Online Timestamp-based Transactional Isolation Checking of Database Systems

Hexu Li[1], Hengfeng Wei[1*], Hongrong Ouyang[2], Yuxing Chen[3], Na Yang[1], Ruohao Zhang[4], Anqun Pan[3]
State Key Laboratory for Novel Software Technology, Nanjing University, China[1];
OceanBase, Ant Group, China[2]; *Tencent Inc., China*[3]; *The Chinese University of Hong Kong, China*[4]
hexuli@smail.nju.edu.cn, hfwei@nju.edu.cn, ouyanghongrong.oyh@oceanbase.com, axingguchen@tencent.com,
nayang@smail.nju.edu.cn, ruohaozhang@link.cuhk.edu.hk, aaronpan@tencent.com

***Abstract*—Serializability (SER) and snapshot isolation (SI) are widely used transactional isolation levels in database systems. The isolation checking problem asks whether a given execution history of a database system satisfies a specified isolation level. However, existing SER and SI checkers, whether traditional black-box checkers or recent timestamp-based white-box ones, operate offline and require the entire history to be available to construct a dependency graph, making them unsuitable for continuous and ever-growing histories.**

This paper addresses online isolation checking by extending the timestamp-based isolation checking approach to online settings. Specifically, we design CHRONOS, an efficient timestamp-based offline SI checker. CHRONOS is incremental and avoids constructing a start-ordered serialization graph for the entire history, making it well-suited for online scenarios. We further extend CHRONOS into an online SI checker, AION, addressing several key challenges unique to online settings. Additionally, we develop AION-SER for online SER checking. Experiments highlight that CHRONOS processes offline histories with up to one million transactions in seconds, greatly outperforming existing SI checkers. Furthermore, AION and AION-SER sustain a throughput of approximately 12K transactions per second, demonstrating their practicality for online isolation checking.

***Index Terms*—Transactional isolation levels, Serializability, Snapshot isolation, Online isolation checking**

I. INTRODUCTION

Database transactions provide an "all-or-none" abstraction and isolate concurrent computations on shared data, greatly simplifying client programming. Serializability (SER) [1], [2], the gold standard for transactional isolation, ensures that all transactions appear to execute sequentially. Prominent database systems offering SER include PostgreSQL [3], CockroachDB [4], and YugabyteDB [5]. For improved performance [6], many databases implement snapshot isolation (SI) [7], a widely adopted weaker isolation level. Examples include YugabyteDB, and Dgraph [8], WiredTiger [9], Google's Percolator [10], MongoDB [11], and TiDB [12].

Implementing databases correctly, however, is notoriously challenging, and many fail to deliver SER or SI as claimed [13], [14], [15], [16], [17], [18]. The isolation checking problem of determining whether a given execution history satisfies a specified isolation level is fundamental to database testing. *Black-box* checkers for SER and SI, such as dbcop [17], Cobra [16], Elle [19], PolySI [20], and Viper [21], have been developed. However, due to the NP-hardness of SER and SI checking in black-box settings [1], [17], [22], these checkers often exhibit high computational complexity and struggle to scale to large histories containing tens of thousands of transactions. Recent advances propose timestamp-based *white-box* checkers [23], [24], [25], which leverage additional transaction start and commit timestamps to infer execution order and read views. Particularly, Emme-SER and Emme-SI [25] enable polynomial-time verification for SER and SI, respectively, alleviating scalability challenges in large histories.

Problem: To our knowledge, existing approaches are all *offline* checkers, requiring the entire history to be available before checking. However, in practice, it is often desirable to perform isolation checking online, i.e., on continuous and ever-growing history. While Cobra [16] supports online SER checking, it does not support online SI checking and requires injecting "fence transactions" into client workloads – an approach often unacceptable in production.

Our Work: In this study, we tackle the problem of *online* transactional isolation checking for database systems by extending the timestamp-based isolation checking approach to online settings. On the one hand, online checkers must be highly efficient to keep pace with the database throughput, which generates a continuous and ever-growing history. However, Emme-SI performs expensive graph construction and cycle detection on the start-ordered serialization graph of the entire history [26], rendering it unsuitable for online use. On the other hand, online checkers *cannot* assume that transactions are collected in ascending order of their start or commit timestamps, as asynchrony in database systems makes this infeasible. This introduces three key challenges unique to online checkers, which we address in our work.

- First, the satisfaction or violation of isolation level rules for a transaction becomes *unstable* due to asynchrony, making it impossible to assert correctness immediately upon transaction collection.
- Second, an incoming transaction T with a smaller start timestamp may require *re-checking* transactions that commit after T started. Efficient data structures are necessary to facilitate this re-checking process.
- Third, to ensure long-term feasibility and prevent unbounded memory usage, online checkers must *recycle*

Hengfeng Wei is the corresponding author.

2375-026X/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDE65448.2025.00279

data structures and transactions no longer needed. However, in the worst-case scenario, asynchrony may prevent the safe recycling of any data structures or transactions.

Contributions: Our approach supports both online SER and SI checking and is general and adaptable to various data types used in generating histories. In this work, we focus on online SI checking, particularly for key-value histories and list histories.[1] Specifically, our contributions are as follows:

- (Section III-B) We design CHRONOS, a highly efficient timestamp-based offline SI checker with a time complexity of $O(N\log N + M)$, where N and M represent the number of transactions and operations in the history, respectively. Unlike Emme-SI, CHRONOS is incremental and avoids constructing the start-ordered serialization graph for the entire history, making it well-suited for online checking.
- (Section III-C) We identify the challenges of online SI checking caused by inconsistencies between the execution and checking orders of transactions. To address these challenges, we extend CHRONOS to support online checking, introducing a new checker named AION.
- (Sections V and VI) We evaluate CHRONOS and AION under a diverse range of workloads. Results show that CHRONOS can process offline histories with up to one million transactions in tens of seconds, significantly outperforming existing SI checkers, including Emme-SI, which either take hours or fail to handle such large histories. Moreover, AION supports online transactional workloads with a sustained throughput of approximately 12K transactions per second (TPS), with only a minor (approximately 5%) impact on database throughput, primarily due to history collection.
- (Section VI) We also develop AION-SER, an online SER checker. Experimental results demonstrate that AION-SER supports online transactional workloads with a sustained throughput of approximately 12K TPS, greatly outperforming Cobra.

II. SEMANTICS OF SNAPSHOT ISOLATION

In this section, we review both the operational and axiomatic semantics of SI. In Section III, we design our timestamp-based SI checking algorithms by relating these two semantics.

We consider a key-value store managing a set K of keys associated with values from V.[2] We use $\mathsf{R}(k, v)$ to denote a read operation that reads $v \in \mathsf{V}$ from $k \in \mathsf{K}$ and $\mathsf{W}(k, v)$ to denote a write operation that writes $v \in \mathsf{V}$ to $k \in \mathsf{K}$.

A. SI: Operational Semantics

SI was first defined in [7]. The original definition is operational: it is describing how an implementation would work. We specify SI by showing a high-level implementation of it in Algorithm 1 [26], [27]. Each procedure in Algorithm 1 is executed atomically. We assume a time oracle $\mathcal{O}$ which returns a unique timestamp upon request and that the timestamps are totally ordered. In the paper, we reference pseudocode lines using the format "algorithm#:line#".

[1]While theoretically similar to online SER checking, online SI checking poses greater engineering challenges.

[2]We assume an artificial value $\perp_v \notin \mathsf{V}$.

Algorithm 1 Operational Semantics of Snapshot Isolation

log: the log of committed transactions

1: **procedure** START(T)
2: $T.start\_ts \leftarrow$ REQUEST($\mathcal{O}$)
3: **return** ok
4: **procedure** WRITE(T, k, v)
5: $T.buffer \leftarrow T.buffer \circ \langle k, v\rangle$
6: **return** ok
7: **procedure** READ(T, k)
8: **return** value of k from $T.buffer$ and log as of $T.start\_ts$
9: **procedure** COMMIT(T)
10: $T.commit\_ts \leftarrow$ REQUEST($\mathcal{O}$)
11: **if** T *conflicts* with some concurrent transaction
12: **return** aborted
13: log $\leftarrow$ log $\circ \langle T.buffer, T.commit\_ts\rangle$
14: **return** committed

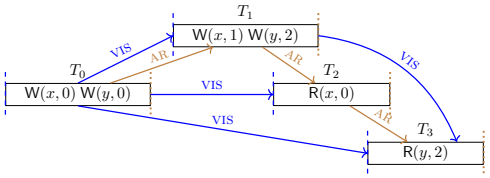


Fig. 1: Illustration of both the operational and axiomatic semantics of SI. The start and commit timestamps are represented by dashed lines and dotted lines, respectively. All timestamps are totally ordered from left to right. (The arrows for AR implied by VIS and transitivity are not shown.)

