Cobra: Making Transactional Key-Value Stores Verifiably Serializable

Cheng Tan and Changgeng Zhao, *NYU;* Shuai Mu, *Stony Brook University;* Michael Walfish, *NYU*

https://www.usenix.org/conference/osdi20/presentation/tan

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Cobra: Making Transactional Key-Value Stores Verifiably Serializable

Cheng Tan, Changgeng Zhao, Shuai Mu*, and Michael Walfish

NYU Department of Computer Science, Courant Institute *Stony Brook University

Abstract. Today's cloud databases offer strong properties, including serializability, sometimes called the gold standard database correctness property. But cloud databases are complicated black boxes, running in a different administrative domain from their clients. Thus, clients might like to know whether the databases are meeting their contract. To that end, we introduce cobra; cobra applies to transactional key-value stores. It is the first system that combines (a) black-box checking, of (b) serializability, while (c) scaling to real-world online transactional processing workloads. The core technical challenge is that the underlying search problem is computationally expensive. Cobra tames that problem by starting with a suitable SMT solver. Cobra then introduces several new techniques, including a new encoding of the validity condition; hardware acceleration to prune inputs to the solver; and a transaction segmentation mechanism that enables scaling and garbage collection. Cobra imposes modest overhead on clients, improves over baselines by 10× in verification cost, and (unlike the baselines) supports *continuous* verification. Our artifact can handle 2000 transactions/sec, equivalent to 170M/day.

1 Introduction and motivation

A new class of cloud databases has emerged, including Amazon DynamoDB and Aurora [2, 4, 133], Azure CosmosDB [7], CockroachDB [9], YugaByte DB [36], and others [16, 17, 21, 22, 69]. Compared to earlier generations of NoSQL databases (such as Facebook Cassandra, Google Bigtable, and Amazon S3), members of the new class offer the same scalability, availability, replication, and geo-distribution but in addition offer *serializable* transactions [55, 110]: all transactions appear to execute in a single, sequential order.

Serializability is the gold-standard isolation level [48, 77], and the correctness contract that many applications and programmers implicitly assume: their code would be incorrect if the database provided a weaker contract [137]. Note that serializability encompasses weaker notions of correctness, like basic integrity: if a returned value does not read from a valid write, that will manifest as a non-serializable result. Serializability also implies that the database handles failures robustly: non-tolerated server failures, particularly in the case of a distributed database, are a potential source of non-serializable results.

However, a user of a cloud database can legitimately wonder whether the database in fact provides the promised contract. For one thing, users often have no visibility into a cloud database's implementation. In fact, even when the source code is available [9, 16, 17, 36], that does not necessarily yield visibility: if the database is hosted by someone else, you can't really be sure of its operation. Meanwhile, any internal corruption—as could happen from misconfiguration, operational error, compromise, or adversarial control at any layer of the execution stack—can cause a serializability violation. Beyond that, one need not adopt a paranoid stance ("the cloud as malicious adversary") to acknowledge that it is difficult, as a technical matter, to provide serializability *and* geo-distribution *and* geo-replication *and* high performance under various failures [40, 78, 147]. Doing so usually involves a consensus protocol that interacts with an atomic commit protocol [69, 96, 103]—a complex combination, and hence potentially bug-prone. Indeed, today's production systems have exhibited serializability violations [1, 18, 19, 25, 26] (see also §6.1).

This leads to our core question: *how can clients verify the serializability of a black-box database?* To be clear, related questions have been addressed before. The novelty in our problem is in combining three aspects:

(a) Black box, unmodified database. In our setting, the database does not "know" it's being checked; the input to the verification machinery will be only the inputs to, and outputs from, the database. This matches the cloud context (even when the database is open source, as noted above), and contrasts with work that checks for isolation or consistency anomalies by using "inside information" [62, 86, 109, 123, 130, 141, 143], for example, access to internal scheduling choices. Also, we target production workloads and standard key-value APIs (§2).

(b) Serializability. We focus on serializability, in contrast to weaker isolation levels. Serializability has a strict variant and a non-strict variant [56, 110]; in the former, the effective transaction order must be consistent with real time. We attend to both variants in this paper. However, the weight is on the non-strict variant, as it poses a more difficult computational problem; the strict variant is "easier" because the real-time constraint diminishes the space of potentially-valid execution schedules.

On the one hand, the majority of databases that offer serializability offer the strict variant. On the other hand, checking non-strict serializability is germane, for two reasons. First, some databases claim to provide the non-strict variant (in general [11], or under clock skew [35], or for read-only workloads [32]), while others don't specify the variant [3, 5]. Second, the strict case can degenerate to the non-strict case. Heavy concurrency, for example, means few real-time constraints, so the difficult computational problem re-enters. As a special case, clock drift causes otherwise ordered transactions to be concurrent (§3.5, §6.1).

(c) Scalability. This means, first, scaling to real-world online transactional processing workloads at reasonable cost. It also means incorporating mechanisms that enable a verifier to work incrementally and to keep up with an ever-growing history.

However, aspects (a) and (b) set up a challenge: checking black-box serializability has long been known to be NP-complete [54, 110]. Recent work of Biswas and Enea (BE) [59] lowered the complexity to polynomial time, under natural restrictions (which hold in our context); see also pioneering work by Sinha et al. [124] (§7). However, these two approaches don't meet our goal of scalability. For example, in BE, the number of clients appears in the exponent of the algorithm's running time (§6, §7) (e.g., 14 clients means the algorithm is $O(n^{14})$). Furthermore, even if there were a small number of clients, BE does not include mechanisms for handling a continuous and ever-growing history.

Despite the computational complexity, there is cause for hope: one of the remarkable developments in the field of formal verification has been the use of heuristics to "solve" problems whose general form is intractable. This owes to major advances in solvers (advanced SAT and SMT solvers) [49, 57, 64, 73, 84, 99, 107, 128], coupled with an explosion of computing power. Thus, our guiding intuition is that it ought to be possible to verify serializability in many real-world cases. This paper describes a system called COBRA, which starts from this intuition, and provides a solution to the problem posed by (a)–(c).

COBRA applies to transactional key-value stores (everywhere in this paper it says "database", this is what we mean). COBRA consists of a third-party, unmodified database that is not assumed to "cooperate"; a set of legacy database clients that COBRA modifies to link to a library; one or more *history collectors* that are assumed to record the actual requests to and responses from the database; and a *verifier* that comprehensively checks serializability, in a way that "keeps up" with the database's (average) load. The database is untrusted while the clients, collectors, and verifier are all in the same trust domain (for example, deployed by the same organization). Section 2 further details the setup and gives example scenarios. COBRA solves two main problems:

1. Efficient witness search (§3). A brute-force way to validate serializability is to demonstrate the existence of a graph G whose nodes are transactions in the history and whose edges meet certain *constraints*, one of which is acyclicity (§2.3). From our starting intuition and the structure of the constraints, we are motivated to use a SAT or SMT solver [34, 50, 73, 127]. But complications arise. To begin with, encoding acyclicity in a SAT instance brings overhead [79, 80, 91] (we see this too; §6.1). Instead, COBRA uses a recent SMT solver, MonoSAT [52], that is well-suited to checking graph properties (§3.4). However, using MonoSAT alone on the aforementioned brute-force search problem is still too expensive (§6.1).

To address this issue, COBRA develops domain-specific pruning techniques and reduces the search problem size. First, COBRA introduces a new encoding that exploits common patterns in real workloads, such as read-modify-write transactions, to efficiently infer ordering relationships from a history (§3.1–§3.2). (We prove that COBRA's encoding is a valid reduction in Appendix B [132].) Second, COBRA uses parallel hardware (our implementation uses GPUs; §5) to compute *all-pairs reachability* over a graph whose nodes are transactions and whose edges are known precedence relationships; then, COBRA resolves some of the constraints efficiently, by testing whether a candidate edge would generate a cycle with an existing path.

2. Scaling to a continuous and ever-growing history (§4). Online cloud databases run in a continuous fashion, where the corresponding history is uninterrupted and grows unboundedly. To support online databases, COBRA verifies in rounds. From round-to-round, the verifier checks serializability on a portion of the history. However, the challenge is that the verifier seemingly needs to involve all history, because serializability does not respect real-time ordering, so future transactions can read from values that (in a real-time view) have been overwritten. To solve this problem, clients issue periodic *fence transactions* (§4.2). The epochs impose coarse-grained synchronization, creating a window from which future reads, if they are to be serializable, are permitted to read. This allows the verifier to discard transactions prior to the window.

We implement COBRA (§5) and experiment with it on production databases with various workloads (§6). COBRA detects all serializability violations we collect from real systems' bug reports. COBRA's core (single-round) verification improves on baselines by 10× in the problem size it can handle for a given time budget. For example, COBRA finishes checking 10k transactions in 14 seconds, whereas baselines can handle only 1k or less in the same time budget. For an online database with continuous traffic, COBRA achieves a sustainable verification throughput of 2k txn/sec on the workloads that we experiment with (this corresponds to a workload of 170M/day; for comparison, Apple Pay handles 33M txn/day [6], and Visa handles 150M txn/day [33], admittedly for a slightly different notion of "transaction"). COBRA imposes modest overhead.

COBRA has several limitations (§8). First, there is no guarantee that COBRA terminates in reasonable time (though our experiments on real workloads do). Second, COBRA supports only a key-value API, and thus does not handle range queries and SQL operations such as "join" and "sum" (though one can translate these queries and operations to a key-value API, as commonly done in research on transactional key-value stores [100, 108, 129, 136, 140, 144]). Third, COBRA does not yet support async (event-driven) I/O patterns in clients (only multithreading). Fourth, COBRA mostly punts fault-tolerance of the verifier and collectors (though modular solutions exist). Finally, we have not identified serializability violations in the wild. Of course, that does not mean that databases unfailingly execute correctly. Indeed, COBRA gives us a way, for the first time, to get confidence in the observed executions of these black box databases.

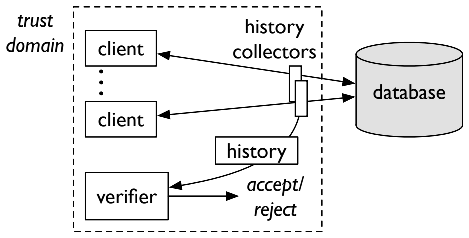


Figure 1: COBRA's architecture. The dashed rectangle is a trust domain. The verifier is off the critical path but must keep up on average.

