RethinkDB 2.1.5

2016-01-04

In this Jepsen report, we'll verify RethinkDB's support for linearizable operations using `majority` reads and writes, and explore assorted read and write anomalies when consistency levels are relaxed. This work was funded by RethinkDB, and conducted in accordance with the Jepsen ethics policy.

1 Background

RethinkDB is an open-source, horizontally scalable document store. Similar to MongoDB, documents are hierarchical, dynamically typed, schemaless objects. Each document is uniquely identified by an `id` key within a table, which in turn is scoped to a DB. On top of this key-value structure, a composable query language allows users to operate on data within documents, or across multiple documents—performing joins, aggregations, etc. However, only operations on a single document are atomic—queries which access multiple keys may read and write inconsistent data.

RethinkDB shards data across nodes by primary key, maintaining replicas of each key across *n* nodes for redundancy. For each shard, a single replica is designated a *primary*, which serializes all updates (and strong reads) to that shard's documents—allowing linearizable writes, updates, and reads against a single key.

If a primary dies, we're in a bit of a pickle. Rethink can still offer stale reads from any remaining replica of that shard, but in order to make any changes, or perform a linearizable read, we need a new primary—and one which is guaranteed to have the most recent committed state from the previous primary. Prior to version 2.1, RethinkDB would not automatically promote a new primary—an operator would have to ensure the old primary was truly down, remove it from the cluster, and promote a remaining replica to a primary by hand. Since every node in a cluster is typically a primary for some shards, the loss of any single node would lead to the unavailability of ~1/n of the keyspace.

RethinkDB 2.1, released earlier this year, introduced automatic promotion of primaries using the Raft consensus algorithm. Every node hosting a shard in a given table maintains a Raft ensemble, which stores table membership, shard metadata, primary roles, and so on. This allows RethinkDB to automatically promote a new primary replica for a shard so long as:

1. A majority of the table's nodes are fully connected to one another, and
2. A majority of the shard's replicas are available to the table's majority component

So: if we have at least three nodes, and at least three replicas per shard, a RethinkDB table should seamlessly tolerate the failure or isolation of a single node. Five replicas, and the cluster will tolerate two failures, and so on. This majority-quorum strategy is common for linearizable systems like etcd, Zookeeper, or Riak's strong buckets—and places Rethink in similar availability territory to MongoDB, Galera Cluster, et al.

RethinkDB also supports *non-voting replicas*, which asynchronously follow the normal replicas' state, are not eligible for automatic promotion, and don't take part in the Raft ensemble. These replicas do not provide the usual Rethink consistency guarantees, and are best suited to geographic redundancy for disaster recovery, or for read-heavy workloads where consistency isn't important. In this analysis, every replica is a voting replica.

2 Consistency guarantees

Like MongoDB, Riak, Cassandra, and other KV stores, RethinkDB does not offer atomic multi-key operations. In this analysis, we'll concern ourselves strictly with single-key consistency.

RethinkDB chooses strong defaults for update consistency, and weak defaults for reads. By default, updates

(inserts, writes, modifications, deletes, etc) to a key are linearizable, which means they appear to take place atomically at some point in time between the client's request and the server's acknowledgement. For reads, the default behavior is to allow any primary to service a request using its in-memory state, which could allow stale or dirty reads.

Like Postgres, RethinkDB defaults will not acknowledge writes until they're fsynced to disk. Users may obtain better performance at the cost of crash safety by relaxing the table's or request's `durability` setting from `hard` to `soft`. As with all databases, the filesystem, operating system, device drivers, and hardware must cooperate for fsync to provide crash safety. In this analysis, we use `hard` durability and do not explore crash safety.

Some databases let you tune write safety on a per-transaction basis, which can lead to confusing semantics when weakly-isolated transactions interleave with stronger ones. RethinkDB, like Riak, enforces write safety at the *table* level: all updates to a table's keys use the same transactional isolation. A table's `write_acks` can be either:

- `single`: A primary can acknowledge a write to a client without the acknowledgement of other replicas, or
- `majority`: A majority of replicas must acknowledge a write first

The difference is in request latency; operations must be fully replicated at *some* point, so both modes have the same throughput, and writes always go to a primary, so a majority quorum must be present for write availability. This differentiates RethinkDB from AP databases like Cassandra, Riak, and Aerospike, which offer total write availability at the cost of linearizability, sequential consistency, etc.

Unlike write safety, read safety is controllable on a *per-request basis*. This makes sense: reads never impact the correctness of other reads, and relaxed consistency is often preferable for large read-only queries which can tolerate some fuzziness—e.g. analytics. RethinkDB offers three `read_mode` flavors:

- `outdated`: The local in-memory state of any replica
- `single`: The local in-memory state of any replica which thinks it's a primary
- `majority`: Values safely committed to disk on a majority of replicas

`Outdated` and `single` are, I believe, equivalent in terms of safety guarantees, though `outdated` will likely exhibit read anomalies *constantly*, where `single` should only show dirty or stale reads during failures. `Outdated` can improve availability, latency and throughput, because all replicas can serve reads, not just primaries. `Majority` is much stronger, offering linearizable reads, but at the cost of a special sync request to every replica, to which a majority must respond before the read can be returned to the client. The sync requests could be omitted, piggybacking leader state on existing write traffic, but any linearizable op must still incur an additional round-trip's worth of latency to ensure consensus.

So, how do these features interact? My interpretation, conferring with the Rethink team, is:

	w=single	w=majority
r=outdated	Lost updates, dirty reads, stale reads	Dirty reads, stale reads
r=single	Lost updates, dirty reads, stale reads	Dirty reads, stale reads
r=majority	Lost updates, stale reads	Linearizable

Rethink's documentation claims that `majority/majority` guarantees linearizability. The other cases are a little trickier.