When a transaction T starts, it is assigned a start timestamp $T.start\_ts$ (line 1:2). The writes of T are buffered in $T.buffer$ (line 1:5). Under SI, the transaction T reads data from its own write buffer and the snapshot (stored in log) of *committed* transactions valid as of its start time $T.start\_ts$ (line 1:8). When T is ready to commit, it is assigned a commit timestamp $T.commit\_ts$ (line 1:10) and allowed to commit if no other *concurrent* transaction T' (i.e., $[T'.start\_ts, T'.commit\_ts]$ and $[T.start\_ts, T.commit\_ts]$ overlap) has already written data that T intends to write (line 1:11). This rule prevents the "lost updates" anomaly. If T commits, it appends its buffered writes, along with its commit timestamp, to log (line 1:13).

We require that for each transaction T, $T.start\_ts \leq T.commit\_ts$.

$$T.start\_ts \leq T.commit\_ts. \tag{1}$$

Note that we allow $T.commit\_ts = T.start\_ts$, which is possible for read-only transactions.

Example 1. *Figure 1 shows a valid execution history of SI according to Algorithm 1 (ignore the edges for now). In this*

history, the snapshot taken by T_2 does not contain the effect of T_1, since $T_1.commit\_ts > T_2.start\_ts$. In contrast, the snapshot taken by T_3 contains the effect of T_1, thereby reading 2 from y written by T_1.

B. SI: Axiomatic Semantics

To define the axiomatic semantics of SI, we first introduce the formal definitions of transactions, histories, and abstract executions [28]. In this paper, we use $(a, b) \in R$ and $a \xrightarrow{R} b$ interchangeably, where $R \subseteq A \times A$ is a relation over the set A. We write $\text{DOM}(R)$ for the domain of R. Given two relations R and S, their composition is defined as $R\ ;\ S = \{(a, c) \mid \exists b \in A : a \xrightarrow{R} b \xrightarrow{S} c\}$. A strict partial order is an irreflexive and transitive relation. A strict total order is a relation that is a strict partial order and total.

Definition 1. *A transaction is a pair (O, po), where O is a set of operations and po is the program order over O.*

Clients interact with the store by issuing transactions during *sessions*. We use a *history* to record the client-visible results of such interactions.

Definition 2. *A history is a pair $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$, where $\mathcal{T}$ is a set of transactions and SO is the session order over $\mathcal{T}$.*

We assume that every history contains a special transaction $\bot_T$ that writes the initial values of all keys [29], [17], [30]. This transaction precedes all the other transactions in SO.

To declaratively justify each transaction of a history, it is necessary to establish two key relations: visibility and arbitration. The visibility relation defines the set of transactions whose effects are visible to each transaction, while the arbitration relation determines the execution order of transactions.

Definition 3. *An abstract execution is a tuple $\mathcal{A} = (\mathcal{T}, \text{SO}, \text{VIS}, \text{AR})$, where $(\mathcal{T}, \text{SO})$ is a history, visibility $\text{VIS} \subseteq \mathcal{T} \times \mathcal{T}$ is a strict partial order, and arbitration $\text{AR} \subseteq \mathcal{T} \times \mathcal{T}$ is a strict total order such that $\text{VIS} \subseteq \text{AR}$.*

Definition 4 (Snapshot Isolation (Axiomatic) [28], [29])**.** *A history $\mathcal{H} = (T, \mathsf{SO})$ satisfies SI if and only if there exists an abstract execution $\mathcal{A} = (T, \mathsf{SO}, \text{VIS}, \text{AR})$ such that the* SESSION*,* INT*,* EXT*,* PREFIX*, and* NOCONFLICT *axioms (explained below) hold.*

Let $r \triangleq \mathsf{R}(k, \_)$ be a read operation of transaction T. If r is the first operation on k 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 a key returns the same value as the last write to or read from this key in the transaction. The *external consistency axiom* EXT ensures that an external read in a transaction T from a key returns the final value written by the last transaction in AR among all the transactions that precede T in terms of VIS and write to this key.

The SESSION axiom (i.e., $\mathsf{SO} \subseteq \text{VIS}$), requires previous transactions be visible to transactions later in the same session.[3] The PREFIX axiom (i.e., AR ; VIS $\subseteq$ VIS) ensures that if the snapshot taken by a transaction T includes a transaction S, than this snapshot also include all transactions that committed before S in terms of AR. The NOCONFLICT axiom prevents concurrent transactions from writing on the same key. That is, for any conflicting transactions S and T, one of $S \xrightarrow{\text{VIS}} T$ and $T \xrightarrow{\text{VIS}} S$ must hold.

Example 2. *Figure 1 shows that the history is valid under the axiomatic semantics of SI. It constructs a possible abstract execution by establishing the visibility and arbitration relations as shown in the figure. For example, T_0 is visible to T_2 but T_1 is not. Both T_0 and T_1 are visible to T_3. Since T_1 is after T_0 in terms of* AR*, T_3 reads the value 2 of y written by T_1.*

III. TIMESTAMP-BASED SI CHECKING ALGORITHMS

We first describe the insights behind our timestamp-based SI checking algorithms and then propose both the offline and online checkers.

A. Insights

Consider a database history $\mathcal{H} = (\mathcal{T}, \text{SO})$ generated by Algorithm 1. We need to check whether $\mathcal{H}$ satisfies SI by constructing an abstract execution $\mathcal{A} = (\mathcal{T}, \text{SO}, \text{VIS}, \text{AR})$ with appropriate relations VIS and AR on $\mathcal{T}$. The key insight of our approach is to infer the actual execution order of transactions to be consistent with their commit timestamps. Formally, we define AR as follows.

Definition 5 (Timestamp-based Arbitration)**.**

$$\forall T_1, T_2 \in \mathcal{T}.\ T_1 \xrightarrow{\text{AR}} T_2 \iff T_1.commit\_ts < T_2.commit\_ts.$$

On the other hand, from start timestamps and the inferred execution order, we can determine the snapshot of each transaction: each transaction observes the effects of all transactions that have committed before it starts. Formally, we define VIS as follows.

Definition 6 (Timestamp-based Visibility)**.**

$$\forall T_1, T_2 \in \mathcal{T}.\ T_1 \xrightarrow{\text{VIS}} T_2 \iff T_1.commit\_ts \leq T_2.start\_ts.$$

Example 3. *The* VIS *and* AR *relations of Figure 1 are constructed according to Definitions 6 and 5, respectively. For example, four transactions in the history are totally ordered (in* AR*) by their commit timestamps. On the other hand, since $T_1.commit\_ts < T_3.start\_ts$, we have $T_1 \xrightarrow{\text{VIS}} T_3$.*

Given the VIS and AR relations, it is straightforward to show that the PREFIX axiom holds. Therefore, by Definition 4, it remains to check that the SESSION, INT, EXT, and NOCONFLICT axioms hold. Since transactions are totally ordered, our checking algorithm can *simulate* transaction execution one by one, following their commit timestamps, and check the axioms on the fly. This simulation approach enables highly efficient checking algorithms that eliminate the need to explore

[3]That is, we consider the *strong session* variant of SI [31], [29] commonly used in practice.

Algorithm 2 CHRONOS: the offline SI checking algorithm

```
    last_sno ∈ SID → ℕ: sno of the last transaction already processed
      in each session, initially −1
    last_cts ∈ SID → TS: commit_ts of the last transaction already
      processed in each session, initially ⊥_ts
    frontier ∈ K → V: the last committed value of each key, initially ⊥_v
    ongoing ∈ K → 2^TID: the set of ongoing transactions on each key,
      initially ∅
    int_val ∈ TID → (K → V): the last read/written value of each key
      in each transaction, initially ⊥_v
    ext_val ∈ TID → (K → V): the last written value of each key
      in each transaction, initially ⊥_v
1:  procedure CHECKSI(H)        ▷ H contains the initial transaction ⊥_T
2:      sort TS in ascending order
3:      for ts ∈ TS
4:          T ← the transaction that owns the timestamp ts
5:          sid ← T.sid      tid ← T.tid      T.wkey ← ∅
6:          if ts = T.start_ts                              ▷ start event of T
7:              if T.sno ≠ last_sno[sid] + 1 ∨ T.start_ts < last_cts[sid]
8:                  report a violation of SESSION
9:              last_sno[sid] ← T.sno
10:             last_cts[sid] ← T.commit_ts
11:             for (op = _(k, v)) ∈ T.ops                  ▷ in program order
12:                 if op = R(k, v)
13:                     if int_val[tid][k] = ⊥_v            ▷ external read
14:                         if v ≠ frontier[k]
15:                             report a violation of EXT
16:                     else if int_val[tid][k] ≠ v         ▷ internal read
17:                         report a violation of INT
18:                 else                                    ▷ op = W(k, v)
19:                     T.wkey ← T.wkey ∪ {k}
20:                     ext_val[tid][k] ← v
21:                     ongoing[k] ← ongoing[k] ∪ {tid}
22:                 int_val[tid][k] ← v
23:         else                                            ▷ commit event of T
24:             if T.start_ts > T.commit_ts                 ▷ check Eq. (1)
25:                 report an error
26:             for k ∈ T.wkey
27:                 ongoing[k] ← ongoing[k] \ {tid}         ▷ except T itself
28:                 if ongoing[k] ≠ ∅
29:                     report a violation of NOCONFLICT
30:                 frontier[k] ← ext_val[tid][k]
31:             int_val[tid] ← ∅                            ▷ gc int_val
32:             ext_val[tid] ← ∅                            ▷ gc ext_val
33:             T ← T \ {T}                                 ▷ gc T
```

all potential transaction execution orders within a history, as required by PolySI [20] and Viper [21], or to conduct expensive cycle detection on extensive dependency graphs of histories, as required by Elle [19] and Emme [25]. Moreover, the incremental nature of the simulation renders our algorithm suitable for online checking.

