Complete and Efficient Checking of Transactional Causal Consistency in Database Engines

Si Liu
ETH Zürich

Long Gu
Nanjing University

Hengfeng Wei
Nanjing University

David Basin
ETH Zürich

ABSTRACT

Transactional causal consistency (TCC) is a prevalent weak isolation guarantee that has emerged as a successful marriage of the distributed computing and database communities. TCC eschews the performance penalties of strong isolation levels while still preventing many undesired data anomalies. However, TCC violations have recently been found in many production database engines that claim to provide TCC or even stronger isolation guarantees.

In this paper we present C4, a complete, efficient black-box checker for TCC that provides understandable scenarios witnessing detected violations. C4 builds on a novel, fine-grained characterization of TCC via *transactional anomalous patterns*, for which we establish its soundness and completeness. C4 builds on two techniques, namely vectors and tree clocks, to accelerate TCC checking. Our extensive assessment shows that C4 can successfully reproduce all 3097 known TCC anomalies, identify new TCC violations in four production database engines along with their causes, report more anomalies than many state-of-the-art black-box checkers, and efficiently validate TCC under a wide range of workloads.

PVLDB Reference Format:
Si Liu, Long Gu, Hengfeng Wei, and David Basin. Complete and Efficient Checking of Transactional Causal Consistency in Database Engines. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
Our tool, experimental data, and technical report are available at https://github.com/dracooooooo/C4.

1 INTRODUCTION

Transactional causal consistency (TCC) [3, 40] has emerged as a successful marriage of the distributed computing and database communities. TCC extends causal consistency [2, 50], the strongest consistency level achievable in an always-available system [5], by providing database transactions. This benefits both database users and developers. TCC eschews the performance penalties of stronger isolation levels such as snapshot isolation and serializability while avoiding many anomalies such as causality violations and diverging data [39] under weaker isolation levels [9, 12]. Moreover, the support for transactions simplifies the programming complexity of concurrency by providing an abstraction for executing concurrent computations on shared data in isolation [13]. Finally, TCC provides intuitive semantics that supports session guarantees [56] like *read your writes*, which is favorable to end users.

As an emerging isolation guarantee, TCC has already attracted the attention of both academia and industry. There has been a torrent of academic advances over the last decade in highly available and performant TCC database systems [3, 10, 23, 24, 39–41, 44, 53, 54]. Facebook advocates the joint force of causal consistency and transactions [42]. Recent production adoptions of TCC and its variants include Neo4j [47], Cosmos DB [45], MongoDB [46], and ElectricSQL [25] (a successful transition from the Cure protocol [3] to production). Unfortunately, TCC anomalies have manifested in many production database engines that claim to provide TCC or even stronger isolation levels such as *snapshot isolation* (SI) or *serializability* (SER) [14, 30, 32]. This raises the concerns of whether database implementations actually deliver the promised isolation levels in practice.

Black-box testing has been adopted as the *de facto* approach to finding isolation bugs and validating isolation fulfilment [14, 30, 35, 55, 59, 60] as database internals are usually unavailable and impenetrable. The SIEGE+ principle [30], extending SIEGE [35] with *completeness*, has recently been proposed to guide the design of black-box isolation checkers. Specifically, an ideal checker is expected to be **S**ound (returning no false positives), **I**nformative (reporting understandable counterexamples), **E**ffective (finding isolation bugs in real-world databases), **G**eneral (compatible with different kinds of databases and general transaction workloads), **E**fficient (adding modest checking time even for workloads with high concurrency and of large size), and **C**omplete (missing no isolation anomalies).

However, existing isolation checkers, including those for causal consistency, fail to fulfill all of the above criteria. In particular, the state-of-the-art TCC tester dbcop [14] misses TCC bugs, e.g., non-repeatable reads labeled as "critical" by MariaDB [6] (**C**), and requires a significant amount of time for TCC validation (**E**); CausalC [59] only supports checking simple transactions of a single read or write operation (**G**). Although checkers for *stronger* isolation guarantees could also be used to detect TCC bugs or validate TCC, they inherently return false positives with respect to TCC (**S**), e.g., an SER checker may report the lost-update anomalies which do not violate TCC, and are prohibitively expensive due to the high computational complexity of the checking problem *per se* (**E**), e.g., checking SI is NP-complete in general [14]. Moreover, few checkers can catch all bugs (**C**) because of either incomplete or coarse-grained characterizations of isolation guarantees [8, 55, 60], or efficiency concerns [29]. Finally, despite the increasing trend of utilizing solvers to check isolation guarantees [30, 55, 60], the checking overhead is still significantly high in practice. In addition, solver-based tools return less understandable counterexamples in general (**I**), such as atomic propositions and unsatisfied clauses, rendering the understanding and debugging of the violations hard.

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

The C4 Checker. We devise a novel black-box TCC checker called C4 to fulfill SIEGE+. C4 builds on two key ideas to address three major challenging criteria: **C**omplete, **E**fficient, and **I**nformative.

First, driven by the building blocks of TCC, such as atomic visibility [9], causal consistency [2, 50], and data convergence [3, 39], and the isolation bugs that manifest in the wild [14, 30, 32], we define 12 fine-grained *transactional anomalous patterns* (TAPs) to characterize TCC. Our characterization overcomes the **C** and **I** challenges. It is *sound* and, critically, *complete*, rendering C4 capable of detecting all possible TCC data anomalies in the collected execution histories[1] of transactions, as in black-box testing of database isolation guarantees. Thanks to C4's completeness, we can identify significantly more anomalies than many state-of-the-art isolation checkers from the same history collection (Section 5). This is highly desirable, especially given that the occurrence of an isolation bug is usually unpredictable due to random test generation commonly adopted by existing black-box checkers.

Our characterization is also fine-grained, with each TAP corresponding to defects of the building blocks underlying a TCC database system. For example, TAP-g (Section 3.2) is typically caused by design or implementation errors on session guarantees (like the incorrect use of vectors) while TAP-h is caused by flaws in the atomic commitment of transactions. Such a fine-grained characterization enables C4 to report informative counterexamples with the full violating scenarios, which aid developers in discovering their root causes. Moreover, these modular TAPs can help us construct sound and complete *weaker* isolation checkers by combining an appropriate subset of TAPs, e.g., six TAPs for *read committed* and additionally with two for *read atomicity*.

Second, although TCC is checkable in polynomial time [14], existing tools still suffer from the high checking overhead in practice due to inefficient searching for cycles (which are used to represent TCC violations [14]) in large, dense dependency graphs (constructed from extensive histories) *or* expensive encoding and solving of graph properties using solvers [30, 55, 59, 60]. C4 leverages two techniques to overcome the **E** challenge.

Inspired by the decades-long practice of using *vectors* [26] to *design* causally consistent databases, we utilize vectors instead to *black-box check* TCC systems. By capturing and storing the causal dependencies among the transactions in a history, vectors enable C4 to accelerate graph traversals for both reachability checking and cycle detection. Moreover, we adapt the *tree clock* [43] data structure, a recent advance in concurrent program analysis, in our utilization of vectors to further accelerate C4. This significantly reduces the time complexity of both comparing and joining vectors, which arises from the large number of client sessions and dense dependencies of transactions.

Main Contributions. Overall, we provide:

(1) a new sound and complete characterization of TCC via TAPs that also facilitates understanding and debugging violations (Section 3);
(2) a complete and efficient TAP-based checking algorithm for TCC, leveraging vectors and tree clocks to accelerate the search for violations (Section 4);
(3) the C4 tool comprising our new checking algorithm and a visualizer for restoring violating scenarios; and
(4) an extensive evaluation of C4, along with the state-of-the-art black-box isolation checkers, that demonstrates C4's fulfilment of SIEGE+ (Section 5). In particular, C4 has (i) reproduced all 3097 known TCC violations, (ii) detected novel TCC bugs in four production database engines of different kinds, (iii) identified their causes, (iv) reported more anomalies than many existing checkers, (v) substantially outperformed the baselines on various benchmarks, and (vi) scaled to large workloads with 1 million transactions, 400 million operations, and 1 billion keys successfully checked under 200 seconds.

[1] A history records the transactional requests to and the responses from a database. See Section 3.1 for its formal definition.

2 TRANSACTIONAL CAUSAL CONSISTENCY

2.1 TCC in a Nutshell

Informally, TCC combines three properties: read atomicity (RA) [9], causal consistency (CC) [2, 50], and convergence [3, 39]. RA ensures that all or none of a transaction's updates are observed by other transactions. It prohibits the *fractured reads* anomaly, as illustrated below.

Example 1 (Read Atomicity)**.** Joe, Monica, and Rachel are planning a road trip. They share a road trip planner. Rachel would like to invite Ross to the trip after two of them become each other's friends. Without the RA guarantee, Joe and Monica may only see one direction of the friendship relation (fractured reads), e.g., Ross is Rachel's friend but Rachel is not Ross' friend.

CC guarantees that two transactions that are causally related must appear to all client sessions in the same *causal order*. CC is define by the following three rules:

- **Session Order (SO).** If two transactions t_1 and t_2 are in the same session and t_1 is performed before t_2, then t_1 is casually before t_2;
- **Write-Read Relation (WR).** If transaction t_2 reads values from transaction t_1, then t_1 is casually before t_2;
- **Transitivity.** If transaction t_1 is causally before t_2 and t_2 is causally before t_3, then t_1 is causally before t_3.

CC prevents the *causality violation* anomaly, as illustrated below.

Example 2 (Causality)**.** On the road trip planner, Joe sets Los Angeles as the starting city. After seeing it, Monica adds Chicago as the destination. Without causal consistency, Rachel may only see the destination but not the starting city.

CC does not constrain transactions that are *not* causally related, which may be observed in different orders by different sessions. As a result, replicas may permanently diverge under concurrent conflicting updates. In TCC, this is forbidden by the *convergence* property, which requires replicas to eventually converge to the same state even under conflicting updates [3, 39]. In practice, convergence can be achieved by applying the LWW (last-writer-wins) rule [17]. Although there is a weaker variant of TCC with no convergence guarantee [41], to the best of our knowledge, all database systems claiming to support TCC provide convergence in practice.

Example 3 (Convergence)**.** Joe and Ross update the meeting place to their own apartment, respectively. With convergence, these two

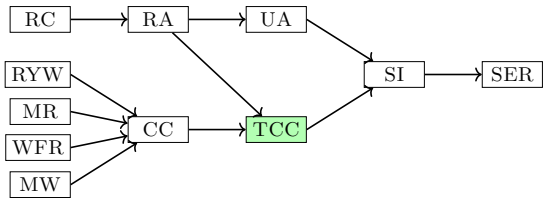


Figure 1: A hierarchy of isolation levels. $A \longrightarrow B$ means that A is strictly weaker than B. RC means read committed [12]. UA means update atomicity [18]. There are four session guarantees [16, 56]: read your writes (RYW), monotonic reads (MR), writes follow reads (WFR), and monotonic writes (MW). CC is the combination of all these session guarantees.

conflicting updates must be handled in a convergent manner: either Joe's or Ross' apartment. Otherwise, Monica and Rachel may head to different apartments.

As shown in Figure 1, TCC is an essential building block of stronger transactional consistency properties such as SI [12] and SER [49]. In particular, both SI and SER comprise *update atomicity* (UA) [18], which forbids lost updates (in addition to fractured reads), and TCC. A TCC violation is therefore a violation for SI and SER; in fact, many isolation bugs found in the production databases that claim to support SI or SER are actually TCC bugs [30, 35, 55, 60]. A history satisfying SI or SER naturally satisfies TCC.

2.2 Checking TCC

Definition 1. The TCC *checking problem* is the decision problem of determining whether a history of transactions satisfies TCC.

