JEPSEN

Aerospike 3.99.0.3

Kyle Kingsbury

2018-03-07

Aerospike is a high-performance distributed document store. Following up on our 2015 analysis, we explored Aerospike's new strong-consistency mode, which offers linearizable operations on single records. We confirmed two documented flaws in Aerospike's homegrown replication algorithm. First, it can lose updates when more than k nodes crash (either concurrently or in sequence). Second, when either process pauses or clock skew exceed 27 seconds, Aerospike could lose committed updates. Aerospike has added an option in 3.99.2.1 which prevents write loss on crashes, and plans to increase the amount of clock skew or pause before data loss can occur. We also discovered a previously unknown issue where Aerospike could inform a client that a write did not *succeed when, in fact, it had; this was fixed in 3.99.1.5. With these fixes in place, Aerospike does appear to provide linearizability through network partitions and process crashes, but data loss due to process pauses and clock skew remains. Aerospike has published a companion post, and made binaries available for reproducing these results. This work was funded by Aerospike, and conducted in accordance with the Jepsen ethics policy.*

1 Background

Aerospike, formerly Citrusleaf, is a distributed, sharded document store designed for high-throughput, low-latency applications. Each record is a map of keys to values, which may be either primitive types (integers, strings, bytes, etc.) or nested collections like lists and maps. Secondary indices are also available for strings and integers. Operations on these records include reads, writes, compare-and-set, and datatype-specific transformations like list append.

Historically, Aerospike has only offered a totally-available (AP) mode, resolving conflicts via wall-clock timestamps or the replica which has seen the greatest number of writes. We tested Aerospike 3.5.4 in 2015, and found that both of these strategies allowed stale reads, dirty reads, and lost updates during network partitions.

In the fall of 2017, Aerospike requested a followup analysis covering their new strong consistency (SC) mode, scheduled for release in 4.0. SC mode offers per-key linearizable updates on a per-namespace basis, in exchange for reduced availability during failures. As with many other CP databases, clients may select between potentially stale reads or, at the cost of an extra round trip, fully linearizable reads. We redesigned and expanded the Jepsen test suite from 2015 to explore linearizable behavior in 3.99.0.3 through 3.99.2.1.

1.1 AP mode

In the 2015 analysis, we found that despite claiming to be an "ACID" database, in the event of network partitions, Aerospike could expose stale versions of records, lose linearizable updates, and even lose commutative updates such as counter increments.

The 2015 analysis spurred Aerospike to re-evaluate their engineering process. Instead of focusing on feature development, they took time to redesign and overhaul core components: improving inter-node messaging performance, preventing permanent split-brain after transient network interruption, reducing traffic used in shard[1] rebalancing after a partition by two orders of magnitude, and so on.

These improvements enabled the development of SC mode in 2017, but also significantly improved life for

[1] We call an Aerospike "partition" a "shard", to prevent confusion with network partitions. We refer to Aerospike "masters" and "proles" as "primaries" and "secondaries". We call a collection of Aerospike nodes which *could* communicate a "cluster"; Aerospike has no dedicated term for this concept. Aerospike uses both "cluster" and "sub-cluster" to refer to group(s) of nodes which can currently exchange messages; we use "sub-cluster" for this concept.

AP users. For instance, performing a rolling restart of an Aerospike cluster without waiting for all shards to migrate could result in lost updates. Overhauling the partition rebalancing system meant that in newer releases of Aerospike, users could restart a new node as soon as the old one had come back into the cluster, and add new nodes to the cluster without temporarily reducing durability. AP users also benefited from a redesign of the migration system to properly handle overlapping migration rounds, while improving performance. The introduction of tombstones also made it possible to delete records without them reappearing during partitions or cold starts. Migrating data between mostly-full shards is dramatically faster with the introduction of delta migrations.

Despite these improvements, Aerospike's documentation continued to be somewhat vague with respect to consistency issues until January 2018. The FAQ accurately explains that records with more changes will be preferred over those with fewer:

> By default, Aerospike uses a generation counter to determine which of 2 values for a key is the correct one. It will choose the value that has a higher generation count.

