JEPSEN

YugaByte DB 1.1.9

Kyle Kingsbury

2019-03-26

YugaByte DB is a distributed, multi-model transactional database based on hybrid logical clocks. We found three safety issues in YugaByte DB: two read skew bugs allowing logical data corruption, in healthy clusters and those with clock skew, respectively; and the occasional loss of small numbers of inserted records during network partitions. These problems were fixed in version 1.2.0, and YugaByte DB now passes tests for snapshot isolation, linearizable counters, sets, registers, and systems of registers, as long as clocks are well-synchronized. Our work also uncovered performance and availability issues, including a leak allowing nodes to rapidly consume all available memory, and a race condition in leader election which could take down the entire cluster indefinitely. As of 1.2.0, YugaByte has fixed all but one, minor issue with new client availability during failure. Users should be aware that, by design, YugaByte may exhibit isolation anomalies, such as stale reads, when node clocks misbehave. We suspect other anomalies may occur, but have not yet experimentally confirmed them. YugaByte has written a companion piece to this report. This work was funded by YugaByte, and conducted in accordance with the Jepsen ethics policy.

1 Errata

2019-04-10: The long fork test used in this analysis contained a bug which caused it to (in many cases) fail to identify long fork anomalies. We have re-checked YugaByte DB 1.1.15.0-b16 with a corrected checker, and its long fork tests still pass.

2 Background

YugaByte DB is an open-source, multi-model, distributed database. It includes a sharded, transactional document store wrapped by multiple interfaces supporting YCQL (a query language derived from Cassandra's CQL), and SQL (presently in beta) APIs. Intended for high-performance systems of record, YugaByte DB is specifically designed for replication across datacenters worldwide.

There are two classes of nodes in YugaByte DB: *masters*,[1] and *tablet servers*. A small number of masters control overall cluster topology, shard assignment, system metadata, and so on. Tablet servers store the actual data records, which are grouped into shards, which YugaByte DB calls *tablets*. Both masters and tablet servers use the Raft consensus algorithm to replicate their state: masters form a single Raft group, and each shard is backed by its own Raft group running on a subset of tablet servers.

In geographically replicated deployments, nodes in each Raft group are spread across datacenters to provide redundancy, as well as fast local reads where linearizability is not required.

2.1 Clocks

Within a shard, Raft ensures linearizability for all operations which go through Raft's log. However, for performance reasons, YugaByte DB does not use Raft's consensus for reads. Instead, it cheats: reads return the local state from any Raft leader immediately, using leader leases[2] to ensure safety. Using `CLOCK_MONOTONIC` for leases (instead of

[1]Where databases use "master" and "slave" terminology, Jepsen typically refers to those roles as "primary" and "secondary". In this case, "primary" and "secondary" more closely map to Raft leaders and followers which are another, orthogonal aspect of YugaByte DB's architecture. We've opted to use "master" here to avoid further confusion.

[2]Diego Ongaro's 2014 dissertation on Raft describes a similar, lease-based safety mechanism for providing linearizable reads by using heartbeat messages to extend leader leases, assuming bounded clock drift across servers.

CLOCK_REALTIME) insulates YugaByte DB from some classes of clock error, such as leap seconds.

Between shards, YugaByte DB uses a complex scheme involving Hybrid Logical Clocks (HLCs): wall clocks which are monotonically propagated with messages between nodes. YugaByte couples those clocks to the Raft log, writing HLC timestamps to log entries, and using those timestamps to advance the HLC on new leaders. This technique eliminates several places where poorly synchronized clocks could allow consistency violations.

YugaByte DB's reliance on clocks for safety is somewhat surprising, as YugaByte's Evaluation Guide says clock skew is "bound to happen", and YugaByte's blog characterizes NTP as unreliable:

> … there's still no guarantee that all nodes will see the exact same time since internet network latency is unpredictable. In reality, nodes exhibit clock skew/drift even with NTP turned on.

So, what exactly does YugaByte DB ask of its clocks? YugaByte's transaction documentation says there are only two requirements for lease safety: bounded monotonic drift rate between servers, and clocks that do not freeze. YugaByte's documentation did not specify how high clock drift is allowed to be, or how long a pause was considered a freeze.

For multi-shard transactions, a November 2018 slide deck explains that multi-row reads of frequently updated records rely on bounded clock skew, and the YugaByte DB deploy checklist confirms that nodes should run NTP. YugaByte's documentation does not provide required bounds on clock skew or drift, but a previously undocumented command line flag (--max_clock_skew_usec) has a default of 50 milliseconds.