Following the common practice in black-box isolation testing [14, 30, 35, 55, 60], we make two assumptions. First, every history contains a special transaction (denoted $t_\perp$) that writes the initial values (denoted $\perp_x, \perp_y, \ldots$) of all keys [14, 19]. This transaction precedes all the other transactions across client sessions. In practice, we can use multiple small write-only transactions to populate the database with initial values for each key *before* the actual testing. These transactions can be considered as a single logical write-only transaction.

Second, for each key, every write to the key assigns a unique value [1, 14, 55]. Hence, each read can be uniquely associated with the transaction that issues the corresponding (dictating) write. The TCC checking problem can then be solved in polynomial time [14]; otherwise, it is NP-hard in general [15]. For database testing, we can use, e.g., the client identifier and local counter, to ensure the uniqueness of the written values.

3 CHARACTERIZING TCC ANOMALIES

In this section we describe our fine-grained characterization of TCC using *transactional anomalous patterns* (TAPs). We first present the formal basis for our characterization.

3.1 Context and Definitions

We consider a key-value store managing a set of keys $\mathsf{K} = \{x, y, z, \ldots\}$ associated with values from 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 $x \in \mathsf{K}$.

Relations and Orderings. 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. We use R^+ to denote the transitive closure of R. A relation $R \subseteq A \times A$ is *acyclic* if $R^+ \cap I_A = \emptyset$, where $I_A \triangleq \{(a, a) \mid a \in A\}$ is the identity relation on A. 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.

Causality and Convergence. Clients interact with the data store by issuing transactions during *sessions*. A *history* records the client-visible results of such interactions.

Definition 2 (History)**.** A *history* is a triple $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$, where T is a set of transactions, $\mathrm{SO} \subseteq T \times T$ is the session order, and $\mathrm{WR} \subseteq T \times T$ is the write-read relation: if transaction t_2 reads the value of key x from transaction t_1, then $t_1 \xrightarrow{\mathrm{WR}} t_2$ (also denoted $t_1 \xrightarrow{\mathrm{WR}(x)} t_2$ to make x explict).

We define the *causal order* as $\mathrm{CO} \triangleq (\mathrm{SO} \cup \mathrm{WR})^+$, which captures the potential causality between transactions. The convergence property of TCC requires a total ordering of transactions, including those concurrent ones that are not causally ordered and possibly write to different keys. Hence, we introduce the *arbitration order* $\mathrm{AO} \subseteq T \times T$, a strict total order such that $\mathrm{CO} \subseteq \mathrm{AO}$. Intuitively, the constraint $\mathrm{CO} \subseteq \mathrm{AO}$ ensures that writes by a transaction t are committed after those that t causally depends on. Note that AO subsumes the well-known *version order* (per key) [1]: assuming that both transactions t_1 and t_2 write to key x, $t_1 \xrightarrow{\mathrm{AO}} t_2$ *if and only if* the version of x written by t_2 is ordered after that written by t_1. In what follows, we use the arbitration order and version order interchangeably when the transactions involved write to *the same key*. In practice, arbitration can be realized by assigning unique timestamps across transactions [9, 41].

Read Atomicity and TCC. Let r be a read of transaction t. If r is the first operation on x in t, then it is called an *external* read operation of t; otherwise, it is an *internal* read operation.

The *internal consistency axiom* Int ensures that, within a transaction, an internal read from key x returns the same value as the last write to or read from key x in the transaction. The *external consistency axiom* Ext ensures that an external read of a transaction t from key x returns the final value written by the last transaction in AO among all the transactions that precede t in terms of CO and write to key x.

Definition 3 (TCC)**.** A history $\mathcal{H} = (T, \mathrm{SO}, \mathrm{WR})$ satisfies TCC if and only if CO is a strict partial order and there exists an arbitration order AO such that the Int and Ext axioms hold.

Our main consideration for basing the TAPs and checking algorithm on the above definition is that it is more suitable for black-box checking over histories since the WR relation can be extracted straightforwardly from a history. However, in order to establish the correctness of our new TCC definition, we prove that it is equivalent to the existing TCC definition by Cerone et al. [18]. The proof is given in [37, Appendix A].

Example 4. Figure 2 visualizes the scenario in Example 2. Since Rachel sees the destination city set by Monica who sees the starting city set by Joe, we have $t_J \xrightarrow{\mathrm{WR}} t_{Mr}$ and $t_{Mw} \xrightarrow{\mathrm{WR}} t_R$, respectively.

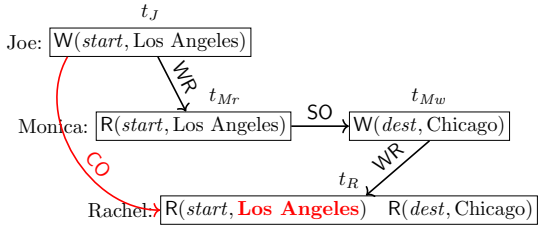


Figure 2: Illustrating TCC's formal definition.

By the definition of CO, $t_J \xrightarrow{\text{CO}} t_R$. Hence, Rachel must also see the starting city set by Joe.

This history satisfies TCC as witnessed by the orders CO = AO such that $t_J \xrightarrow{\text{AO}} t_{Mr} \xrightarrow{\text{AO}} t_{Mw} \xrightarrow{\text{AO}} t_R$.

3.2 Anomalous Patterns for TCC

Arriving at a sound and complete characterization of TCC is non-trivial due to its subtle semantic corner cases. Table 1 describes our characterized 12 TAPs for TCC, defined in terms of WR, CO, and AO. The accompanying visualization is shown in Figure 3.

We derive these fine-grained TAPs along three dimensions. First, we extract different kinds of anomalies by studying an extensive collection of formalizations of isolation levels and consistency properties in the literature [1, 12, 15, 17–19, 22, 38]. Our idea of using fine-grained TAPs to capture TCC anomalies is particularly inspired by the "bad patterns" for formalizing non-transactional CC [15], the forbidden phenomena for characterizing isolation levels (excluding TCC) proposed by Berenson et al. [12], and the data anomalies presented in Adya's thesis [1].

Second, we incrementally devise TAPs in terms of TCC's four semantic building blocks, namely read committed, no fractured reads, causal consistency, and data convergence (see also Section 2). Specifically, six TAPs (a–f) are defined for read committed. Two TAPs (h and i) are used to characterize the fractured reads anomaly. Two TAPs (g and j) are used to capture the causality relation between transactions. Another two TAPs (k and l) characterize the conflicting version orders. All these TAPs additionally underly the modularity of our characterization (resp. checker) for defining (resp. check) weaker isolation guarantees. For example, by combining the eight TAPs (a–f, h, and i), we can soundly and completely characterize RA; accordingly, by disabling checking the other four TAPs, our TCC checking algorithm naturally becomes one for RA.

Finally, to validate our TAPs, we study 3097 anomalous histories collected from executing the production database engines [14, 30, 59, 60] and can successfully associate each TCC anomaly with one TAP. See Table 3, Section 5.3 for the statistics.

Sound and Complete Characterization of TCC. The following theorem shows that, to detect TCC violations in a history, it suffices to find TAPs in the history. Conversely, if there is no TAP found, the history satisfies TCC. Our proof is given in [37, Appendix B].

Theorem 1. A history satisfies TCC if and only if it does not contain any instances of the TAPs.

In the following, we showcase two kinds of TAPs using a single history. As we will see in Section 5.3, many collected histories in practice contain multiple TAPs. Our tool can identify *all* of them. In contrast, most of the existing checkers miss certain anomalies.

Example 5 (TAPs)**.** Figure 4 illustrates both TAP-l and TAP-h in a single history. First, in Figure 4a, as $t_4 \xrightarrow{\text{WR}(y)} t_3$, $t_2 \xrightarrow{\text{SO}} t_3$, and both t_2 and t_4 write to x, we have $t_4 \xrightarrow{\text{AO}} t_2$. Second, as shown in Figure 4b, $t_2 \xrightarrow{\text{CO}} t_5$ (due to $t_2 \xrightarrow{\text{SO}} t_3 \xrightarrow{\text{WR}(z)} t_5$), $t_4 \xrightarrow{\text{WR}(x)} t_5$, and both t_2 and t_4 write to x. Hence, we have $t_2 \xrightarrow{\text{AO}} t_4$. Then there is a cycle in AO, i.e., $t_4 \xrightarrow{\text{AO}} t_2 \xrightarrow{\text{AO}} t_4$. Finally, in Figure 4c, $t_4 \xrightarrow{\text{WR}(x)} t_5$ and $t_4 \xrightarrow{\text{CO}} t_2$ (due to $t_4 \xrightarrow{\text{WR}(x)} t_1 \xrightarrow{\text{SO}} t_2$). We also have $t_2 \xrightarrow{\text{WR}(y)} t_5$ for $y \neq x$ and both t_2 and t_4 write to x. Hence, we have $t_2 \xrightarrow{\text{AO}} t_4$.

In addition to its soundness and completeness, our TAP-based characterization also facilitates understanding and debugging the violations found. We devise our TAPs to correspond to different kinds of database flaws. For example, the above TAP-h is typically caused by a flaw in transaction atomicity maintenance while TAP-l usually results from erroneous conflict resolution when replicas are synchronized. In contrast, existing SAT or SMT-solver-based checkers may return incomprehensible unsatisfied clause [55, 59, 60]; tools based on cycle detection may return a coarse-grained cycle [14], making it difficult to identify the root cause.

4 THE C4 ALGORITHM

C4 detects *all* TAPs in a history. Algorithm 1 shows its pseudocode.

4.1 Overview

C4 identifies TAPs in a directed graph with the node set T and the edge set obtained from SO, WR, CO, and AO. Specifically, in the procedure CheckTCC (line 1), it first builds CO from SO and WR and checks the patterns TAP-a to TAP-h, TAP-j, and TAP-k. Then it builds AO and checks the patterns TAP-i and TAP-l.

To facilitate efficient reachability checking and cycle detection between transactions in the graph, we associate each transaction t 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 $vec(s)$ gives the element of vec on session s. The join operator $\sqcup$ on vectors is defined as $vec_1 \sqcup vec_2 \triangleq \langle \lambda s \in \mathbb{S}.\ \max\{vec_1(s), vec_2(s)\} \rangle$.

C4 maintains the following *invariant* about vectors: transaction t_2 is reachable from t_1 if and only if $Vec[t_1] \sqsubseteq Vec[t_2]$. Therefore, cycle detection amounts to testing whether there are two transactions t_1 and t_2 such that $Vec[t_1] = Vec[t_2]$. Hence, when an edge outgoing from a transaction t is added, we need to join $Vec[t]$ to the vectors of transactions that are reachable from t (line 10). C4 realizes this in a DFS traversal (line 7 in UpdateVec). Thanks to the vector invariant, this traversal can backtrack early once it reaches a transaction t' with a larger vector, i.e., $Vec[t'] \sqsupseteq Vec[t]$ (line 8).

Example 6 (Updating Vectors)**.** Consider the history in Figure 5. It consists of two sessions s_1 and s_2 that contain 100 and 2 transactions, respectively. Suppose that the edge $t_{102} \xrightarrow{\text{WR}} t_3$ is added first. The second component of vectors of t_3, t_4, . . ., and t_{100} has been set to 2.

Table 1: 12 TAPs and their description corresponding to Figure 3.

