Morty: Scaling Concurrency Control with Re-Execution

Matthew Burke
Cornell University
United States
matthelb@cs.cornell.edu

Florian Suri-Payer
Cornell University
United States
fsp@cs.cornell.edu

Jeffrey Helt
Princeton University
United States
jhelt@cs.princeton.edu

Lorenzo Alvisi
Cornell University
United States
lorenzo@cs.cornell.edu

Natacha Crooks
UC Berkeley
United States
ncrooks@berkeley.edu

Abstract

Serializable systems often perform poorly under high contention. In this work, we analyze this performance limitation through a novel take on conflict windows. Through the lens of these windows, we develop a new concurrency control technique that leverages transaction re-execution to improve throughput scalability under high contention. Our system, Morty, achieves up to 1.7x-96x the throughput of state-of-the-art systems, with similar or better latency.

CCS Concepts: • **Computer systems organization** → **Reliability**; • **Information systems** → **Database transaction processing**; **Key-value stores**.

Keywords: replicated databases, concurrency control, multicore scalability, distributed systems

ACM Reference Format:
Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. 2023. **Morty: Scaling Concurrency Control with Re-Execution**. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3552326.3567500

1 Introduction

This chapter presents Morty, a novel storage system that leverages *transaction re-execution* to increase the throughput of serializable and interactive transactions.

The combination of serializability and interactivity is compelling. Serializability lets developers think of their transactions as if they are executing sequentially on a centralized machine, simplifying reasoning about application correctness. Interactivity in turn lets developers write fully general transaction code that is directly interleaved with application code, rather than encapsulated in the database or written in a separate domain-specific language [37].

For scalability, transactional data-stores are usually partitioned such that data and load can be spread across arbitrarily many machines; for availability, they are replicated, either within a datacenter, or across continents, to protect against major correlated failures [12, 47].

How much concurrency does enforcing serializability afford in such systems? The answer depends on the concurrency control mechanism that a system adopts. Yet none of the available choices do well under high contention. Poor performance is especially problematic in geo-replicated settings where high latency between replicas increases the duration of transactions and the likelihood that they will conflict.

In systems that leverage *optimistic concurrency control*, such as TAPIR [54], a transaction executes without blocking, but before it is allowed to commit, a validation phase verifies that serializability is not violated. When a conflict is detected, the transaction is aborted, leading to high abort rates under contention. In contrast, *pessimistic systems* like Spanner [12] preemptively prevent conflicting transactions from executing concurrently by guarding data accesses with locks. Under contention, however, deadlocks and lock thrashing can occur, and latency can significantly increase.

The traditional way to promote progress in the presence of such aborts or deadlocks has been to use exponential backoff: when a conflict is detected, rather than retrying straightaway, the aborted transaction waits a small amount of time, which increases exponentially with successive aborts. Essentially, this amounts to blind guessing how to space transactions temporally to ensure progress: too conservative a guess, and the impediment to progress may persist; too liberal, and opportunities for concurrency are needlessly sacrificed.

To move beyond the guesswork, this chapter proposes to revisit, from first principles, what in serializability fundamentally limits concurrent processing of conflicting transactions.

EuroSys '23, May 8–12, 2023, Rome, Italy
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9487-1/23/05.
https://doi.org/10.1145/3552326.3567500

We capture these requirements with the novel notion of *serialization windows*. Serialization windows are created by transactions that read and modify objects: a transaction T's serialization window for an object x starts at the write of x whose value it observes, and ends when T's own write to x becomes visible. Intuitively, enforcing serializability requires serialization windows to never overlap.

While this observation places a hard upper bound on the concurrency that can be achieved, it also suggests a way forward. First, it identifies an ideal execution pattern for a set of conflicting transactions: rather than rashly attempting to execute concurrently, they should align their execution so that they complete one right after the other, without overlaps. Second, it sheds new light on why existing concurrency control mechanisms perform relatively poorly: to reduce the chances that transactions will abort, exponential backoff can introduce long idle periods in the ideal execution pattern of consecutive serialization windows. In turn, these idle periods significantly limit the system's utilization: we find, for instance, that the CPU utilization of TAPIR and Spanner replicas is less than 17% on a high contention workload.

This chapter proposes Morty[1], a new serializable and replicated storage system that harnesses these spare CPU cycles to virtually eliminate idle periods and significantly improve transactional throughput.

Rather than letting chance determine how serialization windows manifest, Morty takes fate in its own hands and actively rearranges them to avoid overlaps. Specifically, Morty replicas monitor the occurrence of conflicting accesses and, when they detect overlapping serialization windows, trigger *transaction re-execution*: rather than aborting, a transaction T, upon learning of the existence of a conflicting write, partially restarts its execution. This approach effectively nudges serialization windows to be sequential, thus aligning them optimally. Re-execution is made transparent to applications by using a continuation passing style API, already battle-tested in production environments in systems like FaRM [17]; to the best of our knowledge, Morty is the first system to support transparent re-execution for general interactive transactions.

We implement Morty as a geo-replicated system that supports interactive transactions. Morty uses as its starting point for concurrency control multi-versioned timestamp ordering (MVTSO) [9], and extends it to offer efficient and safe transaction re-execution. To minimize latency across wide-area networks, Morty integrates the replication and concurrency control layers [45, 46, 54], thus avoiding the redundant coordination incurred by modular designs [12].

Our results are promising. We find that, on TPC-C, a standard transactional benchmark, Morty achieves 7.4x, 4.4x, and 1.7x higher throughput than Spanner, TAPIR, and a replicated MVTSO baseline respectively. Morty's performance gains are compounded on heavily contended workloads, where it achieves 95x, 52x, and 28x greater throughput than TAPIR, Spanner, and MVTSO respectively.

[1]**M**ulti-core **O**bject-store using **R**e-execution **T**ransactionall**y**.

In summary, we make the following contributions:

- We define *serialization windows* to characterize the maximum concurrency allowed in serializable systems.
- We propose *transaction re-execution* using a continuation passing style API to align serialization windows.
- We design and evaluate Morty, a serializable, replicated storage system that uses re-execution to attain higher throughput on high contention workloads.

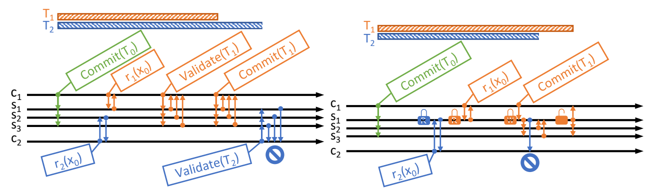
The chapter is organized as follows. We introduce the concepts of serialization windows and validity windows in Section 2, outline Morty's API for re-execution in Section 3, and detail Morty's transaction processing design in Section 4. We evaluate Morty's performance in Section 5, discuss related work in Section 6, and conclude in Section 7.

2 Scraping the Barrel: Limits to Extracting Concurrency

Serializability, the gold-standard correctness condition for transactional storage systems, provides the abstraction of a centralized storage system that executes transactions *sequentially* and ensures they only *read valid data* (data from committed transactions). These properties free developers from reasoning about complex interleavings of operations, simplifying application development [3, 7, 12, 38].

Despite the flexibility that the serializability abstraction affords to the underlying system in processing data accesses, there nevertheless exists a fundamental limitation: transactions cannot concurrently perform conflicting data accesses. Concurrency control mechanisms (CCs) are tasked with preventing such scenarios. How do the design choices of CCs dictate their performance on high contention workloads? In the rest of this section, we introduce a formal framework for reasoning about the performance limitations imposed by the *sequential execution* (§2.1) and *read validity* (§2.2) properties of serializability. Our generic framework can be applied to any serializable system to identify specific design choices that limit its concurrency. We later use insights from this analysis to design a new CC technique that optimizes serializable performance on contended workloads (§3).

Model. Our framework uses Adya's model [1] of a transactional storage system, which is expressed in terms of *histories* consisting of two parts: a partial order of events that reflect the operations of a set of transactions, and a version order that imposes a total order on committed object versions. A transaction T_i's read event $r_i(x_k)$ denotes that T_i observes version k of object x written by transaction T_k. Similarly, a transaction T_i's write event $w_i(x_i)$ denotes that T_i creates version i of x. If a transaction T_i commits, it has a corresponding commit event c_i. Every history H is associated with a



(a) OCC in TAPIR. **(b)** 2PL in Spanner.

Figure 1. Partial executions of two `Payment` transactions, T_1 and T_2, in replicated serializable systems. c_1 and c_2 are application clients issuing T_1 and T_2 respectively; s_1, s_2, and s_3 are storage servers.

directed serialization graph $DSG(H)$, whose nodes are committed transactions and whose edges denote the conflicts (read-write, write-write, or write-read) between them.

2.1 Sequential Execution

While non-conflicting transactions may freely access data, the order of conflicting accesses from transactions must be consistent with a sequential execution to maintain serializability. We explore this intuition with a simple example.

2.1.1 Motivating Example: TPC-C. TPC-C is a benchmark application that simulates the activity of a business that sells a product [49]. Within this workload, the `Payment` transaction represents a customer payment for a given order. As one of several contention hotspots, it generates a high rate of conflicting accesses to the `warehouse` table because it updates a warehouse's year-to-date payment total. We examine the concurrent execution of `Payment` transactions in two canonical CCs: optimistic concurrency control (OCC) [24] and two-phase locking (2PL) [9]. These CCs, which are used in a large number of production systems [6, 8, 10, 12, 13, 16, 19, 21, 27, 31, 34, 35, 39, 42, 43, 53], take opposing approaches to regulating concurrency, and thus provide a strong basis for understanding the fundamental performance limitations.

OCC. Figure 1a shows an execution of two conflicting `Payment` transactions, T_1 and T_2, that update the same warehouse row x in a replicated system with OCC. In OCC, transactions freely read data under the assumption that two transactions will not try to update the same data concurrently. Before a transaction commits, the system validates this assumption by checking that no other transaction committed a more recent write. Since T_1 reads a value for x before T_2 finishes writing its update (in OCC writes are buffered until commit), T_1 observes the same value as T_2. This interleaving of the reads and writes to x is irreconcilable with a sequential ordering of T_1 and T_2, and the system aborts T_2.

2PL. Figure 1b shows a similar execution of T_1 and T_2 in a replicated system that instead uses 2PL for CC. In replicated 2PL, a transaction acquires a read lock before reading an object. Similarly, a transaction acquires a write lock before writing to an object – a process which is typically deferred to commit time. These per-object reader-writer locks prevent two transactions from reading and modifying the same object concurrently. In the execution of Figure 1b, T_1 and T_2 both acquire read locks on x before either acquires a write lock. This would lead to a deadlock when both transactions attempt to upgrade their read locks to write locks. To avoid such deadlocks, systems typically employ a deadlock avoidance strategy. For example, the system in the execution aborts the younger transaction T_2 so that T_1 is able to upgrade its lock.

2.1.2 Serialization Windows. Transactions in serializable systems must appear to take effect sequentially. As demonstrated in Figure 1, if the reads and writes of transactions to the same object interleave, this abstraction may be violated, and at least one of the transactions cannot commit.

Logically, when a transaction T' reads an object x last written by T, T' is choosing to order itself directly after T in the serializable order (the read from T' must appear to immediately follow T's write). No transaction that—through reads and writes to x—would preclude this ordering can commit. In effect, T''s read initiates a period of mutual exclusion: until T' has overwritten x, no other transaction can also read and modify x. We note that such a period of mutual exclusion does not apply to transactions that only read an object.

Figures 1a and 1b explicitly depict this time period with the bars labelled T_1 and T_2 above the timeline. For both executions, the overlap between two such periods intuitively corresponds to a non-serializable interleaving. We refer to this period as a *serialization window* and we formally prove that no two serialization windows can overlap in a system that provides the abstraction of sequential execution.

Formal Definitions. If a transaction T_i reads version k of object x ($r_i(x_k)$) and writes version i of x ($w_i(x_i)$), T_i creates a *serialization window on* x that starts at $w_k(x_k)$ and ends at $w_i(x_i)$. T_i's serialization window on x starts when its read dependency (the version of x it read) is written and ends when it writes the next version of x. [2]

In Adya's model, the sequential execution property is formalized as a statement about the DSG: if $DSG(H)$ is acyclic then a topological sort of the graph is a sequence of transactions that produces an execution equivalent to the one represented by H. A system thus provides the abstraction of *sequential execution* if it only produces histories whose DSGs are acyclic.

For these definitions, the following Theorem holds:

Theorem 2.1. *If $DSG(H)$ is acyclic and T_i and T_j are two committed transactions in H that write object x, then the serialization windows of T_i and T_j on x do not overlap.*

[2]We extend this definition in Appendix **??** to transactions that only write to x.

Proof Sketch. First, consider the case of x_i immediately preceding x_j in the version order and T_j reading x_k (as in Figure 1). If $x_k \neq x_i$, then either x_i is before x_k in the version order or vice versa. In the former case, x_j must precede x_k in the version order because x_i and x_j are directly next to each other. This implies there is a cycle $T_j \xrightarrow{ww} T_k \xrightarrow{wr} T_j$. In the latter case, there is a cycle $T_\ell \xrightarrow{ww} T_j \xrightarrow{rw} T_\ell$ involving the transaction T_ℓ that installs the version x_ℓ that immediately follows x_k in the version order. Both cases contradict the hypothesis that $DSG(H)$ is acyclic, so x_k must equal x_i. This trivially implies that T_i's serialization window ends before T_j's serialization window begins, since they begin and end respectively at the same point in time ($w_i(x_i)$).