If writes are relaxed to `single`, we could see a situation where an isolated primary accepts a write which is later lost, because it has not been acknowledged by a majority of nodes. Subsequently, a disjoint majority of replicas can service reads of an *earlier value*: a stale read. A read against that isolated primary could not, however, succeed with `r=majority`, likely preventing dirty reads.

If writes are performed at `majority`, we prevent lost updates—because any write must be acknowledged by a majority of nodes, and therefore be present on any subsequently elected primary. Stale reads are still possible at `r=single`, because an isolated primary (or

any node, for r=outdated) could serve read requests, while a newer primary accepts writes. We could also encounter *dirty reads*: a single or outdated read could see a majority write while it's being propagated to other replicas, but before those replicas have responded. If the primary does not receive acknowledgement from a majority of replicas, and a new primary is elected without that write, the write would have failed—yet still have been visible to a client.

Finally, single writes and single reads should allow all anomalies we discussed above: lost updates from the lack of majority writes, plus dirty and stale reads.

So, the question becomes: does RethinkDB truly offer linearizable operations at majority/majority? Are writes still safe with single reads? Are these anomalies purely theoretical, or observable in practice? Let's write a test to find out.

2.1 Setup

Installation for Rethink is fairly straightforward. We simply add their debian repository, install the rethinkdb package, and set up a log file. To prevent RethinkDB from exploiting synchronized clocks, we'll use a libfaketime shim to skew clocks and run time at a different rate for each process.

We construct a configuration file based on the stock config. We want to create as many leadership transitions as possible in a short time, so we lower the heartbeat timeout to two seconds once the cluster is up and running.

We tell Jepsen how to install, configure, and start each node, how to clear the logs and data files between runs, and what logfiles to snarf at the end of each test. After starting the DB, we spin awaiting a connection to each node. Jepsen ensures that every node reaches this point before beginning the test.

2.2 Operations

With the database installed, we turn to test semantics. Clients in Jepsen take *invocation operations*, apply those ops to the system under test, and return corresponding *completion operations*. Our operations will consist of writes, reads, and compare-and-sets ("cas", for short). We define *generator functions* that construct these operations over small integers.

```
(defn w   [_ _] {:type :invoke, :f :write,
                 :value (rand-int 5)})
(defn r   [_ _] {:type :invoke, :f :read})
(defn cas [_ _] {:type :invoke, :f :cas,
                 :value [(rand-int 5)
                         (rand-int 5)]})
```

For instance, we might generate a write like {:type :invoke, :f :write, :value 2}, or a compare-and-set like {:type :invoke, :f :cas, :value [2 4]}, which means "set the value to 4 if, and only if, the value is currently 2."

In prior Jepsen analyses, we'd operate on a single key throughout throughout the entire test, which is simple, but comes at a cost. As the test proceeds and Bad Things(TM) happen to the database, more and more processes will time out or crash. We *cannot tell* whether a crashed process's operations will take place now, or in five years, which means as the test goes on, the number of concurrent operations gradually rises. The number of *orders* for those operations rises *exponentially*: at every juncture, we must take all permutations of every possible subset of pending ops. Any more than a handful of crashed processes, and linearizability verification can take *years*.

This places practical limits on the duration and request rate of a linearizability test, which makes it harder to detect anomalies. We need a better strategy.

I redesigned the Knossos linearizability checker based on the linear algorithm described by Gavin Lowe. Extensive profiling and optimization work led to significant speedups for pathological histories that the original Knossos algorithm choked on. This implementation performs the just-in-time linearization partial order reduction proposed by Lowe, with additional optimizations: we precompute the entire state space for the model, which allows us to explore configurations without actually *calling* the model transition code, or allocating new objects. Because we re-use the same model objects, we can precompute their hashcodes and use reference equality for comparisons—which removes the need for Lowe's union-find optimization. Most importantly, determinism allows us to skip the exploration of equivalent configurations, which dramatically prunes the search space.

Unfortunately, these optimizations were not enough: tests of longer than ~100 seconds would bring the checker to its knees. Peter Alvaro suggested a key insight: we may not need to analyze a *single* register over the lifespan of the whole test. If linearizability violations occur on short timescales, we can operate over *several distinct keys* and analyze each one *independently*. Each key's history is short enough to analyze, while the test as a whole encompasses tens to

hundreds of times more operations—each one a chance for the system to fail.

A new namespace, jepsen.independent, supports these kinds of analyses. We can lift operations on a single key, like `{:f :write, :value 3}` to operations on `[key value]` tuples, e.g., to write 3 to key 1, we'd invoke `{:f :write, :value [1 3]}`. We use sequential-generator to construct operations over integer keys, and for each key, emitting a mix of reads, writes, and compare-and-set operations, one per second, for sixty seconds. Then `sequential-generator` begins anew with the next key.

```
:concurrency 10
:generator (std-gen (independent/sequential-generator
             (range)
             (fn [k]
               (->> (gen/reserve 5 (gen/mix [w cas])
                                 r)
                    (gen/delay 1)
                    (gen/limit 60)))))
```

`gen/reserve` assigns five of our ten processes to `(gen/mix [w cas])`: a random mixture of write and compare-and-set operations. The remaining processes perform reads.

`reserve` is critical for identifying dirty and stale reads. When we partition the network, updates can stall for 5-10 seconds because a majority is no longer available. If we don't reserve specific processes for reads, every process would (at some point) attempt a write, block on leader election, and for a brief window *no operations would take place*—effectively blinding the test to consistency violations. By dedicating some processes to reads, we can detect transient read anomalies through these transitions.