TAP	Description	TAP	Description
(a)	A transaction reads a value from thin air.	(g)	The relation SO ∪ WR is cyclic.
(b)	A transaction reads a value written by an aborted transaction.	(h)	Transaction t_3 reads x from t_1 and $y \neq x$ from t_2. t_2 also writes to x such that $t_1 \xrightarrow{\text{CO}} t_2$. This is a *fractured reads* anomaly.
(c)	A transaction reads from a future write within the same transaction.	(i)	Transaction t_3 reads x from t_1 and $y \neq x$ from t_2. t_2 also writes to x such that $t_1 \xrightarrow{\text{AO}} t_2$. This is a *fractured reads* anomaly and is a general case of (h) which requires $t_1 \xrightarrow{\text{CO}} t_2$.
(d)	Transaction t reads value v of x from transaction $t' \neq t$ but t has written $v' \neq v$ to x before this read.	(j)	Transaction t reads the initial value $\perp_x$ of x. However, there exists a transaction $t' \xrightarrow{\text{CO}} t$ that writes $v' \neq \perp_x$ to x.
(e)	Transaction t reads value v of x from transaction $t' \neq t$ but v is not the last value written to x by t' (assuming no write to x before this read in t).	(k)	Transaction t_3 reads x from transaction t_1. There exists a transaction t_2 that also writes to x such that $t_1 \xrightarrow{\text{CO}} t_2 \xrightarrow{\text{CO}} t_3$. This is undesirable as it would result in a cycle in AO.
(f)	A transaction reads from a key twice with different values, but without writing to it in between.	(l)	Transaction t_3 reads x from transaction t_1. There exists a transaction t_2 that also writes to x such that $t_1 \xrightarrow{\text{AO}} t_2 \xrightarrow{\text{CO}} t_3$. This is a general case of (k), which requires $t_1 \xrightarrow{\text{CO}} t_2$.

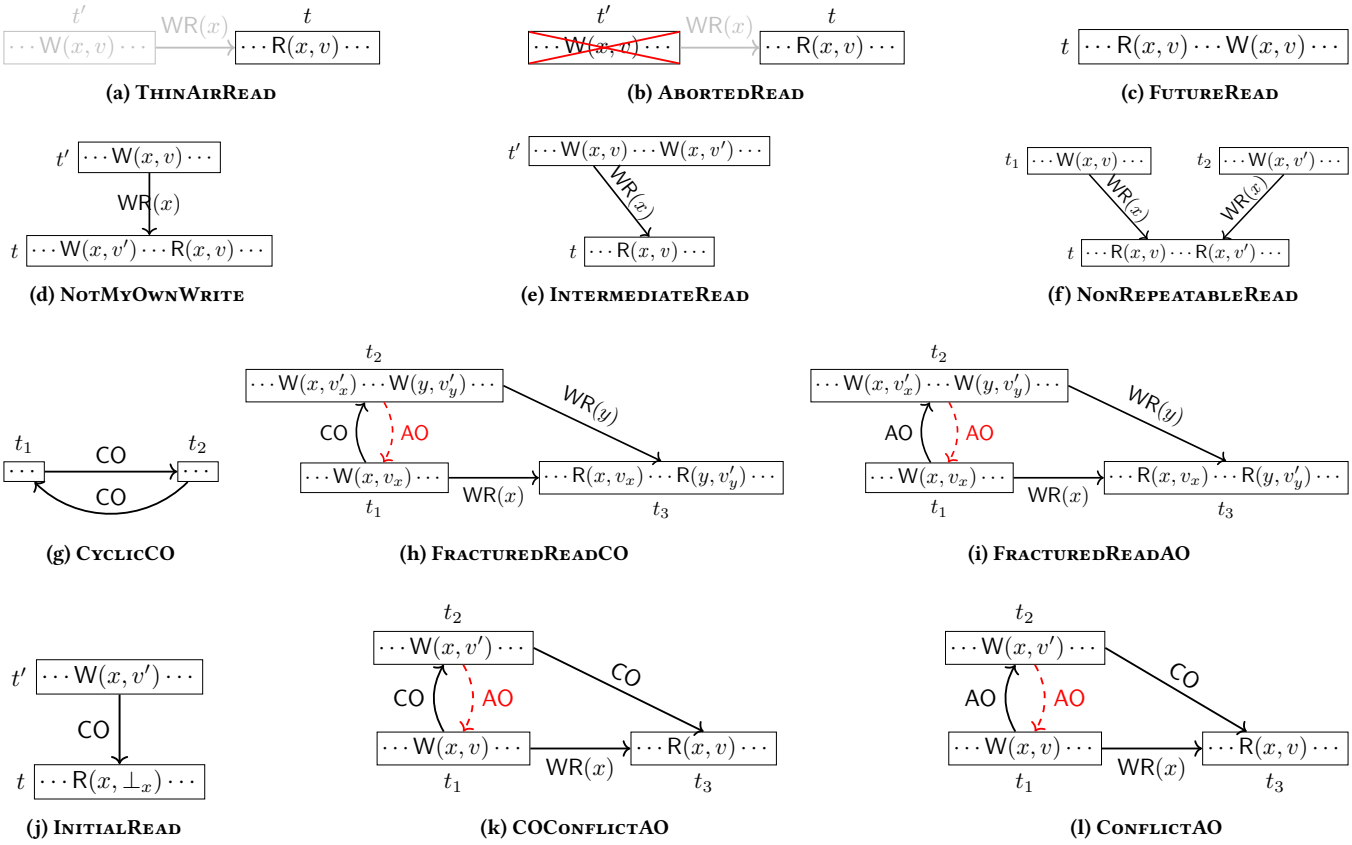
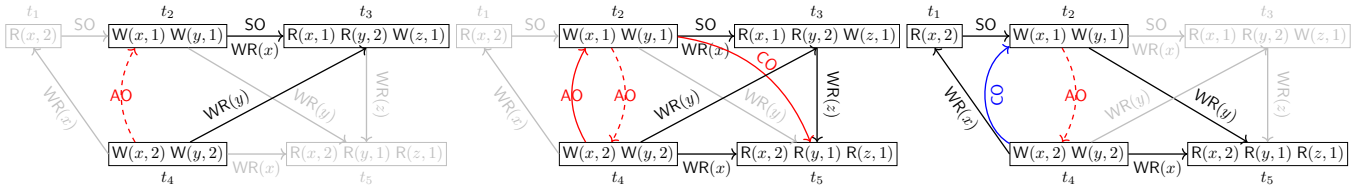


Figure 3: Visualizing the 12 TAPs.



(a) Version order from t_4 to t_2 **(b) ConflictAO (TAP-l)** **(c) FracturedReadCO (TAP-h)**

Figure 4: TAP-l and TAP-h coexist in a single history.

Algorithm 1 The C4 algorithm for checking history $\mathcal{H} = (T, \text{SO}, \text{WR})$ against TCC

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
$Vec[t]$: vector for transaction t
taps: the set of anomalous patterns found, initially $\emptyset$

1: **procedure** CheckTCC()
2: BuildCO() ▷ see [37, Appendix C]
3: CheckCOTAP() ▷ TAP-a to TAP-h, TAP-j, and TAP-k
4: BuildAO()
5: CheckAOTAP() ▷ TAP-i and TAP-l

6: **procedure** UpdateVec(t)
7: **for** $t' \in T$ in DFS traversal from t
8: **if** $Vec[t'] \sqsupseteq Vec[t]$
9: **backtrack**
10: $Vec[t'] \leftarrow Vec[t'] \sqcup Vec[t]$

11: **procedure** BuildAO()
12: **for** $t_1, t_2, t_3 \in T$ such that $\exists x \in \mathsf{K}.\ t_1 \xrightarrow{\text{WR}(x)} t_3 \wedge t_2 \in \text{WT}_x \wedge Vec[t_2] \sqsubseteq Vec[t_3]$
13: $E \leftarrow E \cup \{(t_2, t_1)\}$
14: **for** $t \in T$
15: UpdateVec(t)

16: **procedure** CheckCOTAP()
17: **for** $t \in T$
18: check anomalous patterns TAP-a to TAP-f ▷ details omitted
19: **if** $\exists x \in \mathsf{K}.\ (t \in \text{RT}_x^{\perp_x} \wedge (\exists t' \neq t_\perp \in \text{WT}_x.\ Vec[t'] \sqsubseteq Vec[t]))$
20: $taps \leftarrow taps \cup \{\text{TAP-j}\}$ ▷ InitialRead
21: **if** $\exists t_1, t_2 \in T.\ Vec[t_1] = Vec[t_2]$
22: $taps \leftarrow taps \cup \{\text{TAP-g}\}$ ▷ CyclicCO
23: **if** $\exists x \in \mathsf{K}\ \exists t_1, t_2, t_3 \in T.\ t_1 \xrightarrow{\text{WR}(x)} t_3 \wedge t_2 \in \text{WT}_x \wedge Vec[t_1] \sqsubseteq Vec[t_2] \sqsubseteq Vec[t_3]$
24: **if** $\exists y \neq x \in \mathsf{K}.\ t_2 \xrightarrow{\text{WR}(y)} t_3$
25: $taps \leftarrow taps \cup \{\text{TAP-h}\}$ ▷ FracturedReadCO
26: **else**
27: $taps \leftarrow taps \cup \{\text{TAP-k}\}$ ▷ COConflictAO

28: **procedure** CheckAOTAP()
29: **if** $\exists x \in \mathsf{K}\ \exists t_1, t_2, t_3 \in T.\ t_1 \xrightarrow{\text{WR}(x)} t_3 \wedge t_2 \in \text{WT}_x \wedge Vec[t_1] = Vec[t_2]$
30: **if** $\exists y \neq x \in \mathsf{K}.\ t_2 \xrightarrow{\text{WR}(y)} t_3$
31: $taps \leftarrow taps \cup \{\text{TAP-i}\}$ ▷ FracturedReadAO
32: **else if** $t_2 \xrightarrow{\text{CO}} t_3$
33: $taps \leftarrow taps \cup \{\text{TAP-l}\}$ ▷ ConflictAO

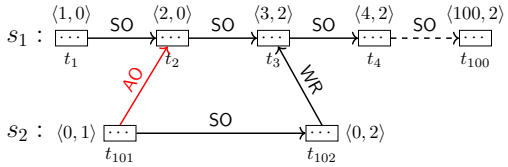


Figure 5: An illustration of updating vectors.

Then an AO edge $t_{101} \xrightarrow{\text{AO}} t_2$ is added. As a result, we should update the vectors of transactions t_2, t_3, ..., and t_{100} with $Vec[t_{101}] = \langle 0, 1\rangle$. However, since $Vec[t_3] = \langle 3, 2\rangle \sqsupseteq Vec[t_{101}]$, the traversal can terminate early at t_3.

4.2 Algorithm

We now describe how C4 works in more details. In the following we abuse the notation WR and $\text{WR}(x)$ for transactions to denote the write-read relation between operations.

Procedure BuildCO. C4 builds CO by examining transactions individually. Consider a transaction t. It first builds SO and initializes $Vec[t]$. Then C4 examines each operation of t and builds WR. Finally, C4 calls the procedure UpdateVec (line 6) where it uses $Vec[t]$ to update vectors of the 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 $\mathcal{G}_{\text{CO}}$. See [37, Appendix C] for more details.

Procedure CheckCOTAP. C4 checks 10 CO-related anomalous patterns on $\mathcal{G}_{\text{CO}}$ (line 16). It is straightforward to check TAP-a to TAP-f given the sets like S_a and $S_{r'}$ constructed during BuildCO (line 18). The details are omitted here. Checking TAP-g, TAP-h, TAP-j, and TAP-k involves reachability checking and cycle detection in terms of CO. Specifically, C4 relies on $Vec[t'] \sqsubseteq Vec[t]$ to capture the condition $t' \xrightarrow{\text{CO}} t$ in TAP-j (line 19). Cycle detection in TAP-g is achieved by checking whether $Vec[t_1] = Vec[t_2]$ (line 21). Both TAP-h and TAP-k 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 23). They differ in whether $t_2 \xrightarrow{\text{CO}} t_3$ is actually a special case $t_2 \xrightarrow{\text{WR}(y)} t_3$ for some key y (line 24).