2 Overview and technical background

2.1 Setup and scenarios

Figure 1 depicts COBRA's architecture. *Clients* issue requests to a *database* (a transactional key-value store) and receive results. The database is untrusted: the results can be arbitrary.

Each client request is one of five operations: start, commit, abort (which refer to *transactions*), and read and write (which refer to *keys*).

History collectors sit between clients and the database, capturing the requests that clients issue and the (possibly wrong) results delivered by the database. This capture is a *fragment of a history*. A *history* is a set of operations; it is the union of the fragments from all collectors.

A *verifier* retrieves history fragments from collectors and attempts to verify whether the history is *serializable*; we make this term precise below (§2.2).

The verifier proceeds in *rounds*; each round consists of a witness search, the input to which is logically the output of the previous round and new history fragments. The verifier must work against an online and continuously available database; however, the verifier performs its work in the background, off the critical path.

The verifier requires the full history including all requests to, and responses from, the database. COBRA assumes that neither the verifier nor the collectors crash (we revisit in §8).

Clients issue operations to a database through *sessions*; a client can have multiple simultaneous sessions. Within a session, transactions do not overlap (requests are blocking). Thus, a client can be multithreaded but not event-driven.

Clients, history collectors, and the verifier are in the same trust domain. This architecture is relevant in real-world scenarios. Consider for example an enterprise web application whose end-users are geo-distributed employees of the enterprise. The application servers run on the enterprise's hardware while the back-end of the web application is a cloud database [27]. Note that our *clients* are the application servers, as clients of the database. A similarly structured example is online gaming, where the main program runs on company-owned servers while the user data is stored in a cloud database [24].

In these scenarios, the verifier runs on hardware belonging to the trust domain. There are several options, meanwhile, for the collectors. Collectors can be middleboxes situated at the edge of the enterprise or gaming company, allowing them to capture the requests and responses between the database clients and the cloud database. Another option is to run the collector in an untrusted cloud, using a Trusted Execution Environment (TEE), such as Intel's SGX. Recent work has demonstrated such a collector [46], as a TLS proxy that logs inbound and outbound messages, thereby ensuring (via the fact that clients expect the server to present a valid SSL/TLS certificate) that clients' requests and responses are indeed logged.

Verifier's performance requirement. The verifier's performance will be reported as capacity (transactions/sec); this capacity must be at least the average offered load seen by the database over some long interval, for example a day. Note that the verifier's capacity need not match the database's *peak* load: because the verifier is off the critical path, it can catch up.

2.2 Verification problem statement

Preliminaries. We work within Adya's canonical framework for specifying isolation levels [38], as summarized below.

First, assume that each database write creates a unique *version* for the given key, and each transaction reads and writes a key at most once; thus, any read can be associated with the transaction that issued the corresponding write. COBRA discharges this assumption in the client library (§5), which embeds a unique id in each write and consumes the id on a read.

In Adya's formalism, a *history* is a set of operations performed by transactions (as in COBRA, §2.1), together with a *version order* [38, §3.1.2]: for each key, a total order of committed versions. The version order comes from within the database and is not exposed externally. In COBRA, history is collected outside the database so doesn't contain a version order. (COBRA's history is also known as a *multi-version log* [54], as discussed in Appendix B [132]).

A history is *serializable*, if there exists a total order on the committed transactions such that executing transactions in this order produces the same result as in the history (in Adya's formalism, an additional requirement for serializability is that the aforementioned total order is consistent with the given version order). *Strictly serializable* [110] is the same as serializable, except that the total order must also obey real time: if a transaction T_i commits before T_j starts in real time, T_i appears earlier than T_j in the total order.

A history imposes *dependencies*. Specifically, a history (without a version order) induces *read-dependencies* (a transaction T_j reads the value written by transaction T_i, denoted $T_i \to T_j$). Adding a version order yields two other kinds of dependencies: *write-dependency* (T_i writes a key, and T_j overwrites it, so $T_i \to T_j$), and *anti-dependency* (T_i reads a value that is overwritten by T_j, so $T_i \to T_j$).

A *serialization graph* (of a history and a given version order) is a graph whose vertices are all transactions in the history and edges are all dependencies described above. Note that aborted and ongoing transactions are not in the serialization graph.

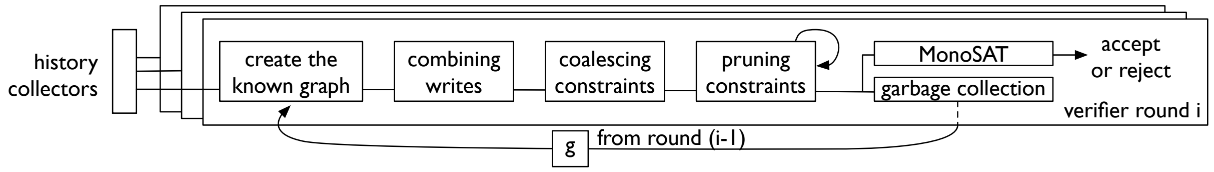


Figure 2: The verifier's process, within a round and across rounds.

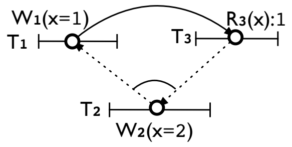
The core problem. An important fact is that a history H is serializable iff there exists a version order such that the serialization graph arising from H and that version order is acyclic [54]. Based on this fact, the core problem is to identify such a serialization graph (that arises from H and some version order), or assert that none exists.

Notice that this problem would be straightforward if the database revealed its internal ordering, thus deciding a version order and all dependencies: one could construct the corresponding serialization graph, and test it for acyclicity. Indeed, that is a well-established family of techniques [65, 138]. But the version order is unavailable in our context, so we have to consider all possible version orders, and test whether any of the implied sets of dependencies yields an acyclic serialization graph.

2.3 Starting point: intuition and brute force

This section describes a brute-force solution, which serves as the starting point for COBRA and gives intuition. The approach relies on a data structure called a *polygraph* [110], which captures unknown dependencies.

In a polygraph, vertices (V) are transactions and edges (E) are read-dependencies. Note that read-dependencies are evident from the history because values are unique, by assumption (§2.2). There is a set, C, which we call *constraints*, that captures possible (but unknown) dependencies. Here is an example polygraph:



It has three vertices $V = \{T_1, T_2, T_3\}$, one known edge in $E = \{(T_1, T_3)\}$ from the known read-dependency $W_1(x) \longrightarrow R_3(x)$, and one constraint $\langle\,(T_3, T_2),\ (T_2, T_1)\,\rangle$ which is shown as two dashed arrows connected by an arc. This constraint captures the fact that T_2 cannot happen in between T_1 and T_3, because otherwise T_3 should have read x from T_2 instead of from T_1. Hence T_2 has to happen either after T_3 or before T_1, but it is unknown which option is the truth.

Formally, a *polygraph* $P = (V,\ E,\ C)$ is a directed graph $(V,\ E)$ which we call the *known graph*, together with a set of *bipaths*, C; that is, pairs of edges—not necessarily in E—of the form $\langle (v,u),\ (u,w)\rangle$ such that $(w,v) \in E$. A bipath of that form can be read as "either u happened after v, or else u happened before w". Now, define *the polygraph* (V, E, C) *associated with a history*, as follows [138]:

- V are all committed transactions in the history
- $E = \{(T_i, T_j) \mid T_j$ reads from $T_i\}$. Notation: $T_i \xrightarrow{\text{wr(x)}} T_j$, for some x.
- $C = \{\langle\,(T_j, T_k),\ (T_k, T_i)\,\rangle \mid (T_i \xrightarrow{\text{wr(x)}} T_j) \wedge$ $(T_k$ writes to $x) \wedge T_k \neq T_i \wedge T_k \neq T_j\}$.

The edges in E capture all read-dependencies, which as noted are evident from the history. C captures how uncertainty is encoded into constraints. Specifically, for each read-dependency in the history, all other transactions that write the same key happen either after the given read or before the given write.

A directed graph is called *compatible* with a polygraph if the graph has the same nodes and known edges in the polygraph, and the graph chooses one edge out of each constraint; one can think of such a graph as a solution to the constraint satisfaction problem posed by the polygraph. Formally, a graph (V', E') is *compatible* with a polygraph (V, E, C) if: $V = V'$, $E \subseteq E'$, and $\forall \langle e_1, e_2\rangle \in C,\ (e_1 \in E' \wedge e_2 \notin E') \vee (e_1 \notin E' \wedge e_2 \in E')$.

A crucial fact (proved in Appendix B [132]) is: there exists an acyclic directed graph that is compatible with the polygraph associated to a history H, iff there exists an acyclic serialization graph G of H. Furthermore, we have seen that if there is such an acyclic serialization graph for H, then H is serializable (§2.2). Putting these facts together yields a brute-force approach for verifying serializability: first, construct a polygraph from a history; second, search for a compatible graph that is acyclic. However, not only does this approach need to consider $|C|$ binary choices ($2^{|C|}$ possibilities) but also $|C|$ is massive: it is a sum of quadratic terms, specifically $\sum_{k \in \mathcal{K}} r_k \cdot (w_k - 1)$, where $\mathcal{K}$ is the set of keys in the history, and each r_k and w_k are the number of reads and writes of key k.

3 Verifying serializability in COBRA

Figure 2 depicts the major components of verification. This section covers one round of verification. As a simplification, assume that the round runs in a vacuum; Section 4 discusses how rounds are linked.

COBRA uses the MonoSAT SMT solver [52], which is geared to graph properties (§3.4). Nevertheless, the brute-force encoding (§2.3) would overwhelm even MonoSAT (§6.1).