If x_i does not immediately precede x_j, then the same reasoning can be applied inductively to the serialization windows of the transactions that created the totally ordered sequence of object versions between x_i and x_j. □

We provide a complete proof in Appendix **??**. Note that non-overlapping serialization windows are necessary, but not sufficient, for serializable execution.

Serialization windows offer a general, yet precise characterization of the throughput limitation that sequential execution imposes. Since serialization windows of committed transactions for the same object cannot overlap in time, the length of serialization windows in a system determine an upper bound on the number of serialization windows of committed transactions that can manifest for the same object in a fixed period of time. Thus, a system's throughput for processing transactions that make conflicting accesses to the same object is bounded by the inverse of the length of its serialization windows. For example, replicated OCC and 2PL have relatively long serialization windows because they buffer writes until commit, which occurs after a round of communication to at least a majority of replicas.

2.2 Read Validity

Besides simulating sequential execution, serializable systems must uphold the abstraction of a failure-free store: they need to ensure that committed transactions only observe the effects of committed transactions, a property commonly referred to as *read validity*.

This property is trivially guaranteed by CCs that only expose committed writes to readers, such as OCC or 2PL. To understand how read validity limits the throughput of serializable systems, we examine the concurrent execution of TPC-C `Payments` in multi-version timestamp ordering (MVTSO) [9, 14, 26, 44, 45, 52], a CC that exposes both committed and uncommitted writes to readers.

Figure 2 shows an execution of T_1 and T_2 in a replicated implementation[3] of MVTSO. T_1 reads the value of x written

[3]A basic extension of MVTSO as described in the literature [9] that uses validation to ensure that a set of replicas agree on the order of writes with respect to reads.

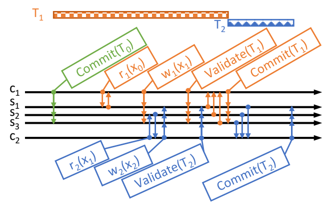


Figure 2. `Payments` in replicated MVTSO.

by a previous transaction T_0, and, to guarantee read validity, waits for T_0 to commit before validating. Likewise, T_2 reads from (and forms a dependency on) T_1's write. Due to this dependency, T_2 waits for T_1 to commit before validating and committing. If T_2 eagerly validates and commits, the system may violate read validity if T_1 subsequently fails to commit.

Read validity, when combined with the sequential execution requirement of serializability, restricts the order in which transactions can commit. This limits how quickly a chain of dependent transactions can commit. We introduce the notion of *validity windows* to quantify this limitation.

Formal Definitions. A history H satisfies *read validity* if for every read $r_i(x_k)$ from a committed transaction T_i, T_k is not aborted. In a real implementation, H satisfies read validity if and only if for every read $r_i(x_k)$ from a committed transaction T_i, T_k is committed and T_k commits before T_i [1]. This is typically referred to as *recoverability* [9].

If a transaction T_i reads version k of object x ($r_i(x_k)$) and writes version i of x ($w_i(x_i)$), T_i creates a *validity window on* x that starts at c_k and ends at c_i. T_i's validity window on x starts when its dependency commits and ends when it commits. [4]

Validity windows on the same object cannot overlap in a system that provides both read validity and sequential execution. We prove the following in Appendix **??**:

Theorem 2.2. *If $DSG(H)$ is acyclic, H satisfies read validity, and T_i and T_j are two committed transactions in H that write object x, then the validity windows of T_i and T_j do not overlap.*

Like serialization windows, validity windows offer a precise characterization of the throughput limitation that read validity imposes in conjunction with sequential execution. A system's throughput for processing transactions that make conflicting accesses to the same object is bounded by the inverse of the length of its validity windows. Thus, a system that processes such transactions at the rate of this bound can achieve higher throughput by reducing the length of its serialization windows and validity windows.

Unlike serialization windows—which can overlap in an execution as long as one of the involved transactions does not commit—validity windows are only defined for committed

[4]We extend this definition in Appendix **??** to transactions that only write to x.

transactions, as their end points correspond to their associated transactions' commit events. A system can seek to avoid overlapping serialization windows of uncommitted transactions to reduce wasted work and idle periods, but there is no analogous goal for validity windows. Instead, the sole performance concern of a serializable CC with respect to read validity is the length of its validity windows.

3 Transaction Re-Execution

To maximize system performance, a serializable CC should ensure that (i) serialization windows are small and not overlapping; and (ii) validity windows are small. These constraints are hard to satisfy efficiently for *interactive transactions*, where the application server executes transactions incrementally using a conversational API (e.g., ODBC) interspersed with application processing. This type of transaction is favored by developers [37], but it prevents a system from knowing a transaction's full access set a priori. Further, asynchrony prevents systems from reliably predicting when outstanding accesses will complete. In this section, we highlight the limitations of existing approaches to providing transactions under asynchrony (§3.1) before introducing *transaction re-execution* to address those shortcomings (§3.2).

3.1 Existing Approaches

3.1.1 Abort & Retry. Existing systems that support interactive transactions immediately process accesses as they are received from the application; the CC then aborts transactions whose reads cause their serialization windows to overlap with that of another transaction. Under high contention, this approach can cause livelock, with transactions repeatedly aborting. Instead, most applications enforce randomized exponential backoff [30]: clients wait a randomized, exponentially growing amount of time before restarting an aborted transaction. Doing so eventually minimizes the likelihood that a transaction's read generates a serialization window that overlaps with the window of another transaction.

Randomized exponential backoff, however, is a rather large hammer applied to a problem that instead benefits from precision. Exponentially increasing the expected times between attempts can introduce artificially long serialization windows where much of the span of a serialization window is from the application server waiting to issue an uncontended read. This limits the maximum throughput of a system even when physical resources are not bottlenecked. For example, in our evaluation of TAPIR (§5), the average CPU utilization of storage servers on a high contention workload at maximum saturation is only about 17%.

3.1.2 Deterministic Databases. Deterministic systems avoid all non-determinism when scheduling operations by pre-ordering transactions [20, 22, 48]. Once the transaction's position in the total order is durably logged, it is forwarded

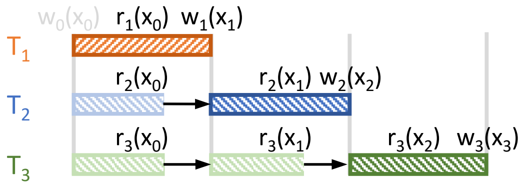


Figure 3. Transaction re-execution.

for execution to the scheduling layer, which then deterministically executes transactions in an order equivalent to the one in which they are logged. As the ordering is known a priori, transactions never read values that cause serialization windows to overlap. Consequently, retries from aborts do not occur and serialization windows are kept relatively short since they do not contain idle periods between retries. Similarly, as transactions commit in a pipelined fashion before being executed, validity windows are also small.

These performance benefits come at the cost of limiting the expressivity of the transactional API: transactions must be written as *stored procedures*, with the transaction's entire program logic submitted on invocation and and stored in the database itself. This tradeoff is unacceptable for most applications [37], as it adds to the developer's burden and complicates deploying updates to the application logic.

3.2 Re-Execution

In this chapter, we ask: can we develop a mechanism that (i) prevents serialization windows from overlapping while minimizing idle time gaps within them; and (ii) minimizes validity windows, all while preserving the expressivity of interactive transactions? To answer these questions, we propose a *transaction re-execution* mechanism that initially schedules transactions best effort, but can dynamically and partially restart execution when overlaps do occur.

In a nutshell, transaction re-execution works as follows. Whenever the serialization windows of two transactions T and T' overlap, transaction re-execution resolves the overlap by changing the read value of T to T''s write, thereby shifting T's window forward. There are two benefits to this approach: (1) it prevents windows from overlapping while ensuring that transactions are processed continuously, without gaps; and (2) it shifts windows locally: re-execution occurs at the granularity of an object (not the full transaction) and thus only requires re-executing operations that access or depend on that particular object. Consider, for instance, Figure 3. Initially, there are three transactions, T_1, T_2 and T_3, whose serialization windows pairwise overlap. Re-execution first shifts the reads of T_2 and T_3 to observe $w_1(x_1)$; and then the read of T_3 once more, after T_2 completes its write.

Two core ideas drive the feasibility of re-execution: *read unrolling* and *a priori ordering*. Below, we briefly discuss these two pillars of re-execution.

Read Unrolling. Transaction re-execution shifts reads forward in time by invalidating the current values read in a given execution and replacing them with others, produced by newer writes. In doing so, however, the read no longer logically corresponds to the ongoing application's thread of execution. The application logic, based on the old value, may have subsequently issued several dependent operations. To avoid inconsistencies, transaction re-execution must provide a means for *unrolling* the effect of prior reads (and all possible dependencies), as well as the ability to partially restart execution in a way that is transparent to the application.

A Priori Ordering. Deterministic databases leverage predefined schedules to streamline execution; while interactive transactions cannot be fully scheduled in advance, determinism can simplify scheduling. By assigning to all transactions a speculative serialization order a priori, overlapping serialization windows are easily identified at runtime: pairwise reads and writes to an object x that appear out of the speculative order induce overlapping serialization windows.

4 Morty Design

Morty is a replicated transactional key-value store explicitly designed to minimize the overlap of serialization windows (§2). Morty's properties and performance rest on two basic mechanisms. First, transaction re-execution (§3), which allows it to realign serialization windows that would otherwise overlap; second, concurrency control and replication techniques that minimize the length of serialization windows and validity windows, especially in geo-replicated settings. The combination of these techniques allows Morty to achieve higher throughput on high contention workloads than existing systems (§5) without sacrificing either strong consistency (serializability) or generality (interactive transactions).

System Model. Morty assumes an asynchronous system, where message delivery and local processing may be delayed for arbitrarily long. Up to f out of $n = 2f + 1$ Morty replicas and any number of its clients may fail by crashing, i.e. permanently cease to send and receive messages. For simplicity, we assume reliable and FIFO message delivery; these properties may be implemented in an unreliable network using retransmissions and message sequence numbers.

Structure. In the rest of this section we describe how Morty implements the two pillars of transaction re-execution (§4.1). Next we walk through a full execution of a transaction in Morty (§4.2). Finally, we discuss how Morty handles failures (§4.3) and garbage collection (§4.4).

4.1 Implementing Re-Execution

4.1.1 Unrolling Reads with a Continuation-based API.

The first pillar of transaction re-execution is the ability to undo the effects of previously completed reads, whether by recomputing intermediate transaction state or by retracting or reissuing operations dependent on those reads.

A simple and general way to rewind the effects of completed reads is to provide the application's logic as input to the system; the effects of undoing a read can then be precisely determined by re-executing that logic with the new read value. To achieve this capability, prior work has required applications to limit themselves to expressing transactions either as stored procedures [50], renouncing interactivity, or via a separate domain-specific language [15], imposing an additional burden on developers [37]. Morty manages to avoid these drawbacks by adopting a different approach: a *continuation passing style* (CPS) API.

Continuation-based API. In CPS, the control flow of a program is specified entirely as function calls. Each function takes a *context* and *continuation* argument. The context stores the program's current state, and the continuation specifies where the program continues executing after finishing the current function. Morty's API mirrors a traditional, imperative API, but adds a context parameter to each database operation; in addition, calls to GET and COMMIT also include a continuation parameter, which defines where to return control (i.e., which logical block to execute next) after completing the database statement.

CPS is widely used for writing asynchronous programs across many languages (JavaScript, Go, C++, Java, Python) and frameworks (NodeJS, LibEvent [29], Tokio Async [41]). It is a good match for networked databases, such as Morty, where the results of GET and COMMIT operations are only available after calls to the network.

We emphasize that networked applications are often already written with the CPS API, and in such cases, Morty imposes no burden on the application developers to rewrite their applications. For example, Microsoft's FaRM transactional system [17] uses the CPS API, and thus, applications written for FaRM could run on Morty with few changes.

Nevertheless, moving traditional imperative code to CPS can be fully automated with the help of a compiler [5, 23]. While Morty does not currently support this capability, our experience suggests that the effort involved in hand-coding such transformations is relatively minor. For example, Figure 4 shows a simplified TPC-C Payment transaction written in an imperative C++ transactional API (Figure 4a) and in Morty's CPS (Figure 4b).

It's all in the context! CPS mostly hides from the application developer the complexity of supporting re-executions. By simply storing old contexts in the client library, Morty can automatically rewind the current execution and re-execute a continuation with a new return value, leaving the application or user none the wiser. No additional effort is asked of the developer beyond what is required in a system that may abort and retry transactions.

```
bool ProcessPayment(uint w_id, uint amt) {
 client.Begin();
 auto wh = client.Get("warehouses", w_id);
 wh.SetCol("ytd", wh.GetCol("ytd") + amt);
 client.Put("warehouses", w_id, wh);
 return cli.Commit();
}
```

(a) Traditional: operations implicitly ordered by program order.

```
void ProcessPayment(uint w_id, uint amt,
    continuation_t cont) {
 auto ctx = make_ptr<PaymentCtx>();
 client.Begin(ctx);
 client.Get(move(ctx), "warehouses", w_id,
   [&client, &cont](ptr<PaymentCtx> ctx,
     string val){
    auto wh = ParseWarehouse(val);
    wh.SetCol("ytd", wh.GetCol("ytd") + amt);
    client.Put(ctx, "warehouses", w_id, wh);
    client.Commit(move(ctx), cont);
 });
}
```

(b) CPS: explicit continuations define control flow dependencies.