B. The CHRONOS Offline Checking Algorithm

1) Input: CHRONOS takes a history $\mathcal{H}$ as input and checks whether it satisfies SI. Each transaction T in the history is associated with the following data:

- $T.tid$: the unique ID of T. We use TID to denote the set of all transaction IDs in $\mathcal{H}$;
- $T.sid$: the unique ID of the session to which T belongs. We use SID to denote the set of all session IDs in $\mathcal{H}$;
- $T.sno$: the sequence number of T in its session;
- $T.ops$: the list of operations in T; and
- $T.start\_ts$, $T.commit\_ts$: the start and commit timestamp of T, respectively. We use TS to denote the set of all timestamps in $\mathcal{H}$ and denote the minimum timestamp in TS by $\bot_{ts}$.

We use $T.wkey$ to record the set of keys written by T. We design CHRONOS with key-value histories in mind, but it is also easily adaptable to support other data types such as lists.

2) Algorithm: CHRONOS simulates the execution of a database assuming that the start and commit events of transactions in $\mathcal{H}$ are executed in the order of their timestamps, while checking the corresponding axioms on the fly. To do this, CHRONOS sorts all the timestamps in TS in ascending order (line 2:2). By Definition 5, we obtain the total order AR between transactions. CHRONOS will traverse TS in this order and process each of the start events and commit events of transactions one by one (line 2:3). During this process, CHRONOS maintains two maps: frontier maps each key to the last (in AR) committed value, and ongoing maps each key to the set of ongoing transactions that write this key.

Let ts be the current timestamp being processed and T the transaction that owns the timestamp ts. If ts is the start timestamp of T, CHRONOS checks whether T violates any of the SESSION, INT, and EXT axioms (line 2:6). Otherwise, it checks whether T violates the NOCONFLICT axiom (line 2:23). CHRONOS will identify *all* violations of these axioms in a history, instead of terminating immediately upon encountering the first one.

SESSION *(lines 2:7 – 2:10):* CHRONOS uses last_sno and last_cts to maintain the sno and $commit\_ts$ of the last transaction already processed in each session, respectively. It checks whether the current transaction T follows immediately after the last transaction already processed in its session and starts after this last transaction commits. If this fails, a violation of SESSION is found.

INT *and* EXT *(lines 2:12 – 2:17):* Let $tid \triangleq T.tid$. CHRONOS uses int_val$[tid][k]$ to track the last value read or written by T on key k (see line 2:22). Let $op = \mathsf{R}(k, v)$ be the read operation of T being checked (line 2:12). If int_val$[tid][k]$ is not the initial artificial value $\bot_v$, then op is an internal read. CHRONOS then checks if int_val$[tid][k] = v$. If this fails, a violation of INT is found (line 2:17).

If int_val$[tid][k] = \bot_v$, then op is an external read (line 2:13). By EXT, op should observe the last committed value of k, i.e., frontier$[k]$. Otherwise, a violation of EXT is found (line 2:15).

For a key k, CHRONOS updates frontier$[k]$ whenever a transaction that writes to k commits. Since a transaction T may write to the same key k multiple times, CHRONOS uses ext_val$[tid][k]$ to track the last value written by T on k (see line 2:20). Therefore, when T commits, CHRONOS updates frontier$[k]$ to ext_val$[tid][k]$ (line 2:30).

NOCONFLICT *(lines 2:27 – 2:29):* Now suppose that ts is the commit timestamp of T (line 2:23). For each key $k \in T.wkey$ written by T (tracked at line 2:19), CHRONOS checks

whether T conflicts with any ongoing transactions on key k by testing the emptiness of ongoing$[k]$ (tracked at line 2:21). If ongoing$[k]$ (except T itself) is non-empty, a violation of NoConflict is found (line 2:29).

GarbageCollect *(lines 2:30 – 2:33):* To save memory, Chronos periodically discard obsolete information from int_val and ext_val and remove old transactions from $\mathcal{T}$. First, it is safe to discard int_val$[tid]$ once the commit event of transaction T has been processed (line 2:31), since the information in int_val will never be used by any other transaction. Second, for a transaction T, all of its writes have been recorded in frontier, ext_val$[tid]$ becomes redundant and can be discarded (line 2:32). Furthermore, $T.sno$ and $T.cts$ are recorded in last_sno and last_cts, respectively, and thus T is no longer needed in $\mathcal{T}$ (line 2:33).

Example 4. *Consider Figure 2's history which consists of five transactions, namely $T_1, T_2, \ldots,$ and T_5.* Chronos *processes the start and commit events of the transactions in the ascending order of their timestamps, namely ❶, ❷, …, and ❿.*

On the start event of T_3 with timestamp ❻, the external read operation $\mathsf{R}(x, 2)$ can be justified by the last committed transaction T_2 that writes 2 to x. This is captured by the current frontier$[x] = \{T_2\}$. Moreover, since T_3 updates key y, Chronos *adds T_3 to ongoing$[y]$, yielding ongoing$[y] = \{T_3\}$. Now comes the commit event of T_5 with timestamp ❼.* Chronos *finds that T_5 conflicts with T_3 on y by checking the emptiness of ongoing$[y] \setminus \{T_5\}$, reporting a violation of* NoConflict. *In addition, since T_5 updates key y,* Chronos *updates frontier$[y]$ to $\{T_5\}$. This justifies the external read operation $\mathsf{R}(y, 1)$ of T_4 when* Chronos *examines the start event of T_4 with timestamp ❽. Note that upon the commit event of T_3 with timestamp ❾,* Chronos *does* not *report a violation of* NoConflict *for T_3 with T_5 on y because this violation has already been reported on the commit event of T_5 and now $T_5 \notin$ ongoing$[y]$ at line 2:27.*

3) Complexity Analysis: Suppose that the history $\mathcal{H}$ consists of N transactions and M operations ($N \leq M$). We assume that the data structures used by Chronos are implemented as hash maps (or hash sets) which offer average constant time performance for the basic operations like put, get, remove, contains, and isEmpty.

The time complexity of Chronos comprises

- $O(N\log N)$ for sorting on TS (line 2:2);
- $O(N) = N \cdot O(1)$ for checking Session on the start events of transactions (lines 2:7 – 2:10);
- $O(M) = M \cdot O(1)$ for checking Ext and Int on each read operation of transactions (lines 2:12 – 2:17) and for updating data structures on each write operation of transactions (lines 2:18 – 2:21);
- $O(N) = N \cdot O(1)$ for checking Eq. (1) on the commit events of transactions (lines 2:24 – 2:25);
- $O(M) = O(M) \cdot O(1)$ for checking NoConflict on the commit events of transactions (lines 2:27 – 2:29).

Therefore, the time complexity is $O(N\log N + M)$.

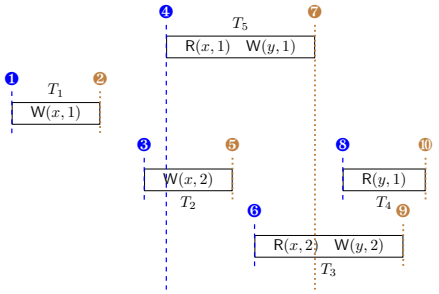


Fig. 2: Illustration of both the offline and online checking algorithms, Chronos and Aion in Example 4 and Example 5.

The space complexity of Chronos is primarily determined by the memory required for storing the history (the memory usage of the frontier and ongoing data structures is relatively small). Thanks to its prompt garbage collection mechanism, the memory usage for storing the history diminishes as transactions are processed.

C. The Aion Online Checking Algorithm

1) Input: In the online settings, the history $\mathcal{H}$ is not known in advance. Instead, the online checking algorithm Aion receives transactions one by one and checks SI incrementally. It is assumed that the session order is preserved when transactions are received.

2) Challenges: Due to asynchrony, Aion *cannot* anticipate that transactions will be collected in ascending order based on their start/commit timestamps. This introduces three main challenges for Aion.

- The determination of satisfaction or violation of the Ext axiom for a transaction becomes *unstable* due to asynchrony. This means that we cannot assert it immediately upon the transaction being collected. In contrast, the Int axiom remains unaffected by asynchrony, while NoConflict violations will eventually be correctly reported as soon as the conflicting transactions are collected.
- An incoming transaction T with a smaller start timestamp may initiate the *re-checking* of transactions that commit after T starts. To facilitate efficient rechecking, it is essential to extend and maintain the data structures, namely frontier and ongoing.
- To prevent unlimited memory usage and facilitate long-running checking, Aion must recycle data structures and transactions that are no longer necessary. Unfortunately, in the worst-case scenario, Aion cannot safely recycle any data structures and transactions due to asynchrony.