2.3 Writing a client

With our operations constructed, we need a client which takes these operations and performs them against a RethinkDB cluster. At startup, one client creates a fresh table for the test, with five replicas—one for each node. We tell Rethink to use a particular write-acks level for the table, and establish a replica on each node. Once the table is configured, we wait for our changes to propagate.

In response to operations, we extract the `[key, value]` tuple from the operation, and construct a Rethink query fragment for fetching that key, using the specified `read-mode`. Then we dispatch based on the type of the operation. Reads simply run a get query, extracting a single field `val` from the document. We return the read value as the `:value` for the completion op.

```
:read (assoc op
             :type  :ok
             :value (independent/tuple
                      id
                      (r/run (term :DEFAULT
                                   [(r/get-field row "val") nil])
                             (:conn this))))
```

Writes behave similarly: we perform an upsert using `conflict: update`, setting the `val` field to the write op's value.

```
:write (do (run! (r/insert (r/table (r/db db) tbl)
                           {:id id, :val value}
                           {"conflict" "update"})
                 (:conn this))
           (assoc op :type :ok))
```

Compare and set takes advantage of Rethink's functional API: we use `update` against the row query fragment, providing a function of the current row which dispatches based on the current value of `val`. If `val` is equal to the compare-and-set predicate `value`, we update the document to `value'`. Otherwise, we abort the update. RethinkDB returns a map with the number of errors and replaced rows, which we use to determine whether the compare-and-set succeeded.

```
:cas (let [[value value'] value
           res (r/run
                 (r/update
                   row
                   (r/fn [row]
                     (r/branch
                       (r/eq (r/get-field row "val") value) ; predicate
                       {:val value'}                        ; true branch
                       (r/error "abort"))))                 ; false branch
                 (:conn this))]
       (assoc op :type (if (and (= (:errors   res) 0)
                                (= (:replaced res) 1))
                         :ok
                         :fail))))))
```

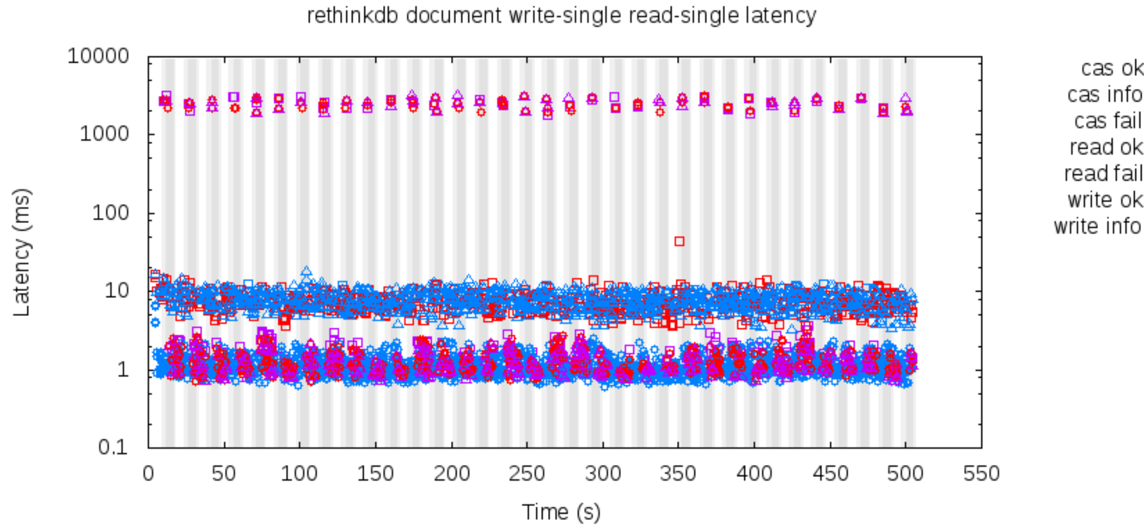
While this is more verbose than a native compare-and-set operator, Rethink's compositional query language and first-class support for functions and control flow primitives allows for fine-grained control over single-document transformations. Like SQL's stored procedures, shipping complex logic to the database can improve locality and cut round-trips. The API, AST, and serialization format are structurally similar to one another, which I prefer to the query-string mangling ubiquitous in SQL libraries—and it's relatively easy to wrap up queries as parameterizable functions. However, the absence of multi-document concurrency control prevents Rethink's query language from reaching its full potential: we cannot (in general) safely read a value from one document and use it to update another, or make two updates and guarantee their simultaneous visibility.

Happily, RethinkDB explicitly distinguishes between *failed* and *indeterminate* operations with a dedicated status code. We use a small macro to trap Rethink's exceptions and construct an appropriate completion operation based on whether the request failed (`:fail`), or *might have failed* (`:info`). Since reads are pure, we can also consider all read errors outright failures, which lowers the number of crashed ops and reduces load on the linearizability checker.

