VIPER: A Fast Snapshot Isolation Checker

Jian Zhang, Ye Ji[†], Shuai Mu[⋆], and Cheng Tan

Northeastern University [†]Cockroach Labs [⋆]Stony Brook University

Abstract

Snapshot isolation (SI) is supported by most commercial databases and is widely used by applications. However, checking SI today—given a set of transactions, checking if they obey SI—is either slow or gives up soundness.

We present VIPER, an SI checker that is sound, complete, and fast. VIPER checks black-box databases and hence is transparent to both users and databases. To be fast, VIPER introduces *BC-polygraphs*, a new representation of transaction dependencies. A BC-polygraph is acyclic iff transactions are SI, a theorem that we prove. VIPER also introduces *heuristic pruning*, an optimization to accelerate checking SI by leveraging common knowledge of real-world database implementations. Besides vanilla SI, VIPER supports major SI variants including Strong SI, Generalized SI, and Strong Session SI. Our experiments show that given the same time budget, VIPER improves over baselines by 15× in the workload sizes being checked.

CCS Concepts: • **Information systems** → **Database design and models**; • **General and reference** → **Verification**.

Keywords: Databases, Verification

ACM Reference Format:
Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. VIPER: A Fast Snapshot Isolation Checker . In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3552326.3567492

1 Introduction

Databases provide concurrency control for client access and guarantee *isolation* among clients: each client has a consistent view of the data and concurrency anomalies are prevented. Because concurrency control is often delicate and complex, and violations of isolation often lead to severe consequences [85], checking the isolation level of a database's outputs becomes an important problem. The checking could be for different purposes: a database user auditing if the backend database meets its claim [83], a database testing team testing if there are bugs causing an isolation level violation [63], a deployment team verifying if the database is configured correctly [85].

Many recent works [4, 34, 35, 40, 83] focus on checking the strongest isolation level, serializability. In reality, there is another isolation level that is also widely used: snapshot isolation (SI). SI is supported in most production databases, including Oracle, MongoDB [77], TiDB [58], SQLServer [10], and YugabyteDB [15]. For some databases [11, 15], SI is their default option.

A legitimate question then is: how can we (efficiently) check SI? There are two main challenges to this question. First, the database to be checked often needs to be treated as a black box. Users often use cloud databases or proprietary databases to which they have no source code level access. Even in the cases where source code is available, such as in database testing, black-box checking is still often preferred because it reduces complexity in engineering and provides more flexibility. However, black-box checking is algorithmically hard [31]. All polynomial time checking solutions require knowing the database's internal schedule; that is, for two writes, which one is ordered after the other. In black-box checking, however, we do not have this information. As a consequence, existing black-box SI checkers [30, 63] either handle only small workloads (hundreds of transactions) or give up soundness (we elaborate existing checkers in §2.3 and §8). To check for real-world workloads and keep soundness, we need a checker with much higher performance.

The second challenge is that SI, from the definition level, is not as universally agreed as serializability. Perhaps affected by its "implementation-before-definition" tradition [26], SI has many variant implementations and definitions over the years. The meaning of SI is different for different databases. Even for the same database, different setups may have different guarantees under SI [73]. This creates a challenge, but it also gives the work a greater sense of relevance, as it can answer the user's question "which SI variant does this database provide"?

In this paper, we present VIPER, an SI checker that is sound, complete, fast, and supports major SI variants. By sound and complete, we mean that VIPER accepts iff the given workload is SI (we prove this in §3.3). By fast, VIPER is three orders of magnitude faster than the second fastest checker, for checking a small workload of 400 transactions (all baselines timed out for larger workloads, §7.1). Also, given the same time budget, VIPER can handle >15× larger workloads than baselines.

The performance improvement of VIPER is inspired by recent works that efficiently check the serializability of black-box

EuroSys '23, May 9–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567492

databases. A major reason why checking serializability can be faster than before is that the problem of checking serializability can be converted to searching for an acyclic serialization graph in a family of graphs. Then this searching problem can benefit from the evolution and optimizations in SAT/SMT solvers such as MonoSAT [25]. However, SI is different. When mapping SI to a serialization graph, SI does not directly translate to simple acyclicity, but rather "cycles with certain edge patterns are disallowed" [16] (§2.2). This keeps us from directly applying existing techniques to checking SI.

To address this issue, VIPER introduces *BC-polygraphs*, a new data structure that represents dependencies between transactions. BC-polygraphs are designed for checking SI and capture two major characteristics of SI in a black-box setting: (i) SI's ordering specifications, for example, a transaction cannot read from concurrent transactions; and (ii) the conceivable but unknown scheduling in the black-box settings, for example, users see two committed transactions writing the same key, but the database may order either one before the other in its internal scheduling. We define BC-polygraphs in section 3.1.

Crucially, we prove that a BC-polygraph is acyclic iff the given transactions are SI (Theorem 5, §3.3). A noteworthy bonus is a necessary and sufficient condition for SI, namely an SI definition (Theorem 4, §3.3). This definition is more intuitive than existing SI definitions (§3.4).

To further accelerate checking, VIPER introduces an optimization called *heuristic pruning*. It is a heuristic approach which works well for real-world workloads and databases. Heuristic pruning is based on an observation that database implementations hardly delay writes for long, or let reads retrieve really old snapshots (though SI allows so). Therefore, VIPER assumes some of the unknown ordering of transactions, and prunes impossible schedules. Of course, if the assumption is false, VIPER needs to backtrack and update the assumption. Nonetheless, if the assumption is correct, VIPER can finish much faster due to a smaller search space.

VIPER also supports range queries (§4) and major SI variants (§5), including Adya SI [16, 17], Generalized SI [49] (an equivalence of ANSI SI [26]), Strong Session SI [43] (an equivalence of Prefix-Consistent SI [49]), and Strong SI [43]. Checking different SI variants requires various ordering restrictions of transactions, which are abstracted as different types of edges in BC-polygraphs. For example, VIPER uses *real-time edges* for checking Strong SI and *session-order edges* for checking Strong Session SI.

To summarize, the contributions of this paper are as follows.

- We introduce BC-polygraphs, a new data structure, for black-box checking SI.
- We discover a new SI definition that has a nice parallel to serializability, and is easier to remember.
- We design and implement VIPER, a black-box SI checker that is sound, complete, and fast.

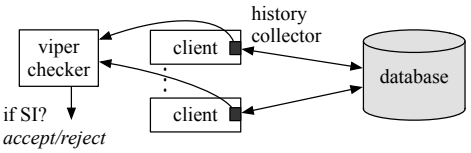

Figure 1. VIPER's architecture. The database is a black box. Inputs to and outputs from the database are captured by history collectors (a library within clients). VIPER checks if the given inputs and outputs (a history) are SI.

- VIPER supports range queries and major SI variants by extending BC-polygraphs with assorted edges.

By our experiments with five baselines and five benchmarks (§7), VIPER outperforms baselines by much, both in terms of checking time for the same workloads and the manageable workload sizes (number of transactions) given the same time budget. In particular, given the workload of 400 transactions of a microbenchmark (BlindW-RW), VIPER finishes in 0.04 seconds while the second best checker (GSI+Z3, §2.3) finishes in 115.98 seconds, a 2900 × speedup. For the same benchmark, VIPER finishes checking 10K transactions in 439.7 seconds. For macrobenchmarks, VIPER finishes 5K-transaction workloads of TPC-C, RUBiS, and Twitter respectively in 2.48, 17.39, and 0.56 seconds (§7.2).

2 Checking SI, status quo

2.1 Problem statement

We target the problem of checking the snapshot isolation (SI) of a *black-box* database. A checker solving this problem has limited information. It only sees the inputs to and outputs from the database, without any internal information. For example, the checker does not know the order of two conflicting writes to the same key because the return values do not reveal which write happens first.

Why black-box checking? There are many scenarios where black-box checking is desired or necessary. For example, cloud database users cannot see the internal of their databases. If they want to check SI, they have to do it the black-box way. Similarly, administrators who host closed source and proprietary databases may not be able to fetch the wanted internal information. Even if the database is open source, it might be hard to instrument and get some extra hints from a distributed database that implements Paxos/Raft and concurrency control protocols, plus many optimizations. Meanwhile, black-box checking is transparent to databases, and can check SI for all these setups, while databases run as-is.

VIPER setup. Figure 1 depicts VIPER's architecture.

Clients send requests to a black-box database and receive responses. The database claims to be snapshot isolation (SI). But clients want to check if the database indeed keeps its promise; therefore, clients forward all the requests and responses to a *checker*. The checker will answer the question: is this set of requests and responses SI?

Each client request includes one or multiple *operations*. There are eight operations: begin, commit, abort (which refer to *transactions*), insert, delete, read, write, and range query (which refer to keys).

All operations referring to keys are wrapped in transactions. For each client, commit/abort operations always pair with begin operations. *History collectors* enforce this logic.

History collectors is a library that logs the operations that clients issue and the values that the database returns. They work as a shim layer between clients and the database, and are transparent to applications because they provide the same key-value semantics. History collectors append a unique write id (a unique hash) to each value, so that all written values are unique. This will piggyback a piece of metadata to each value.

Checker receives logs from history collectors and summarizes them as a *history*. History is the set of all operations clients sent and the corresponding return values from the database. The checker then answers the question—is this history SI? If yes, it accepts; otherwise, it rejects. Answering this question is computationally challenging because the underlying problem is NP-Complete [31].

2.2 SI definitions

A brief history. Snapshot isolation (SI) was first officially introduced by Berenson et al. [26] in 1995. Intuitively, SI requires that all reads in the same transaction read from a consistent database snapshot, and if there are concurrent writes to the same key, only one can commit; others need to abort.

Later, many SI variant proposals arose, including Generalized SI, Prefix-Consistent SI [49], Strong SI, and Strong session SI [43], just to name a few. The definitions until then were descriptive and implementation-based, hence were hard to use for reasoning if a history is SI. Meanwhile, in 1999, Adya gave a graph-based SI definition in his thesis [16] which clarified multiple isolation levels and summarized their relationships. The SI definition is yet based on the full knowledge of the database—it requires the internal interleaving (called *version order*) of transactions.

In 2015, Cerone et al. [37] gave an axiom-based definition of SI and later proved its equivalence to Adya's definition [39]. Similarly, this definition still requires internal information—the order of conflicting transactions (which is defined as a relation called *AR* to arbitrate conflicts). Most recently, in 2017, Crooks et al. [42] proposed a client-centric definition of SI, which was the first definition treating databases as black boxes. Because of the client-centric setting, they proved that some SI variants are equivalent from a client's perspective, and provided a hierarchy of these SI variants [42, Figure 4]:

$$\begin{aligned} \textit{Strong SI} \subset (\textit{PC-SI} \equiv \textit{Strong Session SI}) \\ \subset (\textit{GSI} \equiv \textit{ANSI SI}) \subset \textit{Adya SI} \end{aligned}$$

where $\equiv$ indicates equivalent; and $SI_A \subset SI_B$ means that histories accepted by SI_A are a subset of those by SI_B. In other words, SI_A is "stricter" than SI_B.