The following example illustrates the challenges faced by Aion and our solutions to overcome them.

Example 5. *Consider the history of Figure 2. Suppose that the transactions are collected in the order T_1, T_2, …, and T_5.*

When Aion *collects the first four transactions, it cannot definitively assert that T_4 violates the* Ext *axiom permanently. This uncertainty arises because T_5, from which T_4 reads the value of y, may experience delays due to asynchrony. To tackle*

this instability issue, AION *employs a waiting period during which delayed transactions are expected to be collected. This approach allows for a more accurate determination of whether a violation of* EXT *persists or is a transient result.*

Now, when transaction T_5 arrives, AION *must check the* SESSION, INT, *and* EXT *axioms for T_5 and* re-check *all transactions that commit after T_5 starts (denoted ❹). The re-checking process involves two cases based on the axioms to be verified: (1) re-check the* NOCONFLICT *axiom for transactions overlapping with T_5, i.e., T_2 and T_3 in this example; and (2) re-check the* EXT *axiom for transactions starting after T_5 commits, i.e., T_4 in this example. To facilitate this, the data structures* frontier *and* ongoing *should be* versioned *by timestamps and support timestamp-based search, returning the latest version* before *a given timestamp.*

For instance, to determine the set of transactions overlapping with T_5 (for checking the NOCONFLICT *axiom),* AION *queries the versioned data structure* ongoing *using the timestamp $T_5.commit\_ts$ ❼ (and key y). This query returns the set of ongoing transactions updated at timestamp $T_3.start\_ts$ ❻, namely $\{T_3.tid\}$. Consequently,* AION *identifies a conflict between T_5 and T_3 and reports a violation of* NOCONFLICT.

Similarly, to justify the read operation $\mathsf{R}(y, 1)$ of T_4 (for re-checking the EXT *axiom),* AION *queries the versioned data structure* frontier *using the timestamp $T_4.start\_ts$ ❽ (and key y). This query returns the frontier updated at timestamp $T_5.commit\_ts$ ❼, namely $[y \mapsto 1]$. Thus, T_4 is re-justified by T_5, clearing the false alarm on the violation of* EXT *axiom for T_5. Be cautious that this result is also temporary and subject to change due to asynchrony.*

For violations of the INT, EXT, *and* NOCONFLICT *axiom,* AION *reports the violation and continues checking as if the violation did not occur.*

3) Algorithm: Algorithm 3 provides the pseudocode for AION. AION extends the data structures frontier and ongoing used in CHRONOS to frontier_ts and ongoing_ts, respectively, which are versioned by timestamps. When given a timestamp ts, we use frontier_ts$[\widehat{ts}]$ and ongoing_ts$[\widehat{ts}]$ to denote the latest version of frontier_ts and ongoing_ts *before* ts, respectively.

Upon receiving a new transaction T, AION proceeds in three steps: it first checks the SESSION, INT, and EXT axioms for T, similarly to CHRONOS (lines 3:6 – 3:25), then it re-checks the NOCONFLICT axiom for transactions overlapping with T (lines 3:26 – 3:35), and finally, it re-checks the EXT axiom for transactions starting after T commits (lines 3:37 – 3:57).

Step ① (lines 3:6 – 3:25): The checking for T is similar to that in CHRONOS, with two differences. On one hand, to check the EXT axiom, AION needs to obtain the latest version of frontier_ts before $T.start\_ts$, denoted frontier_ts$[\widehat{T.start_ts}]$ (line 3:14). Moreover, to update frontier_ts at timestamp $cts \leftarrow T.commit\_ts$, it first fetches the latest version of frontier_ts before cts (line 3:23) and then updates the components for each key written by T (line 3:24). On the other hand, if T violates the EXT axiom, AION does not report the violation immediately as it is subject to change due to asynchrony. Instead, it records the temporary violation in variable T.EXT (line 3:15) and waits for a timeout (line 3:59) set at the beginning (line 3:3).

Step ② (lines 3:26 – 3:35): To re-check the NOCONFLICT axiom for transactions overlapping with T, AION iterates through the timestamps ts ranging from $T.start\_ts$ to $T.commit\_ts$ (both *inclusive*) in ascending order (line 3:26). If ts corresponds to the start timestamp of a transaction T' (which could be T), AION updates ongoing_ts$[ts]$ for each key written by T' (line 3:30) to include $T'.tid$. If ts corresponds to the commit timestamp of transaction T', AION reports that T' conflicts with the set of ongoing transactions in ongoing_ts$[ts][k]$ for each key k written by T' (line 3:34). Note that we exclude $T'.tid$ from ongoing_ts$[ts]$ at line 3:33 to prevent reporting self-conflicts (i.e., T' conflicting with itself). This also prevents reporting duplicate conflicts: in the case of conflicting transactions T_1 and T_2, AION reports the conflict only once, considering the transaction with the smaller commit timestamp.

Step ③ (lines 3:37 – 3:57): To re-check the EXT axiom for transactions starting after T commits, AION iterates through timestamps ts greater than $T.commit\_ts$ in ascending order (line 3:37). If ts corresponds to the start timestamp of a transaction T' *and* the timeout for T' has not expired, AION re-checks the EXT axiom for each external read of T' based on ext_val$[T]$ (line 3:43), instead of using the latest version of frontier_ts before ts as in *Step* ①. This optimization is effective because only the recently incoming transaction T impacts the justification of external reads of T'. For further optimization, AION re-checks only the external reads of T' on keys that are written by T (line 3:36 and line 3:43) *and* have not been overwritten by later transactions after T (line 3:53). Essentially, the values of these keys are still influenced by T. Therefore, if ts corresponds to the commit timestamp of transaction T', it suffices for AION to update frontier_ts$[ts]$ only for these keys (line 3:57). The third optimization dictates that once all keys written by T are overwritten by later transactions, the re-checking process for EXT terminates (line 3:55).

The determination of violation or satisfaction of the EXT axiom at this stage is temporary and recorded in T'.EXT (line 3:45 and line 3:47). When the timeout for T' expires, AION reports an EXT violation if $T'.\text{EXT} = \bot$ (line 3:61).

GARBAGECOLLECT *(lines 3:63 – 3:66):* However, in the worst-case scenario, AION cannot safely recycle any data structures or transactions due to asynchrony. To mitigate memory usage, AION performs garbage collection periodically and conservatively: each time it transfers frontier_ts, ongoing_ts, and transactions below a specified timestamp from memory to disk (lines 3:63 – 3:66, marked as ⬇). AION reloads these data structures and transactions as needed later on (refer to code lines marked as ⬆).

4) Complexity and Correctness Analysis: AION behaves similarly to CHRONOS in terms of complexity for (re-)checking the SESSION, INT, EXT, and NOCONFLICT axioms when a new transaction is received. However, instead of

Algorithm 3 AION: the online SI checking algorithm

last_sno, last_cts, int_val, and ext_val are the same as in CHRONOS

1: **procedure** ONLINECHECKSI(T) ▷ upon receiving a new transaction T
2: ⏱ T.EXT $\leftarrow \top$ ▷ T.EXT: whether T satisfies EXT, initially true
3: ⏱ set a timer for re-checking EXT for T
4: **if** $T.start\_ts > T.commit\_ts$ ▷ check Eq. (1)
5: *report an error*
6: $sid \leftarrow T.sid$ $tid \leftarrow T.tid$ $T.wkey \leftarrow \emptyset$

▷ ① *check* SESSION, INT, *and* EXT *for* T, *similarly to* CHRONOS
7: **if** $T.sno \neq$ last_sno$[sid] + 1 \vee T.start_ts <$ last_cts$[sid]$
8: *report a violation of* SESSION
9: last_sno$[sid] \leftarrow T.sno$
10: last_cts$[sid] \leftarrow T.commit_ts$
11: **for** $(op = \_(k, v)) \in T.ops$
12: **if** $op = \mathsf{R}(k, v)$
13: **if** int_val$[tid][k] = \bot_v$ ▷ external read
14: **if** $v \neq$ frontier_ts$[\widehat{T.start_ts}][k]$
15: ⏱ T.EXT $\leftarrow \bot$
16: **else if** int_val$[tid][k] \neq v$ ▷ internal read
17: *report a violation of* INT
18: **else** ▷ $op = \mathsf{W}(k, v)$
19: $T.wkey \leftarrow T.wkey \cup \{k\}$
20: ext_val$[tid][k] \leftarrow v$
21: int_val$[tid][k] \leftarrow v$
22: $cts \leftarrow T.commit\_ts$
23: frontier_ts$[cts] \leftarrow$ frontier_ts$[\widehat{cts}]$
24: frontier_ts$[cts][k] \leftarrow$ ext_val$[tid][k]$ for $k \in T.wkey$
25: int_val$[tid][k] \leftarrow \bot_v$ for $k \in$ DOM(int_val$[tid]$) ▷ reset