Procedure BuildAO. The remaining two anomalous patterns, namely TAP-i and TAP-l, contain AO edges. In BuildAO (line 11), C4 examines each triple of transactions t_1, t_2, and t_3. If $t_1 \xrightarrow{\text{WR}(x)} t_3$, t_2 writes x, and $t_2 \xrightarrow{\text{CO}} t_3$ (in $\mathcal{G}_{\text{CO}}$; captured by $Vec[t_2] \sqsubseteq Vec[t_3]$), then C4 adds an edge $t_2 \xrightarrow{\text{AO}} t_1$. The transitivity of AO is ensured by updating vectors of transactions (line 15). Note that we *cannot* update vectors and add AO edges alternately, because otherwise from $Vec[t_2] \sqsubseteq Vec[t_3]$ we can conclude only $t_2 \xrightarrow{\text{AO}} t_3$ instead of $t_2 \xrightarrow{\text{CO}} t_3$ as desired. Denote the resulting graph by $\mathcal{G}_{\text{AO}}$.

Procedure CheckAOTAP. Both TAP-i and TAP-l contain three transactions t_1, t_2, and t_3 such that $t_1 \xrightarrow{\text{WR}(x)} t_3$ for some key x, t_2

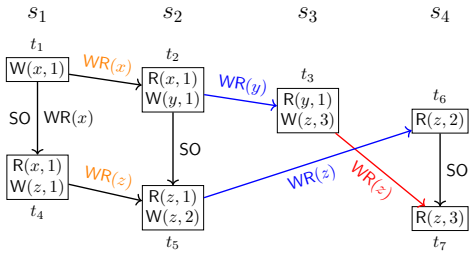


Figure 6: An illustration of tree clocks for updating vectors. Transactions in the same session are vertically aligned.

writes x, and t_1 and t_2 form a AO cycle in $\mathcal{G}_{\text{AO}}$; the last condition is detected by $Vec[t_1] = Vec[t_2]$ (line 29). They differ in the relation between t_2 and t_3: TAP-l requires $t_2 \xrightarrow{\text{CO}} t_3$ (line 32), while TAP-i requires a more fine-grained condition, i.e., $t_2 \xrightarrow{\text{WR}(y)} t_3$ for some key $y \neq x$ (line 30). For TAP-l, we cannot replace $t_2 \xrightarrow{\text{CO}} t_3$ with $Vec[t_2] \sqsubseteq Vec[t_3]$, since the latter implies only $t_2 \xrightarrow{\text{AO}} t_3$ in $\mathcal{G}_{\text{AO}}$.

4.3 Optimization with Tree Clocks

C4 frequently calls UpdateVec to update vectors, where it compares and joins vectors. Each of the two basic operations on vectors requires $\Theta(|\mathbb{S}|)$ time, where $|\mathbb{S}|$ is the number of sessions. UpdateVec will become a computational bottleneck when $|\mathbb{S}|$ is large.

To reduce the time spent on manipulating vectors, we adapt *tree clocks* [43] in our utilization of vectors. While one tree clock per thread (or session) in concurrent program analysis is sufficient, each transaction in our setting has its own tree clock. This is because transactions that are *not* the latest ones in their sessions may also need to update their vectors.

Comparing tree clocks takes constant time and joining tree clocks takes time proportional to the number of entries being modified. The following example illustrates how C4 uses tree clocks in updating vectors, i.e., the UpdateVecprocedure.

Example 7. Consider the history in Figure 6 with four sessions. Suppose C4 examines transactions in BuildCO in the order of t_1, t_2, ..., t_7. When dealing with t_7, C4 adds the edge $t_3 \xrightarrow{\text{WR}(z)} t_7$ and then joins $Vec[t_3]$ with $Vec[t_7]$. With simple vectors, the join operation iterates over *all* entries of $Vec[t_3]$ and $Vec[t_7]$. However, via $t_5 \xrightarrow{\text{WR}(z)} t_6 \xrightarrow{\text{SO}} t_7$, t_7 has learned t_5 on session s_2, which is newer than what t_3 learned about session s_2, namely t_2. Transitively, via $t_4 \xrightarrow{\text{WR}(z)} t_5$, t_7 has learned t_4 on session s_1, which is newer than what t_3 learned (via $t_1 \xrightarrow{\text{WR}(x)} t_2$) about session s_1, namely t_1. By recording a transaction's knowledge of other transactions from different sessions in a hierarchical tree structure, tree clocks can avoid examining certain entries of vectors. For example, the join of $Vec[t_3]$ with $Vec[t_7]$ stops after examining the entry for session s_2, without examining the entry for session s_1.

5 EXPERIMENTS

In this section we conduct a comprehensive evaluation of C4, along with the state-of-the-art black-box isolation checkers, and answer the following questions with respect to the SIEGE+ principle:

(1) **Effective** (Section 5.2): Can C4 detect TCC violations in production database engines?
(2) **Complete** (Section 5.3): Can C4 identify more anomalies than the state-of-the-art checkers?
(3) **Informative** (Section 5.4): Are the violations that C4 report understandable?
(4) **Efficient** (Section 5.5): How efficient is C4? Can C4 outperform the baselines under various workloads and is it scalable for large workloads?

Along with answering these questions, we experimentally validate C4's **soundness** and demonstrate its **generality** by showing its capability for detecting TCC bugs in production database engines of *different kinds*.

5.1 Setup, Histories, and Benchmarks

Experimental Setup. We employ a PostgreSQL v15.2 instance to generate *valid* histories without SI (thus TCC) violations by setting the isolation level to *repeatable read* (implemented as SI in PostgreSQL [51]). We co-locate the client threads and the PostgreSQL instance (or other target databases) on a local machine with an Intel(R) Xeon(R) E5-2620 v3 CPU and 64GB memory.

We adapt PolySI's workload generator and history collector [30] into C4: each client issues to the database a stream of transactions, which are transformed from the generated key-value operations to SQL queries for the interactions with relational databases,[2] and collects the execution history. We prototype C4's core checking algorithm in around 2k LOC in Java. PolySI's adaption, together with the workload generator and the history collector/compiler, includes around 2k LOC in Rust. To meet the *unique value* requirement (Section 2), we use counters for values written to each key.

For the TCC checking over a history, we set the timeout to 10 minutes for each checker.

Histories. We employ the parametric workload generator to produce *general* transaction workloads. Its parameters are: the number of client sessions (25 by default), the number of transactions per session (200 by default), the number of read/write operations per transaction (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 are already sufficient to distinguish C4 from the competitors (see Section 5.5.1). We further demonstrate C4's scalability with significantly larger workloads (Section 5.5.4).

Benchmarks. Our experiments also consider six benchmarks commonly used by existing black-box isolation checkers [30, 55, 60]. First, we include three synthetic benchmarks, each of which contains only serializable histories of at least 10k transactions (also satisfying TCC). TPC-C [57] is a standard benchmark for online transaction processing, consisting of five types of transactions (e.g., new orders and payment). The configuration includes one warehouse, 10 districts, and 30k customers. C-RUBiS [52] is an eBay-like bidding system; users can register and bid for items. The dataset

[2]For multi-column tables, each table cell is represented as a compound key (i.e., "TableName:PrimaryKey:ColumnName") and a single value.

Table 2: Summary of tested database engines. Multi-model in YugabyteDB refers to relational DBMS, document store, and wide-column store. C4 finds more anomalies (highlighted in yellow) from the anomalous histories reported by other checkers.

Database Engine	GitHub Stars	Kind	Checker	Tested Version	Tested Isolation	Violation Cause
New violations:						
AntidoteDB	790	Key-value	C4	v0.2.2	TCC	TAP-g, TAP-h, TAP-k
MariaDB-Galera	4.9k	Relational	C4	v10.4.22	SI	TAP-e, TAP-f, TAP-h, TAP-i, TAP-k, TAP-l
YugabyteDB	8k	Multi-model	C4	v2.11.1	SI	TAP-i, TAP-k, TAP-l
CockroachDB	27.6k	Relational	C4	v21.2.5	SER	TAP-k
Known bugs:						
MySQL-Galera	420	Relational	dbcop	v25.3.26	SI	**TAP-d**, **TAP-f**, **TAP-h**, **TAP-i**, TAP-k, **TAP-l**
			CausalC	v3.2	SI	TAP-j
Dgraph	19.5k	Graph	PolySI	v21.12	SI	**TAP-i**, TAP-k, **TAP-l**
MongoDB	24.3k	Document	Viper	v4.2.6	SI	**TAP-b**, TAP-c, TAP-g, **TAP-l**
CockroachDB	27.6k	Relational	dbcop	v2.1	SER	**TAP-i**, TAP-k, **TAP-l**

contains 20k users and 200k items. C-Twitter [34] is a Twitter clone where users can tweet and follow/unfollow other users, and key accesses follow the zipfian distribution.

Additionally, we benchmark the performance of each checker using three representative *general* datasets archived by [30], i.e., GeneralRH (read-heavy with 95% reads), GeneralWH (write-heavy with 70% writes), and GeneralRW (50% reads). Each history contains 16k transactions and 320k operations.

5.2 Detecting TCC Violations

Reproducing Known Bugs. C4 successfully reproduces *all* known TCC bugs in a substantial collection of 3097 anomalous histories [14, 30, 59, 60] . These histories were collected from five earlier releases of four production databases of different types, i.e., the relational databases MySQL-Galera and CockroachDB, the graph DBMS Dgraph, and the document store MongoDB. Table 2 depicts the details including their popularity (on GitHub), tested releases, tested isolation levels, etc. It is worth mentioning that TCC bugs already manifest extensively when checking *stronger* isolation levels such as SI and SER.

New Violations Found. We also demonstrate C4's effectiveness, along with its generality, by examining recent releases of four popular and heavily-tested database engines of *different kinds*, i.e., the emerging key-value database AntidoteDB [4],[3] YugabyteDB [58] supporting multiple data models, MariaDB-Galera [21], and CockroachDB [36]. These databases claim to provide TCC or even stronger isolation guarantees; see Table 2 for details.

We have detected and reported novel TCC bugs in all these database engines. In particular, both AntidoteDB and MariaDB have confirmed and fixed the reported bugs [6, 7]. MariaDB labeled the bug as "critical" since it affected a wide range of 9 releases (v10.3–v10.11), as well as its Galera cluster. AntidoteDB quickly localized the root cause of our reported bug (a Python client issue with session guarantees), thanks to the informative session dependencies in the counterexample. The YugabyteDB and CockroachDB developers are investigating the reported bugs.

[3]The emerging app development platform ElectricSQL [25] bases its core replication layer on AntidoteDB.

5.3 Complete Violation Detection

An ideal checker will identify *all* violations in a given history. This is highly desirable: more bug coverage can detect more defects in a database and can save developers' time as isolation bug occurrences are highly unpredictable due to random workload (or test case) generation. Through an empirical analysis, we show that, given the same number of anomalous histories, C4 can report significantly more anomalies than almost all the state-of-the-art checkers considered in this paper. With a further investigation, we also show that the current implementations of these checkers inherently cannot detect certain anomalies. Therefore, simply running them multiple times for the same histories or even making them continue to search would not solve the problem.