Adya SI definition. Next, we introduce Adya's SI definition [16, 17]. Adya SI is the most widely used definition in SI checking [7], and is defined on *start-ordered serialization graphs* (SSGs) [16]. SSGs are directed graphs in which nodes are transactions and edges are dependencies between transactions. There are four types of edges (T_i, T_j are transactions and x is a key):

- *read-dependency* edge ($T_i \xrightarrow{\text{wr(x)}} T_j$): T_j reads x's value that T_i writes.
- *write-dependency* edge ($T_i \xrightarrow{\text{ww(x)}} T_j$): T_j overwrites x's value that T_i writes.
- *anti-dependency* edge ($T_i \xrightarrow{\text{rw(x)}} T_j$): T_i reads a version of x, and T_j writes the next version.
- *start-dependency* edge ($T_i \xrightarrow{\text{start}} T_j$): the timestamp of T_i's commit is earlier than T_j's begin timestamp. The timestamps can be either wall-clock timestamps or logical timestamps.

Notice that, in a black-box database setting, write-dependencies, anti-dependencies, and start-dependencies are unknown to clients because databases do not reveal internal execution order. With SSGs, we define SI below.

Definition 1 (Adya SI definition)**.** Given a time-precedes order (a total order) of the begin/commit timestamps of transactions, a history is snapshot isolation iff its start-ordered serialization graph (1) does not have cycles consisting of only start-dependency, read-dependency and write-dependency edges, (2) does not have cycles with exactly one anti-dependency edge, (3) proscribes "G-SIa": there is a read-dependency or write-dependency edge from T_i to T_j but without a start-dependency edge from T_i to T_j.

Definition 1 is a simplification of the original Adya SI [16, Page 81, Theorem 2]. We omit range query (predicate dependencies), which we will add back in section 4.

Other common SI variants—including GSI, Strong SI, and Strong Session SI—are stricter than Adya SI. Besides requirements in Definition 1, GSI and Strong SI enforce that a read must read from transactions that commit *in real time* before the read transaction. In addition, Strong SI requires all reads read from the most recent snapshots in real time, and GSI allows reading from old snapshots. Strong Session SI requires all reads in a session read from the most recent snapshots in the same session in real time.

2.3 Checking SI, status quo

Existing SI checkers. People build several SI checkers with different setups. dbcop [30] is an isolation level checker that can check SI (specifically, Strong Session SI [43]) for black-box databases, the same setup as VIPER. It runs two algorithms for checking SI. One is an SI-checking algorithm introduced by Biswas and Enea [31]; the other uses MiniSAT [47], a SAT solver, plus a SAT encoding according to Cerone's SI definition [37]. Elle [63] is another black-box checker. Different

from dbcop and VIPER, Elle requires atomic update operations that reveal write order in databases. For example, by using the "append" operation, Elle can infer the write (i.e., append) order to a list. Elle checks Adya SI.

Ouyang et al. [73] build a white-box SI checker for verifying SI of MongoDB [77]. They use SI of Cerone et al. [37], and the checker requires full knowledge of the database.

Natural baselines: SI encoding + SAT/SMT solvers. Beyond existing checkers, there are natural baselines for checking SI. One approach is to solve the SI-checking problem by SAT/SMT solvers [29]. This makes sense because many hard problems in practice can be solved by using solvers [23, 36, 61, 83], due to remarkable advances of SAT/SMT solvers [14, 24, 44, 47, 72, 81]. Indeed, this is an established approach used by prior works to check serializability [4, 79, 83]. We implement three baselines—named GSI+Z3, ASI+Z3, and ASI+Mono—with different encodings and solvers. We will elaborate in section 6.

Challenge. The major technical challenge of checking SI is performance. For a small history of 400 transactions with 50% read-only transactions and 50% write-only transactions (a benchmark BlindW-RW, §7), all baselines and existing black-box SI checkers mentioned above (Elle excluded, because it requires write order being manifested) take more than 116 seconds to finish. VIPER is designed to tackle this challenge.

3 Checking SI with BC-polygraph

To accelerate checking SI, we propose BC-polygraphs (meaning Begin and Commit polygraphs). BC-polygraphs are a new transaction dependency representation that captures two pieces of critical information: (1) SI's ordering specifications, for example, a transaction cannot read from concurrent transactions; and (2) the "conceivable-but-unknown" scheduling; for example, if two committed transactions write the same key, one has to commit before the other begins, but clients do not know which transaction commits earlier because of the black-box database setup.

In the following, we first introduce BC-polygraphs (§3.1), and how to check SI using BC-polygraphs (§3.2). Then we show that our SI-checking algorithm is sound and complete with a proof sketch in section 3.3. Finally, we introduce an optimization that works well for real-world workloads (§3.5).

3.1 BC-polygraph

We start with a setup where all the dependencies (§2.2) are known (this is a white-box setup). In such a case, we can construct *BC-graphs* (a variant of BC-polygraphs) from a start-ordered serialization graph (SSG).

Definition 2. Given a history h and its $SSG(h)$, a BC-graph is constructed as follows:

1. for each node (committed transaction) in $SSG(h)$, create two nodes B_i and C_i;
2. for each read-dependency and write-dependency edge $T_i \to T_j$ in $SSG(h)$, create an edge from $C_i \to B_j$; for each anti-dependency edge $T_i \to T_j$, create an edge $B_i \to C_j$.

Notice that BC-graphs do not have to be defined on SSGs. We use this constructive definition to highlight the relationship between SSGs and BC-graphs.

Next, we move to define BC-polygraphs in the black-box setup where write-, anti-, and start-dependencies are unknown. Intuitively, a BC-polygraph can be considered as a combination of a BC-graph and a set of *constraints* (defined below) that captures the unknown but possible edges.

Consider a history: $T_1 : w(x, 1), T_2 : w(x, 2), T_3 : r(x, 1)$ (values are unique and we omit begins/commits for simplicity). From the history, we can conclude $T_1 \xrightarrow{\text{wr(x)}} T_3$ because T_3 reads x from T_1, but we don't know the order between T_1 and T_2 (conflicting writes) and between T_2 and T_3 (a conflicting read and write pair). To capture these conceivable-but-unknown dependencies, BC-polygraphs define *constraints*: a set of bi-edges each of which is an edge pair $\langle v \to u,\ u \to w \rangle$ such that one and only one edge appears in a BC-graph. Figure 2 depicts a BC-polygraph for the example history.

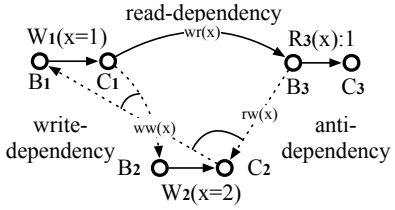

Figure 2. A BC-polygraph example. Nodes represent begin and commit operations (denoted as B_i and C_i for T_i). There are three transactions hence six nodes in this BC-polygraph. A read-dependency edge $C_1 \to B_3$ indicates T_3 reading from T_1. Constraints are indicated by dotted edges connected by an arc. There are two constraints: $\langle C_1 \to B_2,\ C_2 \to B_1 \rangle$ (indicating no two concurrent writes to the same key; one has to commit before the other begins) and $\langle B_3 \to C_2,\ C_2 \to B_1 \rangle$ (indicating R_3 either reads from a snapshot before W_2 or W_2's value has been overwritten).

BC-polygraphs have two types of edges that are known: (i) the edges starting from a transaction's begin to its own commit, called *intra-txn dependencies*, which reflect the program order; and (ii) the read-dependency edges pointing from a commit of *wtx* to the begin of *rtx*, where *rtx* reads from *wtx*. This represents read-dependencies and indicates that SI only allows a read reading from committed transactions.

BC-polygraphs capture write-dependencies and anti-dependencies as constraints. In particular, write-dependency edges point from commits to begins; anti-dependency edges however point from begins to commits. The begin-to-commit order of an anti-dependency indicates that a transaction should not read from a concurrent transaction, as defined in SI. (By concurrent transactions, we mean these transactions all start before any commits.) We define BC-polygraphs below.

Definition 3 (BC-polygraph)**.** A BC-polygraph $P = (V, E, C)$ is a directed graph (V, E) (called the known graph) together with a set of edge pairs C (called constraints).

- V consists of begin operations and commit operations for all committed transactions.
- E includes edges of intra-txn dependencies and read-dependencies.
- C is a set of constraints; each constraint contains two edges (either a write-dependency and an anti-dependency or two write-dependencies), only one of which should exist.

A BC-polygraph can be regarded as a superposition of many directed graphs. By choosing one edge in each constraint of a BC-polygraph pg, we can get a directed graph g. We call such g is *compatible* with pg. In particular, (i) g has the same nodes as pg, and (ii) g includes all edges in pg's known graph, and (iii) g contains one edge in each constraint of pg.

We say a BC-polygraph is *acyclic* if there exists a directed graph of the BC-polygraph that is acyclic. *Crucially, a history is SI iff its BC-polygraph is acyclic.* We prove this as a theorem (Theorem 5) and will show a proof sketch in section 3.3.

Detecting SI violations: a long-fork example. We use an example to demonstrate how BC-polygraphs detect SI violations. Consider a history, $T_1 : w(x,1), w(y,1), T_2 : r(x,1), w(x,2), T_3 : r(y,1), w(y,2), T_4 : r(x,2), r(y,1), T_5 : r(x,1), r(y,2)$ (borrowed from [41, §3.2]). It is known as a long fork [80], which is not SI. In this history, transactions update two disjoint sets concurrently which makes the states fork, and the states are not merged back after they commit. Figure 3 depicts the history's BC-polygraph. (We omitted some irrelevant constraints for simplicity.) No matter how one picks edges in the two constraints, there will always be cycles.

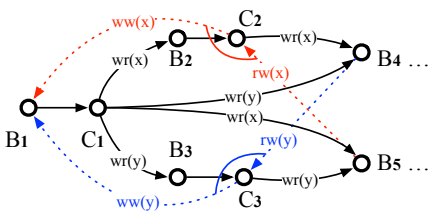

Figure 3. The BC-polygraph of the long fork example. There are two constraints indicated by red and blue dotted lines and arcs. We denote dependencies (§2.2) on edges. There is always a cycle whichever two constraint edges we choose.

3.2 Checking SI

Figure 4 depicts VIPER's SI-checking algorithm. There are three major steps. First, VIPER constructs a BC-polygraph from a given history. Second, VIPER encodes the BC-polygraph into SMT clauses. And finally, VIPER runs MonoSAT [25] to solve the clauses. MonoSAT is an SMT solver that has native supports for graph properties (for example, acyclicity).