COBRA refines that encoding in several ways. It introduces *write combining* (§3.1) and *coalescing* (§3.2). These techniques are motivated by common patterns that impose restrictions on the search space. COBRA's verifier also does its own inference (§3.3) before invoking the solver. This is motivated by observing that (a) all-pairs reachability information (in the "known edges") yields quick resolution of many constraints,

```
procedure ConstructEncoding(history)
    g, readfrom, wwpairs ← CreateKnownGraph(history)
    con ← GenConstraints(g, readfrom, wwpairs)
    con, g ← Prune(con, g) // §3.3, executed one or more times
    return con, g

procedure CreateKnownGraph(history)
    g ← empty Graph                          // the known graph
    wwpairs ← Map {⟨Key,Tx⟩ → Tx}         // consecutive writes
    readfrom ← Map {⟨Key,Tx⟩ → Set⟨Tx⟩} // maps a write to its readers
    for transaction tx in history:
        g.Nodes += tx
        for read operation rop in tx:
            g.Edges += (rop.read_from_tx, tx)      // read-dependencies
            readfrom[⟨rop.key, rop.read_from_tx⟩] += tx

        // detect RMW (read-modify-write) transactions
        for all Keys key that are both read and written by tx:
            rop ← the operation in tx that reads key
            if wwpairs[⟨key, rop.read_from_tx⟩] ≠ null:
                REJECT      // multiple consecutive writes, not serializable
            wwpairs[⟨key, rop.read_from_tx⟩] ← tx

    add session order edges to g      // §4.2
    return g, readfrom, wwpairs

procedure GenConstraints(g, readfrom, wwpairs)
    // each key maps to set of chains; each chain is an ordered list
    chains ← empty Map {Key → Set⟨List⟩}
    for transaction tx in g:
        for write wrop in tx:
            chains[wrop.key] += [ tx ]      // one-element list

    CombineWrites(chains, wwpairs)      // §3.1
    InferRWEdges(chains, readfrom, g) // infer anti-dependency

    con ← empty Set
    for ⟨key, chainset⟩ in chains:
        for every pair {chain_i, chain_j} in chainset:
            con += Coalesce(chain_i, chain_j, key, readfrom) // §3.2

    return con

procedure CombineWrites(chains, wwpairs)
    for ⟨key, tx_1, tx_2⟩ in wwpairs:
        // By construction of wwpairs, tx_1 is the write immediately
        // preceding tx_2 on key. Thus, we can sequence all writes
        // prior to tx_1 before all writes after tx_2, as follows:
        chain_1 ← the list in chains[key] whose last elem is tx_1
        chain_2 ← the list in chains[key] whose first elem is tx_2
        chains[key] \= {chain_1, chain_2}
        chains[key] += concat(chain_1, chain_2)

procedure InferRWEdges(chains, readfrom, g)
    for ⟨key, chainset⟩ in chains:
        for chain in chainset:
            for i in [0, length(chain) − 2]:
                for rtx in readfrom[⟨key, chain[i]⟩]:
                    if (rtx ≠ chain[i+1]): g.Edges += (rtx, chain[i+1])

procedure Coalesce(chain_1, chain_2, key, readfrom)
    edge_set_1 ← GenChainToChainEdges(chain_1, chain_2, key, readfrom)
    edge_set_2 ← GenChainToChainEdges(chain_2, chain_1, key, readfrom)
    return ⟨edge_set_1, edge_set_2⟩

procedure GenChainToChainEdges(chain_i, chain_j, key, readfrom)
    if readfrom[⟨key, chain_i.tail⟩] = ∅:
        edge_set ← {(chain_i.tail, chain_j.head)}
        return edge_set

    edge_set ← empty Set
    for rtx in readfrom[⟨key, chain_i.tail⟩]:
        edge_set += (rtx, chain_j.head)
    return edge_set

procedure Prune(con, g)
    // tr is the transitive closure (reachability of every two nodes) of g
    tr ← TransitiveClosure(g)  // standard algorithm; see [70, Ch.25]
    for c = ⟨edge_set_1, edge_set_2⟩ in con:
        if ∃(tx_i, tx_j) ∈ edge_set_1 s.t. tx_j ⇝ tx_i in tr:
            g.Edges ← g.Edges ∪ edge_set_2
            con −= c
        else if ∃(tx_i, tx_j) ∈ edge_set_2 s.t. tx_j ⇝ tx_i in tr:
            g.Edges ← g.Edges ∪ edge_set_1
            con −= c
    return con, g
```

Figure 3: Cobra's procedure for converting a history into a constraint satisfaction problem (§3). After this procedure, cobra feeds the results (a graph of known edges G and set of constraints C) to a constraint solver (§3.4), which searches for a graph that includes the known edges from G, meets the constraints in C, and is acyclic. We prove the algorithm's validity in Appendix B [132].

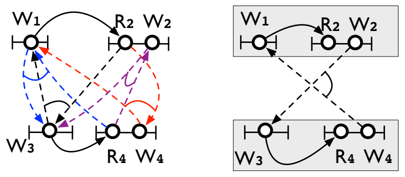
and (b) computing that information is amenable to acceleration on parallel hardware such as GPUs (§5).

Figure 3 depicts the algorithm that constructs cobra's encoding and shows how the techniques combine. Note that cobra relies on a generalized notion of constraints. Whereas previously a constraint was a pair of edges, now a constraint is a pair of *sets of edges*. Meeting a constraint $\langle A,B\rangle$ means including all edges in A and excluding all in B, or vice versa. More formally, we say that a graph (V',E') is *compatible* with a known graph $G=(V,E)$ and generalized constraints C if: $V=V'$, $E\subseteq E'$, and $\forall\langle A,B\rangle\in C, (A\subseteq E'\wedge B\cap E'=\emptyset)\vee(A\cap E'=\emptyset\wedge B\subseteq E')$.

We prove the validity of cobra's encoding in Appx B [132]. Specifically we prove that *there exists an acyclic graph that is compatible with the constraints constructed by* cobra *on a given history if and only if the history is serializable.*

3.1 Combining writes

Cobra exploits the read-modify-write (RMW) pattern, in which a transaction reads a key and then writes the same key. The pattern is common in real-world scenarios, for example shopping: in one transaction, get the number of an item in stock, decrement, and write back the number. Cobra uses RMWs to impose order on writes; this reduces the orderings that the verification procedure would otherwise have to consider. Here is an example:



There are four transactions, all operating on the same key. Two of the transactions are RMW, namely R_2, W_2 and R_4, W_4. On the left is the basic polygraph (§2.3). It has four constraints (each in a different color), derived from the two read-dependencies.

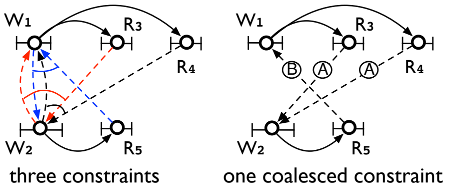
Cobra goes further, inferring *chains*. A single chain comprises a *sequence of transactions whose write operations are consecutive*; in the figure, a chain is indicated by a shaded area. Notice that the only ordering possibilities exist at the granularity of chains (rather than individual writes); in the example, the two possibilities of course are $[W_1, W_2] \to [W_3, W_4]$ and $[W_3, W_4] \to [W_1, W_2]$. This is a reduction in the possibility space; for instance, the original version considers the possibility that W_3 is immediately prior to W_1 (the upward dashed black arrow), but cobra "recognizes" the impossibility of that.

To construct chains, cobra initializes every write as a one-element chain (Figure 3, line 32). Then, cobra consolidates chains: for each RMW transaction t and the transaction t' that contains the prior write, cobra concatenates the chain containing t' and the chain containing t (lines 22 and 44–51).

If a transaction t, which is *not* an RMW, reads from a transaction u, then t requires an anti-dependency edge to u's successor (call it v); otherwise, t could appear in the graph downstream of v, which would mean t actually read from v (or even from a later write), which does not respect history. Cobra creates the needed edge $t \to v$ in InferRWEdges (Figure 3, line 53). Note that in the brute-force approach (§2.3), analogous edges appear as the first component in a constraint.

3.2 Coalescing constraints

This technique exploits the fact that, in many real-world workloads, there are far more reads than writes. At a high level, cobra combines all reads that read-from the same write. We give an example and then generalize.



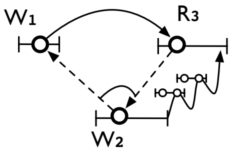
In the above figure, there are five single-operation transactions, to the same key. On the left is the basic polygraph (§2.3), which contains three constraints; each is in a different color. Notice that all three constraints involve the question: which write happened first, W_1 or W_2?

One can represent the possibilities as a constraint $\langle A', B' \rangle$ where $A' = \{(W_1, W_2), (R_3, W_2), (R_4, W_2)\}$ and $B' = \{(W_2, W_1), (R_5, W_1)\}$. In fact, cobra does not include (W_1, W_2) because there is a known edge (W_1, R_3), which, together with (R_3, W_2) in A', implies the ordering $W_1 \to R_3 \to W_2$, so there is no need to include (W_1, W_2). Likewise, cobra does not include (W_2, W_1) on the basis of the known edge (W_2, R_5). So cobra includes the constraint $\langle A, B \rangle = \langle \{(R_3, W_2), (R_4, W_2)\}, \{(R_5, W_1)\} \rangle$ in the figure.

To construct constraints using the above reductions, cobra does the following. Whereas the brute-force approach uses all reads and their prior writes (§2.3), cobra considers particular *pairs of writes*, and creates constraints from these writes and their following reads. The particular pairs of writes are the first and last writes from all pairs of chains pertaining to that key. In more detail, given two chains, $chain_i, chain_j$, cobra constructs a constraint c by (i) creating a set of edges ES_1 that point from reads of $chain_i$.tail to $chain_j$.head (Figure 3, lines 71–72); this is why cobra does not include the (W_1, W_2) edge above. If there are no such reads, ES_1 is $chain_i\text{.tail} \to chain_j\text{.head}$ (Figure 3, line 67); (ii) building another edge set ES_2 that is the other way around (reads of $chain_j$.tail point to $chain_i$.head, etc.), and (iii) setting c to be $\langle ES_1, ES_2 \rangle$ (Figure 3, line 63).

3.3 Pruning constraints

Our final technique mines information that is encoded in paths in the known graph, to cull irrelevant possibilities en masse. The underlying logic is almost trivial. The interesting aspect is that the technique is enabled by a design decision to accelerate the computation of reachability on parallel hardware (§5 and Figure 3, line 77); this can be done since the computation is iterated (Boolean) matrix multiplication. Here is an example:



The constraint is $\langle (R_3, W_2), (W_2, W_1) \rangle$. Having precomputed reachability, cobra knows that the first choice cannot hold, as it creates a cycle with the path $W_2 \rightsquigarrow R_3$; cobra thereby concludes that the second choice holds. Generalizing, if cobra determines that an edge in a constraint generates a cycle, cobra throws away both components of the entire constraint and adds all the other edges to the known graph (Figure 3, lines 78–84). In fact, cobra prunes multiple times, if necessary (§5).

3.4 Solving

The remaining step is to search for an acyclic graph that is compatible with the known graph and constraints, as computed in Figure 3. Cobra does this with a constraint solver. However, traditional solvers do not perform well on this task because encoding graph acyclicity as a set of SAT formulas is expensive (a claim by Janota et al. [91], which we also observed; §6.1).