2.2 Cross-Shard Transactions

YugaByte DB provides cross-shard transactions[3] with a homegrown commit protocol based on two-phase commit. An in-memory *transaction coordinator* creates a durable *transaction status record* in a Raft group, which allows the transaction to be resolved should the coordinator fail. The coordinator then writes *provisional records* to each shard, which are replicated in those shards' respective Raft groups. When all shards have committed their provisional records, the coordinator picks a commit timestamp and updates the status record with that timestamp, marking the transaction as complete. An asynchronous process goes on to promote provisional records to permanent ones, and assigns those permanent records the chosen transaction timestamp.

Reads are somewhat more tricky. In order to obtain a consistent snapshot, reads choose a recent timestamp t (derived from their hybrid logical clock), and query all shards, ignoring records with timestamps higher than t. In order to execute a query at t, that shard waits to ensure that it has a complete view of all transactions up to t. If a read encounters a provisional record for some key, the shard executing that part of the read checks that transaction's status record to determine whether the transaction has been committed, or is still pending.

Transaction coordinators are colocated with the leader of the status record's shard, to eliminate extra round trips. Cross-shard update transactions therefore require four round trips (five, including clients). In the general case, where the leaders of each Raft group might be in different datacenters, YugaByte DB requires at least three cross-datacenter round-trips per cross-shard update transaction.

In the happy case, read transactions can be significantly faster. Thanks to YugaByte DB's use of leader leases for reads, they can bypass normal Raft consensus in each shard, and return the state from a Raft leader without a round trip to its followers. In the uncontested case, reads can complete in as few as one cross-datacenter round trip. If a read encounters a provisional record, it may add a second round-trip to contact that transaction's coordinator. If a tablet server coordinating a read fails to select an appropriate timestamp, it have to restart the process (at most once for each shard), which adds additional message delays.

[3]YugaByte uses "distributed transaction" to mean a transaction involving more than one shard. Since single-shard transactions are also distributed across multiple nodes, we use "cross-shard transaction" in this report.