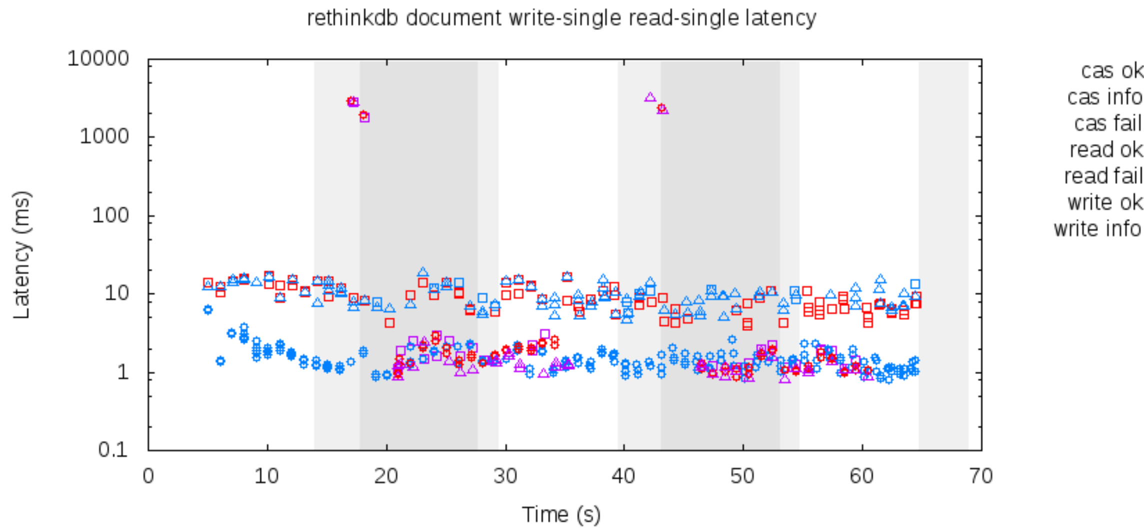
We must also be careful to check the return values for each write: RethinkDB throws exceptions for catastrophic cases, but to allow for partial failures, Rethink's client will return maps with error counts *instead* of throwing for all errors.

2.4 Availability

We'll run these tests for about 500 seconds, while cutting the network in half every ten seconds. Experimentation with partially isolated topologies, SIGSTOP/SIGCONT, isolating primaries only, etc. has so far yielded equivalent results to a simple majority/minority split. Shorter timescales can confuse RethinkDB for longer periods, as it struggles through multiple rounds of leader timeout and election, but it reliably recovers given a stable configuration for ~`heartbeat_timeout` + 10 seconds, and often faster.

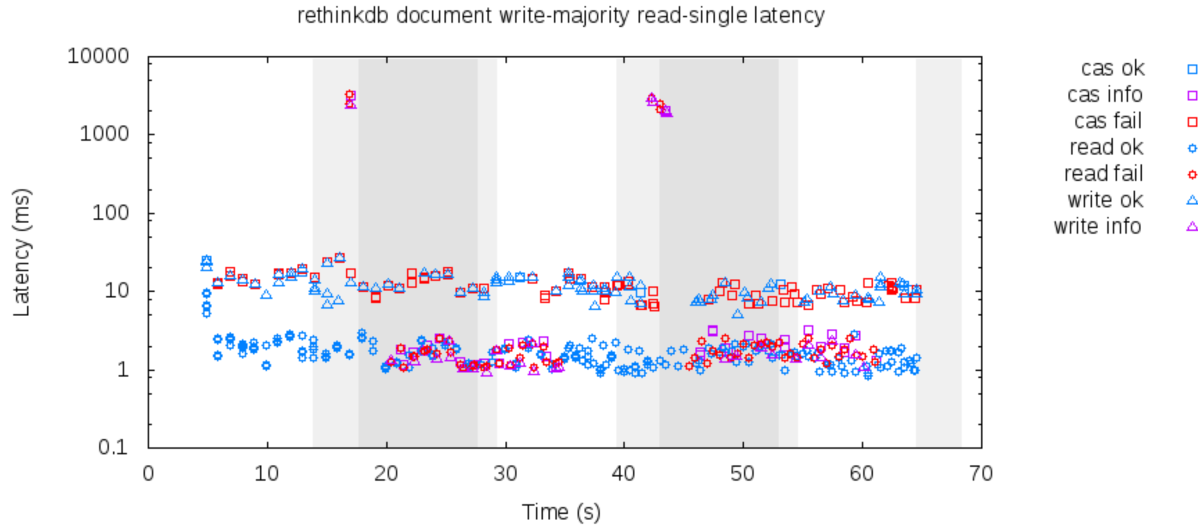


This plot shows the latency of operations over a full 500-second test: blue ops are successful, red definitely failed, and purple ones might have succeeded or might have failed. Grey regions show when the network was partitioned. We can see that each partition is accompanied by a brief spike in latency—around 2.5 seconds—before operations resume as normal.