Figure 4. Payment in traditional (4a) & CPS (4b) APIs.

4.1.2 Pre-Determining an Order with MVTSO.

The second pillar of Morty's transaction re-execution is to execute transactions in a pre-determined order. To this end, Morty adopts MVTSO as its concurrency control protocol. Each transaction is assigned a timestamp when it enters the system; the timestamp determines the transaction's position in a total order. The read, write, and commit protocols attempt to execute transactions in this predefined order by ensuring that the timestamp of the write observed by a read precedes that of the read and follows that of all other writes visible to the read. However, MVTSO can only approximates the perfect deterministic ordering of deterministic databases. Nodes' clocks are only loosely synchronized, and the system still experiences non-deterministic ordering from the network and processors. Thus, a transaction's read may miss the correct write from another transaction whose assigned timestamp was too small (because of clock skew) or whose write arrived too late (because of asynchrony). If in some edge cases these circumstances may force one of the transactions to abort, Morty demonstrates that, in most cases, re-execution reduces throughput loss by allowing both transactions to commit.

4.2 Transaction Execution

In the following, we detail Morty's transaction execution protocol. Morty encapsulates the state and metadata of an executing transaction in a *transaction execution* (or *execution*). Since Morty supports transaction re-execution, multiple executions of the same transaction may exist over the lifespan of a transaction. Thus, the coordinator of a transaction assigns a unique *eid* to each execution that it creates. Figure 5 summarizes the state maintained at each replica.

vstore - map from *key* to a *vrecord* struct:
  reads - set of uncommitted (*ver*, *last_reply*)
  writes - set of uncommitted (*ver*, *val*)
  prepared_reads - set of prepared (*ver*, *exec_id*, *r_ver*)
  prepared_writes - set of prepared (*ver*, *exec_id*)
  committed_reads - set of committed (*ver*, *r_ver*)
  committed_writes - set of committed (*ver*, *val*)
decision_log - map from *ver* to *Commit*/*Abort* decision
erecord - map from (*ver*, *exec_id*) to a struct:
  vote - validation vote cast by replica
  view - view in which replica is prepared to accept finalize decisions
  finalize_decision - accepted commit decision
  finalize_view - view in which replica accepted *finalize_decision*
  decision - learned *Commit*/*Abandon* decision

Figure 5. State at each replica.

Begin(ctx). The coordinator starts a transaction T by assigning it a unique version $ver = (ts, id)$ based on its loosely synchronized local clock ts and unique coordinator identifier id. This version defines T's expected position in a total order for all transactions. The initial execution's *eid* is 0. The coordinator also initializes data structures that will store metadata for the transaction execution. For convenience, these are stored directly in the application *ctx* so that subsequent transaction operations can easily access the metadata associated with the application's current context.

Get(ctx, key, cont). The coordinator creates a mapping between this Get request and the application continuation *cont* in order to call the continuation when the Get request completes. It then sends a *Get*(*ver*, *key*) message to a single replica: in geo-replicated deployments, this minimizes Get latency, as the coordinator (in the common case) can contact the closest replica.

Upon receiving a *Get*, a replica determines the return value by selecting from *key*'s *vrecord* the write with the largest version ver' smaller than *ver*. It then sends to the coordinator a *GetReply*(ver', *val*) message containing the write value *val*, adds the read to the *vrecord*, and records ver' and *val* as the most recent write replied for the read.

When the coordinator receives a *GetReply* it adds (*key*, ver', *val*) to the *read_set* of the execution. Then, the coordinator calls *cont*(*val*) to return the value and control to the application.

Put(ctx, key, val). The coordinator adds (*key*, *val*) to the *write_set* of the execution. It then asynchronously broadcasts a *Put*(*ver*, *key*, *val*) message to all replicas and returns control to the application.

When a replica receives a *Put*, it adds the write to the *vrecord* for *key*. Next, the replica determines whether any read in the *vrecord missed* this new write. A read misses a write if the replica, had it processed the read after the write, would have replied to the read with the value of that write. A read miss happens when the read's version is smaller than *ver* and one of two conditions is met: (i) the version of the most recent write replied for the read is smaller than *ver*; or (ii) the read already observed a write with *ver*, but with a different value. The latter case is possible when re-executing an earlier read in the transaction changes the write value.

The replica sends a new *GetReply*(*ver*, *val*) to the coordinator of any read in a key's *vrecord* that missed the write.

Re-Execution. Upon receiving a *GetReply*, the coordinator considers re-executing T only if T's *current* execution includes a read request that would have prompted that reply. This condition may not be met if the coordinator already had already initiated re-execution and is now operating on an execution branch that either no longer includes the request to read, or is yet to invoke it. To make this determination, the coordinator defines and stores a *reads execution history* within the application-provided context *ctx*. It also maintains a *current context* for the execution that most recently invoked an operation. Only those replies whose reads execution history is a prefix of the execution history of the current context trigger re-execution.

To re-execute T's read and return the new write value to the application, the coordinator uses the copies of T's *ctx* and *cont* that it is storing to implement the CPS asynchronous *Get* calls; supporting re-execution simply requires retaining these copies, for each *Get* of the current execution, until T completes. The coordinator retrieves the stored *ctx* and *cont* that correspond to the read that is to be repeated, and calls the continuation with the new read value.

Commit(ctx, cont). Morty, as in prior work [45, 46, 54], integrates concurrency control with replication to reduce commit latency. The commit protocol requires up to three phases. In the *Prepare* phase, the coordinator requests that all replicas vote on whether or not the transaction execution is serializable. If all replicas agree, the decision is durable; the coordinator immediately performs the *Decide* phase and returns to the application. Otherwise, an intermediate *Finalize* phase is necessary to explicitly make a decision durable before proceeding to the Decide phase.

Abort vs. Abandon. A commit protocol determines one of two possible decisions for a transaction: *Commit* or *Abort*. In Morty, however, the same transaction can trigger multiple re-executions, some of which may start after the transaction's commit protocol has already begun. Thus, Morty refines the commit protocol to operate at the granularity of individual executions of a transaction. Each transaction execution reaches one of two decisions: *Commit* or *Abandon*; these elemental decisions in turn determine the transaction's decision value. For a transaction to commit, at least one of its executions must commit; for it to abort, all of its executions must be abandoned.

Prepare. The coordinator begins the Prepare phase for an *execution* of transaction T with (*ver*, *eid*) by broadcasting *Prepare*(*ver*, *eid*, *read_set*, *write_set*) to replicas.

When a replica receives a *Prepare*, it creates an entry in its *erecord*. Before voting on whether the execution is serializable, the replica checks that all of T's read dependencies are committed. If any version in *read_set* was written by an aborted transaction, the replica votes *Abandon-Final*. Otherwise, if any version in *read_set* is not committed, the replica waits to learn a decision for the corresponding transactions before continuing to process this *Prepare*.

Serializability validation involves four checks:

1. Check that the execution's reads did not miss any writes (§4.2). If a read missed an uncommitted write, the replica votes *Abandon-Tentative*; if a read missed a committed write, the replica votes *Abandon-Final*.
2. Check that other transactions' reads did not miss T's writes. If a committed transaction missed a write from T, the replica votes *Abandon-Final*. Otherwise, if a tentatively prepared transaction missed a write, the replica votes *Abandon-Tentative*.
3. Check for *dirty reads*: a replica confirms that every *ver* and *val* in *read_set* exactly matches a committed write. If not, the offending read must have read from an abandoned execution of a transaction. Therefore, the replica votes *Abandon-Final*.
4. Check that the execution did not read from any truncated transactions, and that the transaction execution itself is not truncated (§4.4). Otherwise, the replica votes *Abandon-Final*.

The first two checks are standard in MVTSO; the third ensures that committed executions only read valid data; finally, the fourth ensures that the execution is validated against committed transactions that have been garbage collected.

If the execution passes all validation checks, the replica *prepares* its reads and writes and votes to *Commit*. In all cases, the replica sends a *PrepareReply*(*vote*) message to the coordinator. If the replica determines that the execution missed a write, it additionally sends a *GetReply* containing the write.

Since at most f replicas are faulty, the coordinator waits to receive at least $f + 1$ *PrepareReplies*. It then determines (i) the decision for the current execution (and, if appropriate, for the corresponding transaction), and (ii) whether or not the decision is durable. Table 1 summarizes how the coordinator aggregates replica votes. An execution of T (and, as a result, T itself) commits only if at least $f + 1$ replica vote to commit: this guarantees that no two conflicting executions

Decision	Skip Finalize?	Need Finalize?
Commit	$2f+1$ *Commit*	$f+1$ *Commit*
Abandon	1 *Abandon-Final*	≥ 1 *Abandon-Tentative*

Table 1. The coordinator aggregates votes and determines a final decision based on the number and types (*Commit*, *Abandon-Tentative*, *Abandon-Final*) of votes.

can both commit, and thus the set of committed transactions is serializable. A decision is considered *durable* if it can be reconstructed from the information stored at any set of $f+1$ replicas. If this is not the case, an untimely failure of T's coordinator may lead a *recovery coordinator* (§4.3) to a different decision from that of T. To avoid this scenario, the coordinator performs an additional Finalize phase.

Finalize. The Finalize phase uses consensus to ensure that replicas agree on the decision for the execution despite coordinator failures. It resembles single-decree Paxos [25] in that the decision for the transaction execution is treated as a write-once register whose value, once determined, will remain unchanged [28]. This is implemented via replicas accepting a *finalize_decision* proposed by coordinators for a *view*. Specifically, the coordinator broadcasts a *Finalize*(*ver*, *eid*, *view*, *decision*) message to all replicas. Upon receiving it, a replica checks in the *erecord* for (*ver*, *eid*) whether its view *view′* is the same as *view*. If so, the replica records *decision* as its *finalize_decision* and sends back a *FinalizeReply*(*view′*) message. The coordinator waits to receive $f+1$ such replies. If they are for the *view* sent by the coordinator, the decision is durable. Otherwise, a recovery coordinator is concurrently attempting to Finalize a decision for the execution and the coordinator itself must perform recovery (§4.3).

Decide. The Decide phase confirms for replicas that the decision for execution *eid* of T (and, if warranted, for T itself) is durable. It also indicates that state associated with T can be safely garbage collected. We discuss garbage collection later (§4.4); for now, we focus on the other actions that a replica takes upon learning a durable decision for (T, *eid*).

To start this phase, the coordinator broadcasts a *Decide*(*ver*, *eid*, *decision*, *abort*?) message to all replicas. Although *decision* applies to *eid*, if it is *Commit*, then the decision's scope extends to T as well. Instead, a *decision* that is *Abandon* applies only to the current execution. However, if the coordinator determines that this is T's only outstanding execution, it sets the *abort*? to True to indicate its decision that T must *Abort*.

When a replica receives a *Decide* with a *Commit* decision, it logs the *Commit* decision in the *erecord* for (*ver*, *eid*) and adds (*ver*, *Commit*) to the *decision_log*. It also adds the *read_set* and *write_set* of the execution to *committed_reads* and *committed_writes* of the appropriate *vstore* entries. This metadata is used for validating future conflicting transactions and is retained until it can be safely garbage collected.

If the *Decide* includes a decision to *Abandon* the execution, but not one to *Abort*, the replica logs the *Abandon* decision in the *erecord* for (*ver*, *eid*) and erases all *prepared_reads* and *prepared_writes* associated with (*ver*, *eid*) in the *vstore*, while retaining all *reads* and *writes* associated with *ver*. This allows subsequent executions of T to continue executing or committing. If *Decide* additionally indicates that T must abort, the replica adds (*ver*, *Abort*) to the *decision_log* and generates new *GetReplies* for all reads that observed T's writes.

Lastly, if the *Decide* implies a *Commit* or *Abort* decision for T (i.e., not just an *Abandon* decision for the current execution), the replica checks whether suspended *Prepares* that depend on T may now move forward.

Commit & Re-Execution. A re-execution for T may be triggered after the commit protocol for T's current execution has already begun. In fact, for geo-replicated deployments, it is during the commit protocol that re-executions are most likely triggered, since it is the first phase that requires a message exchange with at least $f+1$ replicas.

To avoid committing multiple executions from the same transaction, a coordinator abandons all previous executions before attempting to commit its current re-execution. To abandon an execution (*ver*, *eid*) that has reached the commit protocol, a coordinator broadcasts *Finalize*(*ver*, *eid*, 0, *Abandon*) messages to all replicas. In the absence of contending recovery coordinators (§4.3) , $f+1$ replicas accept the *Abandon* decision in view 0, making the decision durable. This message exchange also unprepares any prepared reads and writes from (*ver*, *eid*) – clearing the way for the coordinator's re-execution to proceed through the commit protocol. If the coordinator's *Abandon* proposal fails to be accepted by $f+1$ replicas—because of a concurrent recovery coordinator—the coordinator recovers that decision and proceeds accordingly.

4.3 Handling Failures

Morty tolerates up to f failures among its $2f+1$ replicas. However, the failure of a coordinator poses a potential liveness issue: a transaction that stalls in the middle of its commit protocol may prevent conflicting transactions from committing. Furthermore, transactions that read from a stalled transaction must wait until a decision is reached. Inspired by recent work [45, 46, 54], Morty's coordinator recovery protocol empowers any node in the system to recover a durable decision for a failed coordinator's transaction.