Constructing BC-polygraph. Compared with existing SI checkers and baselines, VIPER's major difference (and advantage) is using BC-polygraphs. VIPER traverses the history and builds a BC-polygraph by adding begins and commits as nodes to the BC-polygraph (line 28, Figure 4). After that, VIPER adds known edges, including intra-txn dependency edges (line 29, Figure 4) and read-dependency edges (line 34, Figure 4). Finally, it adds constraints with regard to write-dependencies (line 46, Figure 4) and anti-dependencies (line 50, Figure 4).

VIPER can build a BC-polygraph with time complexity of $O(n^2)$, where n is the number of read and write operations in the history. $O(n^2)$ comes from line 42–50 in Figure 4. Seemingly, this is a triple nested loop. However, the outermost loop (line 44, Figure 4) and the innermost loop (line 48, Figure 4) together have an $O(n)$ complexity because the inner loop is about the reads (rtx in Figure 4) of the outer loop's writes ($vwtx_1$ in Figure 4)—the two loops combined are all read operations (whose number is $< n$) in this history.

Encoding and solving. VIPER encodes the BC-polygraph into SMT clauses (line 12, Figure 4) by using SMT graph abstraction. In particular, for each edge, VIPER uses a boolean variable to represent if the edge exists: true means existence (for example, line 19, Figure 4); false means otherwise. To encode constraints, VIPER *XORs* (exclusive OR) the constraint's two edges (line 22, Figure 4): *either* one edge's boolean is true *or* the other; one and only one boolean must be true. Finally, VIPER runs MonoSAT to check if there exists any assignment to edges such that the BC-polygraph is acyclic. If so, the history is SI and VIPER accepts; otherwise, VIPER rejects.

3.3 Correctness proof

In this section, we provide proof sketches to our claim: a history is SI iff its BC-polygraph is acyclic (§3.1) which is Theorem 5. This theorem is the core of VIPER's SI-checking algorithm (Figure 4). The full proof can be found in our technical report [13].

Before proving the main Theorem 5, we need a helper theorem that covers a simpler setup—white-box checking when the write-dependencies and anti-dependencies are known.

Theorem 4. *Given a history h and all write- and anti-dependencies h's BC-graph g is acyclic $\iff$ h is SI.*

Proof. "$\Rightarrow$". Since g is acyclic, by topological sorting g, we can have a total order of begins and commits, $\hat{s}$. Next, we can build a $SSG(h)$ according to $\hat{s}$ that proscribes G-SIa by construction. We can also prove that $SSG(h)$ has no cycles with 0 or 1 anti-dependency edge because any such cycle will contradict with $\hat{s}$ (details omitted). By Definition 1, h is SI.

"$\Leftarrow$". If h is SI, by Definition 1, $SSG(h)$ can be either (i) has no cycles or (ii) has cycles with more than one anti-dependency edge. For case (i), by Definition 2, BC-graph does not introduce new cycles with respect to the $SSG(h)$, hence g is also acyclic. For case (ii), we claim that *all* cycles in $SSG(h)$ must have consecutive anti-dependency edges. (we

```
1:  procedure CheckSI(history)
2:      g, cons ← CreateBCPolygraph(history)        // line 25
3:      encoding ← ConstructEncoding(g, cons)       // line 12
4:      ret ← MonoSAT.solve(encoding)               // run MonoSAT
5:      if ret is unsat:
6:          return REJECT
7:      else:
8:          return ACCEPT
9:
10:
11:
12: procedure ConstructEncoding(g, cons)
13:     lits ← empty list
14:     ĝ ← MonoSAT.graph()          // an empty MonoSAT graph
15:     for node n in g:                    // construct nodes
16:         ĝ.Nodes += n
17:     for edge (n1, n2) in g:             // construct known edges
18:         ĝ.Edges += (n1, n2)             // add symbolic edges
19:         lits += ((n1, n2) = True)       // the edge exists
20:     for ⟨(n1, n2), (n2, n3)⟩ in cons:   // encode constraints
21:         ĝ.Edges += {(n1, n2), (n2, n3)}
22:         lits += (n1, n2) XOR (n2, n3)   // one of the edges exists
23:     lits += ĝ.acyclic()
24:     return lits

25: procedure CreateBCPolygraph(history)
26:     readfrom ← Map{⟨Key, Tx⟩ → Set⟨Tx⟩} // map a write to its readers
27:     for transaction tx in history:          // create the known graph
28:         g.Nodes += {tx.begin, tx.commit}
29:         g.Edges += (tx.begin, tx.commit)    // intra-txn dependency
30:         for read operation rop in tx:
31:             // if read from an aborted transaction, reject
32:             if rop.read_from_tx not in history:  REJECT
33:             // add read-dependency (we assume a txn writes a key at most once)
34:             g.Edges += (rop.read_from_tx.commit, tx.begin)
35:             readfrom[⟨rop.key, rop.read_from_tx⟩] += tx
36:
37:     cons ← empty list                       // create constraints
38:     writes ← Map{Key → Set⟨Tx⟩}             // map a key to its writers
39:     for transaction tx in history:
40:         for write operation wrop in tx:
41:             writes[wrop.key] += tx
42:     for all key in writes:
43:         key_writes ← writes[key]
44:         for wtx1 in key_writes:
45:             for wtx2 in key_writes ∧ wtx2 ≠ wtx1:
46:                 cons += ⟨(wtx1.commit, wtx2.begin), (wtx2.commit, wtx1.begin)⟩
47:                 reads ← readfrom[⟨key, wtx1⟩]
48:                 for rtx in reads:
49:                     // wtx2 commits either after rtx begins or before wtx1 begins
50:                     cons += ⟨(rtx.begin, wtx2.commit), (wtx2.commit, wtx1.begin)⟩
51:     return g, cons
```

Figure 4. Viper SI-checking algorithm.

omit the proof of this claim.) By the above claim, any cycle in $SSG(h)$ has at least two consecutive anti-dependencies, say $T_1 \xrightarrow{\text{rw}} T_2 \xrightarrow{\text{rw}} T_3$. This translates to $B_1 \xrightarrow{\text{rw}} C_2$ and $B_2 \xrightarrow{\text{rw}} C_3$ in a BC-graph, where $C_2 \not\to B_2$. Thus, $SSG(h)$'s cycles are not cycles in the BC-graph, meaning the BC-graph is acyclic. □

Next, we prove our main theorem that Viper's SI-checking algorithm (Figure 4) is sound and complete, namely, a history is SI iff its BC-polygraph is acyclic.

Theorem 5. *Given a history h, its BC-polygraph is acyclic* $\iff$ *h is SI.*

Proof. "⇒". Given the BC-polygraph is acyclic, there exists a compatible graph g that is acyclic. By Theorem 4, h is SI. "⇐". By Theorem 4, h is SI implies there exists a BC-graph g that is acyclic. By topological sorting g, we can get a total order of nodes, $\hat{s}$. We can prove that for each constraint, one edge obeys $\hat{s}$ and the other conflicts with $\hat{s}$ (proof details omitted). Now, by choosing the edges that obey $\hat{s}$ in constraints, we construct a compatible graph g such that the g's edges are a subset of $\hat{s}$; this implies that the BC-graph g is acyclic; thus, the corresponding BC-polygraph is acyclic. □

Notice that different from BC-graphs, BC-polygraphs do not have known write-dependencies and anti-dependencies. BC-polygraphs assume they exist but do not know them. Indeed, the SMT solving phase in Figure 4 is to search for a set of write-dependencies, anti-dependencies, and timestamps such that h is SI.

3.4 Theorem 4 revisited

We want to highlight that Theorem 4 gives a necessary and sufficient condition for SI. In other words, Theorem 4 is *an SI definition*. Moreover, Theorem 4 has a neat parallel to serializability. Below we put the canonical way of checking serializability (short as SER) using serialization graphs (short as SER-graphs) and checking SI by Theorem 4 side by side, and highlight differences by underlines.

checking SER: build a SER-graph in which nodes are transactions and edges are read-dependencies, write-dependencies, and anti-dependencies; the graph is acyclic iff the history is SER.	checking SI: build a BC-graph in which nodes are begins/commits and edges are read-dependencies, write-dependencies, and anti-dependencies; the graph is acyclic iff the history is SI.

Compared with the state-of-the-art SI definitions (like Adya SI, Definition 1), Theorem 4 is more intuitive and easier to remember, especially for people who have learned SER.

In fact, this parallel is deeper than their appearances. Here is one way to think of this parallel: given a history h, checking SER is searching for a total order of transactions (call it $\hat{s}_{SER}$), whereas checking SI is searching for a total order of begins and commits (call it $\hat{s}_{SI}$). If h is SER, such $\hat{s}_{SER}$ exists that sequentially executing transactions according to $\hat{s}_{SER}$ reproduces h; if h is SI, such $\hat{s}_{SI}$ exists that sequentially executing begins with all reads (in this transaction) and commits with all writes (in

this transaction) according to $\hat{s}_{SI}$ reproduces h. The $\hat{s}_{SER}$ exists iff the SER-graph is acyclic; The $\hat{s}_{SI}$ exists iff the BC-graph is acyclic. Notice that write-dependencies prevent conflicting concurrent writes in $\hat{s}_{SI}$; read-dependencies prescribe reading from committed transactions in $\hat{s}_{SI}$; and anti-dependencies prevent reading from concurrent transactions in $\hat{s}_{SI}$.

3.5 Heuristic pruning, an optimization

In this section, we introduce an optimization, called *heuristic pruning*, that incorporates some common knowledge from databases. Heuristic pruning significantly accelerates SI checking, sometimes by 36× in our experiments (RUBiS, §7.2).

We observe that in practice, an SI database implementation rarely delays a write for a "long" time or let a read fetch a "really old" snapshot (though SI allows both). Our idea is to heuristically assign orders to those transactions that are "far" from each other because databases are unlikely to reorder them. By adding these hypothetical orders as *heuristic edges* in BC-polygraphs, VIPER can prune constraints that violate these edges, hence reduces the number of constraint decisions to make and accelerates the SI checking. Figure 5 is an illustration of heuristic pruning.

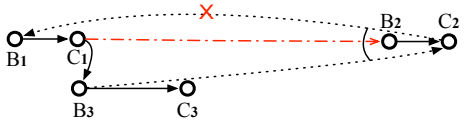

Figure 5. An example of heuristic pruning. There are three transactions: (B_1, C_1), (B_2, C_2), and (B_3, C_3). Solid edges are known edges; dotted edges connected by an arc is a constraint; and the dash-dotted edge (in red) is a heuristic edge.
Assume the heuristic edge exists. VIPER then can prune the constraint by adding $B_3 \rightarrow C_2$ and discarding $C_2 \rightarrow B_1$ (the crossed out edge) because otherwise by choosing $C_2 \rightarrow B_1$, there will be a cycle.

