow{\mathrm{SO}} t_3 \xrightarrow{\mathrm{WR}(z)} t_5$), $t_4 \xrightarrow{\mathrm{WR}(x)} t_5$, and both t_2 and t_4 write to x. Hence, we have $t_2 \xrightarrow{\mathrm{CM}} t_4$ (according to the *Causal* Axiom) and a cycle in CM, i.e., $t_4 \xrightarrow{\mathrm{CM}} t_2 \xrightarrow{\mathrm{CM}} t_4$. Conflicting version orders arise. In Figure 4c, we have $t_4 \xrightarrow{\mathrm{CO}} t_2$ due to $t_4 \xrightarrow{\mathrm{WR}(x)} t_1 \xrightarrow{\mathrm{SO}} t_2$. We also have $t_4 \xrightarrow{\mathrm{WR}(x)} t_5$, $t_2 \xrightarrow{\mathrm{WR}(y)} t_5$, and both t_2 and t_4 write to x. Hence, we have $t_2 \xrightarrow{\mathrm{CM}} t_4$ (according to the *ReadAtomic* Axiom) and another cycle.

3.3 TAP-based Sound and Complete Characterizations

Owing to their modularity, we characterize four prevalent HAT isolation levels, i.e., read committed, cut isolation, read atomicity, and transactional causal consistency, with varying combinations of TAPs. Table 2 shows our characterizations. Below we prove that our TAP-based characterizations of these isolation levels are sound and complete with respect to their formal definitions (§3.1).

Theorem 2 (Soundness and Completeness for CI). A history $(T, \mathrm{SO}, \mathrm{WR})$ satisfies CI if and only if it does not contain any instances of TAP-j.

Proof. This is straightforward by Definition 3. □

Theorem 3 (Soundness and Completeness for RC). A history satisfies RC if and only if it does not contain any instances of TAP-a to TAP-i.

Proof. Let $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$ be a history. Since TAP-h is a special case of TAP-i, we show that $\mathcal{H}$ satisfies RC if and only if it does not contain any instances of TAP-a to TAP-g and TAP-i.

($\Longrightarrow$) Assume that $\mathcal{H}$ satisfies RC. We examine each of the TAPs as follows.

- (TAP-a) ThinAirRead: Subsumed by the definition of history (Definition 2).
- (TAP-b) AbortedRead: Subsumed by the definition of history (Definition 2).
- (TAP-c) FutureRead: Forbidden by RC-(1).
- (TAP-d) NotMyOwnWrite: Forbidden by RC-(2).
- (TAP-e) NotMyLastWrite: Forbidden by RC-(2).
- (TAP-f) IntermediateRead: Forbidden by RC-(3).
- (TAP-g) CyclicCO: Subsumed by the definition of history (Definition 2).
- (TAP-i) NonMonoReadCM: Suppose by contradiction that $\mathcal{H}$ (with the commit order CM) contains an instance of TAP-i with t_1, t_2, and t_3 as in Figure 3i. By the *MonoAtomicView* axiom, $t_2 \xrightarrow{\mathrm{CM}} t_1$. Since $t_1 \xrightarrow{\mathrm{CM}} t_2$ in this instance of TAP-i, we have $t_1 \xrightarrow{\mathrm{CM}} t_2 \xrightarrow{\mathrm{CM}} t_1$, contradicting that CM is a strict total order.

($\Longleftarrow$) We first show that a pair (T, SO) that contains no instances of TAP-a, TAP-b, and TAP-g is indeed a valid history. That is, we show that for such a pair (T, SO), there exists a write-read relation that satisfies the two requirements in Definition 2. We define WR as follows: for each key x, $\mathrm{WR}(x)$ associates each transaction that reads value v of x with the write transaction that writes v to x. Since TAP-a and TAP-b are forbidden, such a write transaction must exist in T. By the assumption that for each key, every write to the key assigns a unique value, the write transaction is unique. Hence, $\mathrm{WR}(x)^{-1}$ is a total function. Since TAP-g is forbidden, $\mathrm{SO} \cup \mathrm{WR}$ is acyclic.

Next we show that if the history $(T, \mathrm{SO}, \mathrm{WR})$ does not contain any instances of TAP-c, TAP-d, TAP-e, and TAP-f, then it satisfies RC-(1), RC-(2), and RC-(3). First, if RC-(1) is violated, then TAP-c would happen. Second, if RC-(2) is violated, then either TAP-d or TAP-e would happen. Third, if RC-(3) is violated, then TAP-f would happen.

Finally we show that if the history $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$ does not contain any instances of TAP-i, then the *MonoAtomicView* axiom holds. Suppose by contradiction that $\mathcal{H}$ (with the commit order CM) violates the *MonoAtomicView* axiom. Therefore, there exist three transactions t_1, t_2, and t_3 such that both t_1 and t_2 write to the same key x, t_3 reads y from t_2 and then reads $x \neq y$ from t_1, but $t_1 \xrightarrow{\mathrm{CM}} t_2$. However, this is an instance of TAP-i. □

Theorem 4 (Soundness and Completeness for RA). A history satisfies RA if and only if it does not contain any instances of TAP-a to TAP-l.

Proof. Let $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$ be a history. Since TAP-h, TAP-i, and TAP-k are special cases of TAP-l, in the following we show that $\mathcal{H}$ satisfies RA if and only if it does not contain any instances of TAP-a to TAP-g, TAP-j, and TAP-l.

($\Longrightarrow$) Assume that $\mathcal{H}$ satisfies RA. By Theorem 3 and the fact that RA is stronger than RC (Definition 5), TAP-a to TAP-g are all forbidden by RA. Moreover, by Theorems 1 and 2, TAP-j is also disallowed by RA. Now we show that TAP-l is forbidden by RA as well. Suppose by contradiction that $\mathcal{H}$ contains an instance of TAP-l with t_1, t_2, and t_3 as in Figure 3l. By the *ReadAtomic* axiom,

$t_2 \xrightarrow{\text{CM}} t_1$. Since $t_1 \xrightarrow{\text{CM}} t_2$ in this instance of TAP-l, we have $t_1 \xrightarrow{\text{CM}} t_2 \xrightarrow{\text{CM}} t_1$, contradicting that CM is a strict total order.

($\Longleftarrow$) Suppose by contradiction that $\mathcal{H}$ (with the commit order CM) violates the *ReadAtomic* axiom. Therefore, there exist three transactions t_1, t_2, and t_3 such that t_3 reads a key x from t_1, t_2 also writes to x such that $t_2 \xrightarrow{\text{SO} \cup \text{WR}} t_3$, but $t_1 \xrightarrow{\text{CM}} t_2$. However, this is an instance of TAP-l. □

Theorem 5 (Soundness and Completeness for TCC). A history satisfies TCC if and only if it does not contain any instances of all the 14 TAPs.

Proof. Let $\mathcal{H} = (T, \text{SO}, \text{WR})$ be a history. Since TAP-h, TAP-i, and TAP-k to TAP-m are special cases of TAP-n, in the following we show that $\mathcal{H}$ satisfies TCC if and only if it does not contain any instances of TAP-a to TAP-g, TAP-j, and TAP-n.

($\Longrightarrow$) Assume that $\mathcal{H}$ satisfies TCC. By Theorems 1 and 4, TAP-a to TAP-g and TAP-j are all forbidden by TCC. Now we show that TAP-n is also forbidden by TCC. Suppose by contradiction that $\mathcal{H}$ contains an instance of TAP-n with t_1, t_2, and t_3 as in Figure 3n. By the *Causal* axiom, $t_2 \xrightarrow{\text{CM}} t_1$. Since $t_1 \xrightarrow{\text{CM}} t_2$ in this instance of TAP-n, we have $t_1 \xrightarrow{\text{CM}} t_2 \xrightarrow{\text{CM}} t_1$, contradicting that CM is a strict total order.

($\Longleftarrow$) Suppose by contradiction that $\mathcal{H}$ (with the commit order CM) violates the *Causal* axiom. Therefore, there exist three transactions t_1, t_2, and t_3 such that t_3 reads a key x from t_1, t_2 also writes to x such that $t_2 \xrightarrow{\text{CO}} t_3$, but $t_1 \xrightarrow{\text{CM}} t_2$. However, this is an instance of TAP-n. □

4 The Checking Algorithm

In this section we present our algorithm for checking weak HAT isolation guarantees. It builds on our TAP-based sound and complete characterizations (§3), leveraging vectors and tree clocks to expedite the checking process.