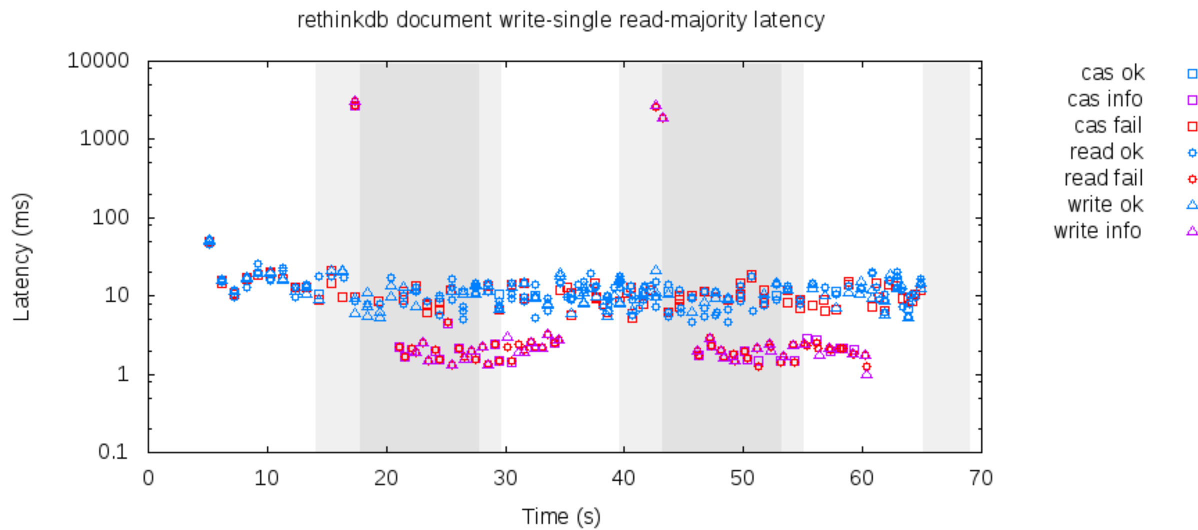


On a shorter timescale, we can see the recovery behavior in finer detail. Rethink delivers a few transient failures or crashes for reads, writes, and compare-and-set operations just after a partition begins. After cluster reconfiguration, we see *partial* errors for all three classes of operations, until the partition resolves and the cluster heals. Once it heals, we return to a healthy pattern. Note that compare-and-set (cas) failures are *expected* in this workload: they only succeed if they guess the current value correctly.

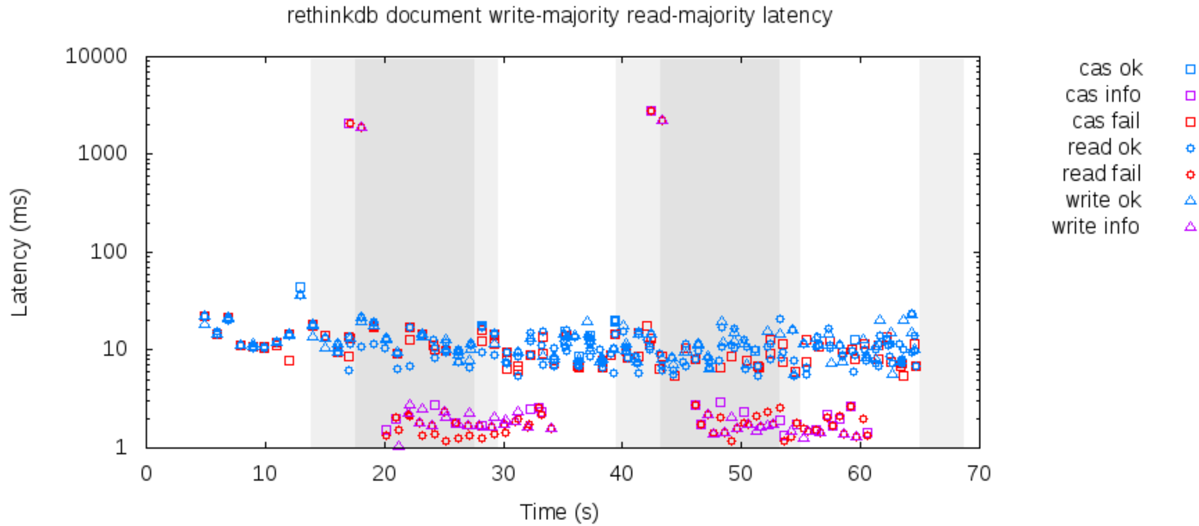
Why do we see partial failure? Because even though reads and writes only require a single node to acknowledge, Rethink's design mandates that those operations take place on a *primary* node—and even after reconfiguration, some nodes won't be able to talk to a primary!



We see a similar pattern for majority writes and single reads: because the window for concurrent primaries is short, there's not a significant difference in availability or latency.



However, a marked change occurs if we choose majority reads and single writes: read latencies, formerly ~2-3 times lower than writes, jump to the level of writes. This is because Rethink implements majority reads by issuing an empty write to ensure that the current primary is still legal. Once the old primary steps down, operations against the minority side fail or crash quickly.



The scale is slightly different for this graph of `majority/majority`, but the numbers are effectively the same. Majority writes don't have a significant impact on availability, because even single writes have to talk to a primary—and in order to have a primary, you need a majority of nodes connected. There should be a difference in client latency, but this network is fast enough that other costs dominate.

3 Linearizability

So, Rethink fails over reliably within a few seconds of a partition, and offers high availability—though not *total* availability. We also suspect that during these transitions, Rethink could exhibit lost updates, dirty reads, and stale reads, depending on the `write_acks` and `read_mode` employed. Only `majority/majority` should ensure linearizability.

3.1 Single writes, single reads

Consider this fragment of a history just after the network partitions, using `single` reads and writes. Each line shows a process (194) invoking and completing (`:ok`), crashing (`:info`), or failing (`:fail`) an operation. Here the key being acted on is 15, and we begin by reading the value 0.

```
9   :ok     :read  [15 0]
...
194 :invoke :write [15 3]
7   :fail   :read  [15 nil]    "Cannot perform read: lost contact with primary
replica"
292 :info   :cas   [15 [0 1]] "Cannot perform write: lost contact with primary
replica"
194 :ok     :write [15 3]
141 :info   :write [15 3]     "Cannot perform write: lost contact with primary
replica"
6   :fail   :read  [15 nil]    "Cannot perform read: lost contact with primary
replica"
8   :fail   :read  [15 nil]    "Cannot perform read: lost contact with primary
replica"
373 :info   :cas   [15 [1 0]] "Cannot perform write: lost contact with primary
replica"
```

