# CockroachDB beta-20160829

2017-02-16

*CockroachDB is a distributed, scale-out SQL database which relies on hybrid logical clocks to provide serializability, given semi-synchronized node clocks. In this Jepsen analysis, we'll discuss multiple serializability violations in CockroachDB beta-20160829 through beta-20160908. As a result of our collaboration, fixes for these issues are included in beta-20160915 and beta-20161013. This work was funded by Cockroach Labs, and conducted in accordance with the Jepsen ethics policy. Cockroach Labs has also written a blog post with more context.*

## 1 Background

CockroachDB is a distributed SQL database, loosely patterned after Google Spanner and designed for semi-synchronous networks. It speaks the PostgreSQL wire protocol, supports a reasonable dialect of SQL, and combines replication for durability with transparent scale-out sharding for large tables. Its most significant limitations (and remember, this database is still in beta) might be the lack of efficient joins and overall poor performance—but in recent months the Cockroach Labs team has made significant progress on these issues, as they work towards a general release.

CockroachDB looks a good deal like Spanner, but has some important distinctions. Both use two-phase commit for transactions across consensus groups, but where Spanner uses locking, CockroachDB uses an optimistic concurrency scheme with transaction aborts. Where Spanner has tight real-time bounds via TrueTime, CockroachDB targets environments with much less reliable clocks, and uses a Hybrid Logical Clock for transaction timestamps. Running on commodity hardware, its clock offset limits are significantly higher; to obtain acceptable performance, it sacrifices external consistency and provides only serializability.[1] Where Spanner waits after every write to ensure linearizability, CockroachDB blocks only on contested reads. As a consequence, its consistency guarantees are slightly weaker.

We've discussed a few SQL databases in the past year: Galera Cluster, which targets snapshot isolation, and VoltDB, which offers strict serializability. CockroachDB falls between the two, offering serializability, with limited real-time constraints, but not *strict* serializability—it's a complicated blend.

At the time of this analysis (October 2016), CockroachDB's consistency documentation said:

> CockroachDB replicates your data multiple times and guarantees consistency between replicas using the Raft consensus algorithm, a popular alternative to Paxos. A consensus algorithm guarantees that any majority of replicas together can always provide the most recently written data on reads.

They also emphasized the absence of stale reads in the FAQ:

> This means that clients always see a consistent view of your data (i.e., no stale reads).

… and on the home page:

> Consistent replication via majority consensus between replicas, with no possibility of reading stale data.

This sounds like it might be strict serializability: all transactions appear to occur atomically at some point between their invocation and completion. However, this is not quite the whole story. Cockroach Labs' blog post Living Without Atomic Clocks tells us that CockroachDB does *not* provide linearizability over the entire database.

[1] In addition to serializability, CockroachDB also provides linearizable transactions in limited cases.

> While Spanner provides linearizability, CockroachDB's external consistency guarantee is by default only serializability, though with some features that can help bridge the gap in practice.

The Raft consensus algorithm ensures that all operations on the system are globally linearizable. However, CockroachDB does not use a single Raft cluster: it runs *many* clusters, each storing a different part of the keyspace. In general, we cannot expect that a transaction which involves operations on multiple Raft clusters will be linearizable; some higher-level commit protocol is required. Moreover, CockroachDB doesn't thread reads through the Raft state machine: it bypasses Raft and reads the state of a leader with a time-based lease on that key range. So how can it provide serializability? And why are stale reads prohibited?

The answer is that on *top* of the Raft layer for discrete key-value storage, CockroachDB implements a transaction protocol which allows for arbitrary, serializable multi-key updates. Consistent snapshots across multiple keys are derived based on *timestamps*, which are derived from hybrid logical clocks: semi-synchronized wall clocks with the assistance of causality tracking. Stale reads are mostly[2] prevented by the fact that transactions which touch the same keys will touch the same *nodes*, and therefore reads on specific keys must obtain higher timestamps than completed prior writes to those keys.

Of course, the safety of this system depends on the correctness of the cluster's local clocks. Should the clock offset between two nodes grow too large, transactions will no longer be consistent and all bets are off.

Therefore: so long as clocks *are* well-synchronized, CockroachDB offers serializable isolation in all cases, and also happens to provide linearizability on individual keys: you can read the latest successfully written value for any single key. However, as we will see in section 2.5, the database as a *whole* is not linearizable: transactions across multiple keys may not observe the latest values. Since CockroachDB can exhibit anomalies in which transactions on disjoint keys are observed contrary to real-time order, it does not provide *strict serializability*—what Spanner terms *external consistency*. Reads *can* in fact be stale, in limited cases.