4.1 Overview

Our algorithm detects instances of *all* TAPs in a history by default, which is equivalent to checking TCC. One can also configure it to check individual TAPs or desired combinations of them such as read committed and read atomicity. Algorithm 1 shows its pseudocode. Our checking algorithm *incrementally* builds a directed transactional dependency graph with a node set of the transactions involved and an edge set obtained from their SO, WR, CO, and CM dependencies, and it identifies the TAPs in the graph. More specifically, the algorithm consists of the following four main steps, as also shown in the procedure IsoCheck (line 1).

(1) BuildCO (line 2) constructs CO from SO and WR. For instance, in the Figure 5 history, the algorithm obtains $t_{101} \xrightarrow{\text{CO}} t_{100}$ based on the SO and WR edges along the path, which can be extracted straightforwardly from the history.
(2) CheckCOTAP (line 3) checks the CO-related patterns TAP-a to TAP-h, TAP-j, TAP-k, and TAP-m. For example, there are no instances of these patterns in the Figure 5 history.
(3) BuildCM (line 4) builds the CM edges based on the axioms in §3.1, e.g., $t_{101} \xrightarrow{\text{CM}} t_2$ according to the *Causal* axiom (Definition 6).
(4) CheckCMTAP (line 5) examines the remaining three CM-related patterns TAP-i, TAP-l, and TAP-n. The history in Figure 5 is also free of these patterns.

Checking TAPs in Steps (2) and (4) is essentially a graph pattern matching problem. This involves reachability checking due to the transitivity of CO and CM, as well as cycle detection for TAPs like TAP-g. Nonetheless, such graph traversals can be significantly time-consuming under workloads of high concurrency. These workloads are, however, commonly used to stress test databases.

Leveraging Vectors. Inspired by the successful applications of *vectors* [Fidge 1988] in designing databases [Didona et al. 2018; Liu et al. 2024b; Lloyd et al. 2013; Lu et al. 2020], we leverage vectors to accelerate our graph traversals by capturing the transitive dependencies among transactions in a dependency graph. This helps avoid unnecessary graph traversals.

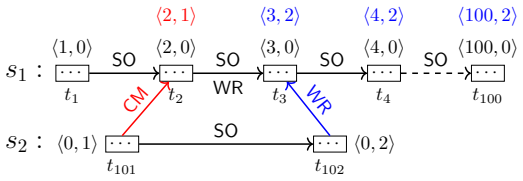


Fig. 5. Illustrating early stop during vector updating. Two sessions s_1 and s_2 are horizontally aligned. For simplicity, we omit the operations inside transactions.

Our algorithm associates each transaction t in the dependency graph with a vector $Vec[t]$, one element per session. Intuitively, $Vec[t]$ encodes the set of transactions that can reach t. Vectors are compared element-wise (similarly for $\sqsupseteq$): $vec_1 \sqsubseteq vec_2 \iff \forall s \in \mathbb{S}.\ vec_1(s) \leq vec_2(s)$, where $\mathbb{S}$ is the set of sessions and $vec(s)$ projects vec's element on session s. The join operator $\sqcup$ on vectors is defined as $vec_1 \sqcup vec_2 \triangleq \langle \lambda s \in \mathbb{S}.\ \max\{vec_1(s), vec_2(s)\}\rangle$.

During the incremental dependency graph construction, our algorithm maintains an important *invariant* on vectors: Transaction t_2 is reachable from t_1 if and only if $Vec[t_1] \sqsubseteq Vec[t_2]$. Cycle detection then amounts to testing whether there exist two transactions t_1 and t_2 such that $Vec[t_1] = Vec[t_2]$. In order to maintain this invariant, whenever an edge outgoing from a transaction t is added, we join $Vec[t]$ with the vectors of transactions reachable from t (line 30). We realize this via DFS traversals (line 27 in the UpdateVec procedure). Notably, owing to this invariant, a traversal can *backtrack early* once it reaches a transaction t' with a larger vector, i.e., $Vec[t'] \sqsupseteq Vec[t]$ (line 28). This is because the vectors of downstream transactions are at least as large as $Vec[t']$.

Example 2 (UpdateVec with early stop)**.** Figure 5 shows a history where two sessions s_1 and s_2 contain 100 and 2 transactions, respectively. Suppose that all the SO and WR edges, along with their vectors (in black), have been constructed except for the edge $t_{102} \xrightarrow{\text{WR}} t_3$. After this edge is added, the second component of the vectors associated with t_3, t_4, ..., t_{100} are set to 2 (in blue). The CM edge $t_{101} \xrightarrow{\text{CM}} t_2$ is then added. As a result, we should update the vectors of downstream transactions t_2, t_3, ..., t_{100} with $Vec[t_{101}] = \langle 0, 1\rangle$. Transaction t_2's vector is thus updated to $\langle 2, 1\rangle$ (in red). However, as $Vec[t_3] = \langle 3, 2\rangle \sqsupseteq Vec[t_{101}]$, the traversal can safely terminate early at t_3.

4.2 Algorithm

We now elaborate on the details of our checking algorithm.

Procedure BuildCO. Our algorithm builds CO by examining transactions individually. Consider a transaction t. It first builds SO and initializes $Vec[t]$. It then examines each operation of t and builds WR. Finally, it calls the procedure UpdateVec (line 26) where it uses $Vec[t]$ to update the vectors of transactions reachable from t in case new edges outgoing from t have been added. This ensures the transitivity of CO. Denote the resulting graph by $\mathcal{G}_{\text{CO}}$.

Procedure CheckCOTAP. Our algorithm checks 11 CO-related TAPs on $\mathcal{G}_{\text{CO}}$ (line 6). It is straightforward to check the patterns a to f and j (line 8); we omit the details. Checking the patterns g, h, k, and m involves reachability checking and cycle detection in terms of CO. Specifically, we detect a cycle in TAP-g if there exist two equivalent vectors (line 9). Patterns h, k, and m contain a $t_1 \xrightarrow{\text{CO}} t_2 \xrightarrow{\text{CO}} t_3$ chain, which is captured by $Vec[t_1] \sqsubseteq Vec[t_2] \sqsubseteq Vec[t_3]$ (line 11). They differ in whether $t_2 \xrightarrow{\text{CO}} t_3$ is actually a special case $t_2 \xrightarrow{\text{SO}} t_3$ (line 12 for TAP-k) or $t_2 \xrightarrow{\text{WR}(y)} t_3$ for some key y (line 14 for TAP-h and TAP-k); otherwise, it is TAP-m (line 20). Note that, as a special case of TAP-k, TAP-h further requires that t_3 reads y from t_2 *before* it reads x from t_1 (line 15).

Algorithm 1 Detecting TAPs in $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$ (the full version is given in [Liu et al. 2024a, Appendix A])

WT_x: the set of transactions that write to key x
RT_x^v: the set of transactions that read value v of key x
$Vec[t]$: the vector for transaction t
taps: the set of TAPs found, initially $\emptyset$

1: **procedure** IsoCheck()
2: BuildCO() ▷ see [Liu et al. 2024a, Appendix A] for details
3: CheckCOTAP() ▷ TAP-a to TAP-h, TAP-j, TAP-k, and TAP-m
4: BuildCM()
5: CheckCMTAP() ▷ TAP-i, TAP-l, and TAP-n
6: **procedure** CheckCOTAP()
7: **for** $t \in T$
8: check TAP-a to TAP-f and TAP-j ▷ details omitted
9: **if** $\exists t_1, t_2 \in T.\ Vec[t_1] = Vec[t_2]$
10: $taps \leftarrow taps \cup \{\text{TAP-g}\}$ ▷ CyclicCO
11: **if** $\exists x \in \mathsf{K}.\ \exists t_1, t_2, t_3 \in T.\ t_1 \xrightarrow{\mathrm{WR}(x)} t_3 \wedge t_2 \in \mathsf{WT}_x \wedge Vec[t_1] \sqsubseteq Vec[t_2] \sqsubseteq Vec[t_3]$
12: **if** $t_2 \xrightarrow{\mathrm{SO}} t_3$
13: $taps \leftarrow taps \cup \{\text{TAP-k}\}$ ▷ FracturedReadCO
14: **else if** $\exists y \neq x \in \mathsf{K}.\ t_2 \xrightarrow{\mathrm{WR}(y)} t_3$
15: **if** $\mathsf{R}(y, \_)$ is before $\mathsf{R}(x, \_)$ in t_3
16: $taps \leftarrow taps \cup \{\text{TAP-h}\}$ ▷ NonMonoReadCO
17: **else**
18: $taps \leftarrow taps \cup \{\text{TAP-k}\}$ ▷ FracturedReadCO
19: **else**
20: $taps \leftarrow taps \cup \{\text{TAP-m}\}$ ▷ COConflictCM