Cobra instead uses MonoSAT, which is a particular kind of SMT solver [57] that includes SAT modulo *monotonic* theories [52]. This solver efficiently encodes and checks graph properties, such as acyclicity.

Cobra represents a verification problem instance (a graph G and constraints C) as follows. Cobra creates a Boolean variable $E_{(i,j)}$ for each vertex-vertex pair; True (resp., False) means the searched-for compatible graph has (resp., does not have) the given edge. For each edge in the known graph G, Cobra sets the corresponding Boolean variable to be True. For the constraints C, recall that each constraint $\langle A, B\rangle$ is a pair of sets of edges, and represents a mutually exclusive choice to include either all edges in A or else all edges in B. Cobra encodes this in the natural way: $((\forall e_a \in A, e_a) \wedge (\forall e_b \in B, \neg e_b)) \vee ((\forall e_a \in A, \neg e_a) \wedge (\forall e_b \in B, e_b))$. (By abuse of notation, we have used e_a and e_b to refer to the corresponding $E_{i,j}$ variable.) Finally, cobra enforces the acyclicity of the searched-for compatible graph (that is, the graph whose edges are given by the $E_{i,j}$ that are set to True); Cobra does so by invoking a primitive provided by the solver.

Cobra vs. MonoSAT. One might ask: if cobra's encoding makes MonoSAT faster, why use MonoSAT? Can we take the domain knowledge further? Indeed, in the limiting case, cobra could re-implement the solver! However, MonoSAT, as an SMT solver, leverages many prior optimizations. One way to understand cobra's decomposition of function is that cobra's preprocessing exploits some of the structure created by the problem of verifying serializability, whereas the solver is exploiting residual structure common to many graph problems.

3.5 On strict serializability

Cobra's verifier checks strict serializability [56, 110] by adding *real-order edges* [38]—which capture the order of non-overlapping transactions in real time—to the known graph. The verifier then performs the serializability checking algorithm of Figure 3; as a result, the serialization order (in the searched-for compatible graph) respects the real-time order.

To get real-order edges, the verifier needs *timestamps* for each operation. The verifier can get them either from the database if it exposes the relevant interface (for example, Google Spanner [69]) or else from cobra's collectors. A naive way to go from timestamps to real-order edges is to examine every pair of transactions, and create a real-order edge when one transaction's commit timestamp is less than another's start timestamp. But this approach runs in time quadratic in the number of transactions. Instead, cobra borrows a prior algorithm [131, Fig. 6], which materializes the time precedence partial order in time $O(n+z)$, where n is the number of transactions and z is the minimum number of real-order edges needed.

A challenge is that clock drift in collectors makes it unsafe to infer real-time precedence relationships from timestamps. To tackle this problem, cobra introduces a *clock drift threshold* (100ms [15] by default). Cobra assumes that clock differences among collectors do not exceed this threshold; if they do, cobra may falsely reject a serializable history. With this assumption, cobra increases transactions' commit timestamps by the threshold. Thus, if two transactions have a real-order edge, one's original commit timestamp is earlier than the other transaction's start timestamp by at least the clock drift threshold. As a consequence, all transactions within a clock drift threshold are concurrent. Within such an interval, the verifier faces the computational expense that exists when there are no real-order edges, which calls for cobra's techniques (§3.1–§3.3) to accelerate verification (see §6.1 for relevant experiments).

4 Garbage collection and scaling

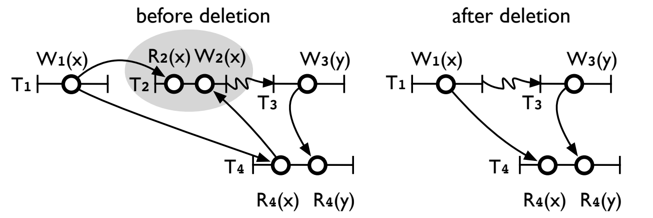
Cobra verifies in rounds. There are two motivations for rounds. First, new history is continually produced, of course. Second, there are limits on the maximum problem size (number of transactions) that the verifier can handle (§6.2); breaking the task into rounds keeps each solving task manageable.

In the first round, a verifier starts with nothing and creates a graph from CreateKnownGraph, then does verification. After that, the verifier receives more client histories; it reuses the graph from the last round (the g in ConstructEncoding, Figure 3, line 5), and adds new nodes and edges to it from the new history fragments received (Figure 2).

The technical problem is to keep the input to verification bounded. So the question cobra must answer is: which transactions can be deleted safely from history? Below, we describe the challenge (§4.1), the core mechanism of fence transactions (§4.2), and how the verifier deletes safely (§4.3). In this section, we describe the general rules and insights; a complete specification and correctness proof are in Appendix C [132].

4.1 The challenge

The core challenge is that past transactions can be relevant to future verifications; specifically, deleting a past transaction could cause the verifier to overlook future cycles.



Suppose a verifier saw three transactions (T_1, T_2, T_3) and wanted to remove T_2 (the shaded transaction) from consideration in future rounds. Later, the verifier observes a new transaction T_4 that violates serializability (and a fortiori, strict serializability) by reading from T_1 and T_3. To see the violation, notice that T_2 is logically subsequent to T_4, which generates a cycle ($T_4 \rightarrow T_2 \rightsquigarrow T_3 \rightarrow T_4$). Yet, if we remove T_2, there is no cycle. Hence, removing T_2 is not safe: future verifications would fail to detect certain kinds of serializability violations.

Note that this example does not require malicious or exotic behavior. For example, consider a geo-replicated database: a client can retrieve a stale version from a local replica.

4.2 Epochs and fence transactions

Cobra addresses this challenge by creating *epochs* that impose a coarse-grained ordering on transactions; the verifier can then discard information from older epochs. To avoid confusion, note that epochs are a separate notion from rounds: a verification round includes multiple epochs.

To memorialize epoch boundaries in history, clients issue *fence transactions*. A fence transaction is a transaction that reads-and-writes a single key named "EPOCH" (a dedicated key that is used by fence transactions only). Each client issues fence transactions periodically (for example, every 20 transactions).

What prevents the database from defeating the point of epochs by placing all of the fence transactions at the beginning of a notional serial schedule? Cobra leverages a property of practical serializable databases: preserved *session order*. That is, the serialization order must obey the execution order within each session (defined in §2.1). Many production databases (for example, PostgreSQL, Azure Cosmos DB, and Google Cloud Datastore) provide this property; for those which do not, cobra requires clients to build the session order, for example, by mandating that all transactions from the same session include a read-modify-write to a distinguished (per-session) key. Since transactions' serialization order obeys the session order, the epoch ordering intertwines with the workload. Indeed, the verifier adds to the known graph *session-order edges* (Figure 3, line 24), which capture the transaction issuing order in each session; the verifier gets that per-session order from collectors, which observe it directly.

The verifier also assigns an *epoch number* to each transaction. To do so, the verifier traverses the known graph (g), locates all the fence transactions, chains them into a list based on the RMW relation (§3.1), and assigns their position in the list as their epoch numbers. Then, the verifier scans the graph again, and for each normal transaction in a session that is between fences with epoch i and epoch j ($j \geq i+1$), the verifier assigns epoch number $j-1$.

During the scan, the verifier keeps track of the largest epoch number that has been seen or surpassed by every session, called $epoch_{agree}$. Then we have the following guarantee.

Guarantee. For any transaction T_i whose epoch $\leq (epoch_{agree}-2)$, and for any transaction (including future ones) T_j whose epoch $\geq epoch_{agree}$, the known graph g contains a path $T_i \rightsquigarrow T_j$.

To see why the guarantee holds, consider the path in three parts. First, for the fence transaction with epoch number $epoch_{agree}$ (denoted F_{ea}), g must have a path $F_{ea} \rightsquigarrow T_j$, through session-order edges. Similarly, for the fence transaction after T_i issued by the same session (denoted $F_{ea-\Delta}$), g has $T_i \rightsquigarrow F_{ea-\Delta}$. Finally, T_i has epoch $\leq (epoch_{agree}-2)$, so $F_{ea-\Delta}$ must have epoch $\leq (epoch_{agree}-1)$. Thus, $F_{ea-\Delta} \rightsquigarrow F_{ea}$ in g.

4.3 Garbage collection

Cobra takes a conservative approach. A transaction T can be safely deleted, if

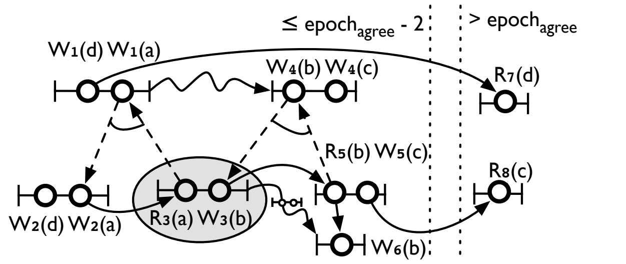
(i) T has been *superseded*, meaning that no future transactions can precede T or directly succeed T in the known graph; and
(ii) T is not involved in any potential cycle that includes edges from constraints whose resolution could be affected by future transactions.

Below, we delve into condition (i), then motivate condition (ii), and finally describe cobra's procedure for garbage collection.

Identifying superseded transactions. Define the *frontier* as the set of transactions that contain the most recent writes to keys among transactions with epoch number $\leq (epoch_{agree} - 2)$. The frontier captures the earliest transactions that future transactions can possibly read. A transaction T is *superseded* if: (1) T does not belong to the frontier, (2) T has epoch number $\leq (epoch_{agree}-2)$, and (3) for any transaction T' that has a path to T in the known graph, T' has epoch number $\leq (epoch_{agree}-2)$. Note that condition (2) does not subsume condition (3): we could have $T' \rightsquigarrow T$ with T' having a larger epoch than T (the Guarantee in §4.2 does not apply to transactions whose epochs differ by one).

At a high level, if a transaction T is superseded, the verifier can conclude that no future transactions should read from T; such a future transaction would have to be ordered before some frontier transaction, which makes a cycle by having a path back to the future transaction, per the Guarantee (§4.2). Thus T is a *candidate* to delete. However, being superseded is not a sufficient condition for safe deletion, as we illustrate next.