VIPER implements heuristic pruning as follows. It first topological sorts the known graph of the BC-polygraph, ignoring the constraints, and gets a total order of begin/commit nodes as $\hat{s}$. Then, VIPER inserts heuristic edges from commit nodes to begin nodes that distance by k nodes in $\hat{s}$, where k is a parameter. If VIPER accepts, the history is SI. If VIPER rejects, we double k until $k = \#nodes$ (namely, no heuristic edges). This optimization works for histories that are SI. For non-SI histories, heuristic pruning plays a negative role in performance because VIPER needs retry multiple times to confirm the results. (We describe more implementation details in §6.)

4 Supporting range query

Adya SI definition supports range queries (predicate dependencies). In this section, we extend VIPER to support them.

Range queries are widely used in applications. VIPER supports one type of range queries that are *key-based*, namely the range refers to keys (instead of values). For example, assume a user uses string keys and considers alphabetical order; a range query RAN("a","b") should return all existing keys that locate in-between key "a" and "b" (regarding alphabetical order).

Challenge. To check SI, range queries have two properties that need to be verified. First, all returned keys must be within the range, which is easy to check. Second, all non-returned keys in the range—potentially many—must be nonexistent. This property is hard to check when intertwined with concurrent transactions. We illustrate the challenge by an example.

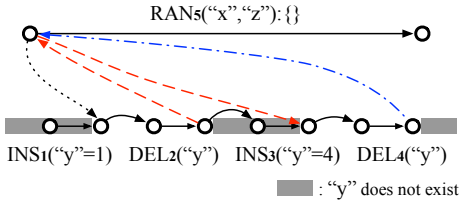

Figure 6. A range query example for non-existing "y". There are five transactions (begin/commit labels omitted): two transactions insert key "y" (INS_1 and INS_3); two transactions delete key "y" (DEL_2 and DEL_4); and one transaction issues a range query (RAN_5) for keys between "x" and "z" (including "y") and gets nothing "{}" ("y" is nonexistent) as the return value.
The return value implies three possible cases when RAN_5 begins: (1) before INS_1 commits (the dotted black edge), (2) after DEL_2 commits and before INS_3 commits (indicated by two dashed red edges), or (3) after DEL_4 commits (the dash-dotted blue edge).

Figure 6 depicts a case that a range query RAN_5("x", "z") does not return key "y", indicating "y" does not exist in the database when RAN_5's transaction begins. The question however is: there are three periods in history that "y" is nonexistent (marked by shaded rectangles in Figure 6); which period does RAN_5 fall into? Moreover, notice that "y" is just one non-returned key. There are potentially many non-returned keys (for example, "ya", "yab", "$yabc$"). All of them have their own versions of "Figure 6".

Tombstones. VIPER addresses this challenge by *tombstones* [22, 65, 69, 71, 76, 78]. Tombstones are special strings that VIPER uses as placeholders for deleted keys. In particular, instead of truly deleting a key from databases, VIPER writes a tombstone to the key, which is enforced by history collectors (§2.1) and is transparent to users. Similarly, when inserting a key, VIPER first reads the key; if the return value is either *NULL* (the key doesn't exist in the database) or a tombstone (the key has been deleted), VIPER updates the key and returns.

With tombstones, a range query will return tombstones associated with the keys that should have been deleted. By studying these tombstones, VIPER knows which transactions delete the keys (because tombstones contain unique write ids). Hence, the checker can figure out the ordering of inserts, deletes, and range queries; and avoids the ambiguity in Figure 6. VIPER's checker handles these operations (inserts/deletes/range queries) as if they are normal reads and writes.

Tombstones have been used by many other systems [22, 65, 69, 71, 76, 78]. Different from prior usages, VIPER uses

tombstones for pinpointing conflicting dependencies between range queries and inserts/deletes.

Pros and cons. Using tombstones has pros and cons. The pro is that it significantly accelerates SI checking for range queries. The cons are threefold. First, using tombstones may harm the performance of aggregation operations (e.g., join) because keys are no longer truly deleted. Second, tombstones also waste disk space, which might be a concern for workloads with many temporary data. Finally, if using VIPER for black-box testing, tombstones skip testing the actual implementation of inserts/deletes, which may reveal fewer concurrency bugs.

5 Checking SI variants

So far, VIPER's SI-checking algorithm (§3.2) checks Adya SI. Yet, SI has many variants, and Crooks [41] provides a clear hierarchy of some major SI variants (§2.2). In the following, we extend VIPER to check major SI variants: Generalized SI [49], Strong SI [43], and Strong Session SI [43] (≡ Prefix-Consistent SI [49]). Note that we follow the interpretation of Crooks et al. [42] for SI variants, in particular, whether "timestamps" are wall-clock time or logical timestamps in these variants.

Generalized SI and Strong SI. Generalized SI (GSI) and Strong SI requires real-time constraints. They both require that a transaction T_i must read from transactions that commit, *in real time*, before T_i begins. In addition, Strong SI requires all reads read from the most recent snapshots in real time, whereas SI allows reading from old (in real time) snapshots.

To check GSI and Strong SI, VIPER needs wall-clock timestamps. We assume clients who together check SI belong to the same organization and run some time synchronization protocols, for example, NTP [8] or PTP [48]. Therefore, the clock drift between clients is bounded. VIPER assumes a *clock drift threshold* which is the maximum possible time difference between clients (this is a parameter in VIPER). However, if clients' actual clock drift exceeds the threshold (VIPER's assumption is false), VIPER may reject histories that are SI (VIPER then fails completeness).

With the bounded clock drift assumption, VIPER can check GSI and Strong SI. For each begin and commit, VIPER's history collectors record a local wall-clock timestamp. We say one event E_i *happens before* another event E_j when E_i's timestamp is smaller than E_j's by at least a clock drift threshold.

When checking Strong SI, for a pair of nodes (N_i, N_j) that N_i happens before N_j, VIPER's checker adds a *real-time edge* from N_i to N_j; N_i and N_j can be begin or commit but cannot be both begins. This is a reminiscence of the time-precede order in Adya SI definition, and start-dependency edges in start-ordered serialization graph [16, Page 79, Definition 20].

GSI relaxes Strong SI by not having to read from the most recent snapshots. So, VIPER's checker will ignore real-time edges from commits to begins; such edges represent that transactions must observe happened-before commits in real time.

VIPER component		LOC written/changed
History collector	history recording	382 lines of Java
	database adapters	1427 lines of Java
	Jepsen test	901 lines of Clojure
VIPER checker	data structures and algos	2977 lines of Python
	history parser and others	964 lines of Python

Figure 7. Components of VIPER implementation.

Instead, for checking GSI, VIPER only adds real-time edges from begins/commits to commits. These edges capture concurrent transactions: if a transaction T_i begins before T_j commits and vice versa, then T_i and T_j are concurrent transactions. With begin-to-commit real-time edges for concurrent transactions, a transaction can no longer read from a concurrent transaction (in real time), hence must read from an older snapshot.

Note that with the bounded clock drift assumption, VIPER's checking of GSI and Strong SI is complete but not sound because VIPER may consider a real-time ordering violation as a clock drift and accepts a non-SI history.

Strong Session SI (≡ Prefix-Consistent SI). Strong session SI requires clients to see the same transaction order in the same session. In other words, one client will always see the same history prefix (hence prefix consistent). This SI variant prohibits "transaction inversion" [43]: a client commits an update but cannot see the update in a following read.

In VIPER, we support sessions (as in Strong Session SI) in the granularity of client-database connections, for example, JDBC connections. We assume sessions are synchronous; clients need to commit or abort a transaction then begin the next transaction. VIPER captures transaction session orders by history collectors. The collectors record each session's transactions into an independent file in their issuing order.

Users can configure VIPER to check Strong Session SI. If so, VIPER's checker adds *session-order edges* and *begin/commit-to-commit edges* (identical to GSI) to the BC-polygraph. The session edges are edges pointing from previous transactions (commit nodes) to latter transactions (begin nodes) according to the transaction sequence in each session. The remaining SI-checking algorithm (§3.2) is unchanged.

If VIPER accepts, it guarantees Strong session SI because there exists some acyclic BC-graph that satisfies session-order edges; by topological sorting the BC-graph, one can get a schedule of begins and commits that obeys session orders. On the contrary, if VIPER rejects, no such acyclic BC-graph exists, hence no valid schedule that obeys Strong Session SI.

6 Implementation

VIPER's checker is written in Python, and history collector is written in Java (clients are written in Java). VIPER's collectors are built on top of Cobra's history collectors. Figure 7 shows VIPER's components.

History collector. As mentioned earlier (§2.1), history collectors are implemented as a library on the client side. They

record operations issued by clients and results returned by the database. In addition, collectors also assign each write with a unique *write id*. These ids allow VIPER's checker to associate read values with their corresponding writes. If the database (accidentally or intentionally) tampers with write ids or replaces one write id to another, VIPER can detect this by matching parameters of the writes (logged directly on the client side) and return values of the reads; then VIPER rejects.

Collectors also implement the tombstones for deletes (§4). As a library, collectors provide a delete wrapper for clients. The wrapper replaces a delete operation with a set of operations that read-modify-write the "deleted" key to a tombstone plus a unique write id. In this case, range queries will return "deleted" keys with tombstones. According to write ids, VIPER's checker knows which transactions delete these keys.

Genesis transaction. To handle reading non-existing keys, we created a genesis transaction, which is a virtual transaction that commits before any transaction in the history. All reads to a key that happen before the key's first write are considered to read from the genesis transaction.

Natural baselines. Section 2.3 introduced three natural baselines. The baselines are designed and implemented by us. The core encodings however are borrowed from existing work, for example, encoding graph acyclicity using SAT/SMT clauses [52, 53, 59]. We elaborate on these baselines below.

GSI+Z3: we encode Generalized SI [49] assuming logical timestamps using SMTs, and use Z3 as the solver. First, we assign each begin and commit with a unique integer id, then define a happen-before relation ($<$) over these ids. For example, a transaction begin must happen-before its commit ($Begin(T_i) < Commit(T_i)$). Then, we assert GSI rules according to the paper, including GSI read rule [49, D1] and GSI commit rules [49, D2]. For example, the rule of no concurrent modifications to the same key can be encoded as, if two transactions write the same key (T_i and T_j), one's commit must happen before the other's begin ($Commit(T_i) < Begin(T_j) \vee Commit(T_j) < Begin(T_i)$), If Z3 can find a legal happen-before relation for all begins and commits, then the history is SI; otherwise, it is not SI.

ASI+Z3: we encode Adya SI (Definition 1) as SMTs and solve them by Z3. First, we assign unique integers to transactions (nodes in the serialization graph), and define two relations over these integers (the "Function" in Z3): one for read-dependencies and write-dependencies (call it Rel_0), and the other for anti-dependencies (call it Rel_1). We encode cycles by using transitive closure (node reachability). Then, we encode timestamps by assigning each begin/commit a timestamp (an unsigned integer), asserting that timestamps respect dependencies, and enforcing a total order of these timestamps.