… and the ACID documentation went on to say that Aerospike is an AP system in CAP terms: on asynchronous networks, every request to a non-faulty node should complete successfully.[2] This implies that Aerospike cannot provide linearizability, serializability, snapshot isolation, or repeatable read.

As we previously discussed, Aerospike's AP system will readily lose committed updates in the event of partitions; however, so long as the network does not fail, Aerospike can (like all databases) ensure consistency. Aerospike's documentation called this approach "high consistency", essentially arguing that partitions are rare enough to not be a significant concern:

> The Aerospike AP system provides high consistency by
>
> - Restricting communication latencies between nodes to the [sic] submillisecond.
> - Eliminating partition formation.

It remains unclear how database software can prevent networks and nodes from delaying or dropping messages.

> - Providing automatic conflict resolution to ensure that during cluster formation, new data overrides old data.

As we saw in 2015, Aerospike's automatic conflict resolution strategies (last-write-wins merge by local timestamp, and max-updates-wins) can also do the exact opposite, preserving old data in favor of new data. Even when conflict resolution *does* work correctly, it will not prevent lost updates: clients on both sides of a partition can independently read, modify, and write back data without seeing each other's changes.

Aerospike's consistency guarantees documentation went on to claim that users "can guarantee complete data consistency by involving all replicas of a record during each transaction". However, Aerospike's team confirms that using consistency level `all` does not prevent consistency anomalies—only reduce their frequency.

These documentation errors were quickly addressed, and Aerospike's web site now accurately describes the current AP behavior.

To recap: AP mode's guarantees are largely unchanged: stale reads, dirty reads, and lost updates are all possible in the face of partitions. However, as a result of their engineering work over the past three years, Aerospike anecdotally reports meaningful reductions in the frequency and magnitude of these anomalies.

Moreover, AP mode remains appropriate for many types of workloads, especially those which are latency sensitive or geographically distributed, where throughput is critical, the importance of individual records is low, contention is limited, or in-place modification of data is infrequent.

However, some users *do* need stronger guarantees. For that, Aerospike 4.0 will offer SC mode.

1.2 SC Mode

Aerospike targets the edge of the performance envelope: systems which would not be cost-effective with a typical database.[3] For instance, telecom data and

[2] Strictly speaking, not *every* request is guaranteed to succeed; there is usually a brief window after failure where a few operations may time out or fail, before the cluster stabilizes.

[3] Documentation for SC mode is not yet public; this section describes Aerospike's behavior based on early-access and Aerospike-internal documents, and discussions with the development team.

web analytics can create enormous data volumes which must be written in tight latency budgets. To meet these goals, Aerospike makes some unconventional design decisions to improve latency and throughput.

First, while Aerospike *does* write records to disk for durability, it does so asynchronously to improve latency; at any given point, some committed transactions are only stored in memory, not on disk. Concurrent crashes of more than *replication-factor* nodes could cause the loss of committed writes.

Second, Aerospike opted to design two custom, overlapping coordination protocols for managing cluster state and replicating data. The first system is loosely adapted from Paxos, but it is decidedly not a consensus system: it does not, for instance, guarantee agreement on a single value, nor does it rely on a fixed set of members. Rather, it provides rough agreement among *whoever showed up* to the most recent round. The goal of this system is to quickly establish a *sub-cluster*—a set of nodes which are currently alive and can see one another—and choosing a *principal* node among them, which can serve as a coordinator. Multiple sub-clusters may be active at any given time. This system is the basis of both the AP and SC modes in Aerospike.

On top of this gossip system, Aerospike's SC mode layers a second, custom replication protocol which aims to provide true consensus. For performance reasons, it is not based on Paxos, Raft, Zab, or any established consensus system. Why? Because most consensus systems use majority quorums, and require a majority of nodes to make progress—this implies a minimum of three replicas for any piece of data. Aerospike deployments are often constrained by storage costs: storing three replicas of data, instead of two, might be too expensive to be practical. Therefore, Aerospike needs to maintain availability through the loss of any single node in a *two*-replica system.