▷ ② *re-check* NOCONFLICT *for transactions overlapping with* T
26: **for** $ts \in$ TS such that $T.start\_ts \leq ts \leq T.commit\_ts$
27: $T' \leftarrow$ the transaction that owns the timestamp ts
28: **if** $ts = T'.start\_ts$
29: **for** $k \in T'.wkey$
30: ongoing_ts$[ts][k] \leftarrow$ ongoing_ts$[\widehat{ts}][k] \cup \{T'.tid\}$
31: **else** ▷ $ts = T'.commit\_ts$
32: **for** $k \in T'.wkey$
33: ongoing_ts$[ts][k] \leftarrow$ ongoing_ts$[\widehat{ts}][k] \setminus \{T'.tid\}$
34: **if** ongoing_ts$[ts][k] \neq \emptyset$
35: *report a violation of* NOCONFLICT

frontier_ts $\in$ TS $\rightarrow$ (K $\rightarrow$ V) : frontier versioned by timestamps
ongoing_ts $\in$ TS $\rightarrow$ (K $\rightarrow 2^{\text{TID}}$) : ongoing versioned by timestamps

▷ ③ *re-check* EXT *for transactions starting after* T *commits*
36: $keys \leftarrow T.wkey$
37: **for** $ts \in$ TS such that $ts > T.commit\_ts$
38: $T' \leftarrow$ the transaction that owns the timestamp ts
39: **if** $ts = T'.start\_ts$
40: **if** TIMEOUT(T') has been called
41: ⏱ **continue**
42: **for** $(op = \_(k, v)) \in T'.ops$
43: **if** $op = \mathsf{R}(k, v) \wedge$ int_val$[T'.tid][k] = \bot_v \wedge k \in keys$
44: **if** ext_val$[tid][k] \neq v$
45: ⏱ T'.EXT $\leftarrow \bot$
46: **else**
47: ⏱ T'.EXT $\leftarrow \top$
48: **if** $k \in keys$
49: int_val$[T'.tid][k] \leftarrow v$
50: **else** ▷ $ts = T'.commit\_ts$
51: **for** $k \in$ DOM(int_val$[T'.tid]$)
52: int_val$[T'.tid][k] \leftarrow \bot_v$ ▷ reset int_val of T'
53: $keys \leftarrow keys \setminus T'.wkey$
54: **if** $keys = \emptyset$
55: **break**
56: **for** $k \in keys$
57: frontier_ts$[ts][k] \leftarrow$ ext_val$[tid][k]$
58: ext_val$[tid] \leftarrow \emptyset$ ▷ gc ext_val of T

59: **procedure** TIMEOUT(T) ▷ ⏱ runs when timeout for T expires
60: **if** T.EXT $= \bot$
61: *report a violation of* EXT

▷ gc frontier_ts, ongoing_ts, and transactions below timestamp ts
62: **procedure** GARBAGECOLLECT(ts) ▷ runs periodically
63: **for** $ts' \leq ts$
64: frontier_ts$[ts'] \leftarrow \emptyset$
65: ongoing_ts$[ts'] \leftarrow \emptyset$
66: $\mathcal{T} \leftarrow \mathcal{T} \setminus \{T \in \mathcal{T} \mid T.commit\_ts \leq ts\}$

sorting transactions based on their timestamps *a priori* as in CHRONOS, AION inserts the new transaction into an already sorted list of transactions, which can be done in logarithmic time using, for example, balanced binary search trees. Therefore, the time complexity of AION for each new transaction is $O(\log N + M)$, where N is the number of transactions received so far, and M is the number of operations in the transactions.

The space complexity of AION is primarily determined by the memory required for storing the ever-growing history, and the versioned frontier_ts and ongoing_ts data structures. Thanks to its garbage collection mechanism, when asynchrony is minimal, AION only needs to keep in memory a few recent transactions and versions of frontier_ts and ongoing_ts.

AION follows a "simulate-and-check" approach, and its correctness is straightforward when all transactions are processed in ascending order based on their start/commit timestamps, ensuring no re-checking is required (by the timeout mechanism described in Section IV-A). However, when out-of-order transactions necessitate re-checking, it is crucial to determine which transactions should be re-checked against which axioms. A formal argument is provided in [32].

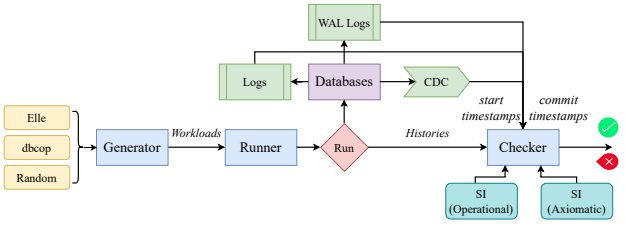


Fig. 3: Workflow of online timestamp-based SI checking.

IV. IMPLEMENTATION

We have implemented the CHRONOS and AION checkers in 3.6k lines of Java code [33]. To evaluate performance, we conducted extensive experiments on three representative

transactional databases: the relational databases TiDB and YugabyteDB and the graph database Dgraph.

A. Workflow

Figure 3 illustrates the workflow of online timestamp-based SI checking. The generator creates a workload of transactions based on specified templates and parameters, which is executed by a database system to generate a history. Our approach extracts the start and commit timestamps of each transaction from the database logs or its Change Data Capture (CDC) mechanism. The history is then checked for violations.

The timeout mechanism sets a deadline for checking and re-checking the EXT axioms for each transaction. Within this period, AION does not report the EXT violation as it is subject to change due to asynchrony. Once the timeout expires, the EXT checking result is considered final and is reported to clients. In our evaluations, we conservatively set 5 seconds.

B. Generating Workloads

To generate key-value histories, we use various workloads tailored to the specific database systems:

- For TiDB and YugabyteDB, we use a schema consisting of a two-column table, where one column stores keys and the other stores values. TiDB and YugabyteDB automatically translate the SQL statements into key-value operations on their underlying storage engines [34].
- For Dgraph, we represent key-value pairs as JSON-like graph nodes with two fields, "uid" and "value". Updates to node data are performed using "mutation" operations, which are transformed into key-value operations by Dgraph's underlying storage engine.

To generate list histories, we adapt our approach based on the data types. For example, in TiDB and YugabyteDB, a comma-separated TEXT field can be used to represent a list value. The append operation can be implemented using an "INSERT ... ON DUPLICATE KEY UPDATE" statement to either insert or concatenate values. Following previous work [19], [16], [20], [28], we consider only committed transactions for verification.

C. Extracting Transaction Timestamps

We summarize the lightweight methods for extracting timestamps from the representative databases as follows: (1) For **TiDB**, we modified its source code for CDC [35], adding simple print statements to expose transaction timestamps. (2) **YugabyteDB** stores transaction timestamps in the Write-Ahead Log (WAL). (3) For **Dgraph**, *start* timestamps are included in the HTTP responses. We further modified Dgraph's source code to include *commit* timestamps in HTTP responses as well, a change that has been officially merged into the main branch of Dgraph [36].

V. EXPERIMENTS ON CHRONOS

In this section, we evaluate the offline checking algorithm CHRONOS and answer the following questions:

TABLE I: Parameters and values of workloads.

Parameters	Values	Default
Number of sessions (#sess)	10, 20, 50, 100, 200	50
Number of transactions (#txns)	5K, 100K, 200K, 500K, 1,000K	100K
Number of operations per transaction (#ops/txn)	5, 15, 30, 50, 100	15
Ratio of read operations (%reads)	10%, 30%, 50%, 70%, 90%	50%
Number of keys (#keys)	200, 500, 1000, 2000, 5000	1000
Distribution of key access (dist)	Uniform, Zipfian, Hotspot	Zipfian

(1) How efficient is CHRONOS, and how much memory does it consume? Can CHRONOS outperform the state of the art under various workloads and scale up to large workloads?
(2) How do the individual components contribute to CHRONOS's performance? Particularly, what impact does GC have on CHRONOS's efficiency and memory usage?
(3) Can CHRONOS effectively detect isolation violations?

A. Setup

We conduct a comprehensive performance analysis of CHRONOS, comparing it to other checkers: **PolySI** [20] and **Viper** [21] are both SI checkers designed for key-value histories. **Elle** [19] serves as a checker for various isolation levels, including SER and SI. **Emme-SI** [25] is a version-order recovery-based SI checker.

1) Workloads: (1) Default workloads. We tune the seven workload parameters, as shown in Table I, during workload generation. The "Default" column presents the default values for these parameters. For the "hotspot" key-access distribution, we mean that 80% of operations target 20% of keys. (2) **Twitter** [37] is a simple clone of Twitter. It allows users to create new tweets, follow/unfollow other accounts, and view a timeline of recent tweets from those they follow. In our study, we involved 500 users, each posting tweets of 140 words. (3) **RUBiS** [38] emulates an auction platform similar to eBay. , allowing users to create accounts, list items, place bids, and leave comments. We initialized the marketplace with 200 users and 800 items.