To see exactly how these properties play out, we'll explore CockroachDB's behavior in several consistency tests.

[2]They mostly commit on time. Mostly.

## 2 Tests

Cockroach Labs designed and evaluated several tests using the Jepsen framework, and found two issues on their own: first, that their clock-offset detection algorithm was insufficiently aggressive, and second, that SQL timestamps should be derived from the underlying KV layer timestamps. To build confidence in their work, Cockroach Labs asked me to review and extend their test suite.

Cockroach Labs had already written an impressive test suite, including a family of composable failure modes and four types of tests. We identified bugs, improved reliability and the resolving power of many tests, added more precise control over clock offset, and added three new tests (*sequential*, *G2*, and *comments*).

The *register* test is a single-key linearizable register—which formed the basis for the etcd and consul tests, among others. *Sequential* looks for violations of sequential consistency across multiple keys, where transaction order is inconsistent with client order. *Bank*, adapted from the Galera snapshot isolation test, verifies that CockroachDB conserves the total sum of values in a table, while transferring units between various rows. *G2* checks for a type of phantom anomaly prevented by serializability: anti-dependency cycles involving predicate reads. All these tests pass, so long as clock offset is appropriately bounded.

Three tests reveal anomalies: *comments* checks for a specific type of strict serializability violation, where transactions on disjoint records are visible out of order. This behavior is by design. The *set* test implements a simple unordered set: we insert many rows into a table and perform a final read of all rows to verify their presence. Finally, *monotonic* verifies that CockroachDB's internal transaction timestamps are consistent with logical transaction order. These tests uncovered two new bugs—double-applied transactions, and a serializability violation.

We'll talk about each of these tests in turn. Because CockroachDB assumes semi-synchronous clocks, unless otherwise noted, we only introduce clock offsets smaller than the default threshold of 250 milliseconds. When clock offset exceeds these bounds, as measured by an internal clock synchronization estimator, all bets are off: some nodes will shut themselves down, but not before allowing various transactional anomalies.

### 2.1 Register

Individual keys within CockroachDB are stored by a single Raft cluster: updates to them are linearizable because they go through Raft consensus. Read safety is enforced by leader leases and hybrid logical clocks. We'll verify that these two systems provide linearizability by performing random writes, reads, and compare-and-sets on single keys, then checking that operations on each independent key form a linearizable history.

Under various combinations of node failure, partitions, and clock offsets, individual keys in CockroachDB appear linearizable. Linearizability violations such as stale reads can occur when the clock offset exceeds CockroachDB's threshold.

### 2.2 Bank

The bank test was originally designed to verify snapshot isolation in Galera Cluster. It simulates a set of bank accounts, one per row, and transfers money between them at random, ensuring that no account goes negative. Under snapshot isolation, one can prove that transfers must serialize, and the sum of all accounts is conserved. Meanwhile, read transactions select the current balance of all accounts. Snapshot isolation ensures those reads see a consistent snapshot, which implies the sum of accounts in any read is constant as well.

In MariaDB with Galera and Percona XtraDB Cluster, this test revealed that neither system offered snapshot isolation. Under CockroachDB, however, bank tests (both within a single table and between multiple tables) passed consistently.

### 2.3 Sequential

CockroachDB does not offer strict serializability. However, as a consequence of its implementation of hybrid logical clocks, all transactions *on a particular node* should observe a strong real-time order. So long as CockroachDB clients are sticky (e.g. bound to the same server), we expect those clients should observe sequential consistency as well: the effective order of transactions should be consistent with the order on every client.

To verify this, we have a single client perform a sequence of independent transactions, inserting $k_1, k_2, \ldots, k_n$ into different tables. Concurrently, a different client attempts to read each of $k_n, \ldots, k_2, k_1$ in turn. Because all inserts occur from the same process, they must also be visible to any single process in that order. This implies that once a process observes $k_n$, any subsequent read must see $k_{n-1}$, and by induction, all smaller keys.

Like G2 and the bank tests, this test does not verify consistency *in general*. However, for this particular class of transactions, CockroachDB appears to provide sequential consistency—so long as clients are sticky and clock offset remains below the critical threshold.