Where most replication systems broadcast a message to every node and commit once a majority have acknowledged, Aerospike's SC mode broadcasts to *replication-factor* replicas (rather than all possible replicas) and waits for unanimous acknowledgement from those nodes before committing. Aerospike handles failure by allowing that set of replicas to *change*, provided certain invariants hold. For each shard, given some set of statically computed *roster replicas* which will store copies of that shard when the cluster is healthy, and a *roster primary* which serves as a tiebreaker, availability is preserved so long as there exists some sub-cluster where:

1. A roster replica is present, and fewer than *replication-factor* nodes are missing, or
2. There exists at least one full replica of that shard, and either:
    a. All roster replicas, or
    b. At least one roster replica and a majority of possible replicas, or
    c. A roster primary and exactly 1/2 of the cluster

These rules allow for the failure or isolation of any single node with a *replication-factor* of only 2, so long as a majority of *potential* replicas are present to ensure only one side of a partition can make progress.

Consider a cluster with three nodes: A, B, and C, where *replication-factor* = 2. Aerospike might choose nodes A and B as replicas, and therefore writes must be acknowledged by both A and B before commit. Now imagine A is partitioned away, leaving a sub-cluster of B and C connected. Since B and C form a majority of the cluster, and a copy of every committed record is present on B, C can become a *temporary* replica, allowing the system to continue with two copies of every write. A, being unable to reach B, cannot commit any further writes. Without a majority, A is also unable to elect a new set of replicas, which prevents divergence. When the partition heals, B and C will bring A up to date; once A has caught up, C can stop being a replica.

There are special rules governing what sub-clusters are eligible to process requests, how to ensure reads are up to date, merging partially divergent replicas, tracking clean vs unclean shutdown, and so on, but this is an essential sketch of 4.0's SC mechanism. The question, as usual, is: "does it work?"

2 Test Design

Our test suite for Aerospike uses the Jepsen testing library, and expands on our work from 2015. We check linearizable reads, writes, and compare-and-set over single registers, each backed by a single record in Aerospike. We test counters by performing continuous increments and reads, ensuring that the counter value remains within the lower and upper bounds given by the number of successful and potential increments, respectively. Finally, a new test, set, verifies that unique integers appended to a single record can be read back later.

We run Aerospike on a five node cluster of Debian Jessie machines in EC2, with *replication-factor* 2 or 3. We enable strong-consistency for our namespace,

and use linearizeRead in the client to ensure reads go through consensus. Since the Aerospike client automatically routes requests to arbitrary nodes in the cluster, we use a custom fork of the Java client which binds each client to a single node. This makes it easier to identify consistency errors that might be masked by all clients routing requests to the same server.

We should note that our tests are illustrate *worst-case* availability, simulating an environment where clients can only reach one server. In normal production deployments, clients may (depending on network topology and server faults) be able to fail over to alternative servers, improving effective availability.