2) Setup: We evaluate all checkers using histories produced by industry databases. Specifically, for Twitter and RUBiS dataset, we collect SER and SI histories from Dgraph (v20.03.1). For default dataset, we use SI histories collected from Dgraph (v20.03.1) on a key-value store. However, since Dgraph lacks support for list data types, we use SI histories collected from TiDB (v7.1.0) to compare CHRONOS with ElleList. For SER checking, we collect histories from YugabyteDB (v2.20.7.1). Both Dgraph and TiDB are deployed on a cluster of 3 machines in a local network. Two of these machines are equipped with a 2.545GHz AMD EPYC 7K83 (2-core) processor and 16GB memory, while the third machine has a 2.25GHz AMD EPYC 9754 (16-core) processor and 64GB memory. Both CHRONOS and AION are deployed on the 16-core machine.

B. Performance and Scalability

1) Runtime and Scalability: As shown in Figure 4, CHRONOS, ElleKV and Emme-SI, significantly outperform

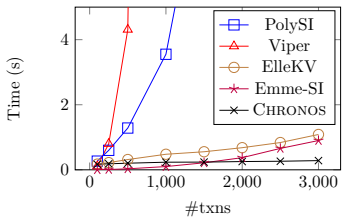
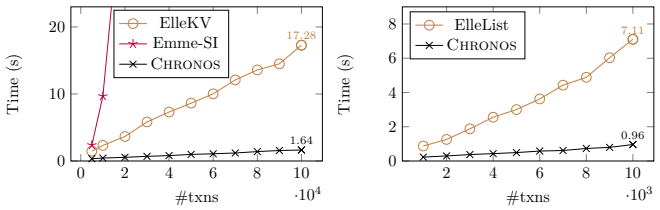


Fig. 4: Runtime comparison on key-value histories under varying number of transactions.



(a) On key-value histories (b) On list histories

Fig. 5: Runtime comparison with Emme-SI and ElleKV/ElleList under varying number of transactions.

PolySI and Viper on key-value histories, which grow super-linearly with transaction numbers. To further compare CHRONOS with ElleKV and Emme-SI on larger key-value histories and ElleList on list histories, we increase the transactions and plot the runtime in Figure 5a and 5b, respectively. Remarkably, CHRONOS can check a key-value history with 100K transactions within 2 seconds (Figure 5a). Emme-SI performance is suboptimal due to its need to construct a start-ordered serialization graph for the entirety of the transaction history. While ElleKV and ElleList show almost linear growth, CHRONOS growth rate is much slower as it processes transactions in timestamp order without cycle detection. Specifically, CHRONOS is about 10.5x faster than ElleKV and 7.4x faster than ElleList. For list histories, which are more complex, CHRONOS takes about one second to check a history with 10K transactions (Figure 5b).

We further investigate the impact of varying workload parameters. The results in Figure 6 (marked with □) align with our time complexity analysis in Section III-B3. Specifically, the runtime of CHRONOS increases almost linearly with both the total number of transactions (a) and the number of operations per transaction (b), while remaining stable for other parameters (c, d). Notably in Figure 6(a), CHRONOS checks key-value histories with up to 1 million transactions in about 17 seconds, showing high scalability, whereas PolySI takes several hours and Viper struggles with large histories [20].

In Figure 6, we also explore the impact of varying GC frequencies on CHRONOS's performance. In these experiments, we trigger GC periodically after processing a certain number of transactions (e.g., 10K and 500K), with 'gc-∞' indicating that no GC is called at all. The results show that as GC is called more frequently, CHRONOS takes longer to check the history. Moreover, while the runtime grows linearly with the

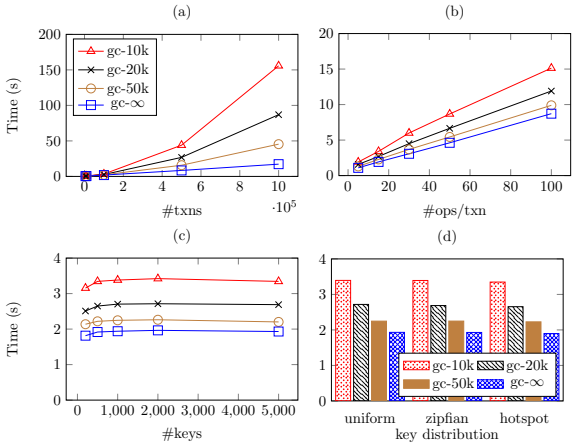


Fig. 6: Runtime of CHRONOS with various GC strategies under varying workload parameters.

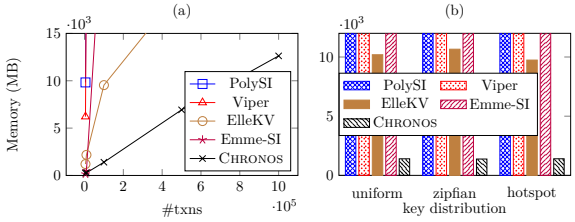


Fig. 7: The maximum memory usage under varying workload parameters.

number of operations per transaction (b), it exceeds linear growth concerning the total number of transactions (a) when GC is called more often, as GC frequencies are tuned based on transactions processed rather than operations.

2) Memory Usage and Scalability: As shown in Figure 7, the *maximum* memory usage of CHRONOS increases linearly with the number of transactions (a) and the number of operations per transaction (b), while it remains stable concerning other parameters (omitted due to space limit). This aligns with our space complexity analysis in Section III-B3.

Figure 7a shows that CHRONOS consumes about 13 GB of memory for checking a key-value history with 1 million transactions. In contrast, PolySI, Viper, and Emme-SI require significantly more memory due to additional polygraph/SSG structures [20], while ElleKV consumes more due to its extensive dependency graphs [19].

C. A Closer Look at CHRONOS

1) Runtime Decomposition of CHRONOS: We measure CHRONOS's checking time in terms of four stages:

- *loading* which loads the whole history into memory;
- *sorting* which sorts the start and commit timestamps of transactions in ascending order;
- *checking* which checks the history by examing axioms;
- *garbage collecting* (GC) which recycles transactions that are no longer needed during checking.

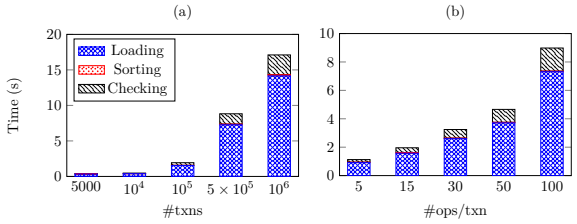


Fig. 8: Runtime decomposition of CHRONOS (without GC) under varying workload parameters.

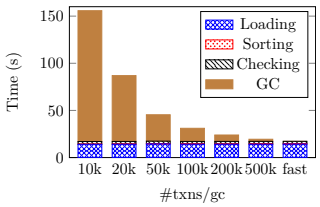


Fig. 9: Runtime decomposition of CHRONOS under varying GC frequencies.

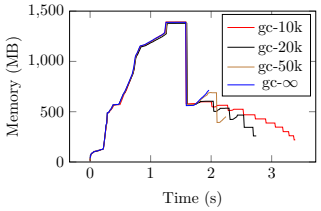


Fig. 10: Memory usage of CHRONOS over time.

Figure 8 shows the time consumption of each stage of CHRONOS *without* GC under varying workload parameters. First, the loading stage, which involves frequent file I/O operations, is the most time-consuming, while the sorting stage is negligible. Second, the time spent on both the loading and checking stages increases almost linearly with the total number of transactions (a) and the number of operations per transaction (b), remaining stable for other parameters (e.g., #session, #keys, read proportion, and key distribution).

Figure 9 shows that varying GC frequencies affect the time consumption of each stage when checking a history of 1 million transactions. Frequent GC calls make it the most time-consuming stage, while the time spent on GC decreases linearly as the frequency of GC decreases.

2) Memory Usage of CHRONOS *over Time:* Figure 10 depicts the memory usage of CHRONOS over time when checking a key-value history of 100K transactions. Initially, memory usage increases steadily, peaking during the loading stage, followed by a sharp decline due to a JVM GC before checking. During the checking stage, the memory usage displays a sawtooth pattern, characterized by intermittent increases followed by decreases after each GC. Overall, the memory usage gradually decreases over time until the checking stage ends. More frequent GC calls lead to smaller memory releases per GC and longer total runtime, indicating that GC should be managed judiciously during the offline checking stage.

D. Checking Isolation Violations

CHRONOS successfully reproduced a clock skew bug in YugabyteDB v2.17.1.0, leading to INT violations under both SI and SER conditions. Additionally, we injected timestamp-related faults into the histories generated by Dgraph, and CHRONOS effectively detected these violations, while non-timestamp-based tools failed. This highlights an important advantage of the timestamp-based checking approach over traditional black-box methods in terms of completeness [25].