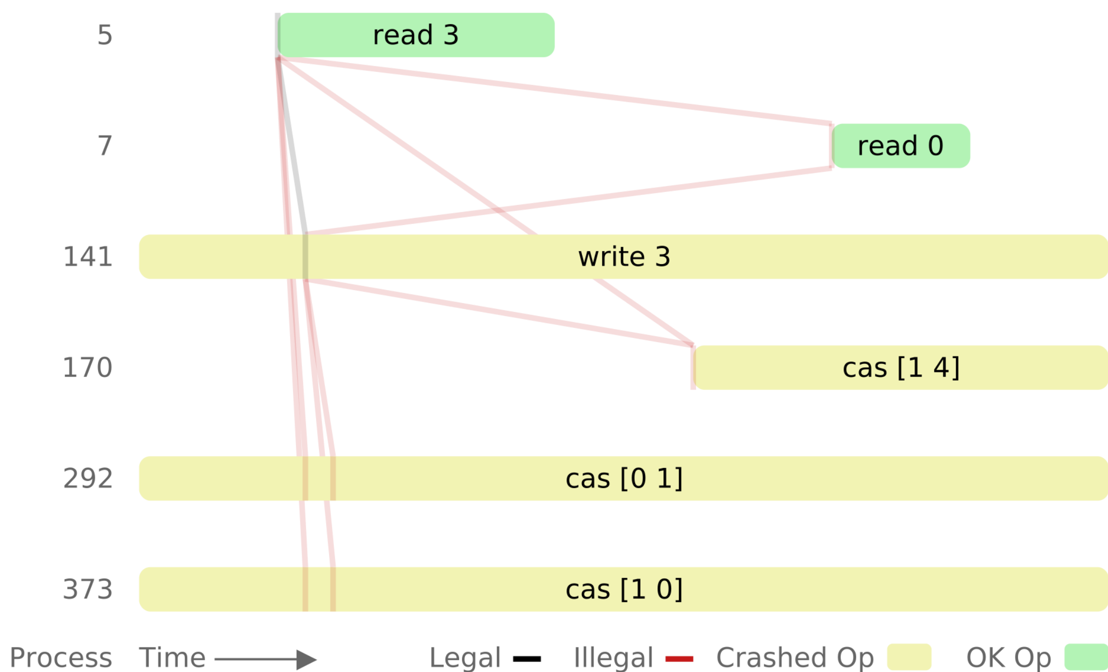
```
5   :invoke :read  [15 nil]
5   :ok     :read  [15 3]
170 :invoke :cas   [15 [1 4]]
170 :info   :cas   [15 [1 4]] "Cannot perform write: The primary replica isn't
connected to a quorum of replicas. The write was not performed."
9   :invoke :read  [15 nil]
9   :fail   :read  [15 nil]   "Cannot perform read: The primary replica isn't
connected to a quorum of replicas. The read was not performed, you can do an
outdated read using `read_mode=\"outdated\"`."
7   :invoke :read  [15 nil]
302 :invoke :cas   [15 [0 0]]
7   :ok     :read  [15 0]
194 :invoke :cas   [15 [3 0]]
302 :fail   :cas   [15 [0 0]]
194 :info   :cas   [15 [3 0]] "Cannot perform write: The primary replica isn't
connected to a quorum of replicas. The write was not performed."
151 :invoke :cas   [15 [1 0]]
6   :invoke :read  [15 nil]
6   :ok     :read  [15 0]
```

Did you catch that? I sure didn't, but luckily computers are good at finding these sorts of errors. Process 194 writes 3, which is read by process 5, and then, for no apparent reason, process 7 reads 0—a value from earlier in the test. Knossos spots this anomaly, and tells us that somewhere between line 73 (read 3) and line 80 (read 0), it was impossible to find a legal linearization. It also knows exactly what crashed operations were pending at the time of that illegal read:

```
:failures
{15
 {:valid? false,
  :configs
  ({:model   {:value 3},
    :pending [{:type :invoke, :f :read, :value 0,     :process 7,   :index 78}
              {:type :invoke, :f :cas,  :value [1 4], :process 170, :index 74}
              {:type :invoke, :f :cas,  :value [1 0], :process 373, :index 48}]}
     ... a few dozen other configurations
    })
  :previous-ok {:type :ok, :f :read, :value 3, :process 5, :index 73},
  :op          {:type :ok, :f :read, :value 0, :process 7, :index 80}}}},
```

This is *still* hard to reason about, so I've written a renderer to show the failure visually. Time flows from left to right, and each horizontal track shows the activity of a single process. Horizontal bars show the beginning and completion of an operation, and the color of a bar shows whether that operation was successful (green) or crashed (yellow).

If the history is linearizable, we should be able to draw *some* path which moves strictly to the right, touching every green operation—and possibly, crashed operations, since they *may* have happened. Legal paths are shown in black, and the resulting states are drawn as vertical lines in that operation. When a transition would be *illegal*, its path and state are shown in red. Hovering over any part of a path highlights every path that touches that component. For clarity, we've collapsed most of the equivalent transitions, so you'll see multiple paths highlighted at once.