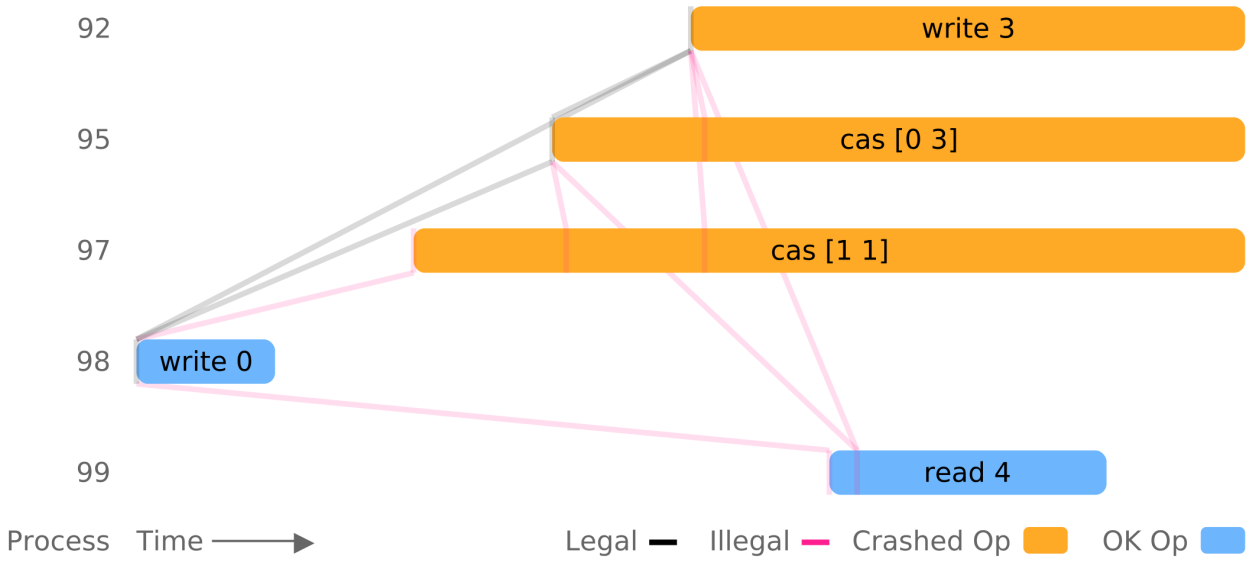
Aerospike's Java client throws exceptions for failure conditions, with an error code that identifies the type of error that occurred. Somewhat unexpectedly, the client can throw exceptions with error code 0: `AS_PROTO_RESULT_OK`, a code which indicates success. Users should be careful not to interpret these results as OK; they are likely indeterminate failures. The Jepsen client considers these errors indeterminate.

3 Results

3.1 Dirty Reads After Partitions

In order to verify linearizability during network failures, we introduce both total and partial network partitions while performing reads, writes, and compare-and-set operations on single registers. However, Aerospike's clustering algorithm requires unanimous agreement between all nodes participating in a sub-cluster, which implies that if if any node can see a different set of nodes than the nodes it's connected to, no sub-cluster can form, and all requests for data on affected shards will fail. We therefore restrict ourselves to total partitions, which Aerospike *can* handle.

Unfortunately, we encountered nonlinearizable histories in 3.99.0.3: after partitions, Aerospike would return apparently impossible values for reads. For instance, the very first successful operation on a key might be a read of, say, 2, when the value *hadn't even been written yet*. Or, as in this test, the value of key 9 could change from 0 to 4 with no concurrent write of 4.



In this case, process 98 wrote 0, just before a partition; processes 92, 95, and 97 crashed during that partition; and process 99 read 4 just after the partition resolved. This makes no sense: where did 4 come from?

Critically, this visualization does not include *failed* operations. A careful investigation of the raw history reveals that just *prior* to the read of 4, process 98 attempted to write 4, and received a failure code `:unavailable`:

```
{:type :invoke, :f :write, :value 4, :process 98, :time 23632780917}
{:type :fail, :f :write, :value 4, :process 98, :time 23633454385,
 :error :unavailable}
{:type :invoke, :f :write, :value 1, :process 192, :time 23759768734}
{:type :fail, :f :write, :value 1, :process 192, :time 23760579493,
 :error :unavailable}
```

```
{:type :invoke, :f :read, :value nil, :process 99, :time 23929420527}
{:type :ok, :f :read, :value 4, :process 99, :time 23931539207}
```

Over many tests a pattern emerges: nonlinearizable operations usually occur during the first few seconds after a partition has resolved. Moreover, those erroneous operations were typically reads of a value that *had* been written before—except that that write had failed with error code 11: `AS_PROTO_RESULT_FAIL_UNAVAILABLE`, which indicates that the given partition was not available for writes.

This occurs because when Aerospike proxies writes from one server to another, it may transparently retry indeterminate failures like timeouts. It then returns the *most recent* failure, not the *most conservative* failure. For instance, say a client issues a write to node A, which proxies it to B, where it is applied. However, B's response is lost due to a network failure. A transparently retries, and the second time around, B responds with a definite error: the partition is unavailable. A dutifully relays this message to the client, saying that the write was rejected, even though it successfully completed.