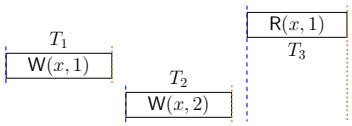


Fig. 11: A history that satisfies SI under traditional black-box SI checking but not under timestamp-based checking.

For instance, in Figure 11, transactions T_1, T_2, and T_3 are committed sequentially. Database developers might expect an SI violation since T_3 reads from a snapshot that excludes the effect of T_2. However, traditional non-timestamp-based SI checkers incorrectly infer an execution order of, i.e., T_1, T_3, T_2, which did not occur.

VI. EXPERIMENTS ON AION

In this section, we evaluate the online checking algorithm AION and answer the following questions:

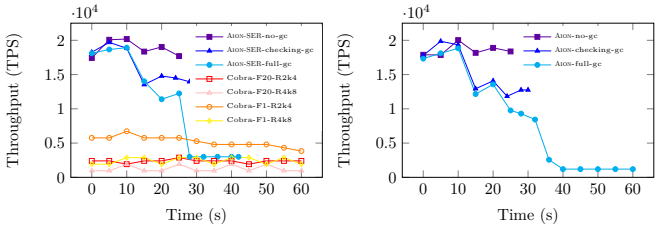
(1) What are the throughputs that AION (for SI checking) and AION-SER (for SER checking) can achieve? Can AION-SER outperform Cobra, the only existing online SER checker?
(2) How does the asynchronous arrival of transaction affect the stability of detecting the EXT violations?
(3) How does the overhead of collecting histories affect the throughput of database systems? How does AION perform with constrained memory resources?

A. Setup

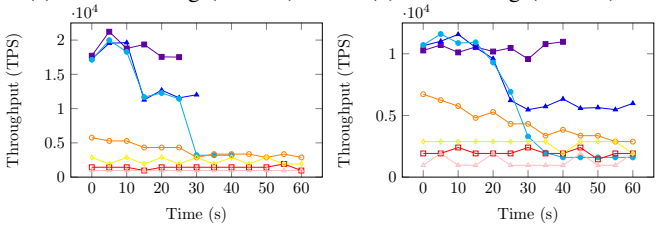
We generate transaction histories using the workload described in Table I, except for the throughput experiments (Section VI-B), where we use #sess=24, #ops/txn=8 for SER and SI checking, and %reads=90% for SER checking (to prevent Cobra from exceeding GPU memory). The configuration of the database systems and the checkers is consistent with the setup described in Section V, except for Cobra, which was tested using an NVIDIA GeForce RTX 3060 Ti GPU, an AMD Ryzen 9 5900 (24-core) CPU, and 64GB of memory. The network bandwidth between the node hosting AION and the database instances is 20Mbps, with an average latency of approximately 0.2ms. History collectors dispatch transaction histories to AION in batches of 500 transactions. The AION-SER algorithm checks whether all transactions appear to execute sequentially in commit timestamp order. Note that start timestamps can be ignored, and there is no need to check the NOCONFLICT axiom. In our tests, database throughput is lower than the checking speed (see Sections VI-C and VI-D). To evaluate AION's throughput limits, we pre-collected logs and then fed historical data exceeding the checkers' throughput (see Section VI-B).

B. Throughput of Online Checking

Figure 12 illustrates the throughput of AION and AION-SER under three different GC strategies, as well as the throughput



(a) SER checking (Default) (b) SI checking (Default)



(c) SER checking (RUBiS) (d) SER checking (Twitter)

Fig. 12: Throughput of online checking over time.

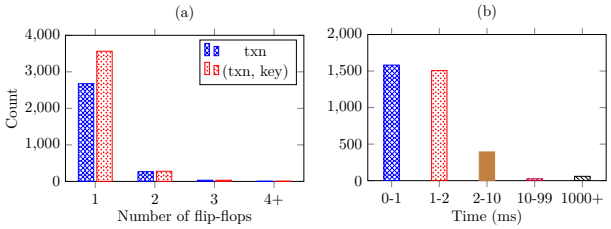


Fig. 13: The number of flip-flops and the time spent on rectifying the false positives/negatives. (The delays injected follow the normal distribution of $N(100, 10^2)$.)

of Cobra under varying fence frequencies (F) and round sizes (R). The total number of transactions is 500K, with an arrival rate exceeding 25K TPS for each checker.

For default benchmark, as shown in Figure 12a, when no GC is performed, AION-SER achieves approximately 18K TPS. When GC is triggered upon reaching a specified threshold for the total number of transactions in memory, AION-SER's throughput decreases to about 12K TPS and stabilizes at this level. In scenarios where a maximum transaction limit is imposed on memory, AION-SER executes GC immediately upon hitting this limit. Since the transaction checking speed surpasses the GC process, AION-SER quickly reaches the limit again, repeatedly triggering GC. Under these conditions, the average throughput drops to roughly 3K TPS.

For Cobra, we tested both the default and optimal configurations of fence frequency and round size, as identified in [16]. With a fixed fence frequency, a round size of 2.4K (the default value) transactions consistently delivers superior performance. In an extreme scenario where a fence is placed between every two transactions, Cobra achieves a peak of around 6K TPS during the first 25 seconds, after which the throughput gradually declines. In all other settings, while Cobra's throughput remains stable over time, it does not exceed 3K TPS.

We also used a history with 500K transactions generated under the SI level to test AION-SER. The result shows that AION efficiently detects all 11,839 violations with a checking speed comparable to that on violation-free histories, while Cobra terminates upon detecting the first violation. We have validated the number of violations by CHRONOS-SER.

Figure 12b shows similar results for AION-SI. Since more information must be stored for SI checking, GC has a greater performance impact compared to SER checking. Notably, when GC is regularly triggered, AION maintains a throughput of approximately 1.2K TPS. Figure 12c and Figure 12d show similar results across datasets. Note that AION's throughput decreases on the Twitter dataset compared to RUBiS, due to the increasing number of keys, requiring high computation and storage to maintain the data structure frontier_ts. Cobra's performance depends on transaction dependencies, influenced by key distribution, read-write ratios, and the number of operations per transaction, rather than the number of keys.

While AION may not always keep up with peak TPS, it can exceed TPS during off-peak periods and eventually catch up. If the sustained TPS consistently exceeds AION's checking speed, we recommend scaling resources or switching to an offline algorithm for efficient verification.

C. Stability of Detecting the EXT Violations

We evaluate the impact of asynchrony on the stability of detecting the EXT violations by counting the number of flip-flops, denoting switches between $T.\text{EXT} \leftarrow \top$ and $T.\text{EXT} \leftarrow \bot$ for each transaction T. As the history collector delivers transactions to the checker in batches (500 transactions per batch), we introduce artificial random delays for each transaction within each batch, following a normal distribution, to mimic asynchrony. For these experiments, we utilize workloads consisting of 10K transactions.

An EXT violation manifests as a transaction-key pair. In Figure 13(a), the right bar denotes the number of EXT violations, while the left bar denotes the number of unique transactions (txn) involved in these violations. In these experiments, we inject delays following a normal distribution with a mean of 100 and a standard deviation of 10, i.e., $N(100, 10^2)$. We observe that 29.8% of transactions exhibit flip-flops, with the vast majority (99%) experiencing them once or twice. Furthermore, Figure 13(b) reveals that over 95% of the false positives/negatives are rectified within 10 ms. Notably, approximately 3% of the false positives/negatives take about 1.5 seconds to be resolved, probably in the subsequent batch. Hence, setting a slightly higher time threshold for reporting violations than the batch latency would help mitigate the impact of these false positives/negatives.

As shown in Figure 14(a), the mean (μ) of delays has a negligible impact on the number of flip-flops since transactions have been equally deferred. Conversely, as shown in Figure 14(b), increasing the standard deviation (σ) of delays leads to a higher number of flip-flops, due to the increased likelihood of transactions arriving out of order.

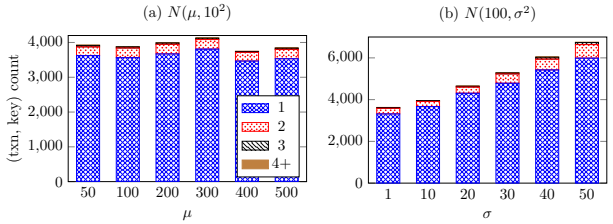


Fig. 14: Number of flip-flops (in terms of (txn, key) count).

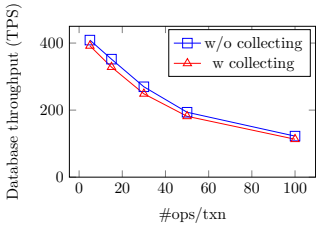


Fig. 15: DB throughput with /without collecting history.

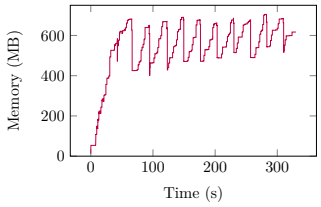


Fig. 16: Constrained memory usage of AION over time.