### 2.4 G2

We can also test for the presence of anti-dependency cycles in pairs of transactions, which should be prevented under serializability. These cycles, termed "G2", are one of the anomalies described by Atul Adya in his 1999 thesis on transactional consistency. It involves a cycle in the transaction dependency graph, where one transaction overwrites a value a different transaction has read. For instance:

```
T1: r(x), w(y)
T2: r(y), w(x)
```

could interleave like so:

```
T1: r(x)
T2: r(y)
T1: w(y)
T2: w(x)
T1: commit
T2: commit
```

This violates serializability because the value of a key could have changed since the transaction first read it. However, G2 doesn't just apply to individual keys—it covers *predicates* as well. For example, we can take two tables…

```
create table a (
  id    int primary key,
  key   int,
  value int);
create table b (
  id    int primary key,
  key   int,
  value int);
```

where `id` is a globally unique identifier, and `key` denotes a particular instance of a test. Our transactions select all rows for a specific key, in either table, matching some predicate:

```
select * from a where
  key = 5 and value % 3 = 0;
select * from b where
  key = 5 and value % 3 = 0;
```

If we find any rows matching these queries, we abort. If there are no matching rows, we insert (in one transaction, to a, in the other, to b), a row which would fall under that predicate:

```
insert into a values (123, 5, 30);
```

In a serializable history, these transactions must appear to execute sequentially, which implies that one sees the other's insert. Therefore, at most one of these transactions may succeed. Indeed, this seems to hold: we have never observed a case in which both of these transactions committed successfully. However, a closely related test, `monotonic`, *does* reveal a serializability violation—we'll talk about that shortly.

## 2.5 Comments

The sequential test demonstrates that a series of inserts by a single process will be observed (by any particular single process) in order. However, CockroachDB does not provide strict serializability—linearizability over the entire keyspace. This means that transactions over different keys may not be observed in their real-time order.

For example, imagine an application which has a sequential stream of comments. Users make comments by inserting new rows into a table. Because each request is load-balanced to a different server, two transactions from the same user may execute on different CockroachDB nodes. Now imagine that a user makes a comment $C_1$ in transaction $T_1$. $T_1$ completes successfully. The user then realizes they made a mistake, and posts a correction comment $C_2$, in transaction $T_2$. Meanwhile, someone attempts to read all comments in a third transaction $T_3$, concurrent with both $T_1$ and $T_2$.

Surprisingly, serializability places no constraints on *when* an insert-only transaction is visible. It is, in fact, legal to reorder both inserts to t = ∞, in which case no read will ever see them. As a performance optimization, a serializable database may opt to throw away those insert-only transactions immediately. This is probably not what we want, but it is legal!

However, as we saw in the register tests, CockroachDB appears to provide linearizability on single keys, which implies a stronger constraint: $C_1$ must become visible to readers sometime between $T_1$'s invocation and completion. Similarly, $C_2$ must be visible by the time $T_2$ returns to the client. Since $T_1$ and $T_2$ do not overlap, one might assume that $C_2$ must never appear without $C_1$. This would be true in a strict serializable system. However, these constraints *only apply to single-key transactions*. Transactions across multiple keys can exhibit unintuitive behavior. For instance, they could interleave like so:

```
T3: r(C1)       (not found)
T1: w(C1)
T1: commit
T2: w(C2)
T2: commit
T3: r(C2)       (found)
T3: commit
```

This is legal because $T_1$ and $T_2$ are causally unrelated and can be evaluated in any order while preserving serializability. A user could observe the followup comment $C_2$, but not the original comment $C_1$. Indeed, we find exactly this behavior in experimental runs:

```
24  :invoke :read   nil
23  :invoke :write  425
23  :ok     :write  425
21  :invoke :write  430
21  :ok     :write  430
24  :ok     :read   #{2 10 15 20 34 35 38 42 43 47 51 53 59 61 71 72 82 88 89
                      113 119 123 129 132 145 146 163 167 176 206 216 224 230
                      238 243 244 255 260 292 294 299 312 316 324 325 327 330
                      350 356 359 360 361 363 366 367 371 376 403 410 419 422
                      430}
```

Process 24 invokes a read. Process 23 invokes and completes a write of 425, inserting a fresh row. Process 21 then invokes and completes a write of 430. Process 24 then completes its read, observing 430 but *not* 425.

```
{:valid? false,
 :errors
 [{:type            :ok,
   :f               :read,
   :process         24,
   :time            11917285915,
   :missing         #{425},
   :expected-count  62}
  {:type            :ok,
   :f               :read,
   :process         22,
   :time            11920656771,
   :missing         #{425},
   :expected-count  62}]},
```