Recovery Protocol. The recovery protocol, like the Finalize phase (§4.2), uses consensus to ensure that a single decision is reached for a transaction execution. Unlike coordinators performing the Finalize phase, a recovery coordinator for an execution *eid* of a transaction with version *ver* must enact a view change to a unique *view′* larger than any previous view by broadcasting a *PaxosPrepare*(*ver*, *eid*, *view′*)

message to all replicas. When a replica receives a *PaxosPrepare*, it checks in the execution's *erecord* entry whether *view′* is larger than its current view *view*, in which it previously promised to not accept decisions in smaller views. If so, the replica updates *view* to *view′*. It then sends a *PaxosPrepareReply*(*view*, *decision*, *finalize_view*, *finalize_decision*, *vote*) to the recovery coordinator.

To propose a durable decision, the recovery coordinator must receive $f + 1$ replies from replicas agreeing to change to *view′*. The actual decision depends on the contents of the replies. If any reply already contains a learned decision, the recovery coordinator simply performs the Decide phase and terminates. Otherwise, it performs the Finalize phase using *view′* and either (i) the *finalize_decision* from among all replies with the highest *finalize_view*, or (ii) if no *finalize_decision* exists, a new decision based on the Prepare phase rules (Table 1). The Finalize and subsequent Decide phase proceed as in normal transaction execution.

4.4 Garbage Collection & Truncation

To be practical, Morty replica state must not grow asymptotically faster than the number of objects stored in the system. This is ensured by a series of garbage collection procedures and a related truncation procedure.

Decide Garbage Collection. Part of the *vstore* is garbage collected when a *Commit* or *Abort* decision for an execution $(T, exec\_id)$ is learned. The uncommitted *reads* with version $ver(T)$ are no longer needed for re-executing T, since T has a durable decision. Similarly, the uncommitted *writes* with version $ver(T)$ are either visible to other transactions as part of *committed_writes* (in the case of *Commit*) or should no longer be visible to any transaction (in the case of *Abort*). Furthermore, regardless of the Decide decision, the *prepared_reads* and *prepared_writes* with $ver(T)$ and matching *exec_id* may be garbage collected.

Truncation. Garbage collection of the *erecord* is more complicated as this state is used to ensure that at most one durable decision is reached for each transaction execution. Morty safely truncates the *erecord* with a truncation protocol, initiated by a *truncation coordinator*, which attempts to establish a durable *truncation_ver* that summarizes all committed state from transactions with smaller versions. Once a safe *truncation_ver* is determined, replicas stop responding to requests for transactions with smaller versions.

The truncation protocol is comprised of the following steps:

1. When the system starts, it establishes truncation versions based on the loosely synchronized clocks of the replicas. These versions are spaced by a configurable amount of time to control how frequently truncation occurs.
2. A replica times out when a configurable amount of time has passed since the most recent truncation version *truncation_ver*. At this time, it stops processing transactions with versions smaller than *truncation_ver* (e.g., by responding *Truncated* to all related messages). In addition, it sends to a pre-established truncation coordinator a *Truncate*(*truncation_ver*, *erecord*) message containing a snapshot of its current *erecord* for transactions with versions smaller than *truncation_ver*.
3. When the truncation coordinator receives $f + 1$ *Truncates* for *truncation_ver*, it merges the *erecords* using the existing voting and coordinator decision procedures. The merging process must maintain the invariant: if a decision could have been reached for a transaction T in one of the constituent *erecords*, then that decision is preserved in the *merged_erecord*. Then the truncation coordinator proposes a consistent *merged_erecord* for this truncation version by broadcasting a *ProposeMerge*(*truncation_ver*, *truncation_view*, *merged_erecord*) message to all replicas.
4. Upon receiving a *ProposeMerge*, a replica checks whether its *truncation_view* is the same as *truncation_view*. If so, the replica records *truncation_view* as *truncation_accept_view* and *merged_erecord* as *truncation_accept_erecord*. In either case, it sends a *ProposeMergeReply*(*truncation_view*) to the truncation coordinator.
5. When the truncation coordinator receives $f+1$ *ProposeMergeReplies* with the same *truncation_view*, the truncation decision is durable and the coordinator informs all replicas of the consistent durable *erecord* by broadcasting a *TruncationFinished*(*truncation_ver*, *merged_erecord*) message.
6. Upon receiving a *TruncationFinished*, a replica applies the *merged_erecord* to its own *erecord*, overwriting any existing metadata for transactions from *merged_erecord*. At this point, it also raises its watermark *truncation_ver* to allow truncated metadata to be garbage collected. Additionally, as part of serializability validation, it thenceforth rejects any transaction executions with versions smaller than *truncation_ver*.
7. If the truncation coordinator receives a *ProposeMergeReply* with a different *truncation_view*, it attempts a view change to a higher view by broadcasting *TruncationPaxosPrepare*. Once $f + 1$ replicas agree to change to the higher view, the coordinator repeats steps 3–5.

Truncated Garbage Collection. Periodically, state in the *erecord* and *vstore* associated with a transaction T whose version $ver(T)$ is smaller than *truncated_ver* may be deleted. Specifically, the entire struct associated with any execution of T may be deleted from *erecord*. In addition, any *committed_reads* and *committed_writes* from T in *vstore* may be deleted, as the truncation check during validation ensures that transactions that would need to be checked against these deleted reads and writes are not allowed to commit.

4.5 Correctness

Using Adya's model of a transactional storage system, the following Theorem holds:

Theorem 4.1. *Morty only produces serializable histories.*

Proof Sketch. Consider a history H produced by Morty. The proof that H is serializable consists of two parts: (i) showing that $DSG(H)$ is acyclic and (ii) showing that committed transactions in H only read valid data.

The proof of (i) reduces to showing that the directions of the edges of $DSG(H)$ are consistent with the version order of transactions, which is a total order. Consider a write-write edge whose direction is determined by the object version order $\ll$. We define $\ll$ to be consistent with the version order of transactions, so the edge direction is trivially consistent with the version order. Similarly, consider a write-read edge: Morty only returns values for reads such that the version of the write value is smaller than the version of the reading transaction, so the edge direction is always consistent with the version order of transactions.

Proving the consistency of a read-write edge $T_i \xrightarrow{rw} T_j$ —where T_i reads some object version x_k and T_j installs the next version after x_k— requires reasoning about the order in which the replicas of the group that stores x perform the validation checks for T_i and T_j. Regardless of whether or not T_i and T_j commit on the fast path, slow path, recovery path, or truncation path, they each must pass the validation check at more than $f + 1$ replicas. This implies that at least one replica validates both T_i and T_j. If the replica validates T_i first, then T_j can only validate successfully if $ver(T_i) < ver(T_j)$. Otherwise, if the replica validates T_j first, then T_i can only validate successfully if $ver(T_i) < ver(T_j)$. The truncation check during validation ensures that this invariant holds even after committed data is truncated.

The proof of (ii) relies on the dirty read check of the validation check and the fact that transaction coordinators only attempt to commit a single execution of a transaction that is produced by the application logic. The former ensures that committed transaction only read from transactions which have been committed and the latter ensures that the only transactions which are committed are those that correspond to a single transaction invocation by the application. □

The full proof of Theorem 4.1 is in Appendix **??**.

5 Evaluation

Our evaluation answers the following questions:

- How do Morty's throughput and latency compare to state-of-the-art systems on high-contention OLTP workloads? (§5.1)
- To what extent do additional CPU resources help Morty (and the baselines) scale throughput? (§5.2)
- How do varying levels of contention affect Morty's throughput (relative to the baselines)? (§5.3)

Our code and experiment scripts are open source [32].

Baselines. We quantify Morty's performance against three baselines: (*i*) TAPIR [54], a state-of-the-art serializable storage system with interactive transactions that uses OCC; (*ii*) Spanner [12], Google's distributed, strictly serializable database that uses 2PL [9] for CC and wound-wait [40] for deadlock prevention; and (*iii*) a replicated implementation of MVTSO inspired by recent work [45, 46] that reuses Morty's replication and execution logic, but does not employ re-execution. We implement Morty, Spanner, and MVTSO in TAPIR's codebase to minimize implementation differences and provide a fair basis for the performance implications of each system's design choices. All systems use TCP for communication, `libevent` for asynchronous I/O, and `libprotobuf` for serialization. Replicas in Morty and MVTSO are multi-threaded; not so in TAPIR and Spanner, as we do not modify their single-threaded replication libraries. To compensate, when measuring system capacity, we configure TAPIR and Spanner with additional replica groups to match the number of cores used by Morty and MVTSO.

Our Spanner implementation is faithful to its documented design, except that we reuse the implementation of viewstamped replication [36] in TAPIR's codebase instead of implementing Multi-Paxos [25]. Notably, we implement several features of Spanner that provide a performance advantage over Morty, as these features are integral to attaining practical performance with Spanner. To support Spanner's non-blocking read-only transactions, we emulate TrueTime with an error of 10ms, the p99.9 value observed in practice [12]. Finally, to better support transactions that read and modify the same key, we implement Spanner's `GetForUpdate`. Although this feature is not in the original description of Spanner's protocol [12], it has since been added [11].

Setup. We run experiments on CloudLab [18] using `c220g5` machines in the Wisconsin cluster. Each machine has two 10-core Intel Xeon CPUs at 2.20 GHz, 192GB of memory, and one Dual-port Intel X520-DA2 10Gb. All experiments use $n = 3$ replicas per replica group, tolerating $f = 1$ replica failures, and use up to six machines for clients, which run as single-threaded applications and send requests to storage in closed loops. Morty and MVTSO use one replica group, while TAPIR and Spanner use 20. Each client is logically co-located with some replica (simulating a local datacenter) and each replica is loaded with the same number of co-located clients. Clients use local replicas for reads, except in Spanner, where clients read from group leaders.

RTT	us-east-1	us-west-1	us-west-2	eu-west-1
us-east-1	0	62ms	68ms	68ms
us-west-1	62ms	0	22ms	138ms

Table 2. Cross-region RTTs in emulated networks.

Transaction	**Characteristics**	**Mix**
New-Order	Medium Read-Write	44%
Payment	Short Read-Write	44%
Delivery	Short Read-Write	4%
Order-Status	Short Read-Only	4%
Stock-Level	Long Read-Only	4%

(a) Transaction mix in TPC-C workload.

Transaction	**Reads**	**Writes**	**Mix**
Add-User	1	2	5%
Follow/Unfollow	2	2	15%
Post-Tweet	3	5	30%
Load-Timeline	[1,10]	0	50%

(b) Transaction mix in Retwis workload.

Measurement. We run all experiments for 90 seconds and exclude measurements from the first and last 15 seconds. We report latency as the time between when the client first begins a transaction and when it is notified that the transaction is committed, including retries after aborts. To avoid livelock, clients perform a random exponential backoff (up to 2.5 s) before retrying. We report *goodput* as the number of committed transactions across all clients over the measurement period.

Network Setup. We use the Linux traffic control (`tc`) utility to emulate wide-area latencies and evaluate the systems across three different network setups. In each case, we replicate the RTTs observed in AWS (Table 2):

1. The regional setup (REG) simulates replicas located in different availability zones of the same region with 10ms inter-replica latency.
2. The continental setup (CON) uses measurements from US-based Amazon Web Services [4] regions (`us-east-1`, `us-west-1`, `us-west-2`) to emulate latencies between replicas in different regions.
3. The global setup (GLO) emulates distributing replicas across the US and Europe and using measurements from AWS regions `us-east-1`, `us-west-1`, and `eu-west-1`.

5.1 OLTP Applications

We evaluate our system against two popular OLTP workloads: (i) TPC-C [49] and (ii) the Retwis-based benchmark [54].

5.1.1 TPC-C. TPC-C is an OLTP workload that simulates an e-commerce service [49]. We run experiments with 100 warehouses, resulting in an initial database size of 8GB; we use the transaction mix of Table 3a, When running with multiple replica groups, we partition all tables, except for `items`, by `warehouse_id`. We replicate the read-only `items` table on each group. We materialize secondary indices with two additional tables that support lookups of orders by customer and of orders with outstanding deliveries [14, 44].

Figure 6 shows goodput and latency for Morty and the baselines as load increases with more clients. In the REG setup (Figure 6a), Morty reaches a maximum goodput of 11.8k txn/s while MVTSO, TAPIR, and Spanner reach only 6.8k, 2.7k, and 1.6k txn/s, respectively.

Morty's higher goodput stems from it re-executing transactions to avoid overlapping serialization windows instead of aborting and retrying. At maximum goodput, Morty's commit rate is over 99%, so very few transaction's serialization windows are artificially elongated by backoff. We measure that Morty performs about 2.9 partial re-executions per transaction on average. Conversely, aborts and retries from overlapping serialization windows in the baselines increase the amount of time between successive writes to contended keys, reducing the number of transactions that can commit in a fixed time period. MVTSO's serialization windows are shorter than TAPIR's because it exposes uncommitted writes; TAPIR's serialization windows are shorter than Spanner's because reads do not need to be processed by a leader replica. Spanner's serialization windows are so long, relative to the other systems, that its latency is an order of magnitude higher. We consequently only show the first three data points in Figure 6a. Its latency at low load is about 151ms, and its maximum throughput is about 1.7k txn/s.