21: **procedure** BuildCM()
22: **for** $t_1, t_2, t_3 \in T$ s.t. $\exists x \in \mathsf{K}.\ t_1 \xrightarrow{\mathrm{WR}(x)} t_3 \wedge t_2 \in \mathsf{WT}_x \wedge Vec[t_2] \sqsubseteq Vec[t_3]$
23: $E \leftarrow E \cup \{(t_2, t_1)\}$
24: **for** $t \in T$
25: UpdateVec(t)

26: **procedure** UpdateVec(t)
27: **for** $t' \in T$ in DFS traversal from t
28: **if** $Vec[t'] \sqsupseteq Vec[t]$
29: **backtrack**
30: $Vec[t'] \leftarrow Vec[t'] \sqcup Vec[t]$

31: **procedure** CheckCMTAP()
32: **if** $\exists x \in \mathsf{K}.\ \exists t_1, t_2, t_3 \in T.\ t_1 \xrightarrow{\mathrm{WR}(x)} t_3 \wedge t_2 \in \mathsf{WT}_x \wedge Vec[t_1] = Vec[t_2]$
33: **if** $t_2 \xrightarrow{\mathrm{SO}} t_3$
34: $taps \leftarrow taps \cup \{\text{TAP-l}\}$ ▷ FracturedReadCM
35: **else if** $\exists y \neq x \in \mathsf{K}.\ t_2 \xrightarrow{\mathrm{WR}(y)} t_3$
36: **if** $\mathsf{R}(y, \_)$ is before $\mathsf{R}(x, \_)$ in t_3
37: $taps \leftarrow taps \cup \{\text{TAP-i}\}$ ▷ NonMonoReadCM
38: **else**
39: $taps \leftarrow taps \cup \{\text{TAP-l}\}$ ▷ FracturedReadCM
40: **else if** $t_2 \xrightarrow{\mathrm{CO}} t_3$
41: $taps \leftarrow taps \cup \{\text{TAP-n}\}$ ▷ ConflictCM

Procedure BuildCM. The remaining three anomalous patterns, i.e., TAP-i, TAP-l, and TAP-n, all contain CM edges. In BuildCM (line 21), our algorithm examines each tuple of transactions t_1, t_2, and t_3. If $t_1 \xrightarrow{\mathrm{WR}(x)} t_3$, t_2 writes x, and $t_2 \xrightarrow{\mathrm{CO}} t_3$ (in $\mathcal{G}_{\mathrm{CO}}$), it adds an edge $t_2 \xrightarrow{\mathrm{CM}} t_1$. The transitivity of CM is ensured by calling UpdateVec to update the associated vectors (line 25). Note that we *cannot* update vectors and add CM edges alternately; otherwise from $Vec[t_2] \sqsubseteq Vec[t_3]$ we would conclude $t_2 \xrightarrow{\mathrm{CM}} t_3$, while $t_2 \xrightarrow{\mathrm{CO}} t_3$ is actually desired (line 28). Denote the resulting graph by $\mathcal{G}_{\mathrm{CM}}$.

Procedure CheckCMTAP. TAP-i, TAP-l, and TAP-n all contain three transactions such that $t_1 \xrightarrow{\mathrm{WR}(x)} t_3$ for some key x, t_2 writes x, and t_1 and t_2 form a CM cycle in $\mathcal{G}_{\mathrm{CM}}$ that can be detected by $Vec[t_1] = Vec[t_2]$ (line 32). They differ in whether the condition $t_2 \xrightarrow{\mathrm{CO}} t_3$ is actually a special case $t_2 \xrightarrow{\mathrm{SO}} t_3$ (line 33 for TAP-l) or $t_2 \xrightarrow{\mathrm{WR}(y)} t_3$ for some key y (line 35 for TAP-i and TAP-l); otherwise, it is TAP-n (line 41). As a special case of TAP-l, TAP-i further requires that t_3 reads y from t_2 *before* it reads x from t_1 (line 36). Note also that, for TAP-n, we cannot replace $t_2 \xrightarrow{\mathrm{CO}} t_3$ with $Vec[t_2] \sqsubseteq Vec[t_3]$, since the latter implies $t_2 \xrightarrow{\mathrm{CM}} t_3$ in $\mathcal{G}_{\mathrm{CM}}$ which is not necessarily $t_2 \xrightarrow{\mathrm{CO}} t_3$.[4]

4.3 Optimization with Tree Clocks

Our algorithm frequently calls the UpdateVec procedure to update vectors where it compares and joins vectors. Both operations on vectors require $O(s)$ time, with s the number of client sessions. UpdateVec would become a computational bottleneck when s is large, which is however desirable for stress testing or validation of databases.

To reduce the time spent manipulating vectors, we adapt the *tree clock* [Mathur et al. 2022], a recent advance in dynamic concurrent program analysis, in our utilization of vectors. Manipulating

[4] In our implementation, we maintain dual vectors to capture reachability in $\mathcal{G}_{\mathrm{CO}}$ and $\mathcal{G}_{\mathrm{CM}}$, respectively. In this case, we can however replace $t_2 \xrightarrow{\mathrm{CO}} t_3$ with $Vec[t_2] \sqsubseteq Vec[t_3]$ *only* in $\mathcal{G}_{\mathrm{CO}}$.

tree clocks is more efficient: comparing tree clocks takes constant time and joining them takes time proportional to the number of entries being modified.

However, in our setting, transactions may be processed in the order *inconsistent* with their happens-before ordering [Lamport 1978]. For instance, transaction t_1 may appear *after* t_2 in the history even if $t_1 \xrightarrow{\text{WR}} t_2$. As a result, t_2's tree clock must be updated when t_1 is processed. This is different from the concurrent program setting where, once a tree clock is computed, it will not be updated later. Thanks to the vector invariant (§4.1) maintained by our algorithm, safely terminating the updating of tree clocks downstream can occur early.

The following example illustrates how our algorithm uses tree clocks in updating vectors.

Example 3. Figure 6 show a history with four sessions. Suppose in BuildCO our algorithm scans the transactions in the order of t_1, t_2, t_4, t_5, t_6, t_7, t_8, and finally t_3. When dealing with t_3, it adds the edge $t_3 \xrightarrow{\text{WR}(z)} t_7$. As a result, it updates t_7's clock, along with those of the downstream transactions (in red). With simple vectors, the join operation iterates over *all* entries of $Vec[t_3]$ and $Vec[t_7]$. However, via $t_5 \xrightarrow{\text{WR}(z)} t_6 \xrightarrow{\text{SO}} t_7$ (in blue), t_7 has learned t_5 on session s_2, which is newer than what t_3 learned about session s_2 (in orange), namely t_2. Transitively, via $t_4 \xrightarrow{\text{WR}(z)} t_5$ (in blue), t_7 has learned t_4 on session s_1, which is newer than what t_3 learned, via $t_1 \xrightarrow{\text{WR}(x)} t_2$ (in orange), about session s_1, namely t_1. By recording a transaction's knowledge of other transactions across sessions in a hierarchical tree structure, tree clocks can avoid examining certain entries of vectors. Here, joining $Vec[t_3]$ with $Vec[t_7]$, and further with $Vec[t_8]$, does not examine the entry for s_1.

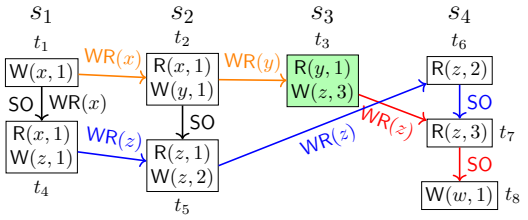


Fig. 6. An illustration of tree clocks for updating vectors. The four sessions are vertically aligned.

4.4 Time Complexity

Consider a history that involves n transactions, s sessions, and k keys. The worst-case time complexity of our algorithm stems mainly from two procedures for building CM and checking TAPs (particularly, TAP-h, TAP-i, TAP-k, and TAP-l), which take $O(n^3 \max(k, s))$ and $O(n^3k^2s)$ time, respectively. As we will see in §5, despite these worst-case times, our algorithm is very efficient in practice. This is due to the use of vectors and tree clocks, along with the common characteristics of database workloads—in particular, the relatively sparse transactional dependency graphs arising under large workloads and the rarity of isolation bugs.