Serializability requires there exists a total order for all transactions, and it's easy to construct one here: $T_2$, $T_3$, $T_1$. However, this order is not consistent with the real-time bounds implied by linearizability: $T_1$ takes effect *after* $T_2$ despite $T_1$ completing before $T_2$ even begins. This system may still be linearizable, but only when discussing single keys. Our intuition breaks down for multi-key transactions—even read-only ones. In these circumstances, we may *fail* to observe the most recent transactions.

Cockroach Labs calls this anomaly a "causal reverse"—we're not sure if there's a formal name for it. Observing inserts out of temporal (even causal!) order is a subtle example of the difference between serializability and strict serializability: CockroachDB chooses the former for performance reasons, although an experimental option can, in theory, recover strict serializable behavior. Users could also obtain causal consistency (up to linearizability) by threading a causality token through any transactions that should be strictly ordered: Living Without Atomic Clocks describes how this feature would work in CockroachDB, but it doesn't exist yet.

This anomaly occurs even when clock offsets remain within bounds.

## 2.6 Monotonic test

The *monotonic* tests are unique to CockroachDB, and verify that CockroachDB's transaction timestamps are consistent with logical transaction order. In a transaction, we find the maximum value in a table, select the transaction timestamp, and then insert a row with a `value` one greater than the maximum, along with the current timestamp, the node, process, and table numbers. When sorted by timestamp, we expect that the values inserted should monotonically increase, so long as transaction timestamps are consistent with the database's effective serialization order.

CockroachDB's design allocates nearby rows in the same table to the same Raft cluster, which linearizes operations on those rows. As tables grow larger, CockroachDB scales by sharding tables into disjoint *ranges*, each backed by an independent Raft cluster. Because the the datasets we work with in Jepsen are small, and we still want to verify the inter-range transaction path, we need a way to force transactions to cross range boundaries. This is why CockroachDB has multi-table variants of the bank and monotonic tests: tables usually occupy distinct ranges, so accessing multiple tables helps ensure we exercise CockroachDB's two-phase commit transaction mechanism.

For the monotonic tests, Cockroach Labs designed *monotonic-multitable*: a variant which splits its operations across multiple tables. This test also verifies its order in a different way from the single-table test: instead of deriving each inserted `value` from the current values in the table, it uses a client-local, monotonic counter (shared between all clients) to generate sequential values, and wraps every operation against the database in a lock. Because inserts occur in strict sequential order, we can verify whether the database's timestamps are externally consistent.

There is, however, a problem with this approach: it relies on the integrity of the lock which prevents two processes from inserting concurrently. When a client's request times out, that process releases its lock—if we didn't have a timeout, the test could block forever as soon as a failure occurred. Timeouts, however, do not guarantee that the client's operation didn't take place. It's possible for a client to insert `value = 1`, time out, insert `value = 2`, complete that insert successfully, *then* have the insert for `1` take place: creating the non-monotonic conditions which lead to a false positive.

Accordingly, we merged the two monotonic tests. To exercise the cross-range pathway, *monotonic* now reads the maximum value across several tables, and inserts `value + 1` to one of those tables at random. To avoid the mutex issue, the test chooses values based on the current table state, not a local counter. We also fixed a number of bugs in the code that evaluates histories; it would fail to recognize non-monotonic histories, and didn't report failures correctly.

We found something new. Something… exciting.

```
(({:val     237,
   :sts     1472846180755263741000000006N,
   :node    4,
   :process 29,
   :tb      1}
  {:val     236,
   :sts     1472846180755263741000000006N,
   :node    1,
   :process 26,
   :tb      1})
 ({:val     1602,
   :sts     1472846340461494553000000009N,
   :node    3,
   :process 28,
   :tb      0}
  {:val     1602,
   :sts     1472846340461494553000000009N,
   :node    2,
   :process 27,
   :tb      1}))
```

Not only do we observe non-monotonicity (decreasing values with increasing timestamps), but there's also a serializability violation! Because each transaction inserts a value strictly larger than any existing value in the table, we should *never* see a duplicate value. However, at the end of this test, we find *two* copies of 1602: both with the same timestamp `:sts`.