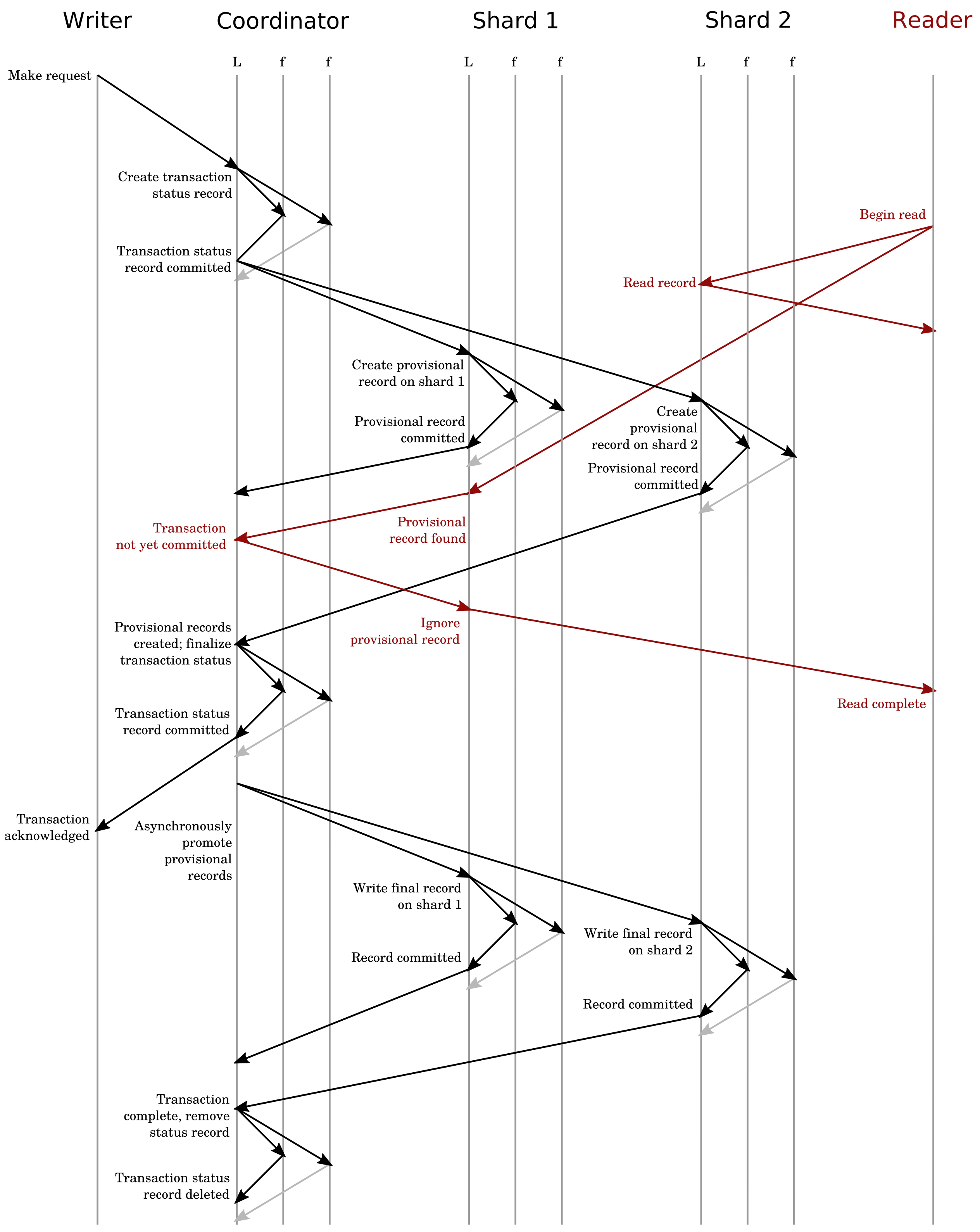

Writer
Coordinator
Shard 1
Shard 2
Reader
L
f
f
L
f
f
L
f
f
Make request
Create transaction status record
Transaction status record committed
Begin read
Read record
Create provisional record on shard 1
Provisional record committed
Create provisional record on shard 2
Provisional record committed
Transaction not yet committed
Provisional record found
Ignore provisional record
Provisional records created; finalize transaction status
Read complete
Transaction status record committed
Transaction acknowledged
Asynchronously promote provisional records
Write final record on shard 1
Record committed
Write final record on shard 2
Record committed
Transaction complete, remove status record
Transaction status record deleted

Figure 1: Request flow for an uncontested multi-shard update transaction (black), and a concurrent read transaction (red). Coordinator, Shard 1, and Shard 2 are independent Raft groups. Within a Raft group, L and f denote Raft leaders and followers.

2.3 Consistency

YugaByte's home page advertises "consistency", "availability", and "performance", with "transactional NoSQL" offering a "full spectrum of ACID compliance". As a key-value database, it promises "zero data loss" using "strongly consistent replication" and "ACID transactions". YugaByte claims that in contrast to FoundationDB their product is "globally consistent across regions".

As an SQL database, YugaByte DB claims to be "fully compatible with PostgreSQL", providing "distributed ACID" multi-shard transactions—again using "strongly consistent replication" which is "Jepsen test suite verified". In fact, YugaByte DB's SQL offering is still in beta, and there is no Jepsen test suite verifying its behavior—the Jepsen tests YugaByte designed only measured the YCQL interface. As far as ACID goes, YugaByte DB supports only snapshot isolation; serializable is under development.

In this analysis, we focus on the Cassandra-inspired YCQL interface for YugaByte DB, leaving SQL for later.

What safety properties does the YCQL interface provide? The Cassandra and MongoDB comparisons claim that YugaByte DB prevents dirty reads, and offers both linearizable and "timeline-consistent (aka bounded staleness)" single-key reads. Linearizability prevents stale reads, whereas timeline consistency means that readers observe states consistent with some total order of updates. Timeline consistency does not constrain the order of reads: a single client could observe, then fail to observe, any given update—even updates that client previously made. Bounded staleness allows stale reads up to δ seconds ago, but YugaByte makes no claims about what δ is.

Indices are "strongly consistent"—YugaByte's engineers say this means linearizable. Multi-key transactions execute at snapshot isolation, and work on serializable transactions is ongoing.

An important caveat: while YugaByte DB claims to be a transactional NoSQL store, YCQL transactions are limited to specific forms. In YCQL, there is no general concept of an interactive or programmatic transaction. Multi-key transactions can only be performed in two ways:

1. A single `SELECT` query, which may cover multiple keys, and
2. A `BEGIN TRANSACTION` statement containing multiple updates.