Proxies, in general, should not retry indeterminate operations; with the exception of idempotent and commutative structures like CRDTs, performing operations multiple times could lead to unsafe effects. 3.99.1.5 fixes this problem by disallowing all proxy retries.[4]

With these patches in place, Aerospike handles majority-minority partitions well. In 3.5.4's AP mode, Aerospike was immediately nonlinearizable and lost counter updates. In hundreds of tests of SC mode through network partitions, 3.99.1.5 and higher versions have not shown any sign of nonlinearizable histories, lost increments to counters, or lost updates to sets.

3.2 Node Crashes

To verify crash-safety, we introduce a mixture of polite (`SIGTERM`) & impolite (`SIGKILL`) process crashes, followed by process restarts.

An immediate problem arises when we crash Aerospike nodes with `SIGKILL`: the concurrent failure of *replication-factor* nodes takes down some fraction of the keyspace indefinitely. The cluster will not recover, even after nodes are restarted—some shards will remain down indefinitely.

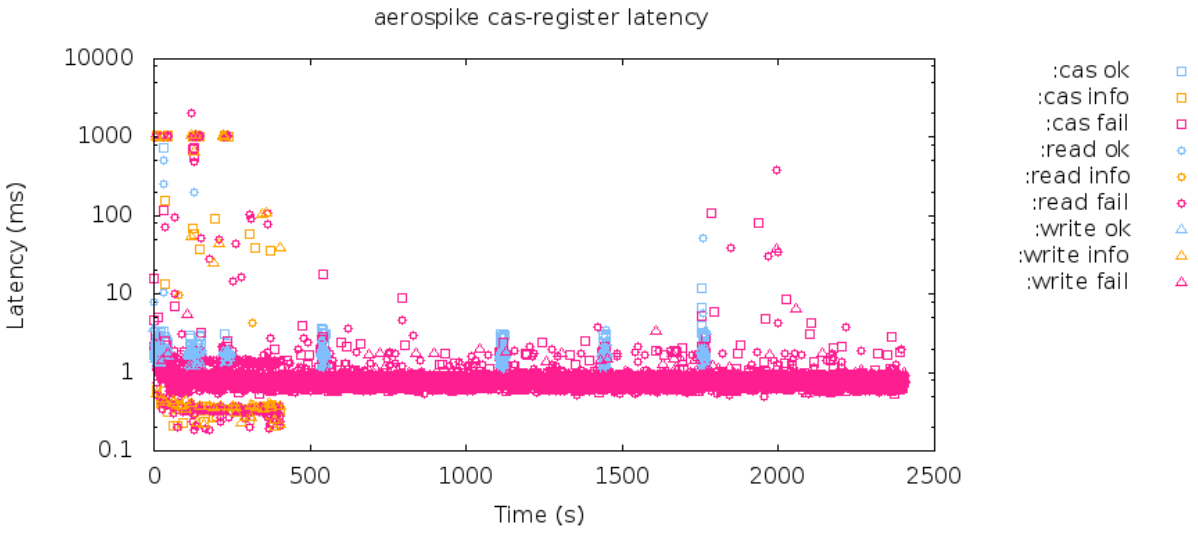


Figure 1: Plot of operation latencies over time, showing more and more shards transitioning to a dead state.

This plot shows the gradual failure of more and more shards due to nodes crashing and restarting, until only

[4]There is an additional issue with Aerospike: the Java client, by default, will retry operations, which could lead to operations being incorrectly applied multiple times, potentially losing updates. This is fixed in version 4.1.1 of the Java client, which defaults to no retries for writes and queries. Jepsen disables all client retries for safety.

a few shards are left alive—operations occasionally succeed as the test rotates to new keys (and therefore new shards) over time. During these failures, Aerospike will log messages like:

```
Nov 28 2017 20:32:34 GMT: WARNING (partition): (partition_balance_ee.c:649)
{jepsen} rebalanced: regime 119 expected-migrations (45,45) expected-signals 33
expected-appeals 0 dead-partitions 4029
```

Here, `dead-partitions` indicates that some shards are unavailable. This occurs when Aerospike suspects that data loss may have been possible and disables the shard to prevent further problems.