First, by utilizing tree clocks, the time complexity of checking TAPs can be reduced from $O(n^3k^2s)$ to $O(n^3k^2)$. Additionally, we check the acyclicity of a dependency graph before identifying specific TAPs. As bugs are rare, we often skip the latter step. Consequently, the time complexity is further reduced to $O(n^2)$ for cycle detection.

Moreover, building CM consists of (i) computing the CM edges and (ii) updating the vectors of downstream transactions. The time complexity of (i) is $O(n * |\text{WR}|)$, with $|\text{WR}|$ the number of WR edges in a dependency graph. Since a WR edge involves two transactions and one key, $|\text{WR}|$ is at most n^2k. For common database workloads, $|\text{WR}| \ll n^2k$. Hence, the complexity of (i) is significantly less than $O(n^3k)$, i.e., $n * |\text{WR}| \ll n * (n^2k) = n^3k$. The time complexity of (ii) is $O(n * (n' + m') * s)$, where n' and m' are the numbers of transactions and dependencies downstream, respectively. With vectors, our algorithm stops early during downstream traversals most of the

time. Therefore, n' and m' are fairly small, and the time complexity of (ii) is much less than $O(n^3 s)$, i.e., $n * (n' + m') * s \ll n * n^2 * s = n^3 s$.

5 Experiments

We have implemented our algorithm in a tool called Plume in around 3k LOC in Java. We conduct a comprehensive evaluation of Plume and the state-of-the-art checkers, and answer the following research questions in terms of the **SIEGE+** principle [Huang et al. 2023] for guiding the design of black-box isolation checkers.

(1) **E**ffective: Can Plume detect isolation bugs in production database engines?
(2) **+** or *complete*: Can Plume identify all isolation anomalies in a history? How about the existing checkers?
(3) **I**nformative: Are the violations reported by Plume understandable?
(4) **E**fficient: Can Plume outperform the baselines under various workloads and is it scalable for large workloads?

Along with answering these questions, we also show that Plume returns no false positives (**S**ound) and can analyze various transaction histories (**G**eneral). We focus on Plume's default mode that detects all 14 TAPs corresponding to TCC, unless explicitly stated otherwise.

5.1 Workloads, Benchmarks, and Setup

Workloads. We incorporate the parametric workload generator of PolySI [Huang et al. 2023] into Plume to produce general transaction workloads. The parameters are: the number of client sessions (#sess=25 by default), the number of transactions per session (#txns/sess=200 by default), the number of read/write operations per transaction (#ops/txn=20 by default), the read proportion (50% by default), the total number of keys (10k by default), and the key-access distribution including uniform (by default), zipfian, and hotspot (80% operations touching 20% keys). For the performance comparison, the default 5k transactions with 100k operations suffice to distinguish Plume from other tools. We further demonstrate Plume's scalability with significantly larger workloads.

Benchmarks. We consider six benchmarks commonly used by existing black-box isolation checkers [Huang et al. 2023; Tan et al. 2020; Zhang et al. 2023]. These include three macrobenchmarks, each containing only serializable histories of at least 10k transactions. TPC-C [TPC 2023] is a standard benchmark for online transaction processing. The configuration includes one warehouse, 10 districts, and 30k customers, with five types of transactions. C-RUBiS [RUBiS 2023] is an eBay-like bidding system. The dataset contains 20k users and 200k items. C-Twitter [Kallen 2023] is a Twitter clone where key accesses follow the zipfian distribution.

Three representative *general* datasets [Huang et al. 2023] are also included: GeneralRH (read-heavy with 95% reads), GeneralWH (write-heavy with 70% writes), and GeneralRW (50% reads). Each history contains 16k transactions and 320k operations.

Experimental Setup. We use a PostgreSQL v15.2 instance to produce *valid* histories without snapshot isolation, thus TCC, violations [PostgreSQL 2023]. We co-locate the client threads (or sessions) and target database on a local machine with an Intel(R) Xeon(R) E5-2620 v3 CPU and 64GB memory. For checking TCC over a history, we set the timeout to 10 minutes, which suffices to distinguish Plume from the other tools.

5.2 Detecting Isolation Bugs

Rediscovering Known Bugs. Plume has reproduced *all* known TCC bugs in a substantial collection of 3097 anomalous histories archived by prior work [Biswas and Enea 2019; Huang et al. 2023; Zennou et al. 2022; Zhang et al. 2023]. These histories were collected from the earlier releases of

Table 3. Summary of tested database engines.

Database	GitHub Stars	Kind	Tested Version	Tested Level	Lowest Lvl Violated	Anomalous History Size	Checking Time
AntidoteDB	830	Key-value	0.2.2	TCC	RC	#sess=6, #txns/sess=20, #ops/txn=20	17ms
MariaDB-Galera	5.8k	Relational	10.4.22	SI	CI, RC	#sess=10, #txns/sess=100, #ops/txn=25	491ms
YugabyteDB	8.7k	Multi-model	2.11.1	SI	RC	#sess=2, #txns/sess=10, #ops/txn=10	1ms

four databases, namely MySQL-Galera, Dgraph, MongoDB, and CockroachDB. During the process of bug rediscovery, we observe that weaker isolation bugs already manifest extensively when checking the stronger isolation guarantees offered by these databases. For example, MongoDB (resp. CockroachDB), claiming to provide snapshot isolation (resp. serializability), violated much weaker isolation guarantees like RC (resp. RA). This new finding owes to our fine-grained TAPs that correspond to *weaker* isolation anomalies. Moreover, Plume also demonstrates high efficiency in detecting these bugs. For instance, it successfully identifies all bugs in over 2k anomalous CockroachDB histories, with 500k transactions and 12 million operations, in around 14 seconds. See [Liu et al. 2024a, Appendix B]) for details.

New Anomalies Found. We also demonstrate Plume's effectiveness by examining recent releases of three popular and heavily-tested databases of *different kinds*, i.e., the emerging key-value database AntidoteDB [AntidoteDB 2023] (atop which ElectricSQL [ElectricSQL 2023] builds its core replication), the relational database MariaDB-Galera [Cluster 2023], and YugabyteDB supporting multiple data models [YugabyteDB 2023]. All three databases claim to provide TCC or even stronger isolation guarantees. Hence, checking these isolation levels can provide a full view of which HAT isolation anomalies can actually occur. Table 3 provides details on these databases including their kinds, the tested releases, tested isolation levels, the lowest level violated, etc.

We have detected and reported new isolation bugs in all these database engines. In particular, MariaDB labeled the bug as "critical" since it affected a wide range of nine releases (v10.3–v10.11), as well as its Galera cluster. This bug had existed for over six years until Plume found it. AntidoteDB quickly localized the root cause of our reported anomaly (a client issue with the session guarantee support). These two issues have been fixed. YugabyteDB labels our reported bug as "medium" and is investigating it. Note that Plume also finds weaker isolation bugs (e.g., for CI or RC) in the three databases, despite their claimed much stronger isolation guarantees. The anomalous histories for these bugs exhibit a moderate size, and Plume completes the checking process instantly.

We also found that many different anomalies exist in these databases: seven types of TAPs in MariaDB-Galera (TAP-f, TAP-h, TAP-i, TAP-j, TAP-k, TAP-l, and TAP-n), five types in YugabyteDB (TAP-i, TAP-k, TAP-l, TAP-m, and TAP-n), and four types in AntidoteDB (TAP-g, TAP-h, TAP-k, and TAP-m). This finding is not unusual as we will see in §5.3.

5.3 Finding All Anomalies

An ideal checker will identify *all* anomalies in a given history. This is highly desirable as anomaly occurrences are unpredictable due to randomized workload generation. Our analysis shows that (i) histories can actually exhibit significantly more anomalies than expected, as captured by Plume, while (ii) the current implementations of many existing checkers inherently miss some of them due to the underlying incomplete characterizations of isolation levels. Therefore, simply running these checkers multiple times for the same histories or even making them continue their search would not solve the problem.

State-of-the-Art Isolation Checkers. Our comparison considers five existing isolation checkers.

Table 4. Distribution of TAPs identified by Plume from 3097 anomalous histories archived by existing checkers and 3 histories corresponding to the new isolation bugs found.