Finally, we assert that there is no cycle for Rel_0 and there are no cycles that have exactly one Rel_1 edge. If Z3 is able to find a satisfiable solution, then the history is SI; otherwise, it is not SI.

ASI+Mono: since MonoSAT natively supports many graph properties, we leverage those to encode Adya SI. We use the graph abstraction provided by MonoSAT for the serialization graph in Adya SI. We assign edges with weights: 0 for read-dependency and write-dependency edges, and 1 for anti-dependency edges. Similar to Z32+ASI, we use bitvectors (bounded integers) as timestamps and enforce their total order. In addition, we assert that timestamps respect dependencies. Then we leverage *node distance*, a primitive provided by MonoSAT, to encode certain cycles. We assert that no node can reach itself by a path (in fact, this is a cycle) with weights ≤ 1, indicating that there is no cycle with 0 or 1 anti-dependency edge (hence is Adya SI).

A note on why VIPER runs faster. The core observation of VIPER is that SI requires a total order of begins and commits, instead of transactions. Therefore, the abstraction of histories should not base on serialization graphs (where transactions are atomic nodes) but on BC-graphs which capture the dependencies between begins and commits. The benefit of this abstraction in implementation is that VIPER's checking algorithm runs faster due to efficient primitives (e.g., graph acyclicity) and fewer constraints.

VIPER checker optimizations. VIPER implements heuristic pruning (§3.5) by topological sorting the known graph of BC-polygraphs to get a total order of nodes, then later uses this to infer unknown order. Usually, there are many valid topological sort results, but not all of them are equally useful. The useful ones are those similar to the (unknown) database execution schedule. To produce high-quality topo-sort results, VIPER implements topological sort using breadth-first search (BFS), and additionally sorts nodes in the same BFS layer by their orders in session logs. Both of these are heuristics to mimic database's execution schedule in practice. VIPER also integrates two optimizations—combining writes [83, §3.1] and coalescing constraints [83, §3.2]—from a serializability checker, Cobra. The two optimizations reduce constraints for workloads with read-modify-writes and many reads.

7 Experimental evaluation

The questions we answer in this section are:

- What is VIPER's performance, and how do these compare to existing checkers and natural baselines?
- How do VIPER's components contribute under different workloads and different setups?
- Can VIPER detect known real-world SI violations?

Baselines. There are five baselines that we experiment with.

- *Elle*: an SI checker that requires revealing write order (in its "sound mode"; we explain its two modes in §8). Elle checks Adya SI.
- *GSI+Z3*: a rule-based SMT encoding of Generalized SI, using Z3 as the solver. (§6)
- *ASI+Z3*: a rule-based SMT encoding of Adya SI, using Z3 as the solver. (§6)

- *ASI+Mono*: a graph-based SMT encoding of Adya SI, using MonoSAT as the solver. (§6)
- *ASI+Mono+Opt*: the baseline ASI+Mono plus Cobra's two optimizations [83, §3.1 and §3.2].

Benchmarks and workloads. We use five benchmarks, two microbenchmarks (V-BlindW and V-Range) and three macrobenchmarks (C-TPCC, C-RUBiS, and C-Twitter). The three macrobenchmarks are borrowed from Cobra Bench [1]: we use their implementations and configurations. We tailor the BlindW [1] to suit our setup, and call it V-BlindW. We also extend BlindW with range queries as a new benchmark, V-Range. We elaborate benchmarks' configurations below.

- *V-BlindW*: there are two types of transactions, read-only transactions and write-only transactions. Each transaction has eight read or write operations to random keys (2K predefined integer keys). By choosing the ratio of read-only and write-only transactions, V-BlindW has two variants: (1) *BlindW-RM* (Read Mostly) with 90% read-only transactions, and (2) *BlindW-RW* (Read-Write) with 50% each.
- *V-Range*: V-Range has five operations: reads, writes, inserts, deletes, and range queries; and keys are integers that are greater than 0. Inserts either add non-existing keys (by increasing the maximum key by 1), or re-insert existing keys. Deletes randomly remove existing keys. A range query will query a random range between 1 and the maximum key inserted. One operation type has one type of transactions; each transaction contains eight operations of the same type. For example, a range query transaction has eight range queries. V-Range has three variants: (1) *Range-B* (Balanced) with 20% transactions of each type, (2) *Range-RQH* (Range Query Heavy) with 50% range query transactions and 12.5% of other types each, and (3) *Range-IDH* (Insert Delete Heavy) with 35% of insert and delete transactions each, and 10% of read, write, and range query each.
- *C-TPCC*: a standard online transaction processing benchmark [12]. We use a configuration of one warehouse, 10 districts, and 30K customers; with transaction types: new order, payment, order status, delivery, and stock level whose frequencies are 45%, 43%, 4%, 4%, and 4%, respectively.
- *C-RUBiS*: a bidding system like eBay. We use a configuration of 20K users and 80K items.
- *C-Twitter*: a simulation of a tiny Twitter. We use a configuration of 1K users.

Experiment setup. To generate histories, we run our five benchmarks on three databases, TiDB [58], SQLServer [10], and YugabyteDB [15]. All databases are configured to be SI. TiDB runs on three Google Cloud machines (each has two 2.25GHz AMD EPYC 7B12 vCPUs, 4GB RAM) with Debian 10. SQLServer runs on a machine (2.20GHz Intel Xeon CPU, 4GB RAM) and Ubuntu 20.04. YugabyteDB runs on a machine (2.20GHz Intel Xeon CPU, 4GB RAM) and Debian 10. We use 24 concurrent clients (threads) for all experiments, unless specified otherwise. We got similar checking results for all three databases, so we only report the results from TiDB.

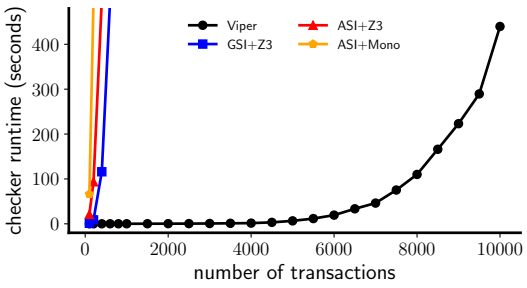

Figure 8. VIPER outperforms all baselines on BlindW-RW workloads. VIPER can handle >10× larger workloads than baselines.

We run VIPER and baselines on a machine with a 12-core processor (AMD Ryzen 9 5900, 3.0GHz), 64GB RAM, and 931 GB SSD. The OS is Ubuntu 20.04. We use Python version 3.8, Z3 solver version 4.8.14.0, and MonoSAT version 1.6.

7.1 Performance and scalability

Compared with natural baselines, using BlindW-RW. In the following experiments, we compare VIPER with GSI+Z3, ASI+Z3, ASI+Mono, and ASI+Mono+Opt on V-BlindW histories. We use V-BlindW because it does not have range queries (none of these baselines support range queries). In our setup, clients run V-BlindW and interact with a distributed TiDB deployed on three machines. History collectors record transactions and store them as files. Checkers (VIPER and baselines) load files, parse them as a history, and check if the history is SI. All histories generated are SI (all checkers accept).

We first compare VIPER with natural baselines that we built, GSI+Z3, ASI+Z3, ASI+Mono, and ASI+Mono+Opt. All baselines check Adya SI. Accordingly, we configure VIPER to check Adya SI (§3.2). All checkers run on BlindW-RW histories of various sizes, where the results are depicted in Figure 8.

VIPER outperforms the second best baseline (ASI+Mono) by 2900× times for a history of size 400 transactions, and VIPER can finish 10K transaction history in 439.7 seconds. VIPER runs faster for several reasons. First, VIPER uses BC-polygraphs which can leverage graph acyclicity primitives that are efficiently implemented by MonoSAT. Second, VIPER has a more succinct encoding: VIPER's SMT variables are about two-thirds of Z3+ASI, Mono+ASI and Mono+ASI+Opt. Finally, heuristic pruning helps reduce the number of constraints (see also §7.2). VIPER's super-linear runtime growth reflects the NP-Complete nature of the problem. ASI+Mono+Opt timed out for all the histories, hence is not plotted in the figure.

Compared with Elle, using an append-only benchmark. We compare VIPER with Elle's "sound mode" (we elaborate Elle's two modes in §8). In this mode, Elle needs to arrange the workloads such that the write order is manifested. We use an *append benchmark* from the Jepsen project [6]. In this benchmark, clients issue atomic append operations to some

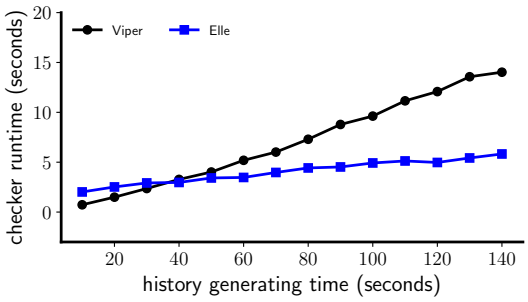

Figure 9. Runtime of VIPER and Elle for Jepsen's append benchmark. The x-axis is how long the benchmark runs; the longer it runs the larger the history it generates. The y-axis is the checking time for VIPER and Elle.

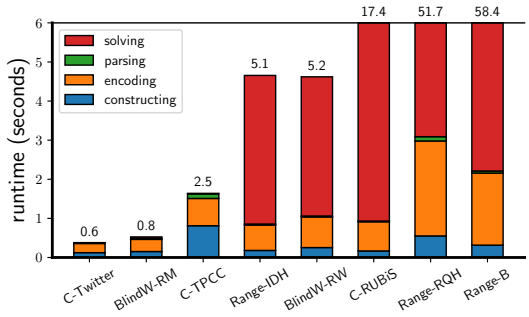

Figure 10. Decomposition of VIPER runtime for all benchmarks, 5K transactions. Phases on a bar (from bottom to top) are constructing, encoding, parsing, and solving.

keyed lists; and by reading the lists which contain all appended values, clients (and checkers) know the write order of appends. To consume Jepsen's logs, we update VIPER by translating the list of values into corresponding write orders and connecting the consecutive writes. Figure 9 shows the results. The x-axis is the history generating time because Jepsen testing framework uses generating time to control the size of a history. Histories grow linearly in time.

From Figure 9, both VIPER and VIPER scale linearly for checking this append benchmark. This is because write order has been revealed, thus checking SI is $O(n)$ (where n is the number of transactions) for both checkers. In particular, by having write order, the BC-polygraph will have no constraints (all write-dependencies and anti-dependencies are known), so it is a BC-graph (directed graph); and VIPER checks the acyclicity of this BC-graph. The performance difference in Figure 9 owes to the graph sizes (BC-graphs double the number of nodes in serialization graphs) and implementations (VIPER uses Python; Elle uses Clojure).