In order for Jepsen to see whether data was *lost*, we have to perform reads. That means we need to bring those dead shards back online. We do this by issuing a *revive* command: an explicitly unsafe operation available to operators for emergency recovery. With revives and reclusters, we observe Aerospike losing updates due to concurrent crashes, especially in set tests:

```
{:valid?          false,
 :lost            "#{648..671}",
 :recovered       "#{251}",
 :ok
 "#{0..251 258..647 1415..2673 2675..2903}",
 :recovered-frac  1/2904,
 :unexpected-frac 0,
 :unexpected      "#{}",
 :lost-frac       1/121,
 :ok-frac         355/484},
```

This occurs because Aerospike does not write records to disk before acknowledging their successful commit to the client. There is a short window of writes which are resident only in memory, and those writes may be lost if all current replicas of a given record crash concurrently.

Like MongoDB, Aerospike made the decision to write asynchronously for performance reasons—disk writes impose a latency penalty. Their rationale is that multiple nodes provide redundancy against single-node crashes, and concurrent crashes are relatively infrequent. However, rack and even datacenter-wide power failures do happen from time to time[5], and being able to guard against that eventuality is helpful for some users. Moreover, even sequential failures can result in data loss, if they happen in quick succession.

```
{:valid?          false,
  :lost           "#{69 109 169..194}",
  :recovered      "#{1770 2462 2634}",
  :ok
  "#{0..54 ... 2832..2937}",
  :recovered-frac  3/2950,
  :unexpected-frac 0,
  :unexpected      "#{}",
  :lost-frac       14/1475,
  :ok-frac         551/590}},
```

Even though every node in this test was fully restarted before another crashed, Aerospike still lost committed updates. When a node restarts from an unclean shutdown, it *knows* that it may have lost some data, and sets an error flag to prevent it from acting as a normal, up-to-date replica. Any committed write must be present on at least *replication-factor* replicas, and should be recoverable by contacting one of those other replicas. However, clearing the error flag allows the original replica to act authoritatively again, and if the other, *truly* up-to-date replicas are inaccessible (due to e.g. partition or crash), then Aerospike could use the original node's obsolete data as the basis for future updates—causing lost writes.

To handle this issue, Aerospike has added an optional feature to ensure writes are flushed to disk before acknowledging them. This feature is present in 3.99.2.1 and higher. With it enabled, node crashes no longer trigger data loss.[6]

```
{:valid?    true,
 :lost      "#{}",
 :recovered "#{201}",
 :ok
 "#{0..201 206 209 212..238 240..246
  248..268 546 549 551 602..682 686
  689 921 925 998..1016 1018..1060
  1064 1233..1278 1743 1769}",
 :recovered-frac  1/1978,
 :unexpected-frac 0,
 :unexpected      "#{}",
 :lost-frac       0,
 :ok-frac         229/989}}
```

3.3 Wall Clocks

We also introduce clock skew over randomized nodes. We use two strategies for adjusting clocks: either

[5] Currently, Aerospike may locate all replicas of a given shard on nodes in a single rack. Aerospike plans to add rack-aware replica placement in an upcoming release.

[6] Aerospike is also working to improve recovery safety for sequential crashes.

bumping the clock once by a random offset (milliseconds to minutes), or strobing the clock back and forth between two monotonically advancing times at a high (~milliseconds) frequency. The latter is useful for triggering timeouts early on selected nodes, and for confusing short-term latency measurements in code that doesn't use `CLOCK_MONOTONIC`. Neither of these randomized schedules revealed safety violations, though they frequently induced downtime—Aerospike pauses writes on nodes which, based on heartbeats, diverge by more than 25 seconds from other nodes in the cluster.

The fact that we didn't observe consistency errors with a randomized test schedule does not, however, imply that Aerospike is necessarily safe. There is a theoretical weakness in Aerospike's replication scheme—one which Aerospike was aware of and factored into their design. To understand this vulnerability, we need to discuss the SC algorithm in more detail.