Superseded does not imply disposable. Here is an example:



The shaded transaction (T_3) is superseded (T_3 and its predecessor T_2 have epochs $\leq epoch_{agree}-2$, and T_3 does not belong to the frontier). Now consider the effect of future transactions T_7 and T_8. T_8 operates on key c; the other operations on this key are $W_4(c)$ and $W_5(c)$. By the guarantee (§4.2), both T_4 and T_5 happen before T_8. Plus, $R_8(c)$ reads from $W_5(c)$, hence $W_4(c)$ must happen before $W_5(c)$ (otherwise, $R_8(c)$ should have read from $W_4(c)$). Consequently, the constraint $\langle (T_5,T_4), (T_4,T_3) \rangle$, which arises from key b, is solved: $T_4 \rightarrow T_3$ is chosen. Similarly, because of $R_7(d)$, the other constraint (concerning key a) is solved and $T_3 \rightarrow T_1$. Thus, there is a cycle ($T_1 \rightsquigarrow T_4 \rightarrow T_3 \rightarrow T_1$). Yet, deleting T_3 would make the cycle undetectable.

The core issue here is that future transactions can affect the resolution of constraints among "old" transactions.

Identifying safe transactions. To garbage collect, the verifier clones the known graph (g in Fig. 3) into g'. Then, for each constraint (con in Fig. 3), the verifier adds all edges in both edge sets to g'. Finally, for each superseded transaction T, if T does not belong to any cycles in g' or belongs to cycles consisting only of superseded transactions, the verifier deletes T. This approach meets conditions (i) and (ii), as argued in Appendix C [132].

COBRA component	LOC written/changed
COBRA client library	
history recording	620 lines of Java
database adapters	900 lines of Java
COBRA verifier	
data structures and algorithms	2k lines of Java
GPU optimizations	550 lines of CUDA/C++
history parser and others	1.2k lines of Java

Figure 4: Components of COBRA implementation.

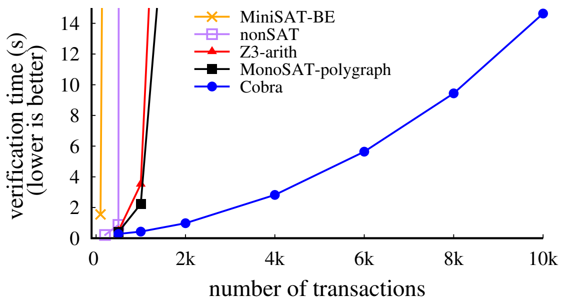


Figure 5: COBRA's running time is shorter than other baselines' on the BlindW-RW workload. The same holds on the other benchmarks (not depicted). Verification runtime grows superlinearly.

5 Implementation

Figure 4 lists the components of COBRA's implementation. COBRA's client library wraps other database libraries: JDBC, Google Datastore library, and RocksJava. It enforces the assumption of uniquely written values (§2.2), by adding a unique id to a client's writes, and stripping them out of reads. It also issues fence transactions (§4.2). Finally, we implement history collection (§2.1) in this library (the library writes operations to disk before sending them to the database); a better implementation would place this function in a proxy.

The verifier iterates the pruning logic within a round, stopping when it finds nothing more to prune or when it reaches a configurable maximum number of iterations (to bound the verifier's work); a better implementation would stop when the cost of the marginal pruning iteration exceeds the improvement in the solver's running time brought by this iteration.

Another aspect of pruning is GPU acceleration. Recall that pruning works by computing the transitive closure of the known edges (Figure 3, line 77). COBRA uses the standard algorithm: repeated squaring of the Boolean adjacency matrix [70, Ch.25] as long as the matrix keeps changing, up to $\log|V|$ matrix multiplications. ($\log|V|$ is the worst case and occurs when two nodes are connected by a ($\geq |V|/2+1$)-step path; at least in our experiments, this case does not arise much.) The execution platform is cuBLAS [12] (a dense linear algebra library on GPUs) and cuSPARSE [13] (a sparse linear algebra library on GPUs), which contain matrix multiplication routines.

COBRA includes several optimizations. It invokes a specialized routine for triangular matrix multiplication (after testing the graph for acyclicity and then indexing the vertices according to a topological sort, creating a triangular matrix). COBRA also exploits sparse matrix multiplication (cuSPARSE), and moves to ordinary (dense) matrix multiplication when the density exceeds a threshold (namely, "5% of the matrix elements are non-zero", the empirical cross-over point that we observed).

When COBRA's verifier detects a serializability violation, it creates a certificate with problematic transactions: either a cycle in the known graph (detected by COBRA's algorithms) or else a set of unsatisfiable clauses (produced by MonoSAT).

6 Experimental evaluation

We answer three questions:

- What are the verifier's costs and limits, and how do these compare to baselines?
- What is the verifier's end-to-end, round-to-round *sustainable capacity*? This determines the offered load (on the actual database) that the verifier can support.
- How much runtime overhead (in terms of throughput and latency) does COBRA impose for clients? And what are COBRA's storage and network overheads?

Benchmarks and workloads. We use four benchmarks:

- *TPC-C* [31] is a standard. A warehouse has 10 districts with 30k customers. There are five types of transactions (frequencies in parentheses): new order (45%), payment (43%), order status (4%), delivery (4%), and stock level (4%). In our experiments, we use one warehouse, and clients issue transactions based on the frequencies.
- *C-Twitter* [8] is a simple clone of Twitter, according to Twitter's own description [8]. Users can tweet a new post, follow/unfollow other users, and show a timeline (the latest tweets from followed users). Our experiments include 1000 users. Each user tweets 140-word posts and follows/unfollows other users based on a Zipfian distribution ($\alpha = 100$).
- *C-RUBiS* [30, 41] simulates bidding systems like eBay [30]. Users can register accounts, register items, bid for items, and comment on items. We initialize the market with 20k users and 200k items.
- *BlindW* measures COBRA's performance in extreme scenarios, specifically those with many blind writes (that is, writes not preceded by a read of the same key in the same transaction); this pattern is the fundamental source of uncertainty in constraints (§3). This benchmark creates 10k keys, and runs read-only and write-only transactions, each with eight operations. It has three variants: (1) *BlindW-RM* (Read Mostly), with 90% read-only transactions; (2) *BlindW-RW* (Read-Write), evenly divided between read-only and write-only transactions; and (3) *BlindW-WM* (Write Mostly), with 90% write-only transactions.

Violation	Database	#Txns	Time
G2-anomaly [19]	YugaByteDB 1.3.1.0	37.2k	66.3s
Disappearing writes [1]	YugaByteDB 1.1.10.0	2.8k	5.0s
G2-anomaly [18]	CockroachDB-beta 20160829	446	1.0s
Read uncommitted [26]	CockroachDB 2.1	20*	1.0s
Read skew [25]	FaunaDB 2.5.4	8.2k	11.4s

Figure 6: Serializability violations that COBRA checks. "Violation" describes the phenomenon that clients experience. "Database" is the database (with version number) that causes the violation. "#Txns" is the size of the violation history. "Time" is the runtime for COBRA to detect the violation.

* The bug report only contains a small fragment of the history.

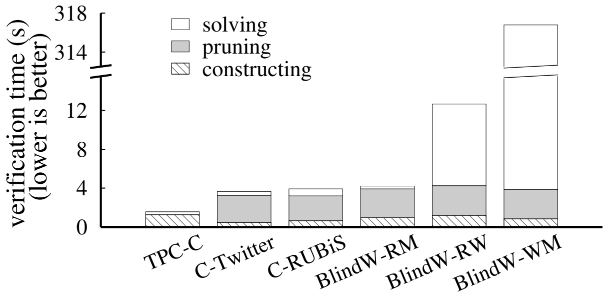


Figure 7: Decomposition of COBRA runtime, on 10k-transaction workloads. Pruning dominates for read-mostly workloads, whereas solving dominates for workloads with many writes.

Databases and setup. We evaluate COBRA on Google Cloud Datastore [21], RocksDB [29, 74] (both provide a key-value API), and PostgreSQL [28, 114] (which only supports the SQL interface, COBRA translates SQL queries to key-value operations). In our experimental setup, clients interact with Google Cloud Datastore through the wide-area Internet, and connect to a PostgreSQL server through a local 1 Gbps network. One client starts one session.

Database clients run on two machines with a 3.3GHz Intel i5-6600 (4-core) CPU, 16GB memory, a 250GB SSD, and Ubuntu 16.04. For PostgreSQL, a database instance runs on a machine with a 3.8GHz Intel Xeon E5-1630 (8-core) CPU, 32GB memory, a 1TB disk, and Ubuntu 16.04. For RocksDB, the same machine hosts the client threads and RocksDB threads, which all run in the same process. We use a *p3.2xlarge* Amazon EC2 instance as the verifier, with an NVIDIA Tesla V100 GPU, a 8-core CPU, and 64GB memory.

6.1 One-shot verification

In this section, we consider "one-shot verification": a verifier gets a history and decides whether that history is serializable. In our setup, clients record history fragments and store them as files; a verifier reads them from the local file system. In this section, the database is RocksDB (PostgreSQL and Google Cloud Datastore give similar results).

Baselines. We have four baselines:

- **A non-SAT serializability-checking algorithm ("nonSAT"):** To the best of our knowledge, the most efficient work for checking serializability that is not based on SAT or SMT solving is Biswas and Enea [59]. In our experiments, we use their Rust implementation [58].
- **SAT solver ("MiniSAT-BE"):** We use the same solving baseline that Biswas and Enea use for their own comparisons [59]: encoding serializability verification into SAT formulas, and feeding this encoding to MiniSAT [76], a popular SAT solver.
- **COBRA, subtracted ("MonoSAT-polygraph"):** We implement the original polygraph (§2.3), directly encode the constraints (without the techniques of §3), and feed them to the MonoSAT SMT solver [52].
- **SMT solver ("Z3-arith"):** An alternative use of SMT, and a natural baseline, is a linear arithmetic encoding: each node is assigned a distinct integer index, with read-from relationships creating inequality constraints, and writes inducing additional constraints (for a total of $O(|V|^2)$ constraints, as in §2.3). The solver is then asked to map nodes to integers, subject to those constraints [80, 91]. We use Z3 [73] as the solver (experiments below use Z3's default configuration; we also experimented with all four builtin linear integer arithmetic tactics, which produce similar results).