7.2 A closer look at VIPER

In this section, we look into VIPER to study how different components contribute to VIPER's runtime, and how optimizations vary VIPER's performance on all five benchmarks.

Decomposition of VIPER's runtime. We run VIPER on all five benchmarks with 5K transactions. We measure the wall clock running time of VIPER's, and break it down into four phases: *parsing* (parsing the logs from history collectors and summarizing transactions as a history), *constructing* (constructing the BC-polygraph from the history), *encoding* (encoding BC-polygraph into SMT clauses), and *solving* (running MonoSAT to find a solution). Figure 10 shows the results.

Figure 10 illustrates that VIPER behaves differently for different benchmarks, and different phases also cost differently. The parsing phase is relatively stable across benchmarks because all histories have 5K transactions, and the time to parse one transaction is relatively stable. Constructing and encoding time varies, which is mainly because benchmarks have a very different number of constraints. Finally, solving is often

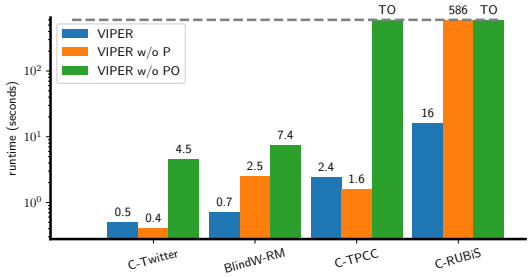

Figure 11. Ablation study of VIPER optimizations. The y-axis is in log-scale and the actual runtime is shown on top of bars. "TO" indicates timeout which is 600 seconds.

the most expensive phase. This is expected given the underlying problem is NP-Complete and the solver essentially does the heavy lifting for solving the problem. One outlier is C-TPCC: it doesn't have solving time. This is because all write transactions are read-modify-write, and the optimization—combining writes [83, §3.1]—will automatically inference the write-dependencies, hence there are no constraints.

Ablation study. VIPER checks Adya SI using MonoSAT that natively supports graph primitives. The baseline ASI+Mono also checks Adya SI using MonoSAT. Compared with ASI+Mono, VIPER has three improvements: (i) encoding histories as BC-polygraphs (§3.1), (ii) applying two optimizations from Cobra (§6), and (iii) using heuristic pruning (§3.5).

In this experiment, we study how these three improvements affect the overall checking time. In particular, we test three variants of VIPER: (1) VIPER itself, (2) VIPER without heuristic pruning (call it "VIPER w/o P"), and (3) VIPER without both heuristic pruning and Cobra's two optimizations (call it "VIPER w/o PO"). We run the three VIPER variants on 5K histories of all benchmarks. Results are shown in Figure 11.

The message we get from Figure 11 is that optimizations can be very effective for some benchmarks but have no effect on others. In other words, there is no "one-optimization-fits-all". For C-Twitter, the VIPER w/o P works well, so heuristic pruning does not stand out. For BlindW-RM, every optimization

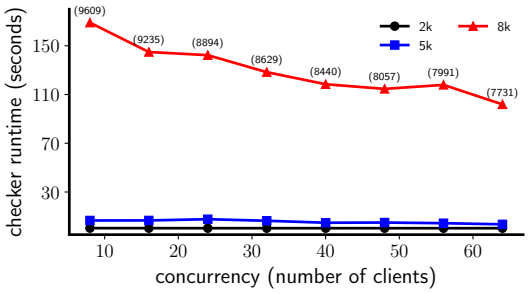

Figure 12. VIPER runtime for different concurrency and history sizes. Each line represents a history size. The numbers in parentheses are the number of constraints in the BC-polygraphs of 8K histories.

#Txns	Trial	Z3-GSI	Z3-GSI-OPT	Z3-ASI	Z3-ASI-OPT
100	1	0.95	0.91	317.56	133.01
	2	1.09	1.15	255.57	50.17
	3	0.94	1.14	21.41	20.72
200	1	10.30	7.89	119.76	104.32
	2	6.03	9.35	91.07	92.62
	3	9.25	11.15	94.14	99.89
400	1	397.48	393.41	TO	TO
	2	413.78	491.17	TO	TO
	3	TO	TO	TO	TO

Figure 13. Apply the heuristic optimization to two Z3 baselines and their runtime in seconds.

contributes meaningfully. For C-TPCC, the Cobra's optimization, combining writes, reduces constraints because TPCC has many read-modify-writes; whereas heuristic pruning even adds some overhead because of topological sorting. Finally, C-RUBiS demonstrates that heuristic pruning is vital.

We also run two baselines, ASI+Mono and ASI+Mono+Opt, on all the above histories. Note that the baseline ASI+Mono is equivalent to VIPER without all of the three improvements. And ASI+Mono+Opt is ASI+Mono by applying Cobra's two optimizations. They both timed out (>10min) in all cases, which shows BC-polygraphs is essential for VIPER.

Client-side concurrency. All prior experiments use 24 concurrent clients (threads). To see how client-side concurrency might affect VIPER's performance, we experiment with BlindW-RW by varying the number of clients (threads) from 8 to 64. We tested three sizes of histories: 2K, 5K and 8K transactions. Note that due to the dynamic of concurrent transaction executions, the history sizes are not exactly the same but we make sure the differences are within 5%. Figure 12 shows the results.

VIPER's performance stays almost the same for 2K and 5K histories. For 8K histories, the runtime decreases as the concurrency increases. This is not because VIPER works better for higher concurrency. It is because concurrency increases contention, which results in higher abort rates and fewer committed transactions per unit of time, which further decreases the number of constraints in BC-polygraphs.

SI violation	Database	#Txns	Time
Lost updates	MongoDB 4.2.6	23.2K	9.75s
Aborted read	MongoDB 4.2.6	2.2K	0.97s
G1c: cyclic information flow	MongoDB 4.2.6	1.1K	0.52s
Read Your Future Writes	MongoDB 4.2.6	4.6K	2.21s
Read Skew	TiDB 2.1.7	9.3K	4.28s

Figure 14. VIPER detects all real-world SI violations. "SI violation" describes the phenomenon that clients see. "#Txns" is the number of transactions in these violation histories. "Time" is how long it takes VIPER to reject.

Heuristic pruning. Heuristic pruning is a general optimization method that is not specific to VIPER; it could be applied to other checkers. We apply the heuristic pruning to two Z3 baselines—Z3+GSI and Z3+ASI—and get two optimized checkers: Z3+GSI+OPT and Z3+ASI+OPT. We experiment with BlindW-RW of sizes of 100, 200, and 400 transactions. We find that the performance of the four checkers fluctuates significantly, so we run three trials for each history size. Figure 13 depicts the results.

It turns out that the heuristic pruning only improves Z3-ASI with 100-transaction histories. For other histories, the optimization almost has no effect. We find two reasons why heuristic pruning does not help. First, the histories are tiny and heuristic pruning works with a small k, hence the number of constraints removed is minor. Second, these checkers produce a large number of constraints and iterating them (to check if can be pruned) outweighs the benefits.

7.3 Checking SI violations

All experiments so far are satisfiable cases that check SI histories. It is unclear whether VIPER finishes in a reasonable time for unsatisfiable cases, where histories violate SI.

Checking real-world SI violations. Next, we experiment with five real-world histories that have known SI violations. We download these histories from Jepsen's bug reports, and they cover databases of MongoDB and TiDB. Though we did not test MongoDB in our performance experiments, VIPER can check MongoDB's histories. Figure 14 shows the results. VIPER detects all SI violations and finishes within 10 seconds.

Checking synthetic anomalies. To see how VIPER performs on different anomalies, we manually inject three types of anomalies into two histories of different sizes generated by BlindW-RW. The three anomalies are: (1) *G1c anomaly*, a cycle with three read-dependency edges in SSG; (2) *long-fork anomaly*, the long-fork example described in section 3.1; and (3) *G-SIb anomaly*, a cycle of exactly one anti-dependency edge in SSG. Note that we only insert one anomaly into each history, so it is pessimistic to checkers because bugs usually trigger many anomalies. Figure 15 shows the results.

Both Elle and VIPER can detect G1c, and VIPER finishes faster. Elle fails to detect G-SIb and long-fork because dislike the append-only benchmark (§7.1), BlindW-RW doesn't reveal

#Txns	SI anomaly	Elle	VIPER
2K	G1c	2.02 (reject)	0.10 (reject)
	long-fork	2.02 (accept)	0.12 (reject)
	G-SIb	1.87 (accept)	3.61–52.91 (reject)
5K	G1c	2.26 (reject)	0.26 (reject)
	long-fork	2.42 (accept)	0.43 (reject)
	G-SIb	2.42 (accept)	40.55–TO (reject)

Figure 15. VIPER rejects all anomalies.

write-write orders. Hence, Elle uses heuristics to infer write-dependencies and anti-dependencies, which is not sound.

We also observe that VIPER has highly unstable performance for non-SI histories: it takes VIPER 3.61–52.91 seconds to detect G-SIb in a 2K history (we run VIPER five times on the same history); for the five runs on G-SIb 5K, VIPER finishes in 40.55, 183.07, 319.32 seconds, and times out twice (>20min). The internal non-determinism of the SMT solver is the major reason why VIPER performance varies for the same history. A mitigation is to fork multiple VIPER instances with different random seeds. It is our future work to study how to provide stable and predictable performance for non-SI histories.

8 Related work

Black-box checking SI. There are a few checkers [30, 63, 73] that check SI, and they target different setups. We briefed them in section 2.3. In particular, two checkers—dbcop [30] and Elle [63]—apply to black-box databases, which are the closest to VIPER. dbcop checks multiple isolation levels. SI is one of them. In fact, dbcop checks Strong Session SI (instead of Adya SI) because it requires transactions to obey session order. Compared with dbcop, VIPER handles larger histories because of using BC-polygraphs and VIPER's optimizations.

Elle is a versatile checker that checks safety constraints (including SI) for distributed systems, and it works for many setups. Elle usually works with the Jepsen project [6], an impactful testing framework that has discovered many bugs and safety violations in real-world production systems [5]. Regarding SI, Elle checks Adya SI and has two modes. One requires to engineer the workloads such that they manifest write order for keys; for example, by only using "append" operations for updates, all versions and their write order will be kept in the value, and clients can then see them. VIPER has the same scalability but worse performance than Elle in this benchmark (§7.1).

The other mode of Elle works on normal reads and writes for black-box databases, the same setup as VIPER. Because write order is unknown, Elle uses heuristics to infer the order [3]. But, because not all write ordering information can be inferred, Elle is not sound (may accept non-SI history) for checking SI in this mode. Compared with Elle (this mode), VIPER is sound.

SI definitions. There are multiple SI definition frameworks [16, 26, 37, 41] (as distinct from SI variants), which result in different SI-checking algorithms and SAT/SMT-based SI encoding (§2.3). We briefed SI definitions in section 2.2. Below we discuss those related to VIPER.