State-of-the-Art Isolation Checkers. Our comparison focuses on the black-box checking of isolation guarantees and considers six state-of-the-art checkers. First, our comparison includes two existing (T)CC checkers. CausalC [15, 59] is based on *answer set programming* (ASP) [27], where causal dependencies are encoded into an ASP logic program before running the Clingo solver [48] to compute the solution. Although it is specifically designed for checking CC, CausalC can also be used to check TCC when histories contain only simple transactions of a single read/write. dbcop [14] is the most efficient TCC checker to date. Given a history, it constructs the causal dependency graph and then searches for a cycle via DFS without using an off-the-shelf solver.

Second, we consider three checkers based on SAT and SMT solving, namely PolySI [30], Viper [60], and Cobra [55]. They all utilize the MonoSAT solver [11], tailored for efficient checking of graph properties such as acyclicity, and employ specific optimizations to accelerate the checking procedure. These checkers are designed for checking *stronger* isolation levels (PolySI and Viper for SI, and Cobra for SER), which can also be used for detecting TCC violations and for TCC validation.

Finally, we also consider Elle [35] which is part of the popular testing framework Jepsen [31]. In contrast to the above five checkers which only work with read-write registers, Elle can also leverage, e.g., list-specific data models and "append" operations, to efficiently infer transaction dependencies. The latest release of Elle does not support checking TCC yet [33]. In our experiments, we therefore

Table 3: Distribution of TAPs identified by C4 from 3097 anomalous histories archived by existing checkers and 4 histories corresponding to the new violations found.

TAP-a	TAP-b	TAP-c	TAP-d	TAP-e	TAP-f	#TAPs
0	3	1	71	1	15	9310
TAP-g	TAP-h	TAP-i	TAP-j	TAP-k	TAP-l	#Hist
2	209	2947	11	3079	2971	3111

use its SI checking component (or Elle-SI), the weakest isolation level supported by Elle which is stronger than TCC. Hence, Elle-SI is expected to detect TCC bugs.

Finding More Anomalies by C4. Thanks to the underlying complete characterization, C4 can detect *all* TCC anomalies (or TAPs) in a history. Surprisingly, the number of identified TAPs (9310 in total) is far more than the number of histories (3111). As the state-of-the-art checkers (except Elle) terminate upon detecting a violation, the number of anomalies reported by them is essentially 3111. That is, C4 finds 6199 more anomalies with the same collection of histories. We highlight in Table 2 the corresponding TAPs. In particular, C4 captures, in total, seven more types of anomalies than dbcop in testing MySQL-Galera and CockroachDB. The advanced SI checkers Viper and PolySI did not report two kinds of anomalies, which are successfully identified by C4.

Table 3 shows the distribution of TAPs identified by C4 from all these 3111 anomalous histories. Most of them involve *fractured reads* (TAP-h and TAP-i) or *version-order conflicts* (TAP-k and TAP-l). However, simple, yet crucial bugs (TAP-b–TAP-f, 91 in total) are not negligible.[4] For example, 15 *non-repeatable reads* anomalies (labeled "critial" by MariaDB) are found; many transactions in MySQL-Galera could not read their own writes (TAP-d, 71 in total); MongoDB even reads aborted writes (TAP-b, three in total). Table 2 shows the distribution of these TAPs among different database engines.

Regarding new violations, C4 detects various TAPs in four database engines. In particular, MariaDB-Galera exhibits six different types of anomalies, including simple bugs like reading intermediate writes and non-repeatable reads; three kinds of TAPs manifest in AntidoteDB and YugabyteDB.

(In)completeness. Given that most of the existing checkers terminate by design upon the first detected bug, would it be possible to find more bugs if we run them multiple times for the same history or even make them continue to search for more, different bugs? The answer is unfortunately *no*, for almost all of them.

We craft for each TAP a simple anomalous history (or test case) to better understand the existing checkers' current limitation in finding TCC bugs. See [37, Appendix D] for the 12 test cases. Table 4 shows the results where none of the checkers, except C4 and PolySI (also complete by design [30]), can catch all the anomalies. In particular, CausalC fails to pass all the test cases, which is expected as it only supports simple transactions, while our test cases are designed with general transactions. dbcop cannot identify reading aborted reads (TAP-b). Viper and Cobra miss both the *intermediate* and *non-repeatable* reads anomalies (TAP-e and TAP-f). Elle fails

[4]We have not found any thin-air reads (TAP-a), which is not surprising. This kind of bug rarely occurs unless there is severe data corruption.

Table 4: Missed bugs by each checker.

Checker	Missed Bugs	Checker	Missed Bugs
C4	none	dbcop	TAP-a, TAP-b, TAP-c
CausalC	all	Viper	TAP-d, TAP-e, TAP-f, TAP-g
PolySI	none	Cobra	TAP-d, TAP-e, TAP-f
		Elle-SI	TAP-c, TAP-g

to detect CO cycles (TAP-g) and future reads (TAP-c).[5] Their incompleteness in finding bugs stems mainly from the underlying incomplete or coarse-grained characterizations of isolation guarantees. Moreover, some checkers such as Elle trade off completeness for less checking overhead in practice [29]. Our analysis also suggests that PolySI should be able to find all anomalies from a history with more runs (the number of runs is however unpredictable) or without termination till the entire history is processed.

5.4 Understanding Violations Found

In addition to the incompleteness, some checkers also return *less informative* counterexamples. Both Viper and Cobra report unsatisfied clauses by the MonoSAT solver. CausalC uses atomic propositions to represent a violation. Such solver-returned counterexamples do not produce semantic interpretations of the violations. The dbcop tool returns a coarse-grained cycle upon finding a violation, rendering the debugging difficult, e.g., causality violation (TAP-g) and fractured reads (TAP-h) are indistinguishable.

Thanks to the underlying TAP-based characterization, C4 is capable of returning fine-grained, informative counterexamples and identifying the violation causes in terms of TAPs. In order to make the counterexamples more understandable, we have integrated into C4 the Graphviz tool [28] to visualize the violating scenarios. Below we showcase four new bugs of different kinds, together with their violating scenarios. We defer representative bugs of other kinds (e.g., TAP-e) to [37, Appendix E].

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.

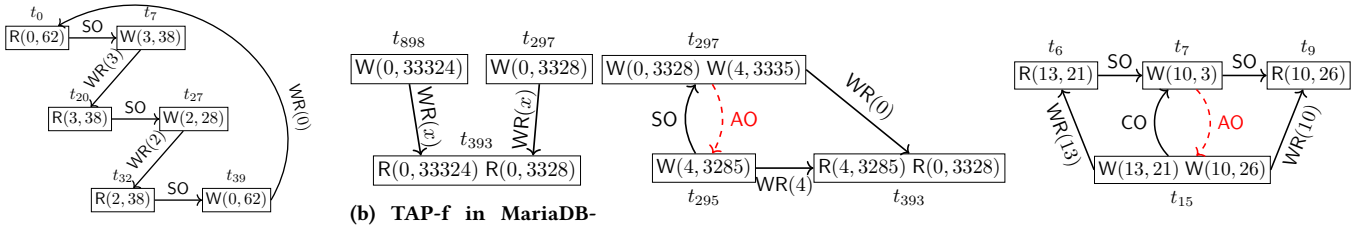
TAP-f and TAP-h in MariaDB-Galera. Figure 7b presents the NonRepeatableRead anomaly where two consecutive reads on the same key 0 in transaction t_{393} fetch different values written by two separate transactions.

Figure 7c depicts how the *fractured reads* occur. Transaction t_{393} has two reads on keys 0 and 4, respectively. $\mathrm{R}(4, 3285)$ fetches the value written by transaction t_{295}; $\mathrm{R}(0, 3328)$ reads the value installed by transaction t_{297} that also writes key 4, i.e., $\mathrm{W}(4, 3335)$. 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-f and TAP-h coexist in this history, with both transactions t_{297} and t_{393} involved. C4 precisely captures these two anomalies from a single history with distinguishable violating scenarios.

TAP-k in YugabyteDB. Figure 7d shows the conflict between the CO and AO orders established from the history. As transaction t_9

[5]Elle's developer quickly confirmed these two bugs [33].



(a) TAP-g in AntidoteDB **(b) TAP-f in MariaDB-Galera** **(c) TAP-h in MariaDB-Galera** **(d) TAP-k in YugabyteDB**

Figure 7: Visualized TCC violations found in production databases by C4. Only involved operations per transaction are shown.

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. However, t_7 has already been causally ordered after t_{15}. Hence, a CO-AO conflict arises.

5.5 Performance Evaluation

We conduct a comprehensive, in-depth performance analysis of C4. Regarding the competitors, as the non-solver tool dbcop is currently the only one specifically designed for checking TCC, we also

- extend CausalC to a new tool, CausalC+, for checking general transactions against TCC via the ASP-based solving;
- implement a *strong baseline* checker called TCC-Mono which leverages the advanced MonoSAT solver to search for cycles; and
- involve PolySI as the representative for checking stronger isolation levels (which has already been shown to outperform Cobra [30]).

Our goal is twofold. First, we want to compare C4 with various techniques utilized by the state of the art, e.g., with or without a solver *and* using different solvers, given the recent trend of applying solving to checking isolation guarantees [30, 55, 59, 60]. Second, although checking stronger isolation levels such as SI is of higher complexity in theory, existing checkers have been shown to be highly efficient in practice with domain-specific optimizations. For example, as we will see, PolySI even outperforms the state-of-the-art TCC checker dbcop. Hence, we want to explore whether stronger isolation checkers could replace C4 for *validating* TCC.[6]

Additionally, we consider Elle, an isolation checker of a different kind that supports list-specific APIs. Given its success in effectively finding isolation bugs in commercial databases and its high checking efficiency [32, 35], we want to make C4 compatible with the Jepsen framework, particularly with its produced "append" histories, to complement Elle with TCC checking.

5.5.1 Performance Comparison. As shown in Figure 8, C4 substantially surpasses all the competitors, including the strong baseline TCC-Mono, with respect to checking efficiency under a wide range of workloads. C4's performance is also fairly stable, at roughly two seconds checking time, for varying read/write ratios and concurrency such as more sessions, transactions per session, operations

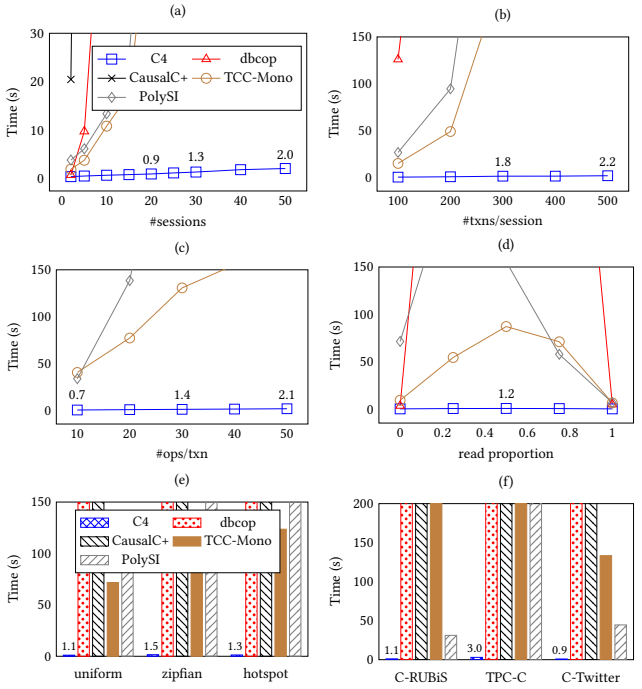


Figure 8: Performance comparison under various workloads. Experiments time out at 10 minutes. For the competitors, data points are not plotted for the timed-out experiments.

per transaction, and skewed key accesses, as well as various benchmarks such as TPC-C. In contrast, the competing tools exhibit exponentially increasing checking time with more concurrency. Note that, 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 outperforms these two checkers given its domain-specific optimizations [30], tailored for certain benchmarks such as C-RUBiS and C-Twitter.