As a special case, there is an alternative baseline for TPC-C that has the same performance as COBRA and beats other baselines. Namely, for RMW transactions, add inferred read-dependency and write-dependency edges to a candidate graph (without constraints, so potentially missing dependency information), topologically sort it, and check whether the result matches history; if not, repeat. This process has even worse order complexity than the brute-force approach (§2.3). However, it works for TPC-C because that workload has *only* RMW transactions. Effectively, all of history coalesces to a single, correctly-ordered chain (§3.1), yielding a serialization graph.

In the experiments below, the baselines and COBRA make use of session order edges (§4.2; also called *program order* in BE [59] and its implementation [58]).

Verification runtime vs. number of transactions. We compare COBRA to other baselines, on the various workloads. We use 24 clients. We vary the number of transactions in the workload, and measure the verification time. Figure 5 depicts the results on the BlindW-RW benchmark. On all five benchmarks, COBRA does better than MonoSAT-polygraph and Z3-arith, which do better than MiniSAT-BE and nonSAT.

Detecting serializability violations. We investigate COBRA's performance on unsatisfiable instances: does COBRA search for an unacceptably long time on real-world workloads? We consider five workloads that are known to have serializability violations [1, 18, 19, 25, 26]. We experiment by downloading the reported histories from their bug repositories and feeding them to COBRA's verifier. Figure 6 shows the results. COBRA detects all violations and finishes in reasonable time.

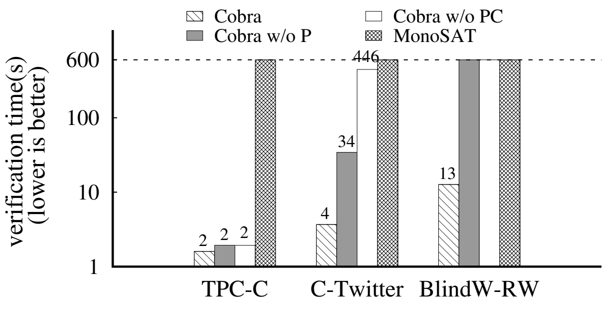


Figure 8: Differential analysis on several workloads, log-scale, with runtime above bars. Experiments time out at 10min (dotted line); no runtime is shown for timed-out experiments. On TPC-C, combining writes exploits the RMW pattern and solves all the constraints. On C-Twitter, each of COBRA's components contributes meaningfully. On BlindW-RW, pruning is essential, because the workload has many blind writes which cannot benefit from the other two techniques.

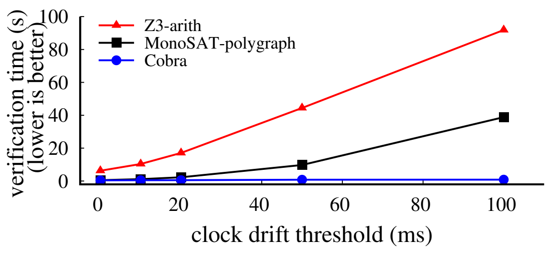


Figure 9: COBRA's running time is shorter than other baselines' on checking strict serializability under clock drift. The workload is 2,000 transactions of BlindW-RW (clock drift threshold of 100 ms).

Decomposition of COBRA's verification runtime. We measure the wall clock time of COBRA's verification, broken into stages: *constructing*, which includes creating the known graph, combining writes, and creating constraints (§3.1–§3.2); *pruning* (§3.3), which includes the time taken by the GPU; and *solving* (§3.4), which includes the time spent within MonoSAT. We experiment with all benchmarks, with 10k transactions.

Figure 7 depicts the results. In benchmarks with RMWs only (the left one), there are no constraints, so COBRA doesn't prune (see also the special case baseline, §6.1). In benchmarks with many reads and RMWs (the second to fourth bars), the dominant component is pruning not solving, because COBRA's own logic identifies concrete dependencies. In benchmarks with many blind writes (the last two), solving is a much larger contributor because COBRA cannot eliminate as many constraints, leading to a larger search space, an effect that grows more pronounced as the fraction of blind writes increases. On the other hand, a majority of writes is not consistent with the patterns in common online transaction processing workloads (OLTP), where reads dominate.

Differential analysis. We experiment with four variants: COBRA itself; COBRA without pruning (§3.3); COBRA without pruning and coalescing (§3.2), which is equivalent to MonoSAT plus write combining (§3.1); and the MonoSAT baseline.

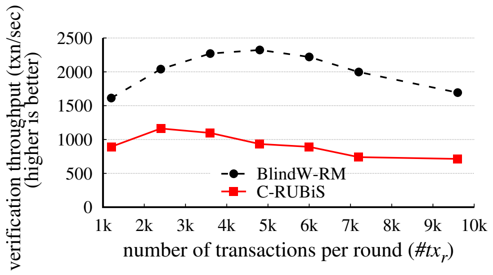


Figure 10: Verification throughput vs. round size (#tx_r). The verification capacity for BlindW-RM (the dashed line) is 2.3k txn/sec when #tx_r is 5k; the capacity for C-RUBiS (the solid line) is 1.2k txn/sec when #tx_r is 2.5k.

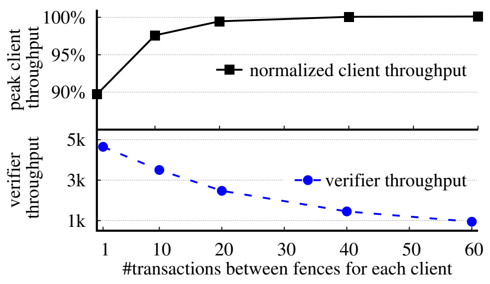


Figure 11: Client and verifier throughputs with different fence frequencies. Client throughput (the solid line) is normalized to the workload without fence transactions. In BlindW-RM, each normal transaction has 8 operations (§6), and fence transactions have 1–2 operations.

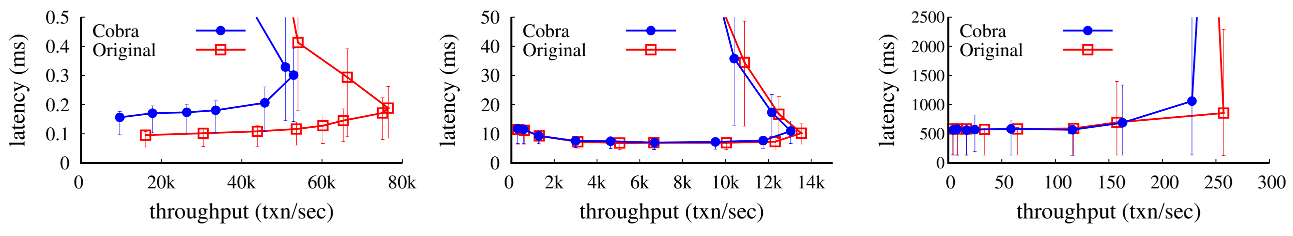
We experiment with three benchmarks, with 10k transactions. Figure 8 depicts the results.

Checking strict serializability under clock drift. Clock drift adds complexity to strict serializability (§1,§3.5). To measure this effect, we experiment with COBRA, MonoSAT-polygraph, and Z3-arith, under different clock drifts, on the same workload. The workload has eight clients running BlindW-RW on 1k keys for one second with a throughput of 2k transaction/sec. To control computational overhead, the clients issue 20 transactions every 10ms. The maximum clock drift threshold is 100 ms [15]; similar thresholds can be found elsewhere [10, 37]. Figure 9 depicts the results; COBRA outperforms the baselines by 45× and 107× in verification time.

6.2 Scaling

What offered load (to the database) can COBRA support on an ongoing basis? To answer this question, we must quantify COBRA's *verification capacity*, in txns/second. This depends on the characteristics of the workload, the number of transactions one round (§4) verifies (#tx_r), and the average time for one round of verification (t_r). Note that the variable here is #tx_r; t_r is a function of that choice. So the *verification capacity* for a particular workload is defined as: $\max_{\#tx_r}(\#tx_r/t_r)$.

To investigate this quantity, we run all our benchmarks on RocksDB with 24 concurrent clients, each configured to issue



(a) RocksDB (b) PostgreSQL (c) Google Datastore

Figure 12: Throughput and latency, for C-Twitter benchmark. For our RocksDB setup, 90th percentile latency increases by 2×, with 50% throughput penalty, an artifact of history collection (disk bandwidth contention between clients and the DB). COBRA imposes minor overhead for our PostgreSQL. For Google Datastore, the throughput penalty reflects a ceiling (a maximum number of operations per second) imposed by the cloud service and the extra operations caused by fence transactions.

fence transactions every 20 transactions. We generate a 100k-transaction history ahead of time. For that same history, we vary $\#tx_r$, plot $\#tx_r/t_r$, and choose the optimum.

Figure 10 depicts the results for two benchmarks (C-RUBiS and BlindW-RM); C-Twitter and TPC-C have similar results (not depicted), but BlindW-RW and BlindW-WM run out of memory (we elaborate below). When $\#tx_r$ is smaller, COBRA does not have enough transactions to garbage collect, hence wastes cycles on redundantly analyzing transactions from prior rounds; when $\#tx_r$ is larger, COBRA suffers from a problem size that is too large (recall that verification time increases superlinearly; §6.1).

History eventually exceeds GPU memory on the BlindW-RW and BlindW-WM benchmarks because blind writes limit COBRA's ability to garbage collect transactions: blind writes cannot benefit from combining writes (§3.1), hence many constraints remain, causing transactions to be involved in uncertain constraints, and thus not collectible (§4.3). Addressing this issue is future work (§8).

Fence frequency. The choice of fence frequency trades off verification capacity and peak client-side throughput. To quantify, we do the same BlindW-RM experiments as in Figure 10, this time fixing round size (at 5k transactions) and varying fence frequency.

Figure 11 depicts the results. The verifier has better throughput if clients issue fence transactions more frequently. The reason is that more fence transactions result in smaller epoch sizes, hence transactions can be garbage collected earlier, and the problem size for the verifier in each round is smaller. Moreover, with more fence transactions, the problem in each round is easier to solve because fence transactions add ordering constraints, which further reduce the number of possibly-valid execution schedules. However, more frequent fence transactions sacrifices peak client-side throughput because more resources are occupied by fence transactions.