To prevent transactions from overwriting data that has been read by some other transaction, CockroachDB stores every read in a structure called the timestamp cache. If a transaction $T_1$ attempts to write a key, and the cache determines that another transaction $T_2$ has read the value *at a higher timestamp*, it might impact $T_2$'s correctness if $T_1$ were allowed to change that key. To avoid breaking $T_2$, $T_1$ aborts. The timestamp cache also ensures that a transaction can update keys it read previously, so long as it still owns the cache entry.

However, there's a bug: when two transactions $T_1$ and $T_2$ have the same timestamp, and access the same key, they are considered equivalent: the transaction ID is not used to discriminate between timestamp cache entries. $T_2$ is allowed to assume $T_1$'s cache entry.

Imagine $T_1$ and $T_2$ intend to scan two tables `A` and `B`, find the largest value `v`, and insert `v + 1` into one of those tables, selected at random. Both have the same transaction timestamp 10. $T_1$ scans `A` and creates a timestamp cache entry $[T_1, A, 10]$. Concurrently, $T_2$ scans `B` and creates $[T_2, B, 10]$. $T_1$ proceeds to `B`, and because the timestamp cache only checks for equivalence on the basis of key range and timestamp, replaces $T_2$'s entry with $[T_1, B, 10]$. $T_2$ does the same on A: $[T_2, A, 10]$. No writes have transpired, so both transactions see identical maximum values `v = 50`, and prepare to insert `v = 51`. $T_1$ inserts into `B`, and the timestamp cache entry for `B` $[T_1, B, 10]$ matches, so it's allowed to write. $T_2$ can do the same on `A`. We wind up with two copies of `v = 51`: a serializability violation.

This anomaly could occur whenever timestamps collide—for instance, due to clock offset, including those well below CockroachDB's threshold. To fix this issue, newer versions of CockroachDB now clear the transaction ID for overlapping timestamp cache entries with identical timestamps. This allows two read-only transactions to proceed with the same timestamp, but if either performs a write, it will be forced to retry.

## 2.7 Sets

The set test inserts a sequence of unique integers into a table, then performs a final read of all inserted values. Normally, Jepsen set tests verify only that successfully inserted elements have not been lost, and that no unexpected values were present, but Cockroach Labs extended the checker to look for duplicated values as well—a multiset test.

This test passed consistently, until a refactor changed its behavior by chance. Originally, every test's database operations went through the same path, including timeouts, error handling, and a `BEGIN ... COMMIT` transaction wrapper. The set test's insertions used this transaction pathway. When we refactored the connection error-handling code to allow for fine-grained control over transaction retries, we replaced the set test's transaction with a plain insert statement, since it has no need for a multi-statement transaction wrapper.

After a few hours of testing, the test emitted this failing case:

```
{:duplicates      [788 785 792 794 793],
 :valid?          false,
 :revived         "#{}",
 :lost            "#{}",
 :recovered
 "#{333 337..485 772..774 781..1047 1051..1052 1055..1056 1058..1061 1063
    1065..1066}",
```

```
  :ok
  "#{0..332 334..336 486..771 775..780 1048..1050 1053..1054 1057 1062 1064}",
  :recovered-frac   431/1067,
  :unexpected-frac  0,
  :revived-frac     1,
  :unexpected       "#{}",
  :lost-frac        0,
  :ok-frac          636/1067},
```

In this test, we found five duplicate values: 788, 785, 792, 794, and 793. These documents were inserted just as a network partition was beginning:

```
78         :invoke :add  785
167        :invoke :add  786
:nemesis   :info ["majring" :start] ...
159        :invoke :add  787
153        :invoke :add  788
151        :invoke :add  789
157        :invoke :add  790
179        :invoke :add  791
175        :invoke :add  792
170        :invoke :add  793
150        :invoke :add  794
```

And all timed out:

```
178 :info   :add  785 :timeout
167 :info   :add  786 :timeout
159 :info   :add  787 :timeout
153 :info   :add  788 :timeout
151 :info   :add  789 :timeout
186 :invoke :add  809
157 :info   :add  790 :timeout
179 :info   :add  791 :timeout
191 :invoke :add  810
175 :info   :add  792 :timeout
170 :info   :add  793 :timeout
150 :info   :add  794 :timeout
```