Similar performance trends occur in the CON and GLO setups. Under all three network configurations, Morty achieves approximately 1.7x and 4.4x the goodput of MVTSO and TAPIR, respectively, with similar latency at low to moderate load. Spanner's serialization windows lengthen with the round-trip latencies between datacenters, so Morty's relative advantage increases from 8x to 18x in CON and GLO, respectively. (We omit Spanner's curves in Figures 6b and 6c to allow easier comparison of the other three systems.)

5.1.2 Retwis. Retwis emulates a social network workload with short read-write and read-only transactions and configurable contention. Table 3b shows the transaction types and mix. We configure the database to contain 10M key-value pairs (8B keys and 8B values). Experiments with multiple replica groups use a static hash function to evenly partition keys. Transactions access keys according to a Zipfian distribution with parameter $\theta = 0.9$, modeling a high contention access pattern.

Figure 7 shows goodput and latency measurements for Morty and the baselines. As with TPC-C, the performance trends for the systems remain similar across all three network setups: Morty achieves approximately 28x, 52x, and 96x the maximum goodput of MVTSO, Spanner, and TAPIR, respectively (note that x-axes are in log scale).

Spanner's fairs better than with TPC-C because of Retwis's shorter read-write transactions and more frequent read-only transactions, which do not acquire locks. The former reduce

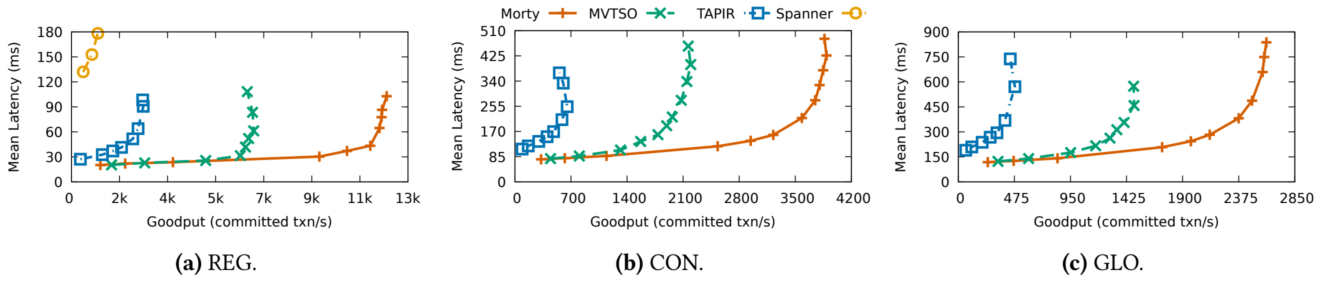


Figure 6. Morty achieves higher goodput at saturation on TPC-C with 100 warehouses.

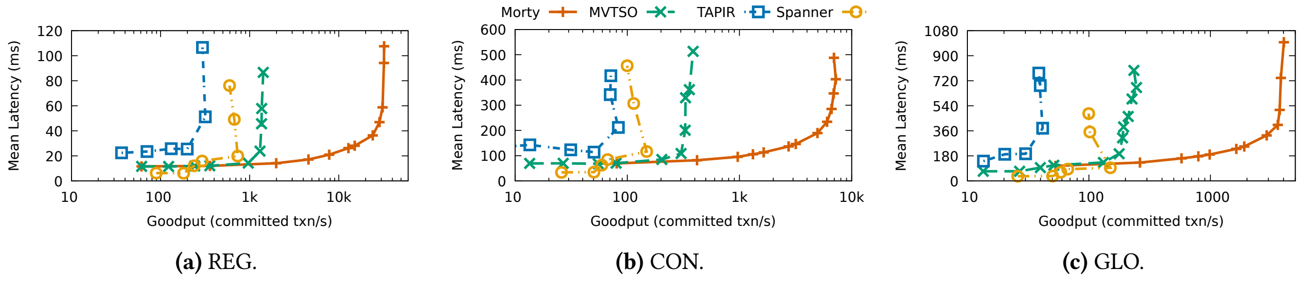


Figure 7. Morty achieves higher throughput at saturation on Retwis with 10M keys and Zipf parameter 0.9.

the number of round trips between clients and group leaders (and thus keeps serialization windows short), and the latter significantly reduce contention.

For REG, (Figure 7a), Morty achieves a maximum goodput of 35.3k txn/s compared to 1.5k, 0.7k, and 0.4k txn/s for MVTSO, Spanner, and TAPIR. Once again, Morty's ability to re-execute and shift serialization windows allows it to avoid aborting most transactions, unlike the baselines.

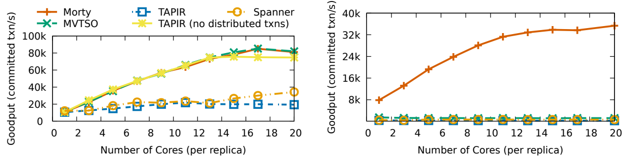
The much larger difference between their peak goodputs and Morty's in Retwis over TPC-C is due to Retwis' higher contention rate. With the Zipfian parameter θ set at 0.9, the probability that two `Post-Tweet` transactions in Retwis both modify the hottest key is at least 2.5%, while two `Payment` transactions modifying the same row in TPC-C conflict with a probability of 1% with 100 warehouses.

5.2 Scalability

To quantify how effectively Morty and the baselines use additional resources to scale goodput, we evaluate their performance on Retwis in the REG setup with an increasing number of server CPUs.

Figure 8 shows the maximum goodput of each system as a function of the number of CPU cores on both a uniform ($\theta = 0$) and Zipfian ($\theta = 0.9$) distribution. Recall that for TAPIR additional cores translate to additional replica groups.

For the uniform Retwis workload (Figure 8a), most transactions do not conflict and additional cores help all systems scale goodput. The codepaths in Morty and MVTSO for execution are nearly identical for non-conflicting transactions, since there are no re-executions. TAPIR and Spanner can



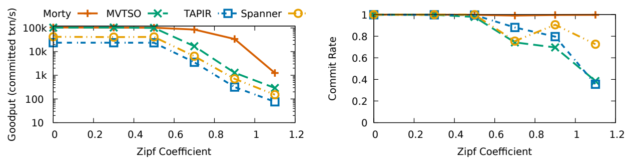
(a) All systems scale with additional cores at low contention.

(b) Morty effectively utilizes additional cores, whereas MVTSO, TAPIR, and Spanner are contention bottlenecked.

Figure 8. Multi-core scalability on Retwis.

also scale goodput despite their single-threaded replication by adding more replica groups; when doing so, there is additional overhead that depends on how frequently transactions span replica groups. For reference, we run TAPIR on a modified uniform workload with no distributed transactions (the best case scenario) and observe similar results: TAPIR can scale with additional cores on a uniform workload.

In contrast, on the heavily-contended Zipfian Retwis workload, only Morty is able to leverage the additional cores to scale its maximum goodput (Figure 8b), from 7.8k txn/s with a single core up to 35.3k txn/s with 20 cores. While Morty leverages additional CPUs to send new *GetReplies* and re-execute, MVTSO, Spanner, and TAPIR remain contention bottlenecked, at 1.5k, 0.7 and 0.4k txn/s, respectively. We emphasize that TAPIR and Spanner's shortcomings here are *not* due to poorer relative CPU utilization per transaction: on the Zipfian workload, nearly every transaction accesses only



(a) Morty's edge over the baselines grows with more contention.

(b) Morty's commit rate remains near 100%.

Figure 9. Varying contention on Retwis.

the hot replica group. We measure that TAPIR and Spanner replicas saturate at most 17% of a single CPU during these experiments because their overlapping serialization windows cause frequent aborts and long exponential backoff periods.

5.3 Microbenchmarks

To better understand the influence that contention has on the performance of these four systems, we measure their maximum goodput and commit rate (an indirect indicator of how often serialization windows overlap) on the Retwis workload for increasing Zipfian parameter θ on the REG network. Figure 9 shows the results. As contention grows, so does the gap in peak goodput between Morty and the baselines (Figure 9a). Though goodput falls when contention on hot keys increases, Morty's near perfect commit rate even under extremely high contention suggests that Morty introduces no unnecessary idle time. As θ grows, instead, transactions in MVTSO, TAPIR, and Spanner abort more often, causing backoffs, longer serialization windows, and falling peak goodput.

6 Related Work

Transaction Re-Execution. Re-execution has been explored by a handful of previous systems, albeit in a more limited fashion. Both TheDB [50] and MV3C [15] make visible only committed values, and thus only trigger re-execution during commit. This increases the length of serialization windows in these systems, both increasing the likelihood of overlap and reducing maximally achievable throughput. In contrast, Morty optimistically makes write values visible as early as possible, shortening serialization windows, and allowing replicas to trigger eager re-execution. Morty's commit and recovery protocols additionally guarantee safe re-execution in a replicated setting; neither TheDB nor MV3C tolerate failures.

Integrated Distributed Commit. To minimize commit latency and avoid redundant coordination, Morty follows recent work [45, 46, 54] in integrating replication, concurrency control, and atomic commit. However, none of these integrated systems supports transaction re-execution. Both TAPIR [54] and Meerkat [46] (unlike Morty) expose only committed writes, resulting in long serialization windows; TAPIR incurs additionally commit latency by using a modular inconsistent replication protocol. Basil [45] instead is Byzantine-fault tolerant and consequently requires signatures and a higher replication degree for safety, resulting in lower relative throughput. Like Morty, Basil is based on MVTSO, but must delay write visibility until prepare time to tolerate Byzantine clients.

Expressivity versus Performance. A wide array of existing systems trade off a restricted transaction model for improved performance. Sinfonia [2] introduces *mini-transactions* that require read and write values to be pre-defined, but minimize latency by piggybacking transaction execution alongside distributed commit. Janus [33] re-orders transactions at commit time to avoid aborts, but does so by requiring transactions to be *stored procedures*, which poses deployment challenges. Calvin [48] also orders transactions before executing them, which requires knowing the read/write sets ahead of time. Carousel [51] instead introduces the *2-round fixed-set interactive* (2FI) model that requires key-sets to be known, but allows write values to depend on reads across shards; this allows the distributed commit and consensus phases to overlap, reducing latency.

7 Conclusion

Traditional approaches for implementing serializable and interactive transactions fair poorly under high contention. This chapter introduces the notion of *serialization and validity windows* to characterize the limitation that serializable systems face, especially in geo-distributed deployments, in concurrently processing conflicting read-write transactions. Using these windows as a guide, we design a serializable, replicated storage system, Morty, that employs *transaction re-execution* to efficiently sequence contending windows and significantly improve throughput.

References

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

[2] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *ACM Symposium on Operating System Principles (SOSP)*.

[3] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*.

[4] Amazon Web Services 2021. https://aws.amazon.com/.

[5] Andrew W. Appel. 1998. SSA is Functional Programming. *ACM SIGPLAN NOTICES* (1998).

[6] Azure SQL Database 2022. https://www.windowsazure.com/en-us/services/sql-database/.

[7] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim

Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*.

[8] BerkeleyDB 2022. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.

[9] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Vol. 370. Addison-Wesley Reading.

[10] Cloud Spanner 2022. https://cloud.google.com/spanner/.

[11] Cloud Spanner's Lock Scanned Ranges 2022. https://cloud.google.com/spanner/docs/reference/standard-sql/query-syntax.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[13] CosmosDB 2022. https://azure.microsoft.com/en-us/services/cosmos-db/.

[14] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[15] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2017. Transaction Repair for Multi-Version Concurrency Control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[16] Db2 2022. https://www.ibm.com/software/data/db2/.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*.

[19] DynamoDB 2022. https://aws.amazon.com/dynamodb/.

[20] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High Performance Transactions via Early Write Visibility. In *Proceedings of the VLDB Endowment (PVLDB)*.

[21] FoundationDB 2022. http://foundationdb.com/.

[22] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *Proceedings of the VLDB Endowment (PVLDB)*.

[23] Richard A Kelsey. 1995. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices* (1995).

[24] Hsiang-Tsung Kung and John T Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (1981), 213–226.

[25] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.

[26] Justin Levandoski, David Lomet, Sedipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. Multi-version range concurrency control in Deuteronomy. In *VLDB*.

[27] LevelDB 2022. http://leveldb.org/.

[28] Harry Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. 2007. The Paxos Register. In *IEEE Symposium on Reliable Distributed Systems*.

[29] libevent 2021. https://libevent.org/.

[30] Robert M. Metcalfe and David R. Boggs. 1976. Ethernet: Distributed Packet Switching for Local Computer Networks. *Commun. ACM* 19, 7 (1976), 395–404.

[31] MongoDB 2022. https://www.mongodb.com/.

[32] Morty Implementation 2022. https://www.github.com/matthelb/morty/.

[33] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[34] MySQL 2022. https://www.mysql.com/.

[35] neo4j 2022. https://neo4j.com/.

[36] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[37] Andy Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Research on Concurrency Control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[38] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[39] RocksDB 2022. http://rocksdb.org/.

[40] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. 1978. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems* 3, 2 (1978), 178–198.

[41] Rust-Tokyo 2021. https://github.com/tokio-rs/tokio.

[42] SQL Server 2022. https://www.microsoft.com/sqlserver/.

[43] SQLite 2022. https://sqlite.org/.

[44] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing Modular Concurrency Control to the Next Level. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[45] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *ACM Symposium on Operating System Principles (SOSP)*.

[46] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. 2020. Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*.

[47] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[48] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[49] TPC-C 2021. http://www.tpc.org/tpcc/.

[50] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. 2016. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[51] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *ACM SIGMOD International Conference on Management of Data (SIGMOD).*

[52] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMD.*

[53] YugabyteDB 2022. https://www.yugabyte.com/.