We also measure the memory overhead for all the checkers. Figure 9 compares results under the same setting as in Figure 8. C4 consumes significantly less memory (for storing generated graphs, vectors, and tree clocks) than the competitors under various workloads and benchmarks. In particular, dbcop, the only competitor that does not rely on solving and therefore stores no constraints, is not competitive with C4. 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.

[6] As a stronger isolation checker may return *false positives with respect to TCC* (e.g., lost updates), validation here means that a history is valid up to the isolation level for which the checker is designed. See also the experimental setup in Section 5.1.

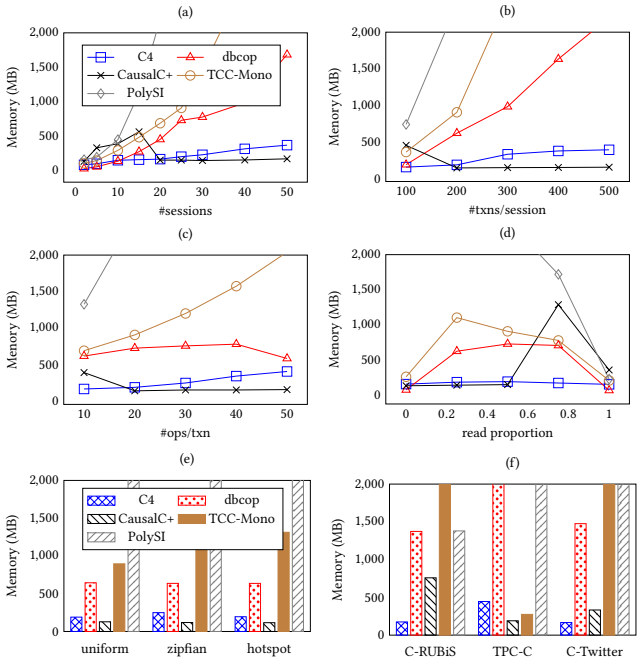


Figure 9: Memory overhead comparison. We only plot each tool's memory usage at timeout (10 minutes).

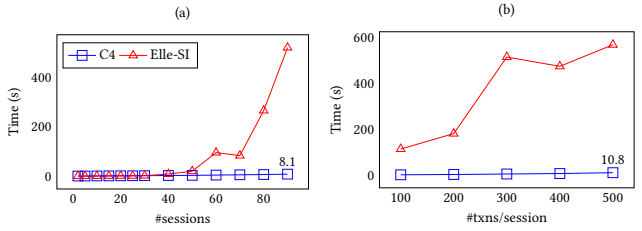


Figure 10: Comparison with Elle-SI (50 operations per transaction, 50% reads, and uniform key-access distribution).

Overall, our experimental results suggest that (i) vectors and tree clocks are more efficient techniques than solving in checking TCC, and (ii) compared to C4, the state-of-the-art stronger isolation checker (on read-write registers) is inefficient in validating TCC.

5.5.2 Compatibility with Jepsen. Elle, as part of the Jepsen framework [31], has shown promising efficiency in checking isolation guarantees. During testing, Jepsen produces workloads of reads and *append* operations, from which Elle can efficiently infer the version order of appends (or writes in our case). To make C4 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. We have integrated C4 into Jepsen. As shown in Figure 10, C4 exhibits high, stable checking efficiency with "append" histories, even when more sessions are involved and the workload becomes large. This suggests that C4 can complement Elle for checking TCC.

As expected, C4 incurs less checking overhead than Elle-SI. C4's high efficiency comes mainly from the reduced complexity of checking TCC and its acceleration in graph traversals by utilizing vectors

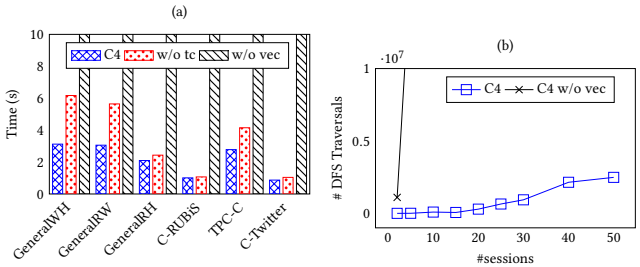


Figure 11: Differential analysis of C4.

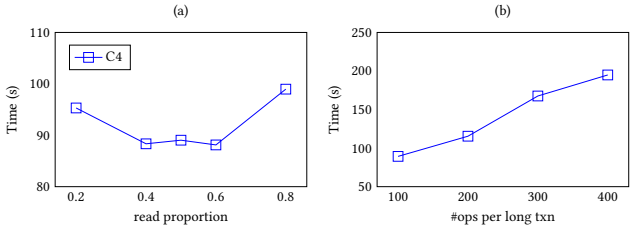


Figure 12: C4's overhead under large workloads with one billion keys and one million transactions.

and tree clocks, as we will show next. Hence, it is fair to say that, for "append" histories, the state-of-the-art stronger isolation checker cannot replace C4 for validating TCC either (apart from read-write registers, as we have seen from the comparison with PolySI).

5.5.3 Differential Analysis. To investigate the contributions of C4's two major design choices, namely vectors and tree clocks, we experiment with two variants: (i) C4 without tree clocks and (ii) C4 with neither tree clocks nor vectors. Figure 11a demonstrates the noticeable acceleration produced by each design choice. Note that C-RUBiS or C-Twitter cannot distinguish C4 from the variant (i) since both contain only a small number of sessions and transactions, and there are fewer comparing and joining operations over vectors during the graph traversals.

Thanks to the utilization of vectors, C4 can also significantly accelerate reachability checking and cycle detection in the UpdateVec procedure by reducing a large number of DFS traversals. As shown in Figure 11b, with more concurrency (in terms of more sessions), the number of DFS traversals performed by the variant (ii) grows dramatically, reaching up to 10^{10} by the timeout.

5.5.4 Scalability. To evaluate C4's scalability, we generate transaction workloads with one billion keys, one million transactions, and up to 400 million operations. We experiment with varying read proportions and operations per long transactions (up to 400). As shown in Figure 12, C4 completes the TCC checking of 400 million operations under 200 seconds. We also observe a linear increase of checking time with larger-sized transactions. To conclude, large workloads appear manageable for C4.

6 DISCUSSION

Versatility. Our fine-grained characterization of TCC provides *modular* patterns for soundly and completely checking many other isolation guarantees. Regarding weaker isolation levels, we only need to check a subset of all the TAPs, e.g., TAP-a–TAP-f for *read*

Table 5: Examining the state-of-the-art isolation checkers through the lens of SIEGE+. We omit soundness and effectiveness as all the checkers meet these two criteria. DSG stands for *direct serialization graph* proposed by Adya [1].

Checker	Property Supported	Informative	General	Efficient	Complete
CausalC	TCC	atomic propositions	simple txns only; rw registers	answer set solving	no
dbcop	multiple incl. TCC	coarse-grained cycles	general txns; rw registers	plain graph traversal	no
PolySI	SI	detailed violating scenarios	general txns; rw registers	MonoSAT solving	yes
Viper	SI	unsatisfied clauses	general txns; rw registers	MonoSAT solving	no
Cobra	SER	unsatisfied clauses	general txns; rw registers	MonoSAT solving	no
Elle	multiple excl. TCC	detailed violating scenarios	general txns; rw registers, lists	list-based DSG inference	no
C4	TCC	detailed violating scenarios	general txns; rw registers, lists	vectors & tree clocks	yes

committed, the default isolation level of many databases, and, additionally TAP-h and TAP-i for *read atomicity* (RA), a promising isolation guarantee that has recently been adopted by Facebook's TAO data store [20]. For other isolation levels, one can design and incorporate new TAP(s) correspondingly, e.g., together with new TAP(s) for "lost updates", the above constructed RA checker would be able to check *update atomicity* [18], and C4 can be used to check SI. Moreover, C4 can be applied directly to check (non-transactional) CC, where each read or write in a history is considered as a transaction of a single operation. This is in contrast to CC checkers such as CausalC, which cannot check TCC in general.

C4's key idea of leveraging vectors and the optimization with tree clocks can also be adapted to *graph-based* approaches for checking other isolation guarantees. This would accelerate the traversals of various dependency graphs, such as *polygraphs* [49] (adopted by Cobra, PolySI, and Viper) and Adya's *direct serialization graph* [1].

Checking Stronger Isolation Levels. Isolation bugs are subtle and tricky. Most of the histories produced by the *de facto* randomized testing approach are often free of anomalies. As testing cannot prove the absence of bugs, developers typically generate and check a large number of histories to gain a higher assurance. This would exhaust stronger isolation checkers due to the high complexity of the checking problems. As we have observed, the state-of-the-art SI checker PolySI requires significant time and memory to validate a moderate-sized history in practice. We argue that, despite being incomplete for checking stronger isolation guarantees, a first attempt with C4 is practicable as it can quickly validate millions of transactions in several minutes against their major building blocks including RC, RA, causality, and data convergence. Moreover, as many TCC bugs already manifest when checking SI or SER (see Table 2), C4 can be used to effectively detect all of them.

Alternatively, to achieve a complete, efficient validation up to a stronger isolation guarantee, adapting vectors and tree clocks into the checking problem is a promising approach. This is in contrast to the existing ASP-based or SAT/SMT-based solving approaches, which have been shown to be less efficient.

7 RELATED WORK

Characterizing TCC. The growing number of TCC databases has attracted the attention of the formal methods community. Cerone et al. [18] propose a framework for declaratively specifying TCC (and other isolation guarantees) with the dual notions of visibility (what transactions can observe) and arbitration (as in our case). Biswas and Enea [14] present an alternative axiomatic framework for characterizing TCC which uses the write-read relation as in our case, instead of the visibility relation as in [18]. This framework is more suitable for black-box testing over database histories as the write-read relation can be naturally extracted from a collection of transactions. We derive TAP-l from its TCC formalization.

We base our fine-grained characterization of TCC on the "bad patterns" for formalizing non-transactional causal consistency [15]. Overall, our new characterization is sound and complete, comprises fine-grained patterns that facilitate understanding and debugging the violations found, and paves the way for the complete checking of other isolation guarantees via modular patterns.

Black-box Checking Isolation Guarantees. We focus on the black-box checkers for *isolation bugs*. As shown in Table 5, we examine each of them using the SIEGE+ principle [30], which strengthens SIEGE [35] with completeness. Our C4 checker meets all of its criteria, as demonstrated by our assessment. Existing tools are all sound and effective. Stronger isolation checkers may report false positives, e.g., the lost-update anomaly, with respect to TCC. CausalC does not support general transactions. Solver-based tools such as CausalC, Viper, and Cobra return to developers less understandable counterexamples in the form of atomic propositions or unsatisfied clauses. PolySI improves them by interpreting unsatisfied clauses as detailed violating scenarios. The coarse-grained cycles reported by dbcop are less informative, rendering understanding and debugging violations hard. As shown by our experimental results, C4 significantly outperforms dbcop, as well as the strong baselines that utilize various solving techniques based on answer set programming and MonoSAT. All the checkers except PolySI are incomplete, thereby missing TCC bugs. C4 joins five of the checkers for dealing with read-write registers, and additionally supports list-specific data models as does Elle.

8 CONCLUSION

We have presented the black-box isolation checker C4, along with a novel characterization of TCC via TAPs. We have also established its soundness and completeness. Our extensive experimental results demonstrate C4's fulfillment of SIEGE+'s criteria.