In the early days, people define SI by describing the rules of SI. For example, when Berenson et al. [26] first define SI, they describe rules like "first-committer-wins": if two concurrent transactions write the same key, the first committed transaction wins, and the other has to abort. Similar rule-based definitions appear for SI variants too, like Generalized SI [49]. VIPER makes no assumptions about databases and treats them as block boxes, so VIPER works for all SI implementations.

Later, Adya [16, 17] gives a graph-based SI definition (§2.2). The definition is based on a serialization graph and checks if there exist cycles with a certain combination of dependency edges. Compared with checking SI using Adya SI (ASI+Z3, ASI+Mono, and ASI+Mono+Opt, §7), VIPER has two major differences. First, VIPER introduces BC-polygraphs which "natively" capture multiple possibilities (by using constraints) and apply to checking black-box databases. Second, checking SI by BC-polygraphs is much simpler: VIPER only checks acyclicity, instead of conditional cycles with certain edges; this results in much simpler SMT encoding, hence faster solving.

Crooks et al. [41, 42] give the first client-centric SI definition, the same setup as VIPER. They also provide a clean hierarchy of SI variants that VIPER uses (§5). One interesting future work is to extend VIPER with the client-centric definitions since these definitions internalize the black-box setting in the first place. The technical challenge is that their definitions ask for an "existence" of certain execution which is likely to trigger an expensive witness search.

Another line of work that helps define SI is to clarify the gap between SI and serializability [28, 74]. In 2005, Fekete et al. [51] proved that if a history is SI but is not serializable, then the history must have cycles in its *serialization graph* and all these cycles must have a certain pattern—having two consecutive *anti-dependency* edges. In 2016, Cerone and Gotsman [38] proved that the above claim of "if-then" in fact is "if-and-only-if".

There are other efforts on defining SI and clarifying SI in various environments and setups. For example, a line of recent research [60, 61, 87, 88] proposes operation-based semantics for SI (and other isolation levels), which is easier to integrate with program logic. Therefore, such an operation-based SI definition helps check the overall correctness guarantees of the program plus the storage. People also extend Adya SI to mixed serialization graph [27], clarify SI in replicated setup [67], and try to unify isolation levels with cache coherence and memory consistency models [82]. VIPER's simpler SI definition (Theorem 4 and §3.4) is one attempt among them.

SI variants. The are many variants of SI. The ones that VIPER supports are Adya SI [16, 17], Generalized SI [49] (≡ ANSI SI [26]), Strong session SI [43] (≡ Prefix-Consistent SI [49]), and Strong SI [43]. Of course, many other SI variants exist, including Parallel SI (PSI) [80], Write-SI [89], Non-Monotonic

SI (NMSI) [20], Clock-SI [45], Posterior SI (PostSI) [92], and Speculative SI (SPSI) [66]. Some have their own isolation guarantees. As an example, PSI is a weaker isolation level than Adya SI; it trades off consistency for performance, especially for those geo-replicated databases. Other "SI variants", though named after SI, may provide stronger isolation guarantees than SI. For example, Write-SI provides serializability [89]. The obscure guarantees (and confusing names) of SI variants motivate tools like VIPER to distinguish what isolation levels the database users actually get.

Check other isolation levels. SI is one of many isolation levels (the *I* in ACID transactional systems). For example, serializability is the strongest isolation level that sometimes is regarded as the gold-standard isolation level [21, 50]. Black-box checking different isolation levels and consistency models attract much attention recently [18, 55, 56, 68, 75, 84, 91], especially along with the booming of cloud storage.

The complexities of checking different isolation levels are different. We have long known that black-box checking serializability is NP-Complete since 1979 [74]. However, it is much recent that people prove the complexity of black-box checking Causal Consistency (in 2017 [33]) and black-box checking SI (in 2019 [31]); both are NP-Complete. Other isolation levels, Read Committed and Read Atomic, are known to be polynomial-time checkable [31].

Among all isolation levels, checking serializability [28, 74, 90] has been studied intensively. There are many black-box checkers, including Cobra [83], Gretchen [4], Sinha et al. [79], dbcop [30, 31], CobraSAT [2]. Beyond black-box checkers (which fight with NP-Completeness), there are checkers that recover internal information. Examples are Elle [63] (described earlier), and Emme [40] which recovers value version order by studying the tested database's concurrency control protocol and providing version order functions.

There is another body of work to check non-transactional databases (like NoSQL) and memory consistency models [32, 46, 54, 86] for properties like linearizability [57] and sequential consistency [64]. These approaches have remote connections to checking SI in how to determine concurrent transactions (their operations).

Testing consistency. In most of the paper, when we say "checking", we mean an examination that is sound and complete, sometimes called "verifying". Of course, there are scenarios where people do not care about soundness, or completeness, or both. For example, for an eventually consistent storage system, people might want to experiment how frequently the storage returns stale values, and how old the returned values could be. We call these "testing".

There is a line of work to test what consistency users can get from cloud storage systems [18, 19, 55, 62, 68, 75, 84, 91]. In addition, for complex and hybrid systems, developers also test their systems to understand the consistency provided under different workloads and environments [70].

SI databases. Many production databases support SI, including Oracle, MongoDB [77], TiDB [11, 58], SQLServer [10], and YugabyteDB [15]. However, in most cases, users do not know which SI variants the databases offer. Sometimes, the same database provides different SI guarantees for different configurations [73]. This motivates tools like VIPER to distinguish which SI variants databases truly provide.

9 Discussion, future work, and conclusion

Checking other isolation levels. VIPER is designed for checking SI and can be extended to check isolation levels that are stricter than Adya SI (§2.2), for example, Strong SI, Strong Session SI, and Serializability. VIPER however cannot support weaker isolation levels than Adya SI, for example, PSI [80]. This is because PSI allows different clients to observe different commit orders but the core of VIPER—BC-polygraph and BC-graph acyclicity—enforces a single commit order. Even weaker isolation levels, like Read Committed, is easy to check [31] and do not need VIPER or BC-polygraphs.

VIPER as a test case generator. Beyond checking black-box databases, VIPER also has the potential to generate SI test cases for grey box testing. For example, by getting the testing transactions, VIPER can randomly produce both SI and non-SI histories based on BC-graphs. Then, we can test these histories on the targeted database and check their results. The technical challenge to deploy such a testing system will be enforcing the scheduling of histories on the tested database because different databases have different concurrency control mechanisms and some mechanisms may reject valid SI histories.

VIPER as a bug detector. We spend some effort testing commercial databases with VIPER, including TiDB [58], SQLServer [10], and YugabyteDB [15]. Though we found some gap between real-world implementations and SI definition [9], we didn't find critical SI bugs or violations. It wasn't surprising because we test these databases using normal benchmarks under normal environments (without failures). To increase the chance of finding bugs, our future work is to design challenging workloads and inject failures (e.g., disk failures, network partitions) into the environment.

Conclusion. SI databases are widely deployed and people use them every day. However, it is surprising how little people know about checking SI: the problem of black-box checking SI was proved to be NP-Complete in 2019 [31], and only a few SI checkers exist. This paper presents BC-polygraphs which are designed for checking SI. Based on BC-polygraphs, we build VIPER, a fast SI checker. In addition, a bonus from these is an SI definition that has a perfect parallel to serializability, which, in our opinion, should be how we teach SI.

Acknowledgement

We thank our shepherd Natacha Crooks and the anonymous reviewers of EuroSys23 for their feedback which substantially improved this paper. Mu's group is supported in part by NSF CNS 2130590 and 2214980.

References

[1] Cobra bench. https://github.com/DBCobra/CobraBench.

[2] Cobrasat. https://github.com/naizhengtan/CobraSAT.

[3] Elle heuristic. https://github.com/jepsen-io/elle/blob/main/src/elle/rw_register.clj.

[4] Gretchen: Offline serializability verification, in clojure. https://github.com/aphyr/gretchen.

[5] Jepsen: Analyses. https://jepsen.io/analyses.

[6] Jepsen: Distributed systems safety research. https://jepsen.io/.

[7] Jepsen: Snapshot isolatin. https://jepsen.io/consistency/models/snapshot-isolation.

[8] Ntp: The network time protocol. http://www.ntp.org/.

[9] A phantom read under snapshot isolation. https://github.com/pingcap/tidb/issues/32142.

[10] Snapshot isolation in sql server. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server.

[11] Tidb transaction isolation levels. https://docs.pingcap.com/tidb/dev/transaction-isolation-levels.

[12] TPC-C. http://www.tpc.org/tpcc/.

[13] Viper: A fast snapshot isolation checker (extended version). https://naizhengtan.github.io/project/viper.

[14] The Yices SMT solver. http://yices.csl.sri.com/.

[15] Yugabytedb: Transaction isolation levels. https://docs.yugabyte.com/preview/architecture/transactions/isolation-levels/.

[16] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[17] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering*, 2000.

[18] A. S. Aiyer, E. Anderson, X. Li, M. A. Shah, and J. J. Wylie. Consistability: Describing usually consistent systems. In *Proc. HotDep*, Dec. 2008.

[19] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. What consistency does your key-value store *actually* provide? In *Proc. HotDep*, Oct. 2010. Full version: Technical Report HPL-2010-98, Hewlett-Packard Laboratories, 2010.

[20] M. S. Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 163–172. IEEE, 2013.

[21] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: virtues and limitations. *PVLDB*, Sept. 2014.

[22] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. Flodb: Unlocking memory in persistent key-value stores. In *Proc. EuroSys*, Apr. 2017.

[23] T. Balyo, M. J. Heule, and M. Jarvisalo. SAT competition 2016: Recent developments. In *Proc. AAAI*, Feb. 2017.

[24] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. CAV*, July 2011.

[25] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. In *Proc. AAAI*, Jan. 2015.

[26] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD*, May 1995.

[27] J. M. Bernabé-Gisbert and F. D. Munoz-Escoı. A compoundable specification of the snapshot isolation level. Technical report, Technical Report ITI-SIDI-2012/007, Instituto Tecnológico de Informática, 2012.

[28] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *TSE*, SE-5(3), May 1979.

[29] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[30] R. Biswas and C. Enea. dbcop source code. https://gitlab.math.univ-paris-diderot.fr/ranadeep/dbcop.

[31] R. Biswas and C. Enea. On the complexity of checking transactional consistency. In *Proc. OOPSLA*, Oct. 2019.

[32] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2-3), Sept. 1994.

[33] A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 626–638, 2017.

[34] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *Proc. POPL*, Jan. 2017.

[35] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Static serializability analysis for causal consistency. In *Proc. PLDI*, 2018.

[36] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, Dec. 2008.

[37] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015)*, Sept. 2015.

[38] A. Cerone and A. Gotsman. Analysing snapshot isolation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 55–64, 2016.