One cannot perform two different read queries in the same transaction; nor can one read and write state in the same transaction. Select queries are limited to a single table; there are no multi-table read transactions. These constraints limit our ability to measure YugaByte DB's underlying transactional protocol. We cannot, for example, construct queries to observe anti-dependency cycles between transactions. However, there are several tests we *can* perform.

3 Test Design

YugaByte designed and ran their own Jepsen test suite prior to our collaboration. Using the YCQL interface, that test suite measured linearizable single-key and multi-key registers, inserts, counters, and a snapshot-isolation test involving a simulated set of bank accounts. It included several failure modes, including tablet server and master crashes; single-node, majority-minority, and non-transitive partitions; as well as instantaneous and stroboscopic changes to clocks, up to hundreds of seconds.

We reviewed and improved upon these tests, and added new ones. Our workloads now include linearizable sets and measurements of long fork, using both key-value operations as well as secondary indices.

We also added failure modes for process pauses, expanded the range of clock skew to hundreds of seconds, and introduced randomized scheduling, mixtures of different, overlapping failures, and periods for node recovery.

3.1 Counter

In the counter test, we create a single record with a counter field, and execute concurrent increments and reads of that counter. We look for cases where the observed value is higher than the maximum possible value, or lower than the minimum possible value, given successful and attempted increment operations.

3.2 Set

The set test inserts a sequence of unique records into a table and concurrently attempts to read all of those records back. We measure how long it takes for a record to become durably visible, or, if it is lost, how long it takes to disappear. A linearizable set should make every inserted element immediately visible. A variant of the set test reads from secondary indices to verify their consistency with the underlying table.

3.3 Long Fork

In snapshot isolated systems, reads should observe a state consistent with a total order of transactions. A *long fork* anomaly occurs when a pair of reads observes contradictory orders of events on distinct records—for instance, T_1 observing record x before record y was created, and T_2 observing y before x. In the long fork test, we insert unique rows into a table, and query small groups of those rows, looking for cases where two reads observe incompatible orders.

3.4 Bank

Another Jepsen staple is the bank test, which creates a series of records, each representing a simulated bank account, and transfers money between randomly selected pairs of accounts. Since YugaByte DB's YCQL transactional system can't express transactions which both read and write accounts, this uses an in-place update of transaction balances. We can't enforce minimum balances; we allow them to become arbitrarily negative, and do not test transactional aborts. Lacking generalized transactions, we cannot test the deletion or creation of accounts: we use a fixed pool of account records throughout.

3.5 Single-Key Linearizable

We verify the linearizability of operations on single keys by performing randomized writes, reads, and compare-and-set operations on individual records, then checking that the resulting history is linearizable using the Knossos linearizability checker.

3.6 Multi-Key Linearizable

To evaluate the correctness of multi-key transactions, we generalize the single-key linearizable test to transactions over a small set of keys. Because YugaByte DB has only limited support for transactions, we can't evaluate this behavior in general—we cannot, for instance, generate transactions which both read and write keys. We can, however, test transactions with multiple reads, and transactions with multiple writes.

4 Results

We evaluated YugaByte DB on a five-node Debian Jessie cluster, with replication factor 3. Three nodes ran masters, and all five ran tablet servers. We tested versions 1.1.9, 1.1.10, 1.1.11, 1.1.13.0-b2, and 1.1.15.0-b16. Prior to publication, YugaByte also tested 1.2.0.0-b7. We'll start by discussing performance issues, including reduced availability and resource consumption, as well as a race condition in table creation. Then we'll present more severe bugs, including cluster-wide stalls and memory leaks. Finally, we'll cover safety violations: read skew including logical state corruption, lost inserts, and behavior under clock skew.

4.1 YCQL Requests Never Time Out

In our tests, we used YugaByte's fork of the Cassandra Java client, which uses a 12-second timeout by default. That timeout was chosen by the Cassandra developers to be slightly longer than Cassandra's default server timeout of 10 seconds. In most Jepsen tests of leader-based replication systems, we observe a burst of timeouts when a network partition or other fault occurs, as nodes partitioned from the leader wait for responses to their queries which will never arrive. After a few seconds, those isolated nodes typically declare themselves unavailable for consensus operations, and requests which would have required a leader will fail immediately, rather than timing out.