Successive configurations of replicas are linked together with a sequence number called the *regime*, which is chosen as a part of each SC election. Regimes are incorporated into a clock structure which is associated with every record, and used to determine which version of a record is most recent.

$$clock = [regime, wall\text{-}clock, counter]$$

Ordinarily, these clocks would be compared by regime, then counter. However, Aerospike must pack this data into a small number of bytes in order to keep record access efficient.[7] At present, there are only six bits allocated to the regime. Rapid-fire elections could overflow the regime counter, causing older data to be have a higher regime than newer data. To avoid this problem, Aerospike relies on wall clocks: whenever a record's wall clock is more than 27 seconds[8] greater than another's, Aerospike assumes the regime may have wrapped, and *ignores* the regime in favor of the record with the higher wall clock.

This theoretical vulnerability is somewhat difficult to reproduce, because it requires that an old primary accept a write with a significantly higher wall clock than a logically newer primary, and primaries tend to step down quickly when their heartbeats expire. Moreover, the same heartbeat mechanism is used for promoting a new primary; so long as *timeouts* proceed in roughly real-time, the likelihood of obtaining concurrent, desynchronized primaries is relatively low.

However, when timeouts do *not* trigger reliably, we can observe consistency anomalies. Consider a single record, initially empty, and an append operation w_1 in flight to node A, the primary for regime 1. Let A pause before processing w_1. Since A is paused, it will not exchange heartbeat messages with the rest of the cluster, and a new primary B will be elected with regime 2. B can process a different append operation w_2 at time t_1, resulting in value "2"—and acknowledge completion of w_2 to the client. Some time later, let A resume. Since A still considers itself to be a primary, it will accept the first append w_1 at t_2, and store the resulting value "1" locally. Since a primary with a newer regime has been elected, A cannot replicate w_1 to its peers. However, when A rejoins the cluster, A and B must compare notes to identify the newest version of each record. Even though A's version comes from a lower regime (1 vs 2), if $t1 \ll t2$, then the regime will be ignored in favor of the wall clock, and A's version "1" will dominate. The acknowledged update w_2 will have been lost!

This scenario requires a quiet period while the process is paused; if additional updates are made to B, they will have a higher wall clock time, and might fall close enough to t_2 to allow the use of regime comparison.

We designed a combined client, generator, and nemesis to explore this possibility, and can confirm that Aerospike can lose updates when a node pauses for more than 27 seconds:

```
{:valid?          false,
 :lost
 "#{193384..193385 193389..193390 193393}",
 :recovered       "#{193230}",
 :ok
  "#{137603 ... 193238}",
 :recovered-frac  1/6118,
 :unexpected-frac 0,
 :unexpected      "#{}",
 :lost-frac       5/6118,
 :ok-frac         321/322}
```

This behavior is a consequence of using wall clocks for conflict resolution, and Aerospike uses wall-clocks because its sequence numbers (the regime) can quickly exhaust their 6-bit allotment, wrapping to 0. Consensus algorithms like Raft or Viewstamped Replication use an unbounded sequence number, which avoids the problem of wrapping at the cost of higher storage space. Aerospike was aware of this limitation during

[7] Master regimes, wall clocks, and counters are stored on a per-record basis in a header structure which should fit within a cache line; this places tight limits on the number of bits which can be allocated to each number.

[8] Aerospike's clock error tolerance is based on the maximum number of elections that can take place in that interval, which is inversely proportional to heartbeat intervals. Increasing the heartbeat interval slows down elections and lets Aerospike handle larger clock skews and pauses.

the design process, and intends to mitigate it by allocating more bits to the regime sometime after version 4.0. This should extend the window of clock error that Aerospike can tolerate.

Aerospike and Jepsen also believe that a node whose clock is bumped into the future, quickly followed by a process crash or network partition, could orphan far-future writes on that isolated node, which could go on to supercede successful writes on the remainder of the cluster. Due to time constraints, we were unable to experimentally confirm this behavior.