TAP-a	TAP-b	TAP-c	TAP-d	TAP-e	TAP-f	TAP-g	TAP-h
0	3	1	71	0	1	2	103
TAP-i	TAP-j	TAP-k	TAP-l	TAP-m	TAP-n	**#TAPs**	**#Hist**
2928	14	1655	2919	3078	2972	**13747**	**3100**

- dbcop [Biswas and Enea 2019] supports checking multiple isolation levels. Given a history of transactions, it constructs a transaction dependency graph by Biswas and Enea [2019] and searches for a cycle (or violation). We consider three of its checking components for RC, RA, and TCC, respectively.
- PolySI [Huang et al. 2023], Viper [Zhang et al. 2023],[5] and Cobra [Tan et al. 2020] are all based on SMT solving, utilizing the MonoSAT solver [Bayless et al. 2015] for efficiently checking graph properties such as acyclicity. These checkers are designed for validating *stronger* isolation levels (PolySI and Viper for SI, and Cobra for serializability), which can also be used for detecting HAT isolation bugs.
- Elle, as part of the testing framework Jepsen [2024a], supports checking a wide range of isolation guarantees, including RC, RA, and TCC [Elle 2023].[6] It builds mainly on Adya's formalism [Adya 1999] and also includes more recent characterizations [Cerone et al. 2015; Cerone and Gotsman 2018]. We test its well-known list-append mode (version 0.1.6) that leverages "append" operations to efficiently infer version orders.

Finding All Anomalies with Plume. Owing to its underlying complete characterization, Plume can detect *all* TCC anomalies (or TAPs) in a history. Surprisingly, the number of identified TAPs (13747 in total) is far more than the number of histories (3100). Table 4 shows the distribution of these TAPs. Most of them involve *non-monotonic reads* (TAP-i), *fractured reads* (TAP-k and TAP-l), or *causality violations* (TAP-m and TAP-n). Moreover, the number of simple, yet crucial bugs are non-negligible (92 in total). For example, 14 *non-repeatable reads* are found (TAP-j); many transactions could not read their own writes (TAP-d, 71 in total); some transactions even read aborted writes (TAP-b, 3 in total). This indicates the importance for isolation checkers to also consider *intra-transaction* dependencies.

(In)completeness. Many checkers report the first or the first few bugs they detect in a history. Can they find more bugs if we run them multiple times for the same history or even have them continue to search for more, different bugs? The answer is unfortunately *no*, for most of them.

We craft for each TAP a simple anomalous history (or test case) by instantiating the patterns in Figure 3. See [Liu et al. 2024a, Appendix C] for the full list of 14 test cases. Table 5 shows our test results where none of the checkers, except Plume and PolySI, can catch all the anomalies. In particular, Viper and Cobra could not identify *intermediate* or *non-repeatable* reads (TAP-f and TAP-j). dbcop fails to detect reading *future* or *non-monotonic* reads (TAP-c, TAP-h, and TAP-i). Elle misses CO cycles (TAP-g). One cause of their failure to detect certain anomalies lies in the underlying incomplete characterizations of isolation guarantees. For example, most of them focus mainly on inter-transaction dependencies, thereby missing anomalies that reside inside transactions like TAP-c and TAP-j. Elle checks RA and TCC via a mix of Adya and non-Adya anomalies, which

[5]We consider its default mode with constraint pruning [Viper 2023].
[6]For TCC, Elle adopts the definition by Cerone et al. [2015]. This is equivalent to the TCC definition by Biswas and Enea [2019] that Plume builds upon.

Table 5. Examining isolation checkers via 14 test cases corresponding to our TAPs. ● denotes the pass of the test case; ○ indicates the checker failing to detect the anomaly; ◐ means that a more recent version has fixed the issue we reported. Elle-RC checks Adya's PL-2 [Adya 1999; Elle 2023] which, unlike Biswas and Enea [2019], does not cover monotonic reads in the same transaction (i.e., TAP-h and TAP-i).

Checker	TAP-a	TAP-b	TAP-c	TAP-d	TAP-e	TAP-f	TAP-g	TAP-h	TAP-i	TAP-j	TAP-k	TAP-l	TAP-m	TAP-n
PolySI (SI)	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Viper (SI)	○	○	◐	◐	◐	○	◐	●	●	◐	●	●	●	●
Cobra (SER)	●	●	●	○	○	○	●	●	●	○	●	●	●	●
dbcop-TCC	●	●	○	●	●	●	●	●	●	●	●	●	●	●
dbcop-RA	●	●	○	●	●	●	●	●	●	●	●	●	–	–
dbcop-RC	●	●	○	●	●	●	○	○	○	–	–	–	–	–
Elle-TCC	●	●	◐	●	●	◐	○	○	◐	●	○	●	○	●
Elle-RA	●	●	◐	●	●	◐	○	○	◐	●	○	●	–	–
Elle-RC*	●	●	◐	○	○	●	○	–	–	–	–	–	–	–
Plume	●	●	●	●	●	●	●	●	●	●	●	●	●	●

are not known to be complete characterizations. We have reported these issues to the developers. Some of them have been fixed, as shown in Table 5.

Note that passing all of our test cases does not guarantee a sound and complete isolation checker as they are by no means exhaustive. For example, Elle-TCC also detected the *lost update* anomalies that are not covered by our test cases; this would result in false positives [Cerone et al. 2015] as such anomalies are only prohibited by stronger isolation levels like snapshot isolation. We reported this issue, and it has since been fixed. Overall, our test result suggests the importance of building an isolation checker on a sound and complete characterization.

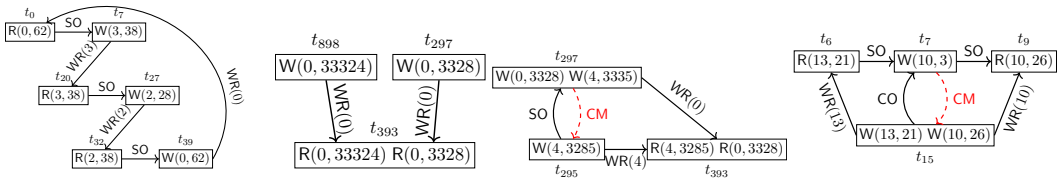
New Anomalies Revisited. Recall that Plume identifies new anomalies of different types in three databases (§5.2). For PolySI, Viper, Cobra, and dbcop-TCC, the anomalous histories reported by Plume contain anomalies these tools can detect, such as TAP-m and TAP-n; consequently, they reject these histories. However, since these tools, except for PolySI, are not designed or implemented to find *all* isolation bugs in a history, it is hard to determine whether they can detect other new anomalies, such as those related to intra-transactional dependencies. Moreover, the anomalous histories involve read-write registers, making them incompatible with Elle's list-append mode.

Nonetheless, we can still make reasonable guesses about whether the aforementioned tools can detect the new anomalies reported by Plume. By cross-checking these anomalies with our examination results in Table 5, it is highly unlikely that Viper, Cobra, and Elle-TCC could detect all of them, such as TAP-f and TAP-g. Although PolySI and dbcop-TCC may potentially find these new anomalies, as indicated by Table 5, additional manual efforts are required for their confirmation. For PolySI, one must analyze which anomalies the returned counterexamples represent; for dbcop-TCC, this is more challenging because it only provides a yes/no answer when finding a bug.

5.4 Understanding Violations Found

Plume returns fine-grained, informative counterexamples in terms of TAPs. We have integrated the Graphviz tool [Graphviz 2023] into Plume to visualize the violating scenarios, which makes returned bugs more understandable. Below we showcase four new isolation bugs of different kinds, together with their violating scenarios. We defer representative bugs of other kinds (e.g., TAP-f, TAP-l, and TAP-n) to [Liu et al. 2024a, Appendix D].

TAP-g in AntidoteDB. As shown in Figure 7a, transaction t_{39} is causally ordered after transaction t_0 via a series of SO and WR dependencies, while t_0 reads t_{39}'s value written to key 0. This results in a cyclic causality anomaly, i.e., TAP-g.



(a) TAP-g in AntidoteDB (b) TAP-j in MariaDB (c) TAP-k in MariaDB (d) TAP-m in YugabyteDB

Fig. 7. Visualized anomalies found in three production databases by Plume. Only the involved operations per transaction are shown.