YugaByte DB 1.1.10 behaved differently: timeouts persisted through the entire duration of a network partition, even if the partition lasted for 500+ seconds.

This could pose performance risks to user-facing systems: a network partition could cause clients to get backed up, waiting for requests that won't complete until the partition is resolved. That could lead to elevated latencies or queue overflows in upstream systems, and prevent clients from moving on to other requests which *could* succeed.

This occurred because the RPC mechanism in YugaByte DB's YCQL interface internally retried queries forever, rather than giving up and returning an error to the client. YugaByte has fixed this issue in version 1.1.13; the new default server timeouts are 60 seconds.

4.2 Repeated Log Messages

During network partitions, one or more tablet servers could find themselves unable to reach the current leader of the master cluster. When this occurred, those nodes could log upwards of 250,000 error messages per minute, all identical:

```
Leader Master has changed, re-trying...
```

On filesystems without compression, this could consume roughly 40 MB of disk per minute, which makes debugging more challenging, and could fill up the disk. We found this issue in 1.1.10, and YugaByte fixed it in 1.1.13.

4.3 Race Condition in Table Creation

When executing concurrent `CREATE TABLE ... IF NOT EXISTS` commands, YugaByte DB 1.1.10 could return successfully before the table was actually created; subsequent operations involving that table could throw "Table Not Found" errors. This occurred when YugaByte DB observed another client in the process of creating that same table; it would return immediately, rather than waiting for the table to be completely created. YugaByte fixed this issue in 1.1.15.

4.4 Connecting to Isolated Servers

When a Cassandra JVM client connects to any server, it checks the `system.peers` table to identify the other nodes in the cluster. This request is synchronous; if it times out or fails, it blocks the client from performing any other requests. Since YugaByte forked Cassandra's client code for use with YugaByte DB, the YCQL JVM client makes the same request.

Unlike Cassandra, YugaByte DB stores the `system.peers` table in the master Raft group. This means that a server must be able to reach a master node, which in turn must able to communicate with a majority of masters, in order to satisfy this initial request. If enough masters are down, or partitioned from one another, or partitioned from a tablet server, JVM clients will be unable to connect to that tablet server until conditions improve. This reduces YugaByte DB's availability: that tablet could be writable, or at least readable, but if it can't reach the masters, only clients with an existing connection will be able to perform work. As clients fail over from other nodes, or rotate connections for performance reasons, more and more clients can find themselves stuck.

Since clients cache the peers table locally, stale reads of the system.peers table are just fine for this use case. We recommended that YugaByte DB cache this table on every node to improve availability.

4.5 Slow Recovery from Network Partitions

After a network partition ended, YugaByte DB generally took 10–30 seconds to recover to a healthy state. However, sometimes it could take 60 seconds or more to recover: a few operations might be able to complete soon after the end of a partition, but resuming normal throughput took significantly longer.