C4 complements the Jepsen testing framework by supporting efficient TCC checking. The modular anomalous patterns and the utilization of vectors and tree clocks can contribute to complete, efficient checking of many other isolation guarantees; building such checkers is our future work. The informative counterexamples can facilitate debugging and help the developers rethink their system designs and implementations.

REFERENCES

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

[2] 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.

[3] 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.

[4] AntidoteDB. 2023. https://www.antidotedb.eu/.

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

[6] The authors of submission 621. Accessed in August, 2023. https://jira.mariadb.org/browse/MDEV-27927.

[7] The authors of submission 621. Accessed in August, 2023. https://github.com/AntidoteDB/antidote/issues/458.

[8] The authors of submission 621. Accessed in May, 2023. Issue #21. https://github.com/jepsen-io/elle/issues/21.

[9] 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.

[10] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *SIGMOD 2013*. ACM, 761–772.

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

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

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

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

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

[16] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. 2004. From session causality to causal consistency. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings.* 152–158. https://doi.org/10.1109/EMPDP.2004.1271440

[17] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. *SIGPLAN Not.* 49, 1 (Jan. 2014), 271–284. https://doi.org/10.1145/2578855.2535848

[18] 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.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] ElectricSQL. 2023. https://electric-sql.com/.

[26] 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.

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

[28] Graphviz. Accessed in May, 2023. Open source graph visualization software. https://graphviz.org/.

[29] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David Basin, Haixiang Li, and Anqun Pan. Accessed in August, 2023. Issue #17. https://github.com/jepsen-io/elle/issues/17.

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

[31] Jepsen. Accessed in May, 2023. https://jepsen.io.

[32] Jepsen. Accessed in May, 2023. Jepsen Analyses. https://jepsen.io/analyses.

[33] Jepsen. Accessed in May, 2023. Supported isolation levels by Elle. https://github.com/jepsen-io/elle/blob/994bc92cbf0802885eb221efc28f160cf30777ca/src/elle/consistency_model.clj.

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

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

[36] Cockroach Labs. Accessed in May, 2023. CockroachDB. https://www.cockroachlabs.com/.

[37] Si Liu, Long Gu, Hengfeng Wei, and David Basin. 2023. *Complete and Efficient Checking of Transactional Causal Consistency in Database Engines.* Technical Report. https://github.com/dracooooo/C4.

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

[39] 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.

[40] 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.

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

[42] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: measuring and understanding consistency at Facebook. In *SOSP 2015*. ACM, 295–310.

[43] 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.

[44] 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.

[45] Microsoft. 2023. Azure CosmosDB DB. https://learn.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[46] MongoDB. 2023. https://www.mongodb.com/.

[47] Neo4j. 2023. https://neo4j.com/.

[48] University of Potsdam. Accessed in May, 2023. Clingo: A grounder and solver for logic programs. https://github.com/potassco/clingo.

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

[50] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. 2016. Causal Consistency: Beyond Memory. *SIGPLAN Not.* 51, 8, Article 26 (2016), 26:1–26:12 pages.

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

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

[53] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2019. PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *ICDCS 2019*. IEEE, 304–316.

[54] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2021. Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. *IEEE Trans. Parallel Distributed Syst.* 32, 3 (2021), 527–542.

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

[56] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems.* IEEE, 140–149. https://doi.org/10.1109/PDIS.1994.331722

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

[58] YugabyteDB. Accessed in May, 2023. https://www.yugabyte.com/.

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

[60] 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.

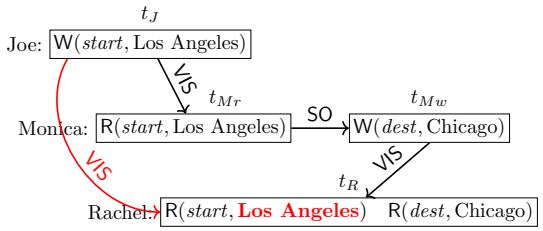


Figure 13: Illustrating the formal definition of TCC [18].

A TCC: FORMAL DEFINITION IN [18]

The framework in [18] uses the *visibility* relation to capture the potential causality between transactions and the *arbitration* relation to capture the commit order of transactions.

Definition 4. An *abstract execution* is a tuple $\mathcal{A} = (T, \text{SO}, \text{WR}, \text{VIS}, \text{AR})$, where $(T, \text{SO}, \text{WR})$ is a history, *visibility* $\text{VIS} \subseteq T \times T$ is a strict partial order such that $(\text{SO} \cup \text{WR}) \subseteq \text{VIS}$, and *arbitration* $\text{AR} \subseteq T \times T$ is a strict total order such that $\text{VIS} \subseteq \text{AR}$.

Note that in this framework, $\text{CO} \triangleq (\text{SO} \cup \text{WR})^+ \subseteq \text{VIS}$. That is, VIS is a general notion of CO.

Definition 5 (TCC [18])**.** A history $\mathcal{H} = (T, \text{SO}, \text{WR})$ satisfies TCC if and only if there is an abstract execution $\mathcal{A} = (T, \text{SO}, \text{WR}, \text{VIS}, \text{AR})$ such that the Int and Ext axioms hold.

Example 8 (TCC)**.** Consider the scenario in Example 2; see Figure 13. Since Monica sees the starting city set by Joe and Rachel sees the destination city set by Monica, we have $t_J \xrightarrow{\text{VIS}} t_{Mr}$ and $t_{Mw} \xrightarrow{\text{VIS}} t_R$, respectively. Since $\text{SO} \subseteq \text{VIS}$ and VIS is transitive, we have $t_J \xrightarrow{\text{VIS}} t_R$. Therefore, Monica should also see the starting city set by Joe.

The following theorem establishes the equivalence between Definitions 3 and 5 of TCC.

Theorem 2 (Equivalence of TCC Definitions)**.** Definitions 3 and 5 of TCC are equivalent.

Proof. Let $\mathcal{H} = (T, \text{SO}, \text{WR})$ be a history. We show that $\mathcal{H}$ satisfies Definition 3 of TCC (denoted TCC_3) if and only if $\mathcal{H}$ satisfies Definition 5 of TCC (denoted TCC_5).

($\Longrightarrow$) Suppose that $\mathcal{H}$ satisfies TCC_3. Then CO is a strict partial order and there exists a strict total order $\text{AO} \supseteq \text{CO}$ such that the Int and Ext axioms hold. Let $\text{VIS} = \text{CO}$ and $\text{AR} = \text{AO}$ in TCC_5. Clearly the Int and Ext axioms also hold for TCC_5. Therefore, $\mathcal{H}$ satisfies TCC_5.

($\Longleftarrow$) Suppose that $\mathcal{H}$ satisfies TCC_5. Then there exists a strict partial order VIS and a strict total order AR such that $\text{CO} \subseteq \text{VIS} \subseteq \text{AR}$ and the Int and Ext axioms hold. Since $\text{CO} \subseteq \text{VIS}$, CO is a strict partial order. Let $\text{AO} = \text{AR}$ in TCC_3. Clearly, the Int axiom holds for TCC_3. In the following, we show that the Ext axiom also holds for TCC_3.

Let r be an external read operation on key x of transaction t. Suppose that r reads from transaction $t' \neq t$. Then t' is the *last* (in AR) transaction among those that precede t in terms of VIS. Since $\text{CO} \subseteq \text{VIS}$ and $\text{AO} = \text{AR}$, t' is also the *last* (in AO) transaction among those that precede t in terms of AO. Therefore, the Ext axiom holds for TCC_3. □

B PROOF OF THEOREM 1

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

($\Longrightarrow$) Assume that $\mathcal{H}$ satisfies TCC. Therefore, $\text{CO} \triangleq (\text{SO} \cup \text{WR})^+$ is a strict partial order and there exists a strict total order $\text{AO} \subseteq T \times T$ such that the Int and Ext axioms hold. We examine each of the anomalous patterns as follows.

(TAP-*a*) ThinAirRead: Forbidden by the Int and Ext axioms.
(TAP-*b*) AbortedRead: Forbidden by the Int and Ext axioms.
(TAP-*c*) FutureRead: Forbidden by the Int and Ext axioms.
(TAP-*d*) NotMyOwnWrite: Forbidden by the Ext axiom.
(TAP-*e*) IntermediateRead: Forbidden by the Ext axiom.
(TAP-*f*) NonRepeatableRead: Forbidden by the Int axiom.
(TAP-*g*) CyclicCO: Since CO is a strict partial order, SO ∪ WR is acyclic.
(TAP-*l*) ConflictAO: Since $t_1 \xrightarrow{\text{WR}(x)} t_3$ and $\text{WR} \subseteq \text{AO}$, $t_1 \xrightarrow{\text{AO}} t_3$. Since $t_2 \xrightarrow{\text{CO}} t_3$ and $\text{CO} \subseteq \text{AO}$, $t_2 \xrightarrow{\text{AO}} t_3$. Since both t_1 and t_2 write to x and $t_1 \xrightarrow{\text{WR}(x)} t_3$, by the Ext axiom, we have $t_2 \xrightarrow{\text{AO}} t_1$. Since $t_1 \xrightarrow{\text{AO}} t_2$, we have $t_1 \xrightarrow{\text{AO}} t_2 \xrightarrow{\text{AO}} t_1$, contradicting the fact that AO is a strict total order.

($\Longleftarrow$) Assume that $\mathcal{H}$ does not contain any of anomalous patterns TAP-a to TAP-g and TAP-l. We prove that $\mathcal{H}$ satisfies TCC by showing that $\text{CO} \triangleq (\text{SO} \cup \text{WR})^+$ is a strict partial order and constructing a strict total order AO such that the Int and Ext axioms hold.

First, since $\mathcal{H}$ does not contain anomalous pattern CyclicCO (TAP-g), CO is a strict partial order.

Then, we define AO as a relation satisfying

- AO is total;
- $\text{CO} \subseteq \text{AO}$;
- For any three transactions t_1, t_2, and t_3, if
  - both t_1 and t_2 writes to the same key, say x,
  - $t_2 \xrightarrow{\text{CO}} t_3$, and
  - t_3 reads x from t_1,

  then $t_2 \xrightarrow{\text{AO}} t_1$.

Since $\mathcal{H}$ does not contain the anomalous pattern ConflictAO (TAP-l), AO is acyclic. Thus, AO is a strict total order.

In the following, we show that the Int and Ext axioms hold. Let $r \triangleq \text{R}(x, v)$ be a read operation of a transaction t. Since $\mathcal{H}$ does not contain anomalous patterns ThinAirRead (TAP-a) or AbortedRead (TAP-b), r have read x from a committed transaction.

- (Int axiom) Suppose that r is an *internal* read of transaction t. Let o_x be the *last* operation on x which is before r in transaction t. (o_x exists because r is an internal read of transaction t.) We distinguish between two cases according to the type of o_x:

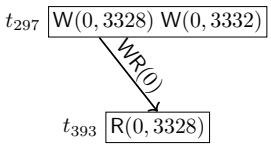


Figure 14: TAP-e found in MariaDB-Galera

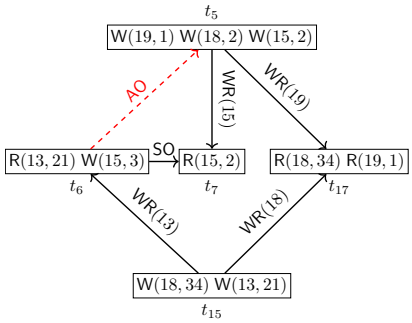


Figure 15: TAP-i found in YugabyteDB

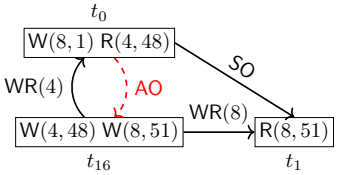


Figure 16: TAP-k found in CockroachDB

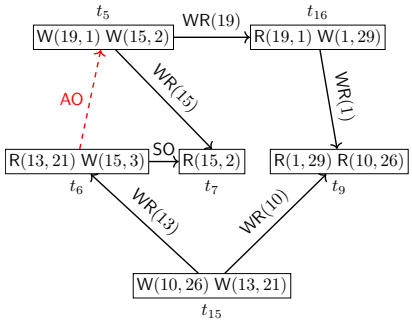


Figure 17: TAP-l found in YugabyteDB

– Suppose that o_x is a read operation on x. Let $o_x \triangleq \mathrm{R}(x, v')$. Since $\mathcal{H}$ does not contain the anomalous pattern NonRepeatableRead (TAP-f), $v' = v$.
– Suppose that o_x is a write operation on x. Let $o_x \triangleq \mathrm{W}(x, v')$. Since $\mathcal{H}$ does not contain the anomalous pattern NotMyOwnWrite (TAP-d), $v' = v$.