4 Discussion

Aerospike has made significant changes since the last Jepsen analysis, overhauling and simplifying their core components to improve stability, performance, and correctness. While the new strong consistency scheme is somewhat unorthodox and lacks any formal description or proof, our experimental research suggests that SC mode in 3.99.2.1 provides linearizable single-key operations so long as processes and clocks are reasonably well-behaved, and can tolerate total network partitions while retaining partial availability.

Foundational work to enable SC mode has also improved AP performance and stability. For many AP users, a small but predictable fraction of lost writes is perfectly acceptable, and reductions in the frequency and magnitude of consistency errors bring commensurate advantages to the application. We recommend that AP users upgrade to take advantage of these improvements, even if they choose not to use SC.

Note that Aerospike's AP mode does not offer a custom merge function or built-in CRDTs, so the only scenario in which AP mode is *not* subject to write loss is with immutable data, e.g. each record is only written once. However, AP mode may also be appropriate when the loss of small windows of writes poses little threat to user happiness. Where concurrent updates are common, and avoiding lost updates, dirty reads, and stale reads is important, users should try SC mode instead. Many Aerospike users have also developed homegrown partition detection and mitigation schemes for AP, such as shutting down isolated subclusters to prevent significant write loss. These users may also be well served by SC.

In SC mode, Jepsen recommends that clients enable `commit-to-device true` to ensure updates are written to disk before commit. This also simplifies operations: Aerospike should smoothly recover from concurrent or sequential crashes without an operator issuing manual, unsafe `revive` commands.

Users should also take care to run Aerospike on semi-realtime networks and computers. For instance, some virtualized environments may pause VMs for multiple minutes to migrate them to other physical nodes; this could cause the loss of data in Aerospike. With the default heartbeat settings, Aerospike can tolerate up to 27 seconds of combined clock skew and process pauses.

As Aerospike's early-access documentation notes, there are other operational concerns that could cause safety violations. Non-durable deletes and TTL expiration interact poorly with SC mode and are disabled by default; we advise operators not to re-enable them without careful thought. Operators should also take care not to remove more than *replication-factor* nodes from the cluster's roster in a single step, or data loss could result. Aerospike may also silently lose data if some sectors or portions of a drive are erased, e.g. due to device failure or operator accident.

SC mode only applies to single-key operations. Although Aerospike bills itself as a "transactional" store, multi-key operations like scans, queries, and aggregates have no notion of a transactional, consistent snapshot across keys, do not follow CP mode's rules for identifying the correct version of divergent copies, and consequently allow dirty and stale reads. User defined functions (UDFs) should be treated with caution; read operations don't yet linearize, and multiple commit points in a transaction may result in partially-completed transactions incompletely replicated between nodes. Aerospike plans to improve these systems in future iterations of SC.

In addition to the standard cluster replication algorithm we've discussed so far, Aerospike has a cross-datacenter replication (XDR) system, which employs log-shipping for flexible uni- and bi-directional replication between different clusters. XDR allows stale reads (since replication is asynchronous), and in bidirectionally replicated clusters, may lose concurrent updates:

> Since XDR allows client writes to happen to both clusters, two clients may simultaneously write to the same key on both clusters, which leads to inconsistent data. As the Aerospike database is agnostic of the application data, it cannot auto-correct this inconsistency.

Our tests did not cover cross-datacenter replication, but Aerospike plans to research XDR strategies for SC clusters going forward.

Aerospike has ported specific Jepsen scenarios to their internal test suite, and begun integrating Jepsen into their test infrastructure. However, experimental testing can only demonstrate the presence of bugs, not their absence. To gain more confidence in Aerospike's safety, a formal specification and proof of the consensus algorithm would be helpful. Aerospike has begun this work, but model-checking and proof will take some time.

In future research, we would like to experimentally confirm the possibility of data loss under clock skew followed by crashes or partitions, and develop tests for UDFs, queries, scans, and aggregations.

This work was funded by Aerospike, and conducted in accordance with the Jepsen ethics policy. Jepsen wishes to thank the entire Aerospike team for their invaluable assistance, especially Andrew Gooding and Kevin Porter. We are also grateful to Brad Greenlee, Camille Fournier, & André Arko for their comments on early drafts.