[54] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *ACM Symposium on Operating System Principles (SOSP).*

A System Model

We use Adya's system model and terminology to write our proofs. We reproduce much of the text from Adya's thesis [1] for convenience.

A transaction is a particular execution of a program that interacts with the objects in the database through read and write operations. When a transaction writes an object x, it creates a new version of x. A transaction T_i can modify an object multiple times; its first update of object x is denoted by $x_{i,1}$, the second by $x_{i,2}$ and so on. Version x_i denotes the final modification of x performed by T_i. That is, $x_i \equiv x_{i,n}$ where $n = \max\{j | x_{i,j}$ exists $\}$. Transactions interact with the database only in terms of objects; the system maps each operation on an object to a specific version of that object. We use $r_i(x_{j,m})$ ($w_i(x_{i,m})$) to denote the execution of a read (write) operation on a specific version of an object x.

Formally, a *transaction* T_i is both a set of operations $T_i \subseteq \{r_i(x_j), w_i(x_{i,m}) | x$ is an object $\} \cup \{c_i, a_i\}$ and a total order on this set which corresponds to the order in which its operations were registered by the database. A transaction T_i may not be a complete execution of a program. However, if $c_i \in T_i$, then $a_i \notin T_i$ and vice versa. Moreover, if T_i commits (aborts), c_i (a_i) must be the last operation of the transaction.

The database state refers to the versions of objects that have been created by committed and uncommitted transactions. The *committed state of the database* reflects only the modifications of committed transactions. When transactions T_i commits, each version x_i created by T_i becomes a part of the committed state and we say that T_i *installs* x_i. If T_i aborts, x_i does not become part of the committed state. Thus, the system needs to prevent modifications made by uncommitted and aborted transactions from affecting the committed database state.

Conceptually, the committed state comes into existence as a result of running a special initialization transaction, T_{init}. Transaction T_{init} creates all objects that will ever exist in the database; at this point, each object x has an initial version, x_{init}, called the *unborn* version. When an application transaction inserts an object x (e.g., inserts a tuple in a relation), we model it as the creation of a *visible* version for x. When a transaction T_i deletes an object x (e.g., by deleting a tuple from some relation), we model it as the creation of a special *dead* version, i.e., in this case, x_i (also called x_{dead}) is a dead version. Thus, object versions can be of three kinds: unborn, visible, and dead.

A *history H* over a set of transactions consists of two parts: a partial order of events E that reflects the operations of those transactions, and a version order, $\ll$, that is a total order on committed object versions.

The partial order of events E in a history obeys the following constraints:

- It preserves the order of all events within a transaction including the commit and abort events.
- If an event $r_j(x_{i,m})$ exists in E, it is preceded by $w_i(x_{i,m})$ in E, i.e., a transaction T_j cannot read version x_i of object x before it has been produced by T_i.
- If an event $w_i(x_{i,m})$ is followed by $r_i(x_j)$ without an intervening event $w_i(x_{i,n})$ in E, x_j must be $x_{i,m}$. This condition ensures that if a transaction modifies object x and later reads x, it will observe its last update to x.

The second part of a history H is the version order, $\ll$, that specifies a total order on object versions created by *committed* transaction in H; there is no constraint on versions due to uncommitted or aborted transactions. We refer to versions due to committed transactions in H as *committed versions* and impose two constraints on H's version order for different kinds of committed versions:

- the version order of each object X contains exactly one initial version, x_{init}, and at most one dead version, x_{dead}.
- x_{init} is x's first version in its version order and x_{dead} is its last version (if it exists); all visible versions are placed between x_{init} and x_{dead}.
- if $r_j(x_i)$ occurs in a history, then x_i is a visible version.

We define three kinds of *direct conflicts* that capture conflicts of two different *committed* transactions on the same object: read-dependency, anti-dependency, and write-dependency. The first type, *read dependency*, specifies write-read conflicts; a transaction T_j depends on T_i if it reads T_i's updates. *Anti-dependencies* capture read-write conflicts; T_j anti-depends on T_i if it overwrites an object that T_i has read. *Write-dependencies* capture write-write conflicts; T_j write-depends on T_i if it overwrites an object that T_i has also modified.

Definition A.1. *A transaction T_j* directly read-depends *on transaction T_i if T_i installs some object version x_i and T_j reads x_i (denoted by $T_i \xrightarrow{wr} T_j$).*

Definition A.2. *A transaction T_j* directly anti-depends *on transaction T_i if T_i reads some object version x_k and T_j installs x's next version (after x_k) in the version order (denoted by $T_i \xrightarrow{rw} T_j$). Note that the transaction that wrote the later version directly anti-depends on the transaction that read the earlier version.*

Definition A.3. *A transaction T_j* directly write-depends *on transaction T_i if T_i installs a version x_i and T_j installs x's next version (after x_i) in the version order (denoted by $T_i \xrightarrow{ww} T_j$).*

Definition A.4. *We define the* direct serialization graph *arising from a history H, denoted $DSG(H)$ as follows. Each node in $DSG(H)$ corresponds to a* committed *transaction in H and directed edges correspond to different types of direct conflicts. There is a* read/write/anti-dependency *edge from transaction T_i if T_j* directly read/write/anti-depends *on T_i.*

There can be at most one edge of a particular kind from node T_i to T_j since the edges do not record the objects that gave rise to the conflict.

Definition A.5. *A history H exhibits phenomenon* G1a *(*aborted reads*) if it contains an aborted transaction T_i and a committed transaction T_j such that T_j has read some object modified by T_i.*

Definition A.6. *A history H exhibits phenomenon* G1b *(*intermediate reads*) if it contains a committed transaction T_j that has read a version of object x written by transaction T_i that was not T_i's final modification of x.*

Definition A.7. *A history H exhibits phenomenon* G1c *(*circular information flow*) if $DSG(H)$ contains a directed cycle consisting entirely of dependency edges.*

Definition A.8. *A history H exhibits phenomenon* G2 *(*anti-dependency cycles*) if $DSG(H)$ contains a directed cycle having one or more anti-dependency edges.*

Definition A.9. *A history H is* serializable *if it does not exhibit G1a, G1b, G1c, and G2.*

Definition A.10. *A history H is* serializable *it it does not exhibit aborted reads and intermediate reads and if $DSG(H)$ is acyclic.*

B Proof of Correctness

We prove the correctness of Morty using Adya's system model and terminology [1].

Definition B.1. *A transaction T_i* commits *if some coordinator reaches a* Commit *decision for T_i.*

Definition B.2. *An Original Slow Path or Recovery coordinator c of transaction T_i* decides in view *v with decision d if at least $f + 1$ servers accept c's proposal of d in v.*

Lemma B.1. *If a coordinator c_1 of transaction T_i decides in view v_1 with decision d_1, then for every coordinator c_2 of transaction T_i that decides in view v_2 with decision d_2 such that $v_2 \geq v_1$, $d_2 = d_1$.*
Proof.

⟨1⟩1. Let c_1 be a coordinator of transaction T_i that decides in view v_1 with decision d_1.
⟨1⟩2. Let c_2 be a coordinator of T_i that decides in view v_2 with decision d_2 such that $v_2 > v_1$.
⟨1⟩3. Q.E.D.

Let $v'_1 = v_1, ..., v'_n = v_2$ be the sequence of $n \geq 2$ views between v_1 and v_2 in which at least one coordinator decides. For each view v'_i in the sequence, we define c'_i as the coordinator thhst decided decision d'_i in v'_i. Since $v'_n > v'_1$, c'_n must be a recovery coordinator.
The inductive hypothesis is that $d_1 = d'_n$.
The base case is when $n = 2$. By the definition of "decide in view", $f + 1$ servers accepted c'_1's proposal of d'_1 in v'_1. This means at least one server that c'_2 receives a *PaxosPrepareReply* from must return c'_1's proposal. Moreover, this must correspond to the highest view v'_1 of any views received in the replies. This and the recovery procedure imply that c'_2 proposes d'_1 as d'_2.
Now we prove the inductive hypothesis using only the fact that $d_1 = d'_{n-1}$. By the definition of "decide in view", $f + 1$ servers accepted c'_{n-1}'s proposal of d'_{n-1} in v'_{n-1}. This means at least one server that c'_n receives a *PaxosPrepareReply* from must return c'_{n-1}'s proposal. Moreover, this must correspond to the highest view v'_{n-1} of any views received in the replies. This and the recovery procedure imply that c'_n proposes d'_{n-1} as d'_n. □

Lemma B.2. *If a coordinator c_1 reaches a decision $d_1 \in \{Commit, Abandon\}$ for transaction T_i, for every other coordinator $c_2 \neq c_1$ that reaches decision d_2 for T_i, $d_2 = d_1$.*
Proof.

⟨1⟩1. Let c_1 be a coordinator that reaches a decision d_1 for T_i.
⟨1⟩2. Let $c_2 \neq c_1$ be a coordinator that reaches a decision d_2 for T_i.
⟨1⟩3. Case: c_1 decides on the Original Fast Path and c_2 decides on the Recovery Path; $d_1 = Commit$.

⟨2⟩1. c_1 receives $2f + 1$ *PrepareReply* messages from servers voting *Commit* for T_i.

By the assumption of ⟨1⟩3 that c_1 decides on the Original Fast Path.

⟨2⟩2. Let v_2 be the view in which c_2 decides.
⟨2⟩3. Every recovery coordinator that reaches a decision decides *Commit.*

⟨3⟩1. Let c be a recovery coordinator that decides d in view v.
⟨3⟩2. c receives $f+1$ *PaxosPrepareReply* messages from servers agreeing to not accept proposals in views smaller than v.
⟨3⟩3. Let v_0 be the smallest view in which a recovery coordinator decides.
⟨3⟩4. Case: $v = v_0$.

⟨4⟩1. No *PaxosPrepareReply* contains a *finalize_decision.*

⟨5⟩1. Suffices Assume: At least one *PaxosPrepareReply* contains a *finalize_decision.*
Prove: False.
⟨5⟩2. Let v' be the view in which the recovery coordinator c' proposed *finalize_decision.*
⟨5⟩3. The server that sent the *PaxosPrepareReply* with *finalize_decision* accepted *finalize_decision* in v' before promising to not accept proposals in views smaller than v.
⟨5⟩4. $v' < v$.
⟨5⟩5. Q.E.D.

By ⟨5⟩4, ⟨3⟩3, and ⟨3⟩4.

⟨4⟩2. Case: At least one *PaxosPrepareReply* contains a *decision.*

⟨5⟩1. Let s be the server that sent the reply containing *decision* to c.
⟨5⟩2. s learned the decision from c_1.

⟨6⟩1. Suffices Assume: s learned of the decision from a coordinator $c' \neq c_1$.
Prove: False
⟨6⟩2. c' is a recovery coordinator.

c' is not the original coordinator by assumption. If c' is truncation coordinator, then T_i is part of the truncated epoch, which implies that s would not respond to *PaxosPrepare* for T_i.

⟨6⟩3. c' decided in view v'.
⟨6⟩4. $v' < v$.

c' sends learned decision implies $f+1$ accepted decision in v'. By ⟨3⟩2, at least one server s' accepted decision in v' and agreed to not accept proposals in views smaller than v. This implies that $v' < v$.

⟨6⟩5. Q.E.D.

By ⟨3⟩3, ⟨3⟩4, and ⟨6⟩4.

⟨5⟩3. *decision* = *Commit*.

By ⟨5⟩2 and the assumption of ⟨1⟩3 that $d_1 = Commit$.

⟨5⟩4. Q.E.D.

By ⟨5⟩3 and the Recovery Decision procedure, $d = Commit$.

⟨4⟩3. CASE: All *PaxosPrepareReply* messages only contain *votes*.

⟨5⟩1. Every server that sent a *PaxosPrepareReply* previously sent a *PrepareReply* with *Commit* vote to c_1.

By ⟨2⟩1.

⟨5⟩2. All $f+1$ votes are *Commit*.

By the fact that a server never changes its vote until the vote is truncated. However, a server would not respond to a *PaxosPrepare* for T_i if it has truncated T_i.

⟨5⟩3. Q.E.D.

By the Recovery Decision procedure and ⟨5⟩2.

⟨4⟩4. Q.E.D.

By ⟨4⟩1, steps ⟨4⟩2 and ⟨4⟩3 are exhaustive.

⟨3⟩5. CASE: 1. $v > v_0$.
2. The decision d' for all $v' < v$ is *Commit*.

By ⟨3⟩1, the assumption of the case, and Lemma B.1.

⟨3⟩6. Q.E.D.

By ⟨3⟩4, ⟨3⟩5, and mathematical induction.

⟨2⟩4. $d_2 = Commit$.

By ⟨2⟩3 and the assumption of ⟨1⟩3 that c_2 is a recovery coordinator.

⟨2⟩5. Q.E.D.

By ⟨2⟩4 and the assumption of ⟨1⟩3 that $d_1 = Commit$, $d_2 = d_1$.

⟨1⟩4. CASE: c_1 decides on the Original Fast Path and c_2 decides on the Recovery Path; $d_1 = Abandon$.

⟨2⟩1. c_1 receives a *PrepareReply* message with an *Abandon-Final* vote for T_i.

By the assumption of ⟨1⟩4 that c_1 decides on the Original Fast Path.