However, in the final read, we found *two* rows for each of these five values. They appear once in insertion order, and *again*, interleaved at the end of the table.

```
477 :ok :read [0 1 2 3 4 5 ... 770 771 772
  773 774 775 776 777 778 779 780 785 788
  792 793 794 ... 1050 1052 1053 1054 1057
  1059 1061 1062 1064 1065 800 805 797 789
  801 796 1051 1056 1060 788 785 795 806
  799 798 793 792 794 787 791 807 782 802
  804 783 1066 1063 1058 1055 803 790 781
  786 784]
```

Could the client have retried these insertions, leading to duplicates? Yes, but only if CockroachDB indicated the transaction had aborted due to conflict and could be safely retried. The history tells us that this is not a client retry at play—the client timed out and gave up inserting these values *well* before values like 1060 were even attempted. Something more subtle is at play here.

In this test, statements like `INSERT INTO sets VALUES (788);` execute with an implicit autocommit, which means the database is free to make some optimizations. In particular, because the DB knows the full scope of the transaction in advance, and it inserts only a single key, no two-phase commit is required. CockroachDB executes the insert in a single phase.

Now things get interesting. The write arrives on some node, and is committed locally. However, due to a network failure, the response is lost, and the CockroachDB node coordinating the transaction doesn't know what happened. When the network heals, the RPC system on the coordinator retries the request, but the retry fails with `WriteTooOldError`: time has advanced since the original write. The RPC layer hands that error to the coordinator's transaction state machine. Since a `WriteTooOldError` indicates that the coordinator (or the client) should retry the transaction, the coordinator picks a new transaction timestamp and retries the insert. It gets a new row ID, is successfully inserted, and now that node has two copies of that value.

This is a common problem in distributed systems: in a normal function call, we assume that an operation either succeeds (and returns) or fails (returning an error, throwing an exception, etc). Across a network, however, there is a third possibility: an indeterminate result. Known failures can simply be retried, but indeterminate results require careful handling to avoid losing or duplicating operations.

CockroachDB now has an `AmbiguousResultError` return type, which indicates that operations cannot be transparently retried.

## 3 Discussion

CockroachDB is aiming for a tough target: a scale-out, transactional, serializable SQL database. To achieve that goal, they've combined Raft consensus for small ranges of keys, timeout-based leases for leader read safety, and bounded-error wall clocks, somewhat like Spanner.

Like Spanner, Cockroach's correctness depends on the strength of its clocks. If any node drifts beyond the clock-offset threshold (by default, 250 ms), all guarantees are basically out the window.[3] Unlike Spanner, CockroachDB users are likely deploying on commodity hardware or the cloud, without GPS and atomic clocks for reference. Their clocks may drift due to VM, IO, or GC pauses, NTP misconfiguration or faults, network congestion, and so on, especially in certain cloud environments.

To mitigate this risk, users should set the CockroachDB clock offset threshold higher than the expected error in their clocks. Higher thresholds increase the maximum allowable latency for read operations which conflict with a write, but do not impose a hard latency floor: unconflicted transactions can proceed faster. When a node exceeds the clock offset threshold, it will automatically shut down to prevent anomalies. However, this mechanism can never be perfect: there will always be a few-second window during which transactional anomalies can occur. This is not necessarily the end of the world: many systems can tolerate occasional invariant violations.

Unlike Spanner, CockroachDB does not provide external consistency (e.g. full-database linearizability, or strict serializability). Transactions on single keys do appear linearizable but the database as a whole does not. This choice is partly a performance optimization: strict serializability requires that CockroachDB block for the full clock offset limit in more circumstances. Since CockroachDB's clocks are generally an order of magnitude less precise than Spanner's, this behavior would impose unacceptable costs for most environments.