The right setting of fence frequency depends on client offered load, peak:average throughput ratio, database capacity, and tolerance for latency. If the frequency is set too high (toward the left side of the x-axis), clients will no longer be able to offer the original workload with acceptable latency. On the

workload	network overhead		history
	traffic	percentage	size
BWrite-RW	227.4 KB	7.28%	245.5 KB
C-Twitter	292.9 KB	4.46%	200.7 KB
C-RUBiS	107.5 KB	4.53%	148.9 KB
TPC-C	78.2 KB	2.17%	1380.8 KB

Figure 13: Network and storage overheads per 1k transactions. The network overheads comes from fence transactions and the metadata (transaction ids and write ids) added by COBRA's client library.

other hand, for too-low frequencies (toward the right side of the x-axis), the verifier will not be able to keep up with the database's average load. Of course, if client load is constant, the fence frequency should be chosen as the point where verifier throughput equals client offered load.

6.3 Online overheads

The baseline in this section is the legacy system; that is, clients use the unmodified database library (for example, JDBC), with no recording of history.

Latency-versus-throughput. We evaluate COBRA's client-side throughput and latency in the three setups, tuning the number of clients (up to 256) to saturate the databases. Figure 12 depicts the results. (Although these results include the overhead of collecting histories in the client library (§5), that overhead is negligible, as the log size is small and disk latency is lower than network latency.)

Network cost and history size. We evaluate the network traffic on the client side by tracking the number of bytes sent over the NIC. We measure the history size by summing sizes of the history files. Figure 13 summarizes.

6.4 Summary of experimental evaluation

COBRA improves by at least 10× on baselines in verification cost (Figure 5), detects real-world issues (Figure 6), gains from its techniques versus the baseline (Figures 7 and 8), and imposes tolerable overhead (Figures 12 and 13).

Furthermore, its sustained throughput of 2k txn/sec (Figure 10) corresponds to large-scale real-world workloads. While 2k/sec might not sound large, recall that the verifier's performance requirement is to match *average* database load (§2.1). An average of 2k/sec corresponds to 170M/day, sufficient to handle Apple Pay [6] (33M txn/day), Visa [33] (150M txn/-day), and others. Of course, a "transaction", in the sense of a payment, might translate to several database transactions, so the comparison is inexact.

7 Related work

As stated earlier (§1), COBRA is the first system that verifies the executions of (a) black box databases, for (b) serializability, under (c) workloads of realistic scale.

Three works that we are aware of tackle (a) and (b) together. Biswas and Enea [59] was covered in Section 1 and compared in Section 6. Sinha et al. [124] use SMT solvers to analyze all possible interleavings for a concurrent program, to search for serializability violations. Finally, Gretchen [23] is an experimental checker that verifies the non-strict serializability of a COBRA-style history. Gretchen encodes the history as constraints [67] (similar to our MiniSAT-BE baseline; §6.1) and solves them with the fzn-gecode [20] solver.

A recent work that deserves special mention is Elle [94], which tests for isolation anomalies, and has found many isolation bugs in production databases. (Elle is part of the impactful Jepsen [14] project, which we discuss later in this section.) Elle has two modes for testing serializability. In one, it verifies Adya's serializability (PL-3 [38]), the same goal as COBRA. But Elle in this mode requires a workload that makes the version order (§2.2) manifest; for example, clients invoke "append", and writes become appends to a list (Elle in this mode also supports counters and sets). In relying on a specific API and a specific workload for testing, this mode does not meet our notion of black box (§1).

In the second mode, Elle works over arbitrary observations of key-value input/output, the same setup as COBRA. Without a determined version order, it applies heuristics to identify bugs. These heuristics are useful but not comprehensive, so this is not verification. For example, if a history contains a set of concurrent transactions that form a cycle through anti-dependencies, Elle's current heuristics do not detect the non-serializability.

Checking consistency. Serializability is a particular *isolation* level in a transactional system—the I in ACID transactions. In shared memory systems and systems that offer replication (but do not necessarily support transactions), there is an analogous correctness contract, namely *consistency*. (Confusingly, the "C(onsistency)" in ACID transactions refers to something else [47].) Example consistency models are linearizability [88], sequential consistency [97], and eventual consistency [112]. Testing adherence to these models is an analogous problem to ours. In both cases, one searches for a total order of operations that fits the ordering constraints of both the model and the history [82]. As in checking serializability, the computational complexity of checking consistency decreases if a stronger model is targeted (for example, linearizability vs. sequential consistency) [81], or if more ordering information can be (intrusively) acquired (by opening black boxes) [123, 139].

Concerto [43] uses *deferred verification*, allowing it to exploit offline memory checking [60] to check online the sequential consistency of a highly concurrent key-value store. Concerto's design achieves orders-of-magnitude performance improvement compared to Merkle tree-based approaches [60, 106], but it also requires modifying the storage layer. (See elsewhere [75, 98] for algorithms related to Concerto.)

A body of work examines cloud storage consistency [39, 42, 83, 101, 102, 115, 135, 142]. These works rely on extra ordering information obtained through techniques like loosely- or well-synchronized clocks [39, 42, 82, 83, 93, 102, 115, 135, 142], or client-to-client communication [101, 122], or by guessing [143] (which risks falsely rejecting honest executions). As another example, a gateway that sequences the requests can ensure consistency by enforcing ordering [90, 113, 122, 125], thereby dramatically reducing concurrency.

Some of COBRA's techniques are reminiscent of these works, such as its use of serialization graphs [42, 82]. However, a substantial difference is that COBRA neither modifies the "memory" (the database) to get information about the actual internal schedule nor depends on external synchronization. COBRA of course exploits epochs (§4.2), but this is for scaling, not core to the verification task, and invokes standard database interfaces.

Execution integrity. Our problem relates to the broad category of *execution integrity*—ensuring that a module in another administrative domain is executing as expected.

One approach is to use trusted components. For example, Byzantine fault tolerant (BFT) replication [66] (where the assumption is that a super-majority is not faulty) and TEEs (trusted execution environments, comprising TPM-based systems [68, 87, 104, 105, 111, 117, 119, 126] and SGX-based systems [44, 45, 51, 89, 95, 118, 121, 125]) ensure that the right code is running. However, this does not ensure that the code itself is right; concretely, if a database violates serializability owing to a bug, neither BFT nor SGX hardware helps.

Other examples are Verena [92], Orochi [131], AVM [85], and Ripley [134]. These systems provide end-to-end assurance that a whole stack is executing as it should, but they are not black box. COBRA is the other way around: it treats the database as a black box, but its purview is limited to the database.

A class of systems uses complexity-theoretic and cryptographic mechanisms [61, 120, 145, 146]. None of these works handle systems of realistic scale, and only one of them [120] handles concurrent workloads. An exception is Obladi [71], which remarkably provides ACID transactions atop an ORAM abstraction by exploiting a trusted proxy that carefully manages the interplay between concurrency control and the ORAM protocol; its performance is surprisingly good (as cryptographic-based systems go) but still pays 1-2 orders of magnitude overhead in throughput and latency.

Detecting application anomalies caused by weak consistency. Several works [63, 109, 116] detect anomalies for applications deployed on weakly consistent storage. Like COBRA, these works use SAT/SMT solvers on graph-related problems. But the similarities end there: these works analyze application behavior, taking the storage layer as *trusted* input. As a consequence, the technical mechanisms are very different.

Testing distributed systems. There is a line of research on testing the correctness of distributed systems under various failures, including network partition [40], power failures [147], and storage faults [78]. Among these, Jepsen [14] is a very successful testing framework (the aforementioned Elle is one of Jepsen's checkers) with active, ongoing innovation, which has detected large numbers of correctness bugs in production distributed systems. COBRA is complementary (and intended to be complimentary) to these works. Indeed, COBRA uses several of Jepsen's traces in Figure 6 (§6.1).

Definitions and interpretations of isolation levels. COBRA of course uses dependency graphs, which are a common tool for reasoning about isolation levels [38, 56, 110]. However, isolation levels can be interpreted via other means such as excluding anomalies [53] and client-centric observations [72]; an open and intriguing question is whether the other definitions yield a more intuitive or more easily-implemented encoding and algorithm than the one in COBRA.

8 Discussion, future work, and conclusion

Applicability. COBRA cannot prevent misbehavior, only detect it. On the other hand, *no* system that we are aware of can detect and prevent serializability violations online. Meanwhile, COBRA could contribute to recovery: given a certificate (§5), the user could supply a candidate serialization order, enabling roll back and replay. See also Concerto's eloquent case for deferred verification [43, §1.1].

Who would use COBRA? We covered some scenarios in Section 2.1. Another is to use COBRA as the checker of a testing framework (for example, Jepsen [14], §7). Then one could insert malfunctions into various layers of the system (OS, storage, network) and avoid instrumenting the database.

One might assume that the verifier needs to be at least as powerful as the database, so why have the database? While they must match in long-term average transactions/sec (§2.1), the two do different kinds of work per transaction. The database provides geo-replication, concurrency control, crash-atomicity, durability, load-balancing, and more; the verifier is a single machine and purely algorithmic.

Limitations and future work. COBRA can be slow for certain workloads (for example, when there are many unconstrained writes, as in the BlindW-WM benchmark; §6.1). In fact, COBRA's worst-case running time is in principle exponential; future work is to investigate whether there are real-world workloads that induce this behavior, or does it just happen under contrived problem instances as in the NP-reduction?

Consistent with our experiments, we expect "real-world" workloads not to trigger this behavior. For intuition, low contention on each key yields a relatively small number of constraints and a small search space; the extreme is that each key is touched once, yielding no dependencies. If there is high contention with sufficient reads, there are more dependencies among transactions, which imposes more ordering. An extreme case is that there is only one key, and transactions read and write this key, so that all transactions are ordered accordingly.

We have assumed that the verifier and the collectors operate fault-free. Future work is to make them fault-tolerant. To that end, COBRA could use standard techniques (for example, transparent state machine replication) or extend its protocols to handle failures. Note that even if some history fragments are lost, COBRA can (with minor modifications) produce meaningful results: a cyclic dependency (serializability violation) in a partial history is also a violation against the full history. Another idea is to use COBRA to infer what the missing transactions would have to be in order to ensure serializability.

COBRA focuses on serializability and strict serializability; future work is extending to other isolation levels. Relatedly, COBRA does not support range queries and other high-level operators (for example, sum and join); if applications want them, they have to rewrite queries (§1). Handling these queries "natively" would require the verifier to analyze both keys that are returned and keys that are *not* returned.

Making garbage collection more aggressive is another area of potential improvement, for example, by allowing the verifier to query the database to resolve certain constraints.

Conclusion. A final critique is that we lack a sensational headline, as we did not identify novel serializability violations. However, validation doesn't always produce a gotcha: from our perspective, it's equally significant to be able to report on a system that gives us confidence that cloud databases do meet serializability. This was something we used to have to *trust*; COBRA, however imperfect, helps us be *sure*.