TAP-j and TAP-k in MariaDB-Galera. Figure 7b shows the *non-repeatable reads* (TAP-j): two consecutive reads on the same key 0 in transaction t_{393} fetch different values written by two separate transactions. Figure 7c depicts how *fractured reads* (TAP-k) occur. Transaction t_{393} has two reads: $\mathrm{R}(4, 3285)$ fetches the value written by transaction t_{295}, and $\mathrm{R}(0, 3328)$ reads the value by transaction t_{297} that also writes key 4. Under *read atomicity*, $\mathrm{W}(4, 3285)$ must be ordered after $\mathrm{W}(4, 3335)$ in the version order. However, $\mathrm{W}(4, 3285)$ is issued before $\mathrm{W}(4, 3335)$ in the same session.

Interestingly, TAP-j and TAP-k coexist in this history, with both involving transactions t_{297} and t_{393}. Plume precisely captures these two anomalies from a single history with distinguishable violating scenarios.

TAP-m in YugabyteDB. Figure 7d shows the conflict between the CO and CM orders established from the history. As transaction t_9 reads the value written by transaction t_{15} on key 10 and is also causally ordered (via SO) after transaction t_7 that also writes to key 10, t_7's write must be installed before t_{15}'s in the database. Nonetheless, t_7 has already been causally ordered after t_{15}.

5.5 Performance Evaluation

We conduct a comprehensive performance analysis of Plume, along with six baselines.

- The non-solver tools dbcop and Elle (its representative "list-append" mode), both supporting checking HAT isolation levels, including RC, RA, and TCC;
- A new tool CausalC+ based on *answer set programming* (ASP) [Gebser et al. 2012], extending CausalC [Zennou et al. 2022] for checking multi-operation transactions against TCC;[7]
- A strong baseline called TCC-Mono, leveraging the advanced MonoSAT solver [Bayless et al. 2015] to search for cycles; and
- The state-of-the-art tools PolySI and Viper, as the representatives for checking *stronger* isolation levels.

Our goal is twofold. First, we want to compare Plume with various techniques utilized by the state-of-the-art, e.g., with or without a solver *and* using different solvers. Second, databases claiming to provide weaker isolation guarantees often deliver stronger consistency in practice [Cheng et al. 2021; Lu et al. 2020], and optimized stronger isolation checkers have already been shown to be highly efficient. Hence, we explore whether these checkers could replace Plume for *validating* weak isolation levels.[8] Note that CausalC+ and TCC-Mono use similar compact and efficient encodings as in CausalC [Zennou et al. 2022] and Cobra [Tan et al. 2020], respectively.

[7] CausalC is specifically designed for checking non-transactional causal consistency.
[8] To do so, we generate histories that are valid up to the isolation level (e.g., snapshot isolation) for which the checker (e.g., PolySI and Viper) is designed.

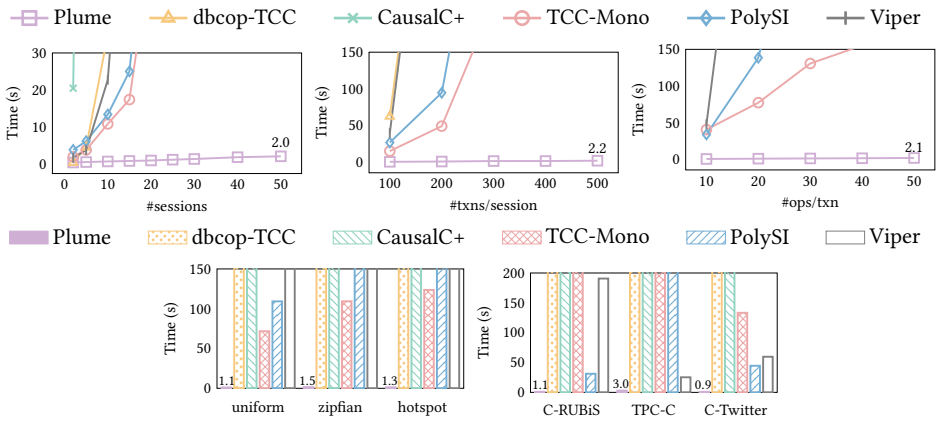


Fig. 8. Validation time comparison. Data points are not plotted for the timed-out experiments.

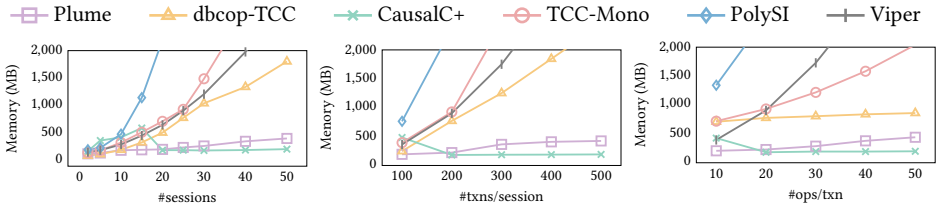


Fig. 9. Memory overhead comparison. We only plot each tool's memory usage at timeout (10 minutes).

5.5.1 Performance Comparison. As shown in Figure 8, Plume substantially surpasses all the baselines (excluding Elle, which we compare with separately in §5.5.2) with respect to checking efficiency under varying workloads. Plume's performance is also fairly stable, at roughly two seconds checking time, even under higher concurrency such as more sessions and skewed key accesses. In most cases, the state-of-the-art TCC checker dbcop and the CausalC extension do not finish within the timeout of 10 minutes. It is not surprising that PolySI and Viper outperform these two checkers given their domain-specific optimizations, tailored for certain benchmarks such as C-Twitter. We have also observed similar results when comparing Plume to dbcop with respect to checking read atomicity; see [Liu et al. 2024a, Appendix E].

In addition, we measure the memory overhead for all the checkers, as shown in Figure 9. The comparison results on other skewed workloads and benchmarks are given in [Liu et al. 2024a, Appendix F]. In summary, Plume consumes significantly less memory (for storing generated graphs, vectors, and tree clocks) than the baselines under a wide variety of workloads and benchmarks. In particular, dbcop-TCC, the only checker that does not rely on solving and thus stores no constraints, is not competitive with Plume. Note that we only plot each tool's memory usage when it times out. That is why CausalC+ has the illusion of less memory overhead.

Overall, our experimental results suggest that (i) vectors and tree clocks are more efficient techniques than solving in checking TCC, and (ii) compared to Plume, the state-of-the-art stronger isolation checkers, such as PolySI and Viper, are inefficient in validating weaker levels like TCC.

5.5.2 Compatibility with Jepsen. Elle is an integrated checker in the Jepsen framework [Jepsen 2024a]. During testing, Jepsen produces workloads of reads and *append* operations (representing

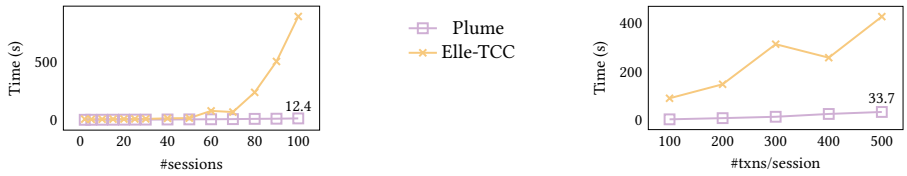


Fig. 10. Performance comparison with Elle-TCC (50 operations per transaction, 50% reads, and the uniform key-access distribution).

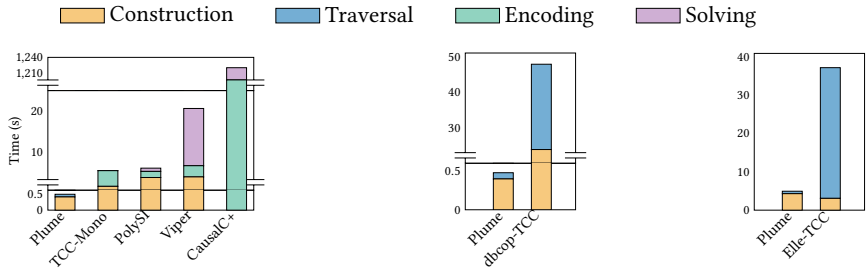


Fig. 11. Decomposing validation time. CausalC+ builds on CausalC's codebase [Zennou et al. 2022] that integrates constructing into encoding.

writes), from which Elle can efficiently infer the version orders of writes. To make Plume compatible with such "append" histories, we extend it with a compiler that translates a list of values per read into the version order of the corresponding writes. As shown in Figure 10, Plume exhibits substantially higher and more stable checking efficiency than Elle-TCC when more sessions are involved and the workloads become large. This also applies to the comparison with Elle on checking RA and RC; see [Liu et al. 2024a, Appendix G].