⟨2⟩2. Let v_2 be the view in which c_2 decides.
⟨2⟩3. Every recovery coordinator that reaches a decision decides *Abandon*.

⟨3⟩1. Let c be a recovery coordinator that decides d in view v.
⟨3⟩2. c receives $f+1$ *PaxosPrepareReply* messages from servers agreeing to not accept proposals in views smaller than v.
⟨3⟩3. Let v_0 be the smallest view in which a recovery coordinator decides.
⟨3⟩4. CASE: $v = v_0$.

⟨4⟩1. No *PaxosPrepareReply* contains a *finalize_decision*.

⟨5⟩1. SUFFICES ASSUME: At least one *PaxosPrepareReply* contains a *finalize_decision*.
PROVE: False.
⟨5⟩2. Let v' be the view in which the recovery coordinator c' proposed *finalize_decision*.
⟨5⟩3. The server that sent the *PaxosPrepareReply* with *finalize_decision* accepted *finalize_decision* in v' before promising to not accept proposals in views smaller than v.
⟨5⟩4. $v' < v$.
⟨5⟩5. Q.E.D.

By ⟨5⟩4, ⟨3⟩3, and ⟨3⟩4.

⟨4⟩2. CASE: At least one *PaxosPrepareReply* contains a *decision*.

⟨5⟩1. Let s be the server that sent the reply containing *decision* to c.
⟨5⟩2. s learned the decision from c_1.

⟨6⟩1. SUFFICES ASSUME: s learned of the decision from a coordinator $c' \neq c_1$.
PROVE: False
⟨6⟩2. c' is a recovery coordinator.

c' is not the original coordinator by assumption. If c' is truncation coordinator, then T_i is part of the truncated epoch, which implies that s would not respond to *PaxosPrepare* for T_i.

⟨6⟩3. c' decided in view v'.
⟨6⟩4. $v' < v$.

c' sends learned decision implies $f+1$ accepted decision in v'. By ⟨3⟩2, at least one server s' accepted decision in v' and agreed to not accept proposals in views smaller than v. This implies that $v' < v$.

⟨6⟩5. Q.E.D.

By ⟨3⟩3, ⟨3⟩4, and ⟨6⟩4.

⟨5⟩3. *decision* = *Abandon*.

By ⟨5⟩2 and the assumption of ⟨1⟩4 that $d_1 = Abandon$.

⟨5⟩4. Q.E.D.

By ⟨5⟩3 and the Recovery Decision procedure, $d = Abandon$.

⟨4⟩3. CASE: All *PaxosPrepareReply* messages only contain *votes*.

⟨5⟩1. There are $\leq f$ votes for *Commit*.

By ⟨2⟩1 and the fact that a server only votes *Abandon-Final* for a transaction T_i if the *Abandon* decision is already durable in that no set of $f+1$ servers can vote to commit T_i.

⟨5⟩2. Q.E.D.

By the Recovery Decision procedure and ⟨5⟩1.

⟨4⟩4. Q.E.D.

By $\langle 4\rangle 1$, steps $\langle 4\rangle 2$ and $\langle 4\rangle 3$ are exhaustive.

$\langle 3\rangle 5$. Case: 1. $v > v_0$.
2. The decision d' for all $v' < v$ is *Commit*.

By $\langle 3\rangle 1$, the assumption of the case, and Lemma B.1.

$\langle 3\rangle 6$. Q.E.D.

By $\langle 3\rangle 4$, $\langle 3\rangle 5$, and mathematical induction.

$\langle 2\rangle 4$. $d_2 = Abandon$.

By $\langle 2\rangle 3$ and the assumption of $\langle 1\rangle 4$ that c_2 is a recovery coordinator.

$\langle 2\rangle 5$. Q.E.D.

By $\langle 2\rangle 4$ and the assumption of $\langle 1\rangle 4$ that $d_1 = Abandon$, $d_2 = d_1$.

$\langle 1\rangle 5$. Case: c_1 decides on the Original Slow Path and c_2 decides on the Recovery Path.

$\langle 2\rangle 1$. Let v_1 be the view in which c_1 decides d_1.

By the hypothesis that c_1 decides on the Original Slow Path.

$\langle 2\rangle 2$. Let v_2 be the view in which c_2 decides d_2.

By the hypothesis that c_2 decides on the Recovery Path.

$\langle 2\rangle 3$. Case: $v_2 > v_1$.

By $\langle 2\rangle 2$, $\langle 2\rangle 3$, and Lemma B.1, $d_2 = d_1$.

$\langle 2\rangle 4$. Case: $v_1 > v_2$.

By $\langle 2\rangle 2$, $\langle 2\rangle 3$, and Lemma B.1, $d_1 = d_2$.

$\langle 2\rangle 5$. Q.E.D.

Steps $\langle 2\rangle 3$ and $\langle 2\rangle 4$ are exhaustive.

$\langle 1\rangle 6$. Case: c_1 decides on the Recovery Path and c_2 decides on the Recovery Path.

$\langle 2\rangle 1$. Let v_1 be the view in which c_1 decides d_1.

By the hypothesis that c_1 decides on the Recovery Path.

$\langle 2\rangle 2$. Let v_2 be the view in which c_2 decides d_2.

By the hypothesis that c_2 decides on the Recovery Path.

$\langle 2\rangle 3$. Case: $v_2 > v_1$.

By $\langle 2\rangle 2$, $\langle 2\rangle 3$, and Lemma B.1, $d_2 = d_1$.

$\langle 2\rangle 4$. Case: $v_1 > v_2$.

By $\langle 2\rangle 2$, $\langle 2\rangle 3$, and Lemma B.1, $d_1 = d_2$.

$\langle 2\rangle 5$. Q.E.D.

Steps $\langle 2\rangle 3$ and $\langle 2\rangle 4$ are exhaustive.

$\langle 1\rangle 7$. Case: c_1 decides on the Original Fast Path, the Original Slow Path, the Recovery Path, or the Truncation Path and c_2 decides on the Truncation Path

The truncation procedure maintains the invariant that: if a decision could have been reached for a transaction T in one of the constituent *erecords*, then that decision is preserved in the *merged_erecord*.

$\langle 1\rangle 8$. Q.E.D.

Steps $\langle 1\rangle 3$, $\langle 1\rangle 4$, $\langle 1\rangle 5$, $\langle 1\rangle 6$, and $\langle 1\rangle 7$ are exhaustive. □

Definition B.3. *A transaction* T_i permanently conflict rejects *at server* s_j *if after* T_i *validates successfully at* s_j *any transaction* T_j *is rejected if:*

- T_i *reads* x_k *and* T_j *writes* x *and* $ver(T_k) < ver(T_j) < ver(T_i)$, *or*
- T_i *writes* x *and* T_j *reads* x_k *and* $ver(T_k) < ver(T_i) < ver(T_j)$.

Lemma B.3. *If a transaction* T_i *commits,* T_i *permanently conflict rejects at* $m \geq f + 1$ *servers.*
Proof.

$\langle 1\rangle 1$. Let T_i be a transaction that commits.
$\langle 1\rangle 2$. Let T_j be a transaction that validates at server s such that:
- T_i reads x_k and T_j writes x and $ver(T_k) < ver(T_j) < ver(T_i)$, or
- T_i writes x and T_j reads x_k and $ver(T_k) < ver(T_i) < ver(T_j)$.

$\langle 1\rangle 3$. If T_i is prepared at a server s when T_j validates at s, s rejects T_j.

By the Validation algorithm's prepared reads and writes check.

$\langle 1\rangle 4$. If T_i is committed at a server s when T_j validates at s, s rejects T_j.

By the Validation algorithm's committed reads and writes check.

$\langle 1\rangle 5$. If T_i has been truncated at a server s when T_j validates at s, s rejects T_j.

By the Validation algorithm's truncation check.

$\langle 1\rangle 6$. $m \geq f + 1$ servers validate T_i successfully and prepare T_i.

Every Commit path requires that T_i be successfully validated at $f + 1$ servers. When a server successfully validates a transaction, it always immediately adds it to its prepared set.

$\langle 1\rangle 7$. Let s be one of the m servers that validate T_i successfully and prepare T_i.
$\langle 1\rangle 8$. s only unprepares T_i when it receives a durable decision (either through the original coordinator, recovery coordinator, or truncation coordinator).

Since T_i commits, Lemma B.2 implies that no coordinator could have reached a durable Abandon decision for T_i. This implies that s can only receive a durable commit decision. If s receives such a decision, it removes T_i from its prepared set and add T_i to its committed set.

$\langle 1\rangle 9$. s only uncommits T_i when it truncates T_i.

The only place in the Algorithm where a server removes a transaction from its committed set is when truncating.

$\langle 1\rangle 10$. Q.E.D.

Proof: By $\langle 1\rangle 3$, $\langle 1\rangle 4$, $\langle 1\rangle 5$, $\langle 1\rangle 6$, and $\langle 1\rangle 7$. □

Lemma B.4. *If* H *is a history produced by Morty,* $DSG(H)$ *is acyclic.*
Proof.

⟨1⟩1. Let $\prec$ be a total order on the transactions in H: $T_i \prec T_j \iff ver(T_i) < ver(T_j)$.

⟨1⟩2. Let the version order $\ll$ for H be: $x_i \ll x_j \iff T_i \prec T_j$.

⟨1⟩3. If the edge $T_i \rightarrow T_j$ is in $DSG(H)$, then $T_i \prec T_j$.

⟨2⟩1. Case: $T_i \xrightarrow{ww} T_j$.

⟨3⟩1. T_i installs a version x_i and T_j installs x's next version x_j in the version order.

Proof: By the hypothesis that the edge $T_i \xrightarrow{ww} T_j$ exists in $DSG(H)$ and Definition A.3.

⟨3⟩2. $x_i \ll x_j$.

Proof: By ⟨3⟩1.

⟨3⟩3. Q.E.D.

Proof: By ⟨3⟩2 and ⟨1⟩2.

⟨2⟩2. Case: $T_i \xrightarrow{wr} T_j$.

⟨3⟩1. T_i installs a version x_i and T_j reads x_i.

Proof: By the hypothesis that the edge $T_i \xrightarrow{wr} T_j$ exists in $DSG(H)$ and Definition A.1.

⟨3⟩2. $ver(T_i) < ver(T_j)$.

Proof: By ⟨3⟩1 and that Morty servers, for a read $r_\ell(x)$ from transaction T_ℓ, only return object versions x_k written by transaction T_k such that $ver(T_k) < ver(T_\ell)$.

⟨3⟩3. Q.E.D.

Proof: By ⟨3⟩2 and ⟨1⟩1.

⟨2⟩3. Case: $T_i \xrightarrow{rw} T_j$.

⟨3⟩1. T_i reads some object version x_k and T_j installs the version x_j after x_k in the version order.

Proof: By the hypothesis that the edge $T_i \xrightarrow{rw} T_j$ exists in $DSG(H)$ and Definition A.2.

⟨3⟩2. $m_i \geq f + 1$ servers permanently prepare T_i.

Proof: By the hypothesis that T_i is committed and Lemma B.3.

⟨3⟩3. $m_j \geq f + 1$ servers permanently prepare T_j.

Proof: By the hypothesis that T_j is committed and Lemma B.3.

⟨3⟩4. There exists a server s that permanently conflict rejects both T_i and T_j.

Proof: By ⟨3⟩2, ⟨3⟩3, and that there are only $n = 2f + 1$ that store object x.

⟨3⟩5. s permanently conflict rejects T_i and T_j sequentially, either preparing T_i first or T_j first.

Proof: By ⟨3⟩4 and the multi-threaded locking that ensures accesses to an object's metadata are sequential at a server,

⟨3⟩6. Case: T_i permanently conflict rejects at s first.

⟨4⟩1. After T_i validates successfully at s, s rejects any transaction T_ℓ that writes x such that $ver(T_k) < ver(T_\ell) < ver(T_i)$.

Proof: By Definition B.3 and ⟨3⟩1.

⟨4⟩2. s does not reject T_j.

Proof: By ⟨3⟩4.

⟨4⟩3. $ver(T_j) < ver(T_k)$ or $ver(T_i) < ver(T_j)$.

Proof: By ⟨4⟩1 and ⟨4⟩2.

⟨4⟩4. $ver(T_k) < ver(T_j)$.

Proof: By ⟨3⟩1, ⟨1⟩1, and ⟨1⟩2.

⟨4⟩5. $ver(T_i) < ver(T_j)$.

Proof: By ⟨4⟩3, and ⟨4⟩4.

⟨4⟩6. Q.E.D.

Proof: By ⟨4⟩5 and ⟨1⟩1.

⟨3⟩7. Case: T_j permanently conflict rejects at s first.

⟨4⟩1. After T_j validates successfully at s, s rejects any transaction T_ℓ that reads x_k such that $ver(T_k) < ver(T_j) < ver(T_\ell)$.

Proof: By Definition B.3 and ⟨3⟩1.

⟨4⟩2. s does not reject T_i.

Proof: By ⟨3⟩4.

⟨4⟩3. $ver(T_j) < ver(T_k)$ or $ver(T_i) < ver(T_j)$.

Proof: By ⟨4⟩1 and ⟨4⟩2.

⟨4⟩4. $ver(T_k) < ver(T_j)$.

Proof: By ⟨3⟩1, ⟨1⟩1, and ⟨1⟩2.

⟨4⟩5. $ver(T_i) < ver(T_j)$.

Proof: By ⟨4⟩3, and ⟨4⟩4.

⟨4⟩6. Q.E.D.