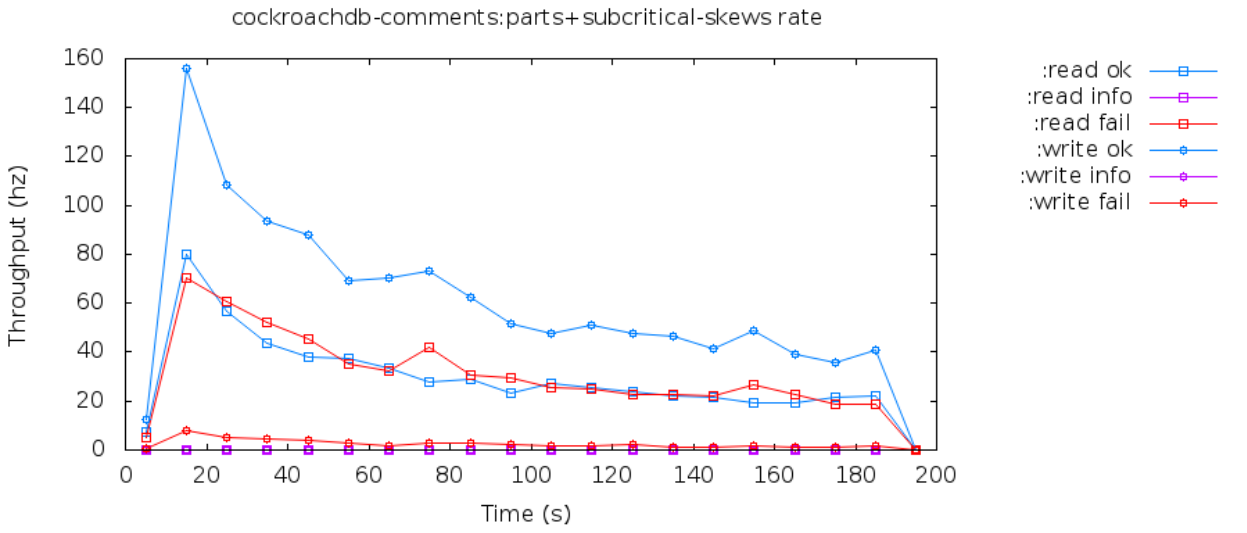
CockroachDB's test suite and pre-existing Jepsen tests caught most of the low-hanging fruit, but we were able to uncover two serializability violations over the past few months. In one case, timestamp collisions exposed an edge case in the data structure used to detect transaction conflicts, allowing two transactions to read and insert to the same range of keys. In the other, a low-level RPC retry mechanism converted an indeterminate network failure to an (apparently) definite logical failure, allowing a higher-level transaction state machine to retry the entire transaction, applying it *twice*.

Both of these faults are relatively infrequent: they required minutes to hours of testing to reproduce at moderate (~20 txns/sec) throughput. Jepsen's resolving power in these tests is limited by two factors.

First, Jepsen's linearizability checker, Knossos, is not fast enough to reliably verify long histories, or histories over many keys—and verifying serializability, in general, is an even harder problem. This means we must design a family of dedicated, special-case tests for various anomalies, and each is sensitive to transaction scope, retry strategy, timing, and key distribution.

Second, CockroachDB is still in beta, and as of the time of testing (Fall 2016) performed relatively slowly on Jepsen's workloads. For instance, on a cluster of five m3.large nodes, an even mixture of processes performing single-row inserts and selects over a few hundred rows pushed ~40 inserts and ~20 reads per second (steady-state). These figures were typical across most tests. High latency windows and low throughput gives Jepsen fewer chances to see consistency violations. As database performance improves, so will test specificity and reproducibility.

[3] Game over man, game over!




In the latest betas, CockroachDB now passes all the Jepsen tests we've discussed above: sets, monotonic, g2, bank, register, and so on—under node crashes, node pauses, partitions, and clock offset up to 250 milliseconds, as well as random interleavings of those failure modes. The one exception is the comments test, which verifies a property (strict serializability) which CockroachDB is not intended to provide. Cockroach Labs has merged many of these test cases back into their internal test suite, and has also integrated Jepsen into their nightly tests to validate their work.

In recent months, Cockroach Labs has put significant investment into both correctness and performance. There's still a lot of work to do before CockroachDB is suitable for general release, but putting time into safety early in the development process has paid off: CockroachDB uses established algorithms and storage systems, has an extensive internal test suite, and has performed their own Jepsen tests in addition to the present work. CockroachDB's reliance on semi-synchronized clocks must be considered by operators, but the fact that nodes shut down due to high clock offset means that operators will have a good idea whether constraints are being violated–and the window for invariant violation is limited by that shutdown process as well. I look forward to seeing Cockroach Labs make their first general release.

*This research was funded by Cockroach Labs, and conducted in accordance with the Jepsen ethics policy. I am indebted to their entire team for their help understanding CockroachDB's semantics, internals, and existing tests. I would especially like to thank Raphael 'kena' Poss, Arjun Narayan, Tobias Schottdorf, Andrei Matei, Matt Tracy, Peter Mattis, Ben Darnell, and Irfan Sharif for their time, expertise, and good cheer.*