5.5.3 Decomposition Analysis. We decompose each tool's checking time into stages: *construction* (building up a certain dependency graph from a given history), *traversal* (for non-solver tools, searching for certain graph patterns like cycles that represent bugs), *encoding* (encoding the dependency graph into constraints), and *solving* (running the solver to find a solution).

For the comparison with solving-based tools, we generate a small workload of 2k transactions to obtain a full view of the checking overheads for their individual stages. Figure 11 depicts our analysis results. Constructing a graph is generally inexpensive. Solving-based checkers exhibit variations in the efficiency of encoding and solving, with the overall overhead ranging from a few seconds to 20 minutes. Compared to the non-solver tools dbcop and Elle, Plume demonstrates a substantial reduction in the time required for graph traversals due to the utilization of vectors and the optimization with tree clocks.

5.5.4 Ablation Analysis. To see the contributions of Plume's two major design choices, i.e., vectors and tree clocks, we experiment with two variants: (i) Plume without tree clocks (to assess the performance improvement on top of vectors), and (ii) Plume with neither tree clocks nor vectors (thus using standard DFS traversals to search for instances of TAPs). Figure 12 demonstrates the noticeable acceleration produced by each design choice. In particular, Variant (ii) is significantly less efficient because each invocation of cycle detection involves a DFS over the entire dependency graph with no early stop. Note that C-RUBiS or C-Twitter cannot distinguish Plume from Variant

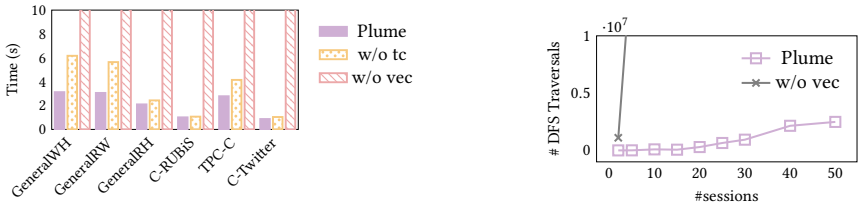


Fig. 12. Ablation analysis of Plume. "w/o tc" and "w/o vec" refer to Variants (i) and (ii), respectively.

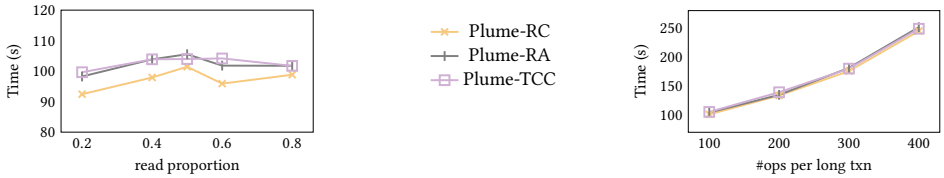


Fig. 13. Plume's overhead under large-sized workloads of 1 million transactions and over 50 million operations.

(i) as both contain only a small number of sessions and transactions, and there are fewer comparing and joining operations over vectors during the graph traversals.

Owing to the utilization of vectors, Plume can also significantly accelerate reachability checking and cycle detection in the UpdateVec procedure by enabling early stops and reducing a large number of DFS traversals. With more concurrency (or sessions), the number of DFS traversals in Variant (ii) grows dramatically, reaching up to 10^{10} by the timeout, as shown in Figure 12.

5.5.5 Scalability. To evaluate Plume's scalability, including its three checking components for RC, RA, and TCC, we generate workloads of one million transactions and over 50 million operations. We experiment with varying read ratios and operations per long transactions, up to 400, for which Plume completes the checking in under five minutes, as shown in Figure 13. We also observe a modest increase of checking time with larger-sized transactions. These three components exhibit quite similar checking overheads as the optimizations with vectors and tree clocks mask the difference. To conclude, large workloads appear manageable for Plume.

6 Related Work

Characterizing Database Isolation. Along with the advances in designing reliable, highly available database transactions, considerable efforts [Adya 1999; Bernstein et al. 1979; Biswas and Enea 2019; Cerone et al. 2015; Crooks et al. 2017] have been devoted to formalizing their isolation guarantees. Cerone et al. [2015] propose a framework for declaratively specifying isolation levels at least as strong as read atomicity, with the dual notions of visibility (what transactions can observe) and arbitration (the commit order in our case). We base our TAP-based characterization of HAT isolation levels on an alternative axiomatic framework by Biswas and Enea [2019]. This framework is more suitable for black-box testing as the write-read relation (playing the role of the visibility relation [Cerone et al. 2015]), along with session orders, can be naturally extracted from histories. We are also inspired by prior work [Adya 1999; Berenson et al. 1995; Bouajjani et al. 2017] in the same spirit that uses anomalies to characterize isolation/consistency criteria. However, Berenson et al. [1995] and Adya [1999] do not cover more recent isolation levels such as read atomicity and TCC; Bouajjani et al. [2017] only consider (non-transactional) causal consistency. We complement this line of research by soundly and completely characterizing weak HAT isolation levels.

Table 6. Examining the state-of-the-art black-box isolation checkers through the lens of SIEGE+ [Huang et al. 2023]. We omit soundness (returning no false positives) and effectiveness (finding real-world bugs) as all the checkers meet these two criteria except for Elle-TCC (see §5.3).

Checker	Property	Informative (bugs)	General	Efficient (core tech.)	Complete
CausalC	CC	unsatisfied clauses	simple txns; rw reg.	answer set solving	no
dbcop	multiple	true or false	general txns; rw reg.	graph traversal	no
PolySI	SI	detailed violating scenarios	general txns; rw reg.	SMT solving	yes
Viper	SI	unsatisfied clauses	gen. txns, range qry.; rw reg., lists	SMT solving	no
Cobra	SER	unsatisfied clauses	general txns; rw reg.	SMT solving & GPU acc.	no
Elle	multiple	detailed violating scenarios	gen. txns; rw reg., lists, counters	list-append inference	no
Plume	HAT	detailed violating scenarios	general txns; rw reg., lists	vectors & tree clocks	yes

Recently, Crooks et al. [2017] and Xiong et al. [2020] introduce alternative formalizations of isolation levels via client-observable states. An open question is whether they would yield a more efficient isolation checking algorithm.

Black-box Checking Isolation. We focus on the state-of-the-art black-box checkers for database isolation, thereby excluding works that rely on white-box approaches [Clark et al. 2024; Liu et al. 2019b]. We examine these black-box checkers using SIEGE+ [Huang et al. 2023] (see also §5), as shown in Table 6.

Plume meets all the SIEGE+ criteria, with a focus on complete and efficient anomaly detection, as demonstrated by our assessment. Other checkers are all sound (no false positives) and effective (finding bugs in real databases) except for Elle-TCC that may report false positives (i.e., lost updates). CausalC focuses on causal consistency and does not support general transactions. Solver-based tools like Viper and Cobra return hard-to-understand counterexamples in general, e.g., unsatisfied clauses. PolySI improves them by constructing the violating scenarios from unsatisfied clauses. Although dbcop also utilizes graph traversal to search for isolation bugs, its current implementation only returns "true" (no bugs) or "false" (a bug found). This is uninformative with respect to understanding how the bug occurred. In contrast, both Elle and Plume report detailed violation scenarios in terms of Adya's anomalies and instances of TAPs, respectively. Additionally, as shown by our experimental results, Plume significantly outperforms dbcop, as well as the strong baselines utilizing various solving techniques based on answer set programming and the advanced MonoSAT solver. Plume also outperforms Elle under large-sized workloads with higher concurrency (e.g., more clients). Other checkers, except PolySI, miss HAT isolation anomalies, despite some of them being designed for checking stronger isolation criteria. All the checkers can handle read-write registers. Plume and Viper join Elle in supporting list-append histories as well. Moreover, Viper can deal with a specific type of range queries, where a range refers to keys, instead of values.

7 Discussion

Beyond HAT Isolation. As we have observed, many databases claiming to provide stronger isolation guarantees actually violate much weaker HAT isolation guarantees. This suggests that a first analysis with Plume could be effective and, more importantly, efficient as it can quickly check millions of transactions in minutes.

Plume's key idea of leveraging vectors and tree clocks could be adapted to other *graph-based* approaches for checking stronger isolation guarantees. This would accelerate the traversals of various dependency graphs, such as *polygraphs* [Papadimitriou 1979] (adopted by Cobra, PolySI, and Viper) and Adya's *direct serialization graph* [Adya 1999] (adopted by Elle). Specifically, for anomalies (e.g., *lost updates*) that are forbidden only by a stronger isolation level L (e.g., snapshot isolation), it is necessary to design new TAPs. Alongside the existing TAPs (owing to their modularity), a

sound and complete characterization of L must also be established. Let us denote this resulting set of TAPs as S.