[39] A. Cerone, A. Gotsman, and H. Yang. Algebraic laws for weak consistency. In *28th International Conference on Concurrency Theory (CONCUR 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[40] J. Clark. Verifying serializability protocols with version order recovery. Master's thesis, ETH Zurich, 2021.

[41] N. Crooks. *A client-centric approach to transactional datastores*. PhD thesis, The University of Texas at Austin, 2020.

[42] N. Crooks, Y. Pu, L. Alvisi, and A. Clement. Seeing is believing: a client-centric specification of database isolation. In *Proc. PODC*, July 2017.

[43] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 715–726, 2006.

[44] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, Mar. 2008.

[45] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 173–184. IEEE, 2013.

[46] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *Proc. TCC*, Mar. 2009.

[47] N. Eén and N. Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*. Springer, 2003.

[48] J. C. Eidson, M. Fischer, and J. White. Ieee-1588™ standard for a precision clock synchronization protocol for networked measurement and control systems. In *Proceedings of the 34th Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 243–254, 2002.

[49] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, 2005.

[50] A. Fekete, S. N. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. *Proceedings of the VLDB Endowment*, 2(1):467–478, 2009.

[51] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[52] M. Gebser, T. Janhunen, and J. Rintanen. Answer set programming as SAT modulo acyclicity. In *Proc. ECAI*, 2014.

[53] M. Gebser, T. Janhunen, and J. Rintanen. SAT modulo graphs: acyclicity. In *Proc. JELIA*, 2014.

[54] P. B. Gibbons and E. Korach. Testing shared memories. *SIJC*, 26(4), Aug. 1997.

[55] W. Golab, X. Li, and M. Shah. Analyzing consistency properties for fun and profit. In *Proc. PODC*, June 2011.

[56] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *Proc. ICDCS*, June 2014.

[57] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), July 1990.

[58] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[59] M. Janota, R. Grigore, and V. M. Manquinho. On the quest for an acyclic graph. *CoRR*, abs/1708.01745, Aug. 2017.

[60] G. Kaki. *Automatic Reasoning Techniques for Non-Serializable Data-Intensive Applications*. PhD thesis, Purdue University Graduate School, 2019.

[61] G. Kaki, K. Nagar, M. Najafzadeh, and S. Jagannathan. Alone together: compositional reasoning and inference for weak isolation. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[62] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Proc. S&P*, May 2015.

[63] K. Kingsbury and P. Alvaro. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554*, 2020.

[64] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *TC*, C-28(9), Sept. 1979.

[65] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proc. SOSP*, Oct. 2019.

[66] Z. Li, P. Van Roy, and P. Romano. Speculative transaction processing in geo-replicated data stores. Technical report, Technical Report 2, INESC-ID, 2017.

[67] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. E. Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Transactions on Database Systems (TODS)*, 34(2):1–49, 2009.

[68] Q. Liu, G. Wang, and J. Wu. Consistency as a service: Auditing cloud consistency. *TNSM*, 11(1), Mar. 2014.

[69] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. NSDI*, Apr. 2013.

[70] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *Proc. SOSP*, Oct. 2015.

[71] Y. Matsunobu, S. Dong, and H. Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *PVLDB*, 13(12), 2020.

[72] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*, June 2001.

[73] H. Ouyang, H. Wei, Y. Huang, H. Li, and A. Pan. Verifying transactional consistency of mongodb. *arXiv preprint arXiv:2111.14946*, 2021.

[74] C. H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4), Oct. 1979.

[75] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie. Toward a principled framework for benchmarking consistency. In *Proc. HotDep*, Oct. 2012.

[76] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A tunable delete-aware lsm engine. In *Proc. SIGMOD*, June 2020.

[77] W. Schultz, T. Avitabile, and A. Cabral. Tunable consistency in mongodb. *Proceedings of the VLDB Endowment*, 12(12):2071–2081, 2019.

[78] W. Shen, A. Khanna, S. Angel, S. Sen, and S. Mu. Rolis: a software approach to efficiently replicating multi-core transactions. In *Proc. EuroSys*, Apr. 2022.

[79] A. Sinha, S. Malik, C. Wang, and A. Gupta. Predicting serializability violations: SMT-based search vs. DPOR-based search. In *Haifa Verification Conference*, 2011.

[80] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. SOSP*, Oct. 2011.

[81] A. Stump, C. W. Barrett, and D. L. Dill. CVC: a cooperating validity checker. In *Proc. CAV*, July 2002.

[82] A. Szekeres and I. Zhang. Making consistency more consistent: A unified model for coherence, consistency and isolation. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, pages 1–8, 2018.

[83] C. Tan, C. Zhao, S. Mu, and M. Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[84] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *Proc. CIDR*, Jan. 2011.

[85] T. Warszawski and P. Bailis. ACIDRain: Concurrency-related attacks on database-backed web applications. In *Proc. SIGMOD*, May 2017.

[86] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *JPDC*, 17(1-2), Jan. 1993.

[87] S. Xiong. *Parametric Operational Semantics for Consistency Models*. PhD thesis, Imperial College London, 2019.

[88] S. Xiong, A. Cerone, A. Raad, and P. Gardner. Data consistency in transactional storage systems: a centralised approach. In *Proc. ECOOP*, 2020.

[89] M. Yabandeh and D. Gómez Ferro. A critique of snapshot isolation. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 155–168, 2012.

[90] M. Yannakakis. Serializability by locking. *JACM*, 2, Apr. 1984.

[91] K. Zellag and B. Kemme. How consistent is your cloud application? In *Proc. SoCC*, Oct. 2012.

[92] X. Zhou, X. Zhou, Z. Yu, and K.-L. Tan. Posterior snapshot isolation. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 797–808. IEEE, 2017.

A Artifact Appendix

A.1 Abstract

The artifacts contain two parts: a VIPER checker and VIPER clients. The VIPER checker determines whether a given history is SI. The VIPER clients are responsible for generating transactions, distributing transactions to black-box databases and collecting their responses to form histories.

A.2 Description & Requirements

A.2.1 How to access. VIPER's artifacts are available at https://github.com/Khoury-srg/Viper. Instructions are provided in the `Readme.md` for preparing the running environment and reproducing the major results. A Docker image is also provided with all the required dependencies of VIPER.

A.2.2 Hardware dependencies. VIPER does not have specific hardware requirements. All our experiments were run on a desktop with a 12-core processor (AMD Ryzen 9 5900, 3.0GHz), 64GB RAM, and 931 GB SSD.

A.2.3 Software dependencies.

- Ubuntu 20.04
- python3.8, python3-setuptools python3-pip
- jdk11
- cmake
- g++
- libgmp
- zlib

A.2.4 Benchmarks. VIPER's benchmark repository is here: https://github.com/Khoury-srg/ViperBench.

We use five benchmarks, two microbenchmarks (V-BlindW and V-Range) and three macrobenchmarks (C-TPCC, C-RUBiS, and C-Twitter). The three macrobenchmarks are borrowed from Cobra Bench [1]: We tailor the BlindW [1] to suit our setup, and call it V-BlindW. We also extend BlindW with range queries as a new benchmark, V-Range.

A.3 Set-up

Please follow the instructions in the `Readme.md` in the VIPER repository. The provided `Dockerfile` is recommended for easier setup.

A.4 Evaluation workflow

A.4.1 Major Claims.

- VIPER outperforms all the natural baselines and can finish checking large history(>5K) in a reasonable time while all the natural baselines time out. This is proved by experiment E1 (see descriptions of experiments E1–8 in §A.4.1).
- The performance difference of VIPER and Elle is not fundamental. This is proven by experiment E2.
- VIPER behaves differently for different benchmarks, and different phases also cost differently. The parsing phase is relatively stable across benchmarks and solving is often the most expensive phase. This is proven by experiment E3.
- Optimizations can be very effective for some benchmarks but have no effect on others. This is proven by experiment E4.
- As the concurrency increases, VIPER's runtime stays almost the same for small histories and may decrease for large histories. This is proven by experiment E5.
- The heuristic optimization does not work for the baselines for most histories. This is proven by experiment E6.
- VIPER can successfully detect real-world SI violations and finishes fast. This is proven by experiment E7.
- VIPER can successfully detect synthetic SI violations while Elle can only detect some of them. This is proven by experiment E8.

A.4.2 Experiments. You can use the bash scripts in the `src/ae` folder of the VIPER repo (https://github.com/Khoury-srg/Viper) to reproduce the results. Each script corresponds to a figure in the paper.

Experiment (E1): [Figure 8] [10 human-minutes + 3 compute-hours]: This experiment compares VIPER's performance with natural baselines, including *GSI+Z3*, *ASI+Z3*, *ASI+Mono* and *ASI+Mono+Opt*. The expected result is VIPER outperforms all the natural baselines.
[How to]: In the VIPER home directory, call run_fig8.sh.

Experiment (E2): [Figure 9] [10 human-minutes + 10 compute-minutes]: This experiment compare VIPER with Elle. The expected result is that both of them scale linearly and hence the performance difference is not fundamental.
[How to]: Call run_fig9.sh.

Experiment (E3): [Figure 10] [10 human-minutes + 10 compute-minutes]: This experiment studies how different phases contribute to VIPER's runtime, and how various optimizations may vary VIPER's performance on all five benchmarks. The expected results are: different benchmarks, and different phases also cost differently. The parsing phase is relatively stable and the solving phase is often the most expensive one.
[How to]: Call run_fig10.sh.

Experiment (E4): [Figure 11] [10 human-minutes + 1 compute-hour]: This experiment studies how the three optimizations affect the overall checking time. The expected result is that optimizations can be very effective for some benchmarks but have no effect on others.
[How to]: Call run_fig11.sh.

Experiment (E5): [Figure 12] [5 human-minutes + 30 compute-minutes]: This experiment tests how the runtime

changes against concurrency. The expected result is that VIPER's runtime stays almost the same for 2K and 5K histories and decreases for 8K histories.
[How to]: Call run_fig12.sh.

Experiment (E6): [Figure 13] [10 human-minutes + 3 compute-hours]: This experiment tests how the heuristic optimization works with the baselines. The expected result is that it does not have improvements for most of the histories.
[How to]: Call run_fig13.sh.

Experiment (E7): [Figure 14] [10 human-minutes + 30 compute-minutes]: This experiment downloaded some snapshot isolation violations cases from Jepsen's public bug reports and tests VIPER's usefulness. The expected result is that VIPER can detect the violations fast.
[How to]: Call run_fig14.sh.

Experiment (E8): [Figure 15] [10 human-minutes + 5 compute-minutes]: This experiment manually injects some snapshot isolation violations into histories and test VIPER's usefulness. The expected result is that VIPER is able to detect all the violations while Elle can only detect some of them.
[How to]: Call run_fig15.sh.

Generating new histories. The above experiments run on the histories that we generated ahead of time. If you want to generate histories from benchmarks, please refer to the `README.md` in ViperBench (https://github.com/Khoury-srg/ViperBench) for instructions.