Proof: By ⟨4⟩5 and ⟨1⟩1.

⟨3⟩8. Q.E.D.

Proof: By ⟨3⟩5, ⟨3⟩6, and ⟨3⟩7.

⟨2⟩4. Q.E.D.

Proof: By ⟨2⟩1, ⟨2⟩2, and ⟨2⟩3.

⟨1⟩4. Suffices Assume: There exists a cycle in $DSG(H)$.
Prove: False.

Proof: By the assumption of ⟨1⟩4, ⟨1⟩3, and ⟨1⟩1.

⟨1⟩5. Q.E.D.

Proof: By ⟨1⟩4. □

Lemma B.5. *If H is a history produced by Morty, H does not exhibit aborted reads.*
Proof.

Let T_j be a committed transaction in H such that T_j has read some object modified by T_i. Since T_j is committed, it must have passed the validation check on at least one server. The dirty read check of the validation check implies that each read of T_j is of a write from a committed transaction. This implies that T_i is also committed in H. □

Lemma B.6. *If H is a history produced by Morty, H does not exhibit intermediate reads.*

Proof.

Let T_j be a committed transaction in H such that T_j has read a version of object x written by transaction T_i. Since T_j is committed, it must have passed the validation check on at least one server. The dirty read check of the validation check implies that each read of T_j is of a final write from a committed transaction. □

Theorem B.1. *Morty only produces serializable histories.*
Proof.

$\langle 1 \rangle 1$. Let H be a history produced by Morty.
$\langle 1 \rangle 2$. Q.E.D.

PROOF: By $\langle 1 \rangle 1$, Definition A.10, Lemma B.5, Lemma B.6, and Lemma B.4. □

C Proof of Serialization Windows and Validity Windows

We prove that serialization windows and validity windows must be non-overlapping using Adya's system model and terminology [1].

C.1 Serialization Windows

Definition C.1. *A transaction T_i that writes to object x creates a* serialization window *on x represented by the interval $[a_i, b_i]$. If T_i reads x_k before it writes x, $a_i = \min(w_k(x_k), b_j)$ where b_j is right endpoint of the serialization window on x for the transaction T_j that writes the version x_j that immediately follows x_i in the version order $\ll$; otherwise $a_i = b_i$. $b_i = \min(w_i(x_i), b_j)$*

Note that the definition of a serialization window is a well-formed interval (i.e., $a_i \le b_i$) since a_i is defined to be at most as large as b_i.

A transaction which only reads x does not create a serialization window on x as reading is not a conflicting operation in isolation.

First we prove that the definition of a serialization window is a valid interval.

Lemma C.1. *If $[a_i, b_i]$ is the serialization window of a transaction T_i in H, then $a_i \le b_i$.*

Proof. There are two cases:

- $r_i(x_k) <_H w_i(x_i)$. By the definition of a_i, $a_i = \min(w_k(x_k), b_j)$ and by the definition of b_i, $b_i = \min(w_i(x_i), b_j)$. By the definition of H, $w_k(x_k) <_H r_i(x_k)$, so by the assumption of the case, $w_k(x_k) <_H w_i(x_i)$. Combined with the definitions of a_i and b_i, this implies that $a_i \le b_i$.
- $w_i(x_i) <_H r_i(x_k)$ or $r_i(x_k) \notin T_i$. By the definition of a_i, $a_i \le b_i$.

□

Next, we prove that in a history with an acyclic *DSG*, a transaction that reads and writes an object must read from the version that immediately precedes its version in the version order.

Lemma C.2. *If $DSG(H)$ is acyclic, T_i is a transaction that writes version x_i of object x, and T_j is a transaction that reads version x_k and writes version x_j of object x, and x_j immediately follows x_i in the version order $\ll$ of H, then $x_k = x_i$.*

Proof. Assume for a contradiction that this is not the case, i.e., that $x_k \ne x_i$. There are two sub-cases depending on the version order of x_k and x_i.

Case** $x_i \ll x_k$**: Since x_j immediately follows x_i in the version order, x_k must come after x_j in $\ll$. This implies that there is a sequence of $\xrightarrow{ww}$ edges from T_j to T_k in $DSG(H)$. In addition, there exists a $T_k \xrightarrow{wr} T_j$ edge in $DSG(H)$ because T_j reads x_k from T_k. These imply that there is a cycle in $DSG(H)$. However, there are no cycles in $DSG(H)$ by assumption. Thus, there is a contradiction.

Case** $x_k \ll x_i$**: Let T_ℓ be the transaction that installs the version x_ℓ that immediately follows x_k in the version order. By the assumption of the case and the assumption that x_j immediately follows x_i in $\ll$, there is a sequence $T_k \xrightarrow{ww} T_\ell \xrightarrow{ww} \dots \xrightarrow{ww} T_j$ of $\xrightarrow{ww}$ edges in $DSG(H)$. In addition, there exists a $T_j \xrightarrow{rw} T_\ell$ edge in $DSG(H)$ because T_j reads x_k from T_k and T_ℓ installs the version x_ℓ immediately after x_k. These imply that there is a cycle in $DSG(H)$. However, there are no cycles in $DSG(H)$ by assumption. Thus, there is a contradiction.

In either case, the assumption that $x_k \ne x_i$ implies a contradiction, so $x_k = x_i$. □

This allows us to prove more generally that the order of serialization windows must match the version order in a history with an acyclic *DSG*.

Lemma C.3. *If $DSG(H)$ is acyclic, T_i and T_j are transactions that write object x with serialization windows $[a_i, b_i]$ and $[a_j, b_j]$, and $x_i \ll x_j$, then $b_i \le a_j$.*

Proof. Let $x_1 = x_i, ..., x_n = x_j$ be the sequence of $n \ge 2$ versions between x_i and x_j in $\ll$. For each transaction T_ℓ that creates a version in this sequence, if T_ℓ reads x, then we define x_{k_ℓ} as the version of x that T_ℓ reads.

The inductive hypothesis is that $b_1 \le a_n$.

The base case is when $n = 2$. There are no versions between x_i and x_j in the sequence ($T_1 = T_i$, $T_2 = T_j$). There are two sub-cases:

1. T_2 reads x before writing x. Since T_2 reads x, $a_2 = \min(w_{k_2}(x_{k_2}), b_2)$. Lemma C.2 implies that $x_{k_2} = x_1$. Therefore, $a_2 = \min(w_1(x_1), b_2)$. Since the definition of b_1 is also $\min(w_1(x_1), b_2)$, $b_1 \le a_2$.
2. T_2 does not read x before writing x. Since T_2 does not read x, $a_2 = b_2$. The definition of $b_1 = \min(w_1(x_1), b_2)$ and the definition of a_2, imply that $b_1 \le a_2$.

Now we prove the inductive hypothesis using only the fact that $b_1 \le a_{n-1}$. There are two-sub-cases:

1. T_n reads x before writing x. Since T_n reads x, $a_n = \min(w_{k_n}(x_{k_n}), b_n)$. Lemma C.2 implies that $x_{k_n} = x_{n-1}$. Therefore, $a_n = \min(w_{n-1}(x_{n-1}), b_n)$. Since the definition of b_{n-1} is also $\min(w_{n-1}(x_{n-1}), b_n)$, $b_{n-1} \le a_n$. This, the fact that $a_{n-1} \le b_{n-1}$, and the fact that $b_1 \le a_{n-1}$ imply that $b_1 \le a_n$.
2. T_n does not read x before writing x. Since T_n does not read x, $a_n = b_n$. The definition of $b_{n-1} = \min(w_{n-1}(x_{n-1}), b_n)$ and the definition of a_n imply that $b_{n-1} \le a_n$. This, the fact that $a_{n-1} \le b_{n-1}$, and the fact that $b_1 \le a_{n-1}$ imply that $b_1 \le a_n$.

□

Finally, these lemmas give the result that serialization windows cannot overlap in a history with an acyclic *DSG*.

Theorem C.1. *If $DSG(H)$ is acyclic and T_i and T_j are two transactions in H that write object x, then the serialization windows of T_i and T_j do not overlap.*

Proof. Lemma C.1 implies that for T_i's serialization window $[a_i, b_i]$ to not overlap with T_j's serialization window $[a_j, b_j]$, it must be the case that either $b_i \leq a_j$ or $b_j \leq a_i$. The version order $\ll$ is a total order, so either $x_i \ll x_j$ or $x_j \ll x_i$. In the former case, Lemma C.3 implies that $b_i \leq a_j$. In the latter case, Lemma C.3 implies that $b_j \leq a_i$. □

C.2 Validity Windows

Definition C.2. *A transaction T_i that writes to object x creates a* validity window *on x represented by the interval $[a_i, b_i]$. If T_i reads version x_k before it writes x, $a_i = \min(c_k, b_j)$ where b_j is the right endpoint of the validity window on x for the transaction T_j that writes the version x_j that immediately follows x_i in the version order $\ll$; otherwise $a_i = b_i$. $b_i = \min(c_i, b_j)$.*

First we prove that the definition of a validity window is a valid interval. This requires that the history is recoverable.

Lemma C.4. *If H is a recoverable history and $[a_i, b_i]$ is the validity window of a transaction T_i in H, then $a_i \leq b_i$.*

Proof. By the assumption that H is recoverable, $c_k <_H c_i$. There are three sub-cases:

- $c_k \leq c_i \leq b_j$. By the definitions of a_i and b_i, $a_i = c_k$ and $b_i = c_i$. By the assumption of the case, $a_i \leq b_i$.
- $c_k \leq b_j \leq c_i$. By the definitions of a_i and b_i, $a_i = c_k$ and $b_i = b_j$. By the assumption of the case, $a_i \leq b_i$.
- $b_j \leq c_k \leq c_i$. By the definitions of a_i and b_i, $a_i = b_j$ and $b_i = b_j$. This implies $a_i \leq b_i$.

□

This allow us to prove that the order of validity windows must match the version order in a recoverable history with an acyclic *DSG*.

Lemma C.5. *If $DSG(H)$ is acyclic, H is a recoverable history, T_i and T_j are transactions that write object x with validity windows $[a_i, b_i]$ and $[a_j, b_j]$, and $x_i \ll x_j$, then $b_i \leq a_j$.*

Proof. Let $x_1 = x_i, ..., x_n = x_j$ be the sequence of $n \geq 2$ versions between x_i and x_j in $\ll$. For each transaction T_ℓ that creates a version in this sequence, if T_ℓ reads x, then we define x_{k_ℓ} as the version of x that T_ℓ reads. Let T_{n+1} be the transaction that writes x_{n+1}, the next version after x_n according to $\ll$.

The inductive hypothesis is that $b_1 \leq a_n$.

The base case is when $n = 2$. There are no versions between x_i and x_j in the sequence ($T_1 = T_i$, $T_2 = T_j$).

1. T_2 reads x before writing x. Since T_2 reads x, $a_2 = \min(c_{k_2}, b_3)$. Lemma C.2 implies that $x_{k_2} = x_1$, so $c_{k_2} = c_1$. The definition of $b_1 = \min(c_1, b_2)$ and the definition of $b_2 = \min(c_2, b_3)$, so $b_1 = \min(c_1, c_2, b_3)$. Since this is a minimum over a superset of the elements in the definition of a_2, this implies that $b_1 \leq a_2$.
2. T_2 does not read x before writing x. Since T_2 does not read x, $a_2 = b_2$. This equality and the definition of $b_1 = \min(c_1, b_2)$ imply that $b_1 \leq a_2$.

Now we prove the inductive hypothesis using only the fact that $b_1 \leq a_{n-1}$. First, we show that $b_{n-1} \leq a_n$. There are two sub-cases:

1. T_n reads x before writing x. Since T_n reads x, $a_n = \min(c_{k_n}, b_{n+1})$. Lemma C.2 implies that $x_{k_n} = x_{n-1}$, so $c_{k_n} = c_{n-1}$. The definition of $b_{n-1} = \min(c_{n-1}, b_n)$ and the definition of $b_n = \min(c_n, b_{n+1})$, so $b_{n-1} = \min(c_{n-1}, c_n, b_{n+1})$. Since this is a minimum over a superset of the elements in the definition of a_n, this implies that $b_{n-1} \leq a_n$.
2. T_n does not read x before writing x. Since T_n does not read x, $a_n = b_n$. This equality and the definition of $b_{n-1} = \min(c_{n-1}, b_n)$ imply that $b_{n-1} \leq a_n$.

Finally, Lemma C.4 implies that $a_{n-1} \leq b_{n-1}$. The facts that $b_1 \leq a_{n-1}$, $a_{n-1} \leq b_{n-1}$, and $b_{n-1} \leq a_n$ imply that $b_1 \leq a_n$. □

Finally, these lemmas give the result that validity windows cannot overlap in a recoverable history with an acyclic *DSG*.

Theorem C.2. *If $DSG(H)$ is acyclic, H is a recoverable history, and T_i and T_j are two transactions in H that write to x, then the validity windows of T_i and T_j do not overlap.*

Proof. Lemma C.4 implies that for T_i's validity window $[a_i, b_i]$ to not overlap with T_j's validity window $[a_j, b_j]$, it must be the case that either $b_i \leq a_j$ or $b_j \leq a_i$. The version order $\ll$ is a total order, so either $x_i \ll x_j$ or $x_j \ll x_i$. In the former case, Lemma C.5 implies that $b_i \leq a_j$. In the latter case, Lemma C.5 implies that $b_j \leq a_i$. □