Acknowledgments

Sebastian Angel, Miguel Castro, Pete Chen, Byron Cook, Andreas Haeberlen, Dennis Shasha, Ioanna Tzialla, Thomas Wies, and Lingfan Yu made helpful comments and gave useful pointers. We thank the anonymous reviewers (including at SOSP and NSDI) for careful and constructive comments, and likewise our shepherd Chris Hawblitzel. We thank the anonymous artifact evaluators for their patience and attention to detail. This work was supported by NSF grants CNS-1423249 and CNS-1514422, ONR grant N00014-16-1-2154, AFOSR grants FA9550-15-1-0302 and FA9550-18-1-0421, and DARPA under Agreement HR00112020022.

References

[1] Acknowledged inserts can be present in reads for tens of seconds, then disappear. https://github.com/YugaByte/yugabyte-db/issues/824.

[2] Amazon Aurora. https://aws.amazon.com/rds/aurora/.

[3] Amazon Aurora MySQL Reference. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraMySQL.Reference.html.

[4] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[5] Amazon DynamoDB Transactions. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/transaction-apis.html.

[6] Apple pay transaction volume and new user growth outpacing paypal, tim cook says. https://9to5mac.com/2019/07/30/apple-pay-transactions-users-paypal/.

[7] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/.

[8] Big data in real time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter.

[9] CockroachDB: Distributed SQL. https://www.cockroachlabs.com.

[10] CockroachDB: What happens when node clocks are not properly synchronized? https://www.cockroachlabs.com/docs/stable/operational-faqs.html#what-happens-when-node-clocks-are-not-properly-synchronized.

[11] CockroachDB's consistency model. https://www.cockroachlabs.com/blog/consistency-model/.

[12] cuBLAS: Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas.

[13] cuSPARSE: Sparse Linear Algebra on GPUs. https://developer.nvidia.com/cusparse.

[14] Distributed system safety research. https://jepsen.io/.

[15] Executive summary: Computer network time synchronization. https://www.eecis.udel.edu/~mills/exec.html.

[16] FaunaDB. https://fauna.com.

[17] FoundationDB. https://www.foundationdb.org.

[18] G2: anti-dependency cycles. https://github.com/cockroachdb/cockroach/issues/10030.

[19] G2-item anomaly with master kills. https://github.com/YugaByte/yugabyte-db/issues/2125.

[20] Gecode: Flatzinc. https://www.gecode.org/flatzinc.html.

[21] Google Cloud Datastore. https://cloud.google.com/datastore/.

[22] Google Cloud Spanner. https://cloud.google.com/spanner/.

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

[24] How Halo 5 implemented social gameplay using Azure Cosmos DB. https://azure.microsoft.com/en-us/blog/how-halo-5-guardians-implemented-social-gameplay-using-azure-documentdb/.

[25] Jepsen: Faunadb 2.5.4. http://jepsen.io/analyses/faunadb-2.5.4.

[26] Lessons learned from 2+ years of nightly jepsen tests. https://www.cockroachlabs.com/blog/jepsen-tests-lessons/.

[27] Norwegian electronics giant scales for sales, sets record with cloud-based transaction processing. https://customers.microsoft.com/en-us/story/elkjop-retailers-azure.

[28] PostgreSQL. https://www.postgresql.org/.

[29] RocksDB. https://rocksdb.org/.

[30] RUBiS. https://rubis.ow2.org/.

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

[32] Transactions, cloud Spanner. https://cloud.google.com/spanner/docs/transactions.

[33] Visa: Small business retail. https://usa.visa.com/run-your-business/small-business-tools/retail.html.

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

[35] YugaByte db 1.3.1, undercounting counter. http://jepsen.io/analyses/yugabyte-db-1.3.1.

[36] YugaByte DB: Home. https://www.yugabyte.com.

[37] yugabyte source code. https://github.com/yugabyte/yugabyte-db/blob/3b90e8560b8d8bc81fba6ba9b9f2833e83e2244e/src/yb/util/physical_time.cc#L36.

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

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

[40] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proc. OSDI*, Oct. 2018.

[41] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *Proc. IEEE WWC*, Nov. 2002.

[42] 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.

[43] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: a high concurrency key-value store with integrity. In *Proc. SIGMOD*, May 2017.

[44] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell, et al. SCONE: Secure Linux containers with Intel SGX. In *Proc. OSDI*, Oct. 2016.

[45] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch. LibSEAL: Revealing service integrity violations using trusted execution. In *Proc. EuroSys*, Apr. 2018.

[46] A. Awad and B. Karp. Execution integrity without implicit trust of system software. In *ACM Workshop on System Software for Trusted Execution (SysTEX)*, 2019.

[47] P. Bailis. Linearizability versus serializability. `http://www.bailis.org/blog/linearizability-versus-serializability/`, Sept. 2014.

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

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

[50] 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.

[51] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proc. OSDI*, Oct. 2014.

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

[53] 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.

[54] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[55] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

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

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

[58] R. Biswas and C. Enea. dbcop source code. `https://zenodo.org/record/3367334`.

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

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

[61] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proc. SOSP*, Nov. 2013.

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

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

[64] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT solver. In *Proc. CAV*, July 2008.

[65] M. A. Casanova. *The concurrency control problem for database systems*. Number 116. Springer Science & Business Media, 1981.

[66] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. OSDI*, Feb. 1999.

[67] 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.

[68] C. Chen, P. Maniatis, A. Perrig, A. Vasudevan, and V. Sekar. Towards verifiable resource accounting for outsourced computation. In *Proc. VEE*, Mar. 2013.

[69] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *TOCS*, 31(3), June 2013.

[70] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, third edition*. The MIT Press, 2009.

[71] N. Crooks, M. Burke, E. Cecchetti, S. Harel, L. Alvisi, and R. Agarwal. Obladi: Oblivious serializable transactions in the cloud. In *Proc. OSDI*, Oct. 2018.

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

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

[74] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in RocksDB. In *Proc. CIDR*, Jan. 2017.

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

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

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

[78] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Proc. FAST*, Feb. 2017.

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

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

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

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

[83] 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.

[84] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proc. DATE*, Mar. 2002.

[85] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proc. OSDI*, Oct. 2010.

[86] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *Proc. ICSE*, May 2008.

[87] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: end-to-end security via automated full-system verification. In *Proc. OSDI*, Oct. 2014.

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

[89] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: a distributed sandbox for untrusted computation on secret data. In *Proc. OSDI*, Oct. 2016.

[90] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *Proc. ICDE*, Apr. 2013.

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

[92] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for Web applications. In *Proc. S&P*, May 2016.

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

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

[95] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In *Proc. EuroSys*, Apr. 2018.

[96] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proc. EuroSys*, Apr. 2013.

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

[98] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. SIGMOD*, June 2006.

[99] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *Proc. AAAI*, Feb. 2016.

[100] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proc. SIGMOD*, May 2017.

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

[102] 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.

[103] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9), July 2013.

[104] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proc. S&P*, May 2010.

[105] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. EuroSys*, Apr. 2008.

[106] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proc. Crypto*, Aug. 1987.

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

[108] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Proc. OSDI*, Oct. 2014.

[109] K. Nagar and S. Jagannathan. Automated detection of serializability violations under weak consistency. *arXiv preprint arXiv:1806.08416*, 2018.

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

[111] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping trust in modern computers*. Springer, 2011.

[112] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP*, Oct. 1997.

[113] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *Proc. USENIX ATC*, June 2011.

[114] D. R. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *PVLDB*, 5(12), Aug. 2012.

[115] 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.

[116] K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan. Clotho: directed test generation for weakly consistent database systems. In *Proc. OOPSLA*, Oct. 2019.

[117] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proc. USENIX Security*, Aug. 2004.

[118] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. S&P*, May 2015.

[119] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proc. SOSP*, Oct. 2005.

[120] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proc. OSDI*, Oct. 2018.

[121] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In *Proc. NDSS*, Feb. 2017.

[122] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Proc. CCSW*, Oct. 2010.

[123] A. Sinha and S. Malik. Runtime checking of serializability in software transactional memory. In *Proc. IPDPS*, Apr. 2010.

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

[125] R. Sinha and M. Christodorescu. VeritasDB: High throughput key-value store with integrity. *IACR Cryptology ePrint Archive*, 2018.

[126] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proc. SOSP*, Oct. 2011.

[127] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. SAT*, June 2009.

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

[129] C. Su, N. Crooks, C. Ding, L. Alvisi, and C. Xie. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, May 2017.

[130] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *Proc. RV*, Sept. 2011.

[131] C. Tan, L. Yu, J. Leners, and M. Walfish. The efficient server audit problem, deduplicated re-execution, and the web. In *Proc. SOSP*, Oct. 2017.

[132] C. Tan, C. Zhao, S. Mu, and M. Walfish. Cobra: Making transactional key-value stores verifiably serializable (extended version). arXiv:1912.09018, https://arxiv.org/abs/1912.09018, Dec. 2019.

[133] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon Aurora : Design considerations for high throughput cloud-native relational databases. In *Proc. SIGMOD*, May 2017.

[134] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proc. CCS*, Nov. 2009.

[135] 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.

[136] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proc. SIGMOD*, June 2016.

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

[138] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

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

[140] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *Proc. SOSP*, Oct. 2015.

[141] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Notices*, 40(6), 2005.

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

[143] K. Zellag and B. Kemme. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *The VLDB Journal*, 23(1), Feb. 2014.

[144] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. SOSP*, Oct. 2015.

[145] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *Proc. S&P*, May 2017.

[146] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *Proc. CCS*, Oct. 2015.

[147] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proc. OSDI*, Oct. 2014.

A Artifact Appendix

This artifact contains two parts: a cobra verifier and cobra clients. The cobra verifier checks serializability of a set of transactions (called a history). Cobra clients include database clients and cobra's client library. Database clients are benchmark programs that interact with a black-box database (not part of cobra) and generate histories. Cobra's client library wraps database libraries, encodes and decodes values to and from the database, and records histories to logs.

Cobra's artifact, including source code and comprehensive instructions for running the code and reproducing results, is released at: https://github.com/DBCobra/CobraHome. Cobra's verifier requires an NVIDIA GPU to run, and cobra depends on Linux (tested on Ubuntu 18.04), Java (1.8 or higher), CUDA (tested on 10.0.130), and MonoSAT (1.6.0).