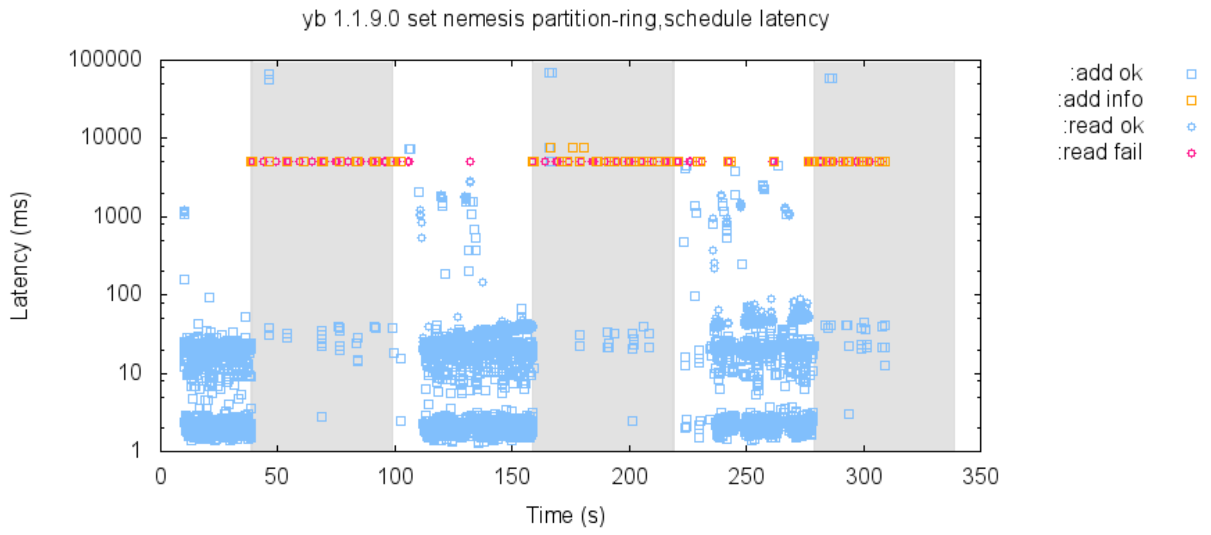


Figure 2: After network partitions (grey regions), YugaByte DB could take 30–50 seconds to recover.

For instance, in this set test, we attempt a mixture of single-row inserts and table-wide reads, while creating and healing network partitions which always leave a majority of the cluster connected. Partitions lasted for 60 seconds, followed by 60 seconds of total network connectivity, then a new partition. Recovery to a completely healthy state could take 20–60 seconds.

YugaByte has been exploring ways to speed up recovery, by making optimizations to elections and timeouts. In our testing, version 1.1.15.0-b16 typically recovers from partitions in ~25 seconds, and cases of persistent elevated latencies are significantly less frequent than in version 1.1.9.

4.6 Indefinite Master Stalls

In many of our network partition tests, YugaByte DB 1.1.9.0 through 1.1.13.0-b2 exhibited long windows of unavailability—despite total network connectivity and all nodes being online. Normal recovery times for YugaByte DB were on the order of tens of seconds, but in some cases, even 5 minutes without any faults was insufficient for recovery.

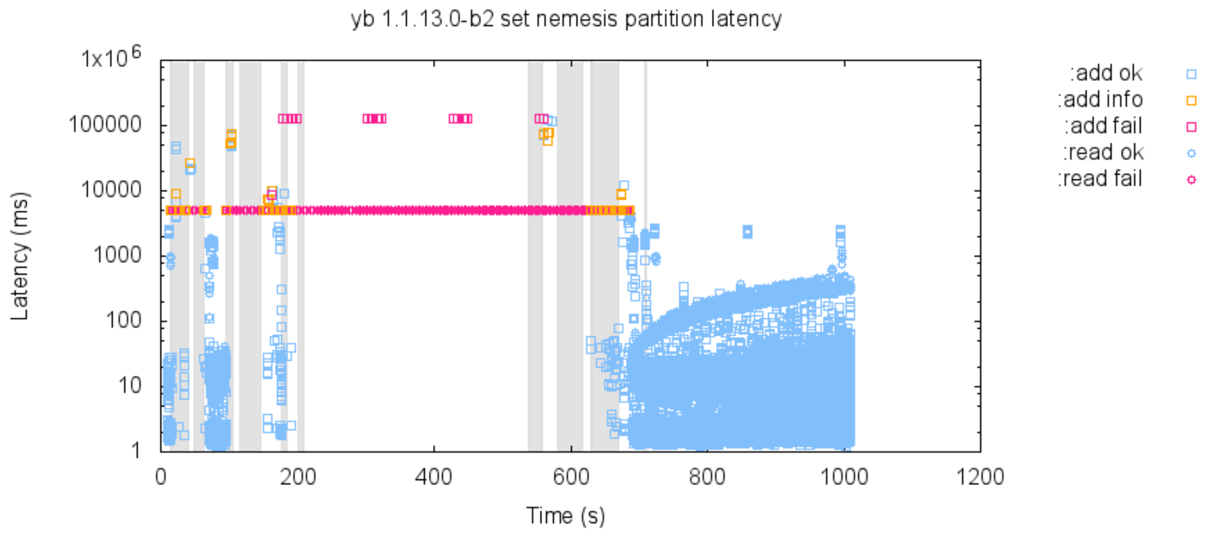


Figure 3: After an initial sequence of network partitions, the cluster is completely unavailable for 300 seconds; additional partitions resolve the outage.

Unusually, additional network partitions could cause the cluster to recover. We were perplexed by this phenomenon, until YugaByte identified that some master nodes had gotten stuck in a way which made them appear healthy to other nodes (thus precluding a leader election), but prevented them from processing any requests. In this particular test, nodes n1 and n2 both got stuck, but a third master node, n3, remained alive. Node n3 was elected after the second series of network partitions, and operations resumed.