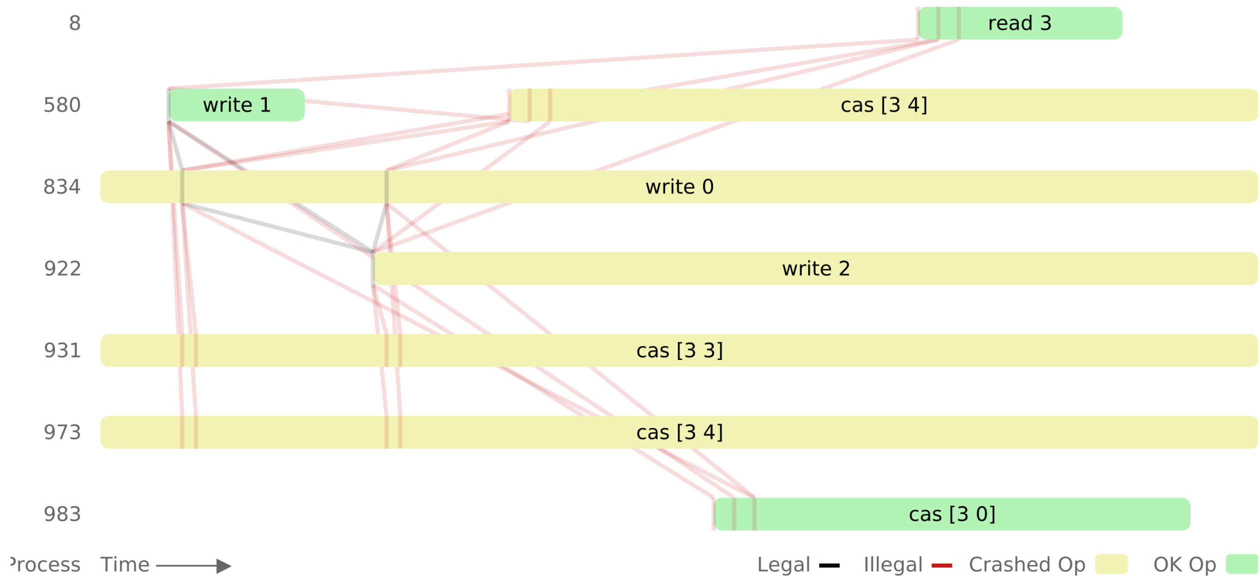


Hovering over the top line tells us that we can't simply read 3 then read `0`: there has to be a write in between in order for the state to change. We have a few crashed operations from earlier in the history that *might* take effect during this time—a write of 3 by process 141, and three compare-and-set operations. The write could go through, as shown by the black line from `read 3` to `write 3`, but that doesn't help us reach `0`. *No linearization exists*. Given subsequent reads see `0`, and multiple processes attempted to write 3 just prior, I suspect the read of 3 is either a lost update or a stale read: both expected behaviors for `single/single` mode.

3.2 Single writes, majority reads

When we use single writes and majority reads, it's still possible to see *lost updates*.

```
{:valid? false,
 :previous-ok {:type :ok, :f :write, :value 1, :process 580, :index 181},
 :op          {:type :ok, :f :read,  :value 3, :process 8,   :index 200}}}},
 :configs
    ({:model {:value 1},
      :pending [{:type :invoke, :f :read, :value 3,     :process 8,   :index 199}
                {:type :invoke, :f :cas,  :value [3 4], :process 580, :index 196}
                ... eighty zillion lines ...]})}
```

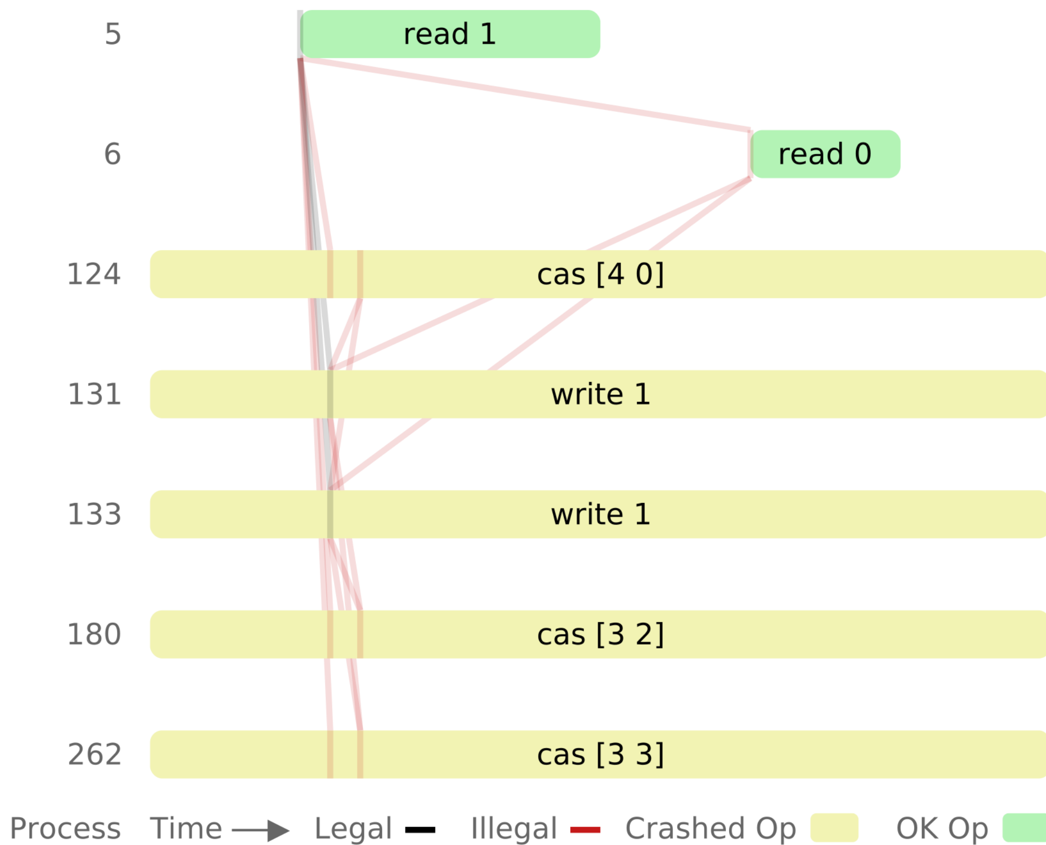


In this history, process 580 successfully writes 1, and two subsequent operations complete which require the value to be 3. None of the crashed operations allow us to reach a state of 3; it's as if the write of 1 never happened. This is a lost write, which occurs when a primary acknowledges a write *before* having fully replicated it, and a new primary comes to power without having seen the write. To prevent this behavior, we can write with `majority`.