- (Ext axiom) Suppose that r is an *external* read of transaction t. Since $\mathcal{H}$ does not contain the anomalous pattern FutureRead (TAP-c), r reads x from a different transaction $t' \neq t$. Since $\mathcal{H}$ does not contain the anomalous pattern IntermediateRead (TAP-e), the write $\mathrm{W}(x, v)$ which writes the value v is the last write to x in transaction t'.
  Let V_x be the set of transactions that precede t in terms of CO and write to x. Since $t' \xrightarrow{\mathrm{WR}(x)} t$, we have $t' \in V_x$. We show that t' is the last (in AO) transaction in V_x. Suppose by contradiction that there is another transaction $t'' \neq t'$ in V_x such that $t' \xrightarrow{\mathrm{AO}} t''$. Since $t'' \in V_x$, $t'' \xrightarrow{\mathrm{CO}} t$. Then there is an anomalous pattern ConflictAO (TAP-l) involving t, t', and t'', contradicting the fact that $\mathcal{H}$ does not contain such an anomalous pattern.

□

C THE FULL C4 ALGORITHM

Algorithm 2 presents the full pseudocode of C4.

Procedure BuildCO. C4 builds CO by examining transactions one by one (line 6). Consider a transaction t. It first builds SO, initializing $Vec[t]$ using its immediate predecessor transaction t' in the same session (lines 10–14). Then C4 examines each operation, denoted by o, of t and builds WR. There are three cases: (1) If o is a read operation but its dictating write has not been encountered (line 17), C4 temporarily adds it to $S_{r'}$; (2) If o is a read operation and its dictating write, denoted t', has been encountered (line 19), C4 updates $Vec[t]$ using $Vec[t']$ and adds an WR edge (t', t) if t is unreachable from t' in the graph (decided on $Vec[t]$ and $Vec[t']$ at line 22); and (3) If o is a write operation (line 25), C4 adds an WR edge from o to each of its dictated read operations in $S_{r'}$. Finally, C4 uses $Vec[t]$ to update vectors of the transactions reachable from t in case new edges outgoing from t have been added (line 32). This ensures the transitivity of CO. Denote the resulting graph by $\mathcal{G}_{\mathrm{CO}}$.

D TEST CASES FOR THE 12 TAPS

Table 6 shows 12 test cases (or anomalous histories) that we craft for the 12 TAPs, respectively.

E OTHER REPRESENTATIVE TCC BUGS FOUND BY C4

Figures 14–17 depict the violations found by C4 in the production database engines we tested.

Algorithm 2 The C4 algorithm for checking history $\mathcal{H} = (T, \text{SO}, \text{WR})$ against TCC

$\mathbb{S}$: the set of sessions in $\mathcal{H}$
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
$R(t)$: the set of read operations of transaction t
$W(t)$: the set of write operations of transaction t
$O(t)$: the set of operations of transaction t
$txn(o)$: the transaction containing operation o
$key(o)$: the key accessed by operation o
ite(c, t, e): if-then-else with condition c, "then" part t, and "else" part e

1: **procedure** CheckTCC()
2: BuildCO()
3: CheckCOTAP() ▷ TAP-a to TAP-h, TAP-j, and TAP-k
4: BuildAO()
5: CheckAOTAP() ▷ TAP-i and TAP-l
6: **procedure** BuildCO()
7: **for** $t \in T$
8: **if** t is aborted
9: $S_a \leftarrow S_a \cup O(t)$ **continue**
10: **suppose** t is the i-th transaction in its session s
11: $Vec[t] \leftarrow \langle \lambda f \in \mathbb{S}.\ \textbf{ite}(f = s, i, 0) \rangle$
12: **if** $i > 0$
13: **let** $t' \leftarrow$ the $(i-1)$-st transaction in session s
14: $Vec[t] \leftarrow Vec[t] \sqcup Vec[t']$
15: $E \leftarrow E \cup \{(t', t)\}$
16: **for** $o \in O(t)$
17: **if** $o \in R(t) \wedge \neg(\exists w \in S_w.\ w \xrightarrow{\text{WR}} o)$
18: $S_{r'} \leftarrow S_{r'} \cup \{o\}$
19: **if** $o \in R(t) \wedge \exists w \in S_w.\ w \xrightarrow{\text{WR}} o$
20: $S_r \leftarrow S_r \cup \{o\}$
21: **let** $t' \leftarrow txn(w)$
22: **if** $t' \neq t \wedge \neg(Vec[t] \sqsupseteq Vec[t'])$
23: $Vec[t] \leftarrow Vec[t] \sqcup Vec[t']$
24: $E \leftarrow E \cup \{(t', t)\}$ $E_{wr} \leftarrow E_{wr} \cup \{(t', t, key(o))\}$
25: **if** $o \in W(t)$
26: $S_w \leftarrow S_w \cup \{o\}$
27: **for** $\forall r \in S_{r'}.\ o \xrightarrow{\text{WR}} r$
28: $S_{r'} \leftarrow S_{r'} \setminus \{r\}$ $S_r \leftarrow S_r \cup \{r\}$
29: **let** $t' \leftarrow txn(r)$
30: **if** $t' \neq t$
31: $E \leftarrow E \cup \{(t, t')\}$ $E_{wr} \leftarrow E_{wr} \cup \{(t, t', key(o))\}$
32: UpdateVec(t)

$Vec[t]$: vector for transaction t
$taps$: the set of anomalous patterns found, initially $\emptyset$
E: the set of SO, WR, and AO edges
E_{wr}: the set of WR(_) edges associated with keys
S_w: the set of write operations encountered, initially $\emptyset$
S_r: the set of read operations whose dictating writes are in S_w, initially $\emptyset$
$S_{r'}$: the set of read operations whose dictating writes are not in S_w, initially $\emptyset$
S_a: the set of operations in aborted transactions encountered, initially $\emptyset$

33: **procedure** BuildAO()
34: **for** $t_1, t_2, t_3 \in T$ such that $\exists x \in \mathsf{K}.\ t_1 \xrightarrow{\text{WR}(x)} t_3 \wedge t_2 \in \text{WT}_x \wedge Vec[t_2] \sqsubseteq Vec[t_3]$
35: $E \leftarrow E \cup \{(t_2, t_1)\}$
36: **for** $t \in T$
37: UpdateVec(t)
38: **procedure** CheckCOTAP()
39: **for** $t \in T$
40: check anomalous patterns TAP-a to TAP-f ▷ details omitted
41: **if** $\exists x \in \mathsf{K}.\ (t \in \text{RT}_x^{\perp_x} \wedge (\exists t' \neq t_\perp \in \text{WT}_x.\ Vec[t'] \sqsubseteq Vec[t]))$
42: $taps \leftarrow taps \cup \{\text{TAP-j}\}$ ▷ InitialRead
43: **if** $\exists t_1, t_2 \in T.\ Vec[t_1] = Vec[t_2]$
44: $taps \leftarrow taps \cup \{\text{TAP-g}\}$ ▷ CyclicCO
45: **if** $\exists x \in \mathsf{K}\ \exists t_1, t_2, t_3 \in T.\ t_1 \xrightarrow{\text{WR}(x)} t_3 \wedge t_2 \in \text{WT}_x \wedge Vec[t_1] \sqsubseteq Vec[t_2] \sqsubseteq Vec[t_3]$
46: **if** $\exists y \neq x \in \mathsf{K}.\ t_2 \xrightarrow{\text{WR}(y)} t_3$
47: $taps \leftarrow taps \cup \{\text{TAP-h}\}$ ▷ FracturedReadCO
48: **else**
49: $taps \leftarrow taps \cup \{\text{TAP-k}\}$ ▷ COConflictAO
50: **procedure** CheckAOTAP()
51: **if** $\exists x \in \mathsf{K}\ \exists t_1, t_2, t_3 \in T.\ t_1 \xrightarrow{\text{WR}(x)} t_3 \wedge t_2 \in \text{WT}_x \wedge Vec[t_1] = Vec[t_2]$
52: **if** $\exists y \neq x \in \mathsf{K}.\ t_2 \xrightarrow{\text{WR}(y)} t_3$
53: $taps \leftarrow taps \cup \{\text{TAP-i}\}$ ▷ FracturedReadAO
54: **else if** $t_2 \xrightarrow{\text{CO}} t_3$
55: $taps \leftarrow taps \cup \{\text{TAP-l}\}$ ▷ ConflictAO
56: **procedure** UpdateVec(t)
57: **for** $t' \in T$ in DFS traversal from t
58: **if** $Vec[t'] \sqsupseteq Vec[t]$
59: **backtrack**
60: $Vec[t'] \leftarrow Vec[t'] \sqcup Vec[t]$

Table 6: Test cases for the 12 TAPs. For example, [w(1,2,3,4) r(2,1,3,4)] refers to a transaction of one write operation and one read operation. The write operation writes value 2 to key 1 by transaction 4 of session 3, while the read operation reads value 1 on key 2 by transaction 4 of session 3.

TAP	Test Case
TAP-a	[r(1,1,0,0) r(2,1,0,0)]
TAP-b	[w(1,1,0,0) w(2,1,0,0)] [r(1,1,0,1)]
TAP-c	[r(1,1,0,0) w(1,1,0,0)]
TAP-d	[w(1,1,0,0)] [w(1,2,1,1) r(1,1,1,1)]
TAP-e	[w(1,1,0,0) w(1,2,0,0)] r(1,1,1,1)]
TAP-f	[w(1,2,0,0)] [w(1,1,1,1) r(1,1,1,1) r(1,2,1,1)]
TAP-g	[r(1,1,0,0) w(2,1,0,0)] [r(2,1,1,1)] [w(1,1,1,2)]
TAP-h	[w(1,2,0,0)] [r(1,2,1,1)] [w(1,1,1,2) w(2,1,1,2)] [r(1,2,2,3) r(2,1,2,3)]]
TAP-i	[w(1,1,0,0) w(2,1,0,0)] [w(1,2,1,1) w(2,2,1,1)] [r(1,1,2,2) r(2,2,2,2)]
TAP-j	[w(1,0,0,0)] [w(1,1,0,1) w(2,1,0,1)] [r(1,0,0,2)]
TAP-k	[w(1,2,0,0)] [r(1,2,1,1)] [w(1,1,1,2) w(2,1,1,2)] [r(2,1,2,3)] [r(1,2,2,4)]
TAP-l	[w(1,2,0,0)] [r(1,2,1,1)] [w(1,1,1,2) w(2,1,1,2)] [r(2,1,2,3)] [r(1,2,2,4)] [r(1,2,3,5)] [r(1,1,3,6)]