Vectors and tree clocks can then be applied to check L. The major distinction when checking L is the need to "guess" the version order of writes (and thus the associated transactions). Consider W_1 and W_2 from two different transactions, both writing to the same key. The new checking algorithm first assumes that W_1 is ordered before W_2 during the construction of a dependency graph and searches for any TAPs in S. If a TAP is found, the algorithm then switches to the alternative version order and searches for any TAPs in S again. For each guess, vectors and tree clocks can be utilized in the same way as in Plume. The history violates L if TAPs are identified in both cases.

Beyond Database Engines. We have thus far focused on checking isolation guarantees in database engines. Many database-backed applications, such as social network and E-commerce, also provide weak HAT semantics and are known to exhibit data anomalies [Gan et al. 2020; Warszawski and Bailis 2017]. Moreover, we have also seen HAT semantics, such as RA and TCC, adopted in promising computing architectures like serverless computing [Lykhenko et al. 2021; Wu et al. 2020] and microservices [Ferreira Loff et al. 2023; Pereira and Silva 2023]. Plume could be naturally applied to detecting isolation violations in these application domains due to its black-box nature. One only needs to collect system execution histories as an outsider before feeding them to Plume.

Limitations and Future Work. Plume can miss bugs not covered by HAT semantics, e.g., lost update and write skew [Cerone et al. 2015; Liu et al. 2019a]. Plume also cannot handle range queries, for which we plan to explore the possibility of applying *tombstones* [Zhang et al. 2023], adopted by Viper for checking SI, and *SQL-level instrumentation* [Jiang et al. 2023], devised by TxCheck for detecting logic database bugs in general. All black-box isolation checkers, including Plume, rely on the randomized generation of large workloads or test cases (Step ① in Figure 2). However, as we have observed, counterexamples like those in Figure 7 are often much smaller than the generated workloads. Designing concise, yet effective, test cases would improve overall checking performance.

8 Conclusion

Plume is the first efficient, complete, black-box checker for weak isolation levels. It complements two lines of research: at the theoretical level, characterizing data consistency via anomalies, and at the practical level, designing performant black-box isolation checkers. For databases that provide multiple isolation levels, one can naturally employ Plume and the aforementioned checkers, such as PolySI, to respectively check weak and strong levels. This owes to the black-box nature of these checkers that take as input histories of essentially the same format. We have integrated Plume, along with several strong isolation level checkers, into our database isolation checking system IsoVista [Gu et al. 2024], featuring user-friendly interfaces. Moreover, Plume's informative counterexamples could facilitate debugging and help developers rethink their system designs and implementations. Our TAPs and test cases could also help solidify the development of isolation checkers. We expect this work also to shed light on checking data consistency beyond databases.

Acknowledgments

We appreciate the anonymous reviewers for their valuable feedback. We also extend our thanks to Zihe Song for her contribution in identifying isolation bugs in MariaDB-Galera. Si Liu was supported by an ETH Zurich Career Seed Award.

References

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

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Comput.* 9, 1 (1995), 37–49.

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.

AntidoteDB. Accessed in October, 2023. https://www.antidotedb.eu/.

Hagit Attiya, Faith Ellen, and Adam Morrison. 2015. Limitations of Highly-Available Eventually-Consistent Data Stores. In *PODC 2015*. ACM, 385–394.

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013a. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (nov 2013), 181–192.

Peter Bailis, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013b. HAT, Not CAP: Towards Highly Available Transactions. In *HotOS XIV*. USENIX Association.

Peter Bailis, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (2016), 15:1–15:45.

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

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD '95*. ACM, 1–10.

Philip A. Bernstein, David W. Shipman, and Wing S. Wong. 1979. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. Software Eng.* 5, 3 (1979), 203–216.

Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28.

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

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, 271–284.

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

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

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

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.

Jack Clark, Alastair F. Donaldson, John Wickerson, and Manuel Rigger. 2024. Validating Database System Isolation Level Implementations with Version Certificate Recovery. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. ACM, 754–768.

MariaDB Galera Cluster. Accessed in October, 2023. https://mariadb.com/kb/en/what-is-mariadb-galera-cluster/.

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.

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

Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *SoCC 2014*. ACM, 4:1–4:13.

ElectricSQL. Accessed in December, 2023. https://electric-sql.com/.

Elle. Accessed in November, 2023. Consistency Models in Elle. https://github.com/jepsen-io/elle/blob/main/src/elle/consistency_model.clj.

João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. 2023. Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications. In *SOSP '23*. ACM, 298–313.

C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66.

Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *Proc. VLDB Endow.* 13, 12 (2020), 2773–2786.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2012. *Answer Set Solving in Practice*. Morgan & Claypool Publishers.

Graphviz. Accessed in December, 2023. Open source graph visualization software. https://graphviz.org/.

Long Gu, Si Liu, Tiancheng Xing, Hengfeng Wei, Yuxing Chen, and David Basin. 2024. IsoVista: Black-box Checking Database Isolation Guarantees. *Proc. VLDB Endow.* 17, 12 (2024).

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.

Jepsen. Accessed in March, 2024a. https://jepsen.io.

Jepsen. Accessed in March, 2024b. Jepsen Analyses. https://jepsen.io/analyses.

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

Nick Kallen. Accessed in December, 2023. Big Data in Real Time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter/.

Kyle Kingsbury. 2023. MySQL 8.0.34. https://jepsen.io/analyses/mysql-8.0.34.

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

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

Si Liu. 2022. All in One: Design, Verification, and Implementation of SNOW-Optimal Read Atomic Transactions. *ACM Trans. Softw. Eng. Methodol.* 31, 3 (2022).

Si Liu, Long Gu, Hengfeng Wei, and David Basin. 2024a. *Plume: Efficient and Complete Black-Box Checking of Weak Isolation Levels*. Technical Report. https://github.com/dracooooo/Plume.

Si Liu, Luca Multazzu, Hengfeng Wei, and David Basin. 2024b. NOC-NOC: Towards Performance-optimal DistributedTransactions. *Proc. ACM Manag. Data* 2, 1 (2024).

Si Liu, Peter Csaba Ölveczky, Qi Wang, Indranil Gupta, and José Meseguer. 2019a. Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. *Formal Aspects Comput.* 31, 5 (2019), 503–540.

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

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP 2011*. ACM, 401–416.

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.

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

Taras Lykhenko, Rafael Soares, and Luis Rodrigues. 2021. FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing. In *Middleware '21*. ACM, 159–171.

Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *ASPLOS'22*. ACM, 710–725.

Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *NSDI 2017*. USENIX Association, 453–468.

Microsoft. Accessed in December, 2023. Azure CosmosDB DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

Neo4j. Accessed in December, 2023. https://neo4j.com/.

Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (1979), 631–653.

Pedro Pereira and António Rito Silva. 2023. Transactional Causal Consistent Microservices Simulator. In *Distributed Applications and Interoperable Systems - 23rd IFIP WG 6.1 International Conference, DAIS 2023 (LNCS, Vol. 13909)*. Springer, 57–73.

Matthieu Perrin, Achour Mostefaoui, and Claude Jard. 2016. Causal consistency: beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, Article 26, 12 pages.

PostgreSQL. Accessed in December, 2023. Transaction Isolation. https://www.postgresql.org/docs/current/transaction-iso.html.

Potassco. Accessed in December, 2023. Clingo: A grounder and solver for logic programs. https://github.com/potassco/clingo.

RUBiS. Accessed in December, 2023. Auction Site for e-Commerce Technologies Benchmarking. https://projects.ow2.org/view/rubis/.

Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: Making Transactional Key-Value Stores Verifiably Serializable *(OSDI'20)*. USENIX Association, Article 4.

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

Viper. Accessed in December, 2023. Configurations in Viper. https://github.com/Khoury-srg/Viper/blob/main/src/config.yaml.

Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *SIGMOD'17*. ACM, 5–20.

Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *SIGMOD '20*. ACM, 83–97.

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)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:31.

YugabyteDB. Accessed in December, 2023. https://www.yugabyte.com/.

Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2022. Checking causal consistency of distributed databases. *Computing* 104, 10 (2022), 2181–2201.

Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. ACM, 654–671.

Received 2024-03-31; accepted 2024-08-18