3.3 Majority writes, single reads

As expected, we can also find linearization anomalies with `majority` writes and `single` reads. Before a partition begins, this register is 0. We have a few crashed writes of 1, which are visible to processes 5 and 8—then process 6 reads 0 again. Eliding some operations:

```
8        :ok      :read    [10 0]
...
133      :invoke :write    [10 1]
131      :invoke :write    [10 1]
:nemesis         :info    :start   "Cut off {:n4 #{:n3 :n2 :n5},
:n1 #{:n3 :n2 :n5}, :n3 #{:n4 :n1}, :n2 #{:n4 :n1}, :n5 #{:n4 :n1}}"
5        :invoke :read    [10 nil]
5        :ok     :read    [10 1]
9        :invoke :read    [10 nil]
8        :invoke :read    [10 nil]
8        :ok     :read    [10 1]
5        :invoke :read    [10 nil]
5        :ok     :read    [10 1]
8        :invoke :read    [10 nil]
8        :ok     :read    [10 1]
...
5        :invoke :read    [10 nil]
5        :ok     :read    [10 1]
...
6        :invoke :read    [10 nil]
141      :invoke :cas     [10 [4 0]]
6        :ok     :read    [10 0]
```



It's difficult to tell exactly what happened in these kinds of histories, but I suspect process 131 and/or 133 attempted a write of 1, which became visible on a primary *just before* that primary was isolated by a partition. Those writes crashed because their acknowledgements never arrived, but until the old primary steps down, other processes on that side of the partition can continue to read that uncommitted state: a dirty read. Once the new primary steps up, it *lacks* the write of `1` and continues with the prior state of `0`.

3.3.1 Majority writes, majority reads

I've run hundreds of test against RethinkDB at `majority/majority`, at various timescales, request rates, concurrencies, and with different types of failures. Consistent with the documentation, I have never found a linearization failure with these settings. If you use `hard` durability, `majority` writes, and `majority` reads, single-document ops in RethinkDB appear safe.

3.4 Discussion

As far as I can ascertain, RethinkDB's safety claims are accurate. You can lose updates if you write with anything less than `majority`, and see assorted read anomalies with `single` or `outdated` reads, but `majority/majority` appears linearizable.

Rethink's defaults prevent lost updates (offering linearizable writes, compare-and-set, etc), but do allow dirty and stale reads. In many cases this is a fine tradeoff to make, and significantly improves read latency. On the other hand, dirty and stale reads create the potential for lost updates in non-transactional read-modify-write cycles. If one, say, renders a web page for a user based on dirty reads, the user could take action based on that invalid view of the world, and cause invalid data to be written back to the database. Similarly, programs which hand off state to one another through RethinkDB could lose or corrupt state by allowing stale reads. Beware of sidechannels.

Where these anomalies matter, RethinkDB users should use `majority` reads. There is no signifi-

cant availability impact to choosing `majority` reads, though latencies rise significantly. Conversely, if read availability, latency, or throughput are paramount, you can use `outdated` reads with essentially the same safety guarantees as `single`—though you'll likely see continuous, rather than occasional, read anomalies.

Rethink's safety documentation is generally of high quality—the only thing I'd add is a description of the allowable read anomalies for weaker consistency settings. Most of my feedback for the RethinkDB team is minor: I'd prefer standardizing on either thrown or checked errors for all types of operations, instead of a mix, to prevent cases where users assume exceptions and forget to check results. Error codes for indeterminate vs definite failures are new in the Clojure client, and requires knowing some magic constants, but I appreciate their presence nonetheless.

I've hesitated to recommend RethinkDB in the past because prior to 2.1, an operator had to intervene to handle network or node failures. However, 2.1's automatic failover converges reasonably quickly, and its claimed safety invariants appear to hold under partitions. I'm comfortable recommending it for users seeking a schema-less document store where inter-document consistency isn't required. Users might also consider MongoDB, which recently introduced options for stronger read consistency similar to Rethink—the two also offer similar availability properties and data models. For inter-key consistency, a configuration store like Zookeeper, or a synchronously replicated SQL database like Postgres might make more sense. Where availability is paramount, users might consider an AP document or KV store like Couch, Riak, or Cassandra.

I've quite enjoyed working with the Rethink team on this analysis—they'd outlined possible failure scenarios in advance, helped me refine tests that weren't aggressive enough, and were generally eager to see their system put to the test. This sort of research wanders through all kinds of false starts and dead ends, but Rethink's engineers were always patient and supportive of my experimentation.

You can reproduce these results by setting up your own Jepsen cluster, checking out the Jepsen repo at `6cf557a`, and running `lein test` in the `rethink/` directory. Jepsen will run tests for all four read/write combinations and spit out analyses in `store/`. This test relies on unreleased features from clj-rethinkdb `0.12.0-SNAPSHOT`, which you can build locally by cloning their repo and running `lein install`.

This work was funded by RethinkDB, and conducted in accordance with the Jepsen ethics policy. I am indebted to Caitie McCaffrey, Coda Hale, Camille Fournier, and Peter Bailis for their review comments. My thanks as well to the RethinkDB team, especially Daniel Mewes, Tim Maxwell, Jeroen Habraken, Michael Lucy, and Slava Akhmechet.