D. Overhead

Figure 15 compares the database throughputs with and without history collection. We observe that collecting (and transmitting) the history leads to a roughly 5% decrease in TPS, indicating a minor impact. In the experiment shown in Figure 16, we set AION to trigger garbage collection (GC) when memory usage exceeds 700M, using a workload of 100K transactions. Initially, memory consumption rises gradually until it hits the threshold, then oscillates between 400M and 700M due to periodic GC, completing all transactions in about 310 seconds. Thus, AION effectively manages continuous online checking with limited memory resources.

VII. Related Work

The SER and SI checking problems are known to be NP-complete for general key-value histories with unknown transaction execution orders [1], [17]. Existing checkers are offline and fall into two categories: those for general histories and those for histories with additional information.

Checkers for General Histories: Algorithms for SER and SI checking, such as those in [17], operate in polynomial time under fixed input parameters, but their efficiency is limited due to potentially large parameters [16], [20]. Cobra encodes the SER problem into an SAT formula and solves it with the MonoSAT solver [39]. Similarly, PolySI encodes the SI problem into a SAT formula, also solved using MonoSAT. Viper [21] introduces BC-polygraphs to reduce the SI checking problem to cycle detection on BC-polygraphs, solved via MonoSAT. However, these checkers require exploring all potential transaction execution orders, leading to exponential computational costs with the number of transactions.

Checkers for Histories with Additional Information: Elle [19] can infer transaction execution orders in histories with specific data types and the "unique-value" assumption, scaling well to tens of thousands of transactions but struggling with larger histories due to cycle detection costs. While Elle effectively handles certain data types, it has limited capabilities for others, such as key-value pairs. Clark et al. [24], [25] introduced version order recovery, leading to the development of the timestamp-based checker Emme for SER and SI checking. However, Emme relies on costly cycle detection, making it unsuitable for online checking.

To our knowledge, no existing checkers support online SI checking like our AION. Cobra is the only checker that supports online SER checking but requires "fence transactions", which is often impractical in production environments.

VIII. Conclusion, Discussions, and Future Work

In this work, we address the problem of online transactional isolation checking in database systems by extending the timestamp-based offline isolation checking approach to online settings. Specifically, we develop an efficient offline SI checking algorithm, CHRONOS, which is inherently incremental. We extend CHRONOS to the online setting, introducing the AION algorithm. We also develop AION-SER, an online SER checker. Experimental results demonstrate that AION and AION-SER can handle online transactional workloads with a sustained throughput of approximately 12K TPS.

Unlike black-box methods [16], [20], CHRONOS and AION require some database knowledge and kernel modifications, trading off for improved checking efficiency. We plan to extend CHRONOS and AION to support more data types and complex queries, such as SQL queries with predicates, by inferring the commit/version order [25] and/or transaction snapshots from their start and commit timestamps/IDs.

Our work do not directly work for database systems which implement SI/SER using mechanisms distinct from the timestamp-based methods. However, our approach can be generalized to handle more database systems. For example, Emme [25] leverages PostgreSQL's logical streaming replication and the Debezium's CDC tool to recover the version order, which is defined as the commit order of transactions. In their SI implemenations, databases like SQL Server [40], PostgreSQL [3], and WiredTiger [9] rely on the visibility rules to compute a transaction's snapshot. They associate each transaction with a unique ID, but the transaction IDs, such as those used in WiredTiger, may not directly correlate with the timestamps employed in the database systems considered in this paper. We will explore how to obtain the snapshots in a lightweight way in these databases in future work.

The intra-transaction savepoints [41] are useful for partial rollbacks which can be supported by the underlying recovery algorithm. However, we do not test the recovery mechanism and simply assume it works correctly in this paper and plan to address this in future work.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work is supported by NSFC (62472214) and Natural Science Foundation of Jiangsu Province (BK20242014).

REFERENCES

[1] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, oct 1979. [Online]. Available: https://doi.org/10.1145/322154.322158

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] PostgreSQL, "Database concurrency in PostgreSQL," https://www.red-gate.com/simple-talk/databases/postgresql/database-concurrency-in-postgresql/.

[4] CockroachDB, https://www.cockroachlabs.com/product/.

[5] YugabyteDB, https://www.yugabyte.com/.

[6] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, nov 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732237

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *SIGMOD '95*. ACM, 1995, pp. 1–10. [Online]. Available: https://doi.org/10.1145/223784.223785

[8] Dgraph, https://dgraph.io/.

[9] "WiredTiger (version 10.0.2): Snapshot," http://source.wiredtiger.com/mongodb-5.0/arch-snapshot.html.

[10] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI'10*. USA: USENIX Association, 2010, pp. 251–264.

[11] MongoDB, https://www.mongodb.com/.

[12] TiDB, https://en.pingcap.com/tidb/.

[13] Jepsen testing of PostgreSQL 12.3, https://jepsen.io/analyses/postgresql-12.3.

[14] Jepsen testing of MongoDB 4.2.6, http://jepsen.io/analyses/mongodb-4.2.6.

[15] Jepsen testing of TiDB 2.1.7, https://jepsen.io/analyses/tidb-2.1.7.

[16] C. Tan, C. Zhao, S. Mu, and M. Walfish, "Cobra: Making transactional key-value stores verifiably serializable," in *OSDI'20*, 2020.

[17] R. Biswas and C. Enea, "On the complexity of checking transactional consistency," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360591

[18] Jepsen testing of CockroachDB beta-20160829, https://jepsen.io/analyses/cockroachdb-beta-20160829.

[19] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies from experimental observations," *Proc. VLDB Endow.*, vol. 14, no. 3, pp. 268–280, Nov. 2020.

[20] K. Huang, S. Liu, Z. Chen, H. Wei, D. Basin, H. Li, and A. Pan, "Efficient black-box checking of snapshot isolation in databases," *Proc. VLDB Endow.*, vol. 16, no. 6, p. 1264–1276, apr 2023. [Online]. Available: https://doi.org/10.14778/3583140.3583145

[21] J. Zhang, Y. Ji, S. Mu, and C. Tan, "Viper: A fast snapshot isolation checker," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 654–671. [Online]. Available: https://doi.org/10.1145/3552326.3567492

[22] L. Gu, S. Liu, T. Xing, H. Wei, Y. Chen, and D. A. Basin, "IsoVista: Black-box checking database isolation guarantees," *Proc. VLDB Endow.*, vol. 17, no. 12, pp. 4325–4328, 2024.

[23] H. Ouyang, H. Wei, Y. Huang, H. Li, and A. Pan, "Verifying transactional consistency of mongodb," 2022. [Online]. Available: https://arxiv.org/abs/2111.14946

[24] J. Clark, "Verifying serializability protocols with version order recovery," Master's thesis, ETH Zurich, 2021, https://doi.org/10.3929/ethz-b-000507577.

[25] J. Clark, A. F. Donaldson, J. Wickerson, and M. Rigger, "Validating database system isolation level implementations with version certificate recovery," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24, 2024, p. 754–768. [Online]. Available: https://doi.org/10.1145/3627703.3650080

[26] A. Adya, "Weak consistency: A generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, USA, 1999.

[27] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP '11*. ACM, 2011, pp. 385–400. [Online]. Available: https://doi.org/10.1145/2043556.2043592

[28] A. Cerone, G. Bernardi, and A. Gotsman, "A framework for transactional consistency models with atomic visibility," in *CONCUR'15*, ser. LIPIcs, vol. 42, 2015, pp. 58–71.

[29] A. Cerone and A. Gotsman, "Analysing snapshot isolation," *J. ACM*, vol. 65, no. 2, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3152396

[30] R. Biswas, D. Kakwani, J. Vedurada, C. Enea, and A. Lal, "MonkeyDB: Effectively testing correctness under weak isolation levels," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485546

[31] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *VLDB'06*. VLDB Endowment, 2006, pp. 715–726.

[32] H. Li, H. Wei, H. Ouyang, Y. Chen, N. Yang, R. Zhang, and A. Pan, "Online timestamp-based transactional isolation checking of database systems (extended version)," 2025. [Online]. Available: https://arxiv.org/abs/2504.01477

[33] AION *and* CHRONOS, March 2025, https://github.com/FertileFragrance/TimeKiller.

[34] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang, "TiDB: A Raft-based HTAP database," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3072–3084, aug 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415535

[35] Tsunaou, *TiDB timestamp acquisition*, February 2025, https://github.com/Tsunaou/tiflow/commit/82446cd9bd8a2feb52a7a3f403039620aaec9618.

[36] pydgraph issue, "return 'commit_ts' in the function commit() in txn.py (#213)," https://github.com/dgraph-io/pydgraph/pull/213.

[37] Twitter, *Big Data in Real Time at Twitter*, February 2025, https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter/.

[38] sguazt, *Rice University Bidding System (RUBiS)*, February 2025, https://github.com/sguazt/RUBiS.

[39] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu, "SAT modulo monotonic theories," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15, 2015, pp. 3702–3709.

[40] Microsoft SQL Server, "Snapshot isolation in SQL Server," Accessed April, 2024, https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server.

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