YugaByte traced this problem to multiple race conditions in the leader election process for master nodes. For example, when becoming a leader, a master would abort and block for all previously created tasks. However, concurrently added tasks might not receive the message to abort, causing the master to block on those tasks indefinitely. This issue was fixed in version 1.1.15.

4.7 Memory Leaks

In our tests of YugaByte DB 1.1.10 through 1.1.13.0-b2, with verbose logging enabled, roughly one master per day encountered a catastrophic memory leak, allocating several gigabytes per second, until the OOM killer (or operators) intervened. In one case, the process allocated 60 GB and remained stable.

These allocations occurred despite a configured `--memory_limit_hard_bytes` of 4 GB, in scenarios with network partitions, process pauses, and even with no failures at all. We identified no obvious correlation with data volume or throughput; tests with only a handful of integer values could still result in memory leaks. Curiously, these leaks did not appear in tcmalloc's allocation statistics.

Valgrind identified `libbacktrace` as the cause of this leak, and YugaByte believes the problem stems from a thread safety bug in `libbacktrace`. As of 1.1.15.0-b16, YugaByte DB now uses Google's `glog` for stack traces instead, and we have not observed memory leaks since.

4.8 Frequent Read Skew

Our first test run with 1.1.9 revealed a significant problem: under normal operating conditions, transactional reads in YugaByte DB frequently exhibited read skew: seeing part, but not all, of other transactions' effects. In the bank test, for instance, reads of a set of accounts containing $100 total could return $102, $85, or $180. Moreover, those totals would *drift* over time, suggesting that skewed reads affected read-write transactions, even when those transactions wrote every value they read.

For example, in this test, the total balance started at $100, but quickly dropped to $95, then drifted as high as $154. A snapshot isolated system would never change the total value of accounts, and observe $100 in every read transaction.

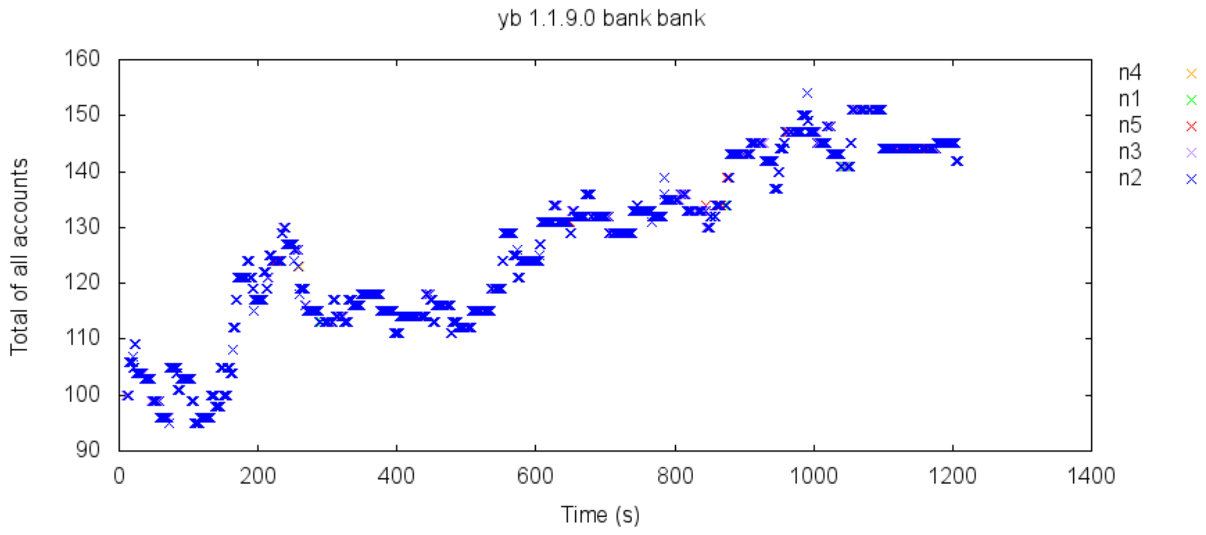


Figure 4: Plot of the total of all accounts over time. In a snapshot-isolated system, every total should be exactly 100.

As it turns out, YugaByte had independently discovered the cause of this problem: a race condition affecting concurrent multi-shard update transactions. Consider two transactions T_1 and T_2, which both interact with key k. T_1 writes a provisional record for k, which is observed by T_2. T_2 goes to checks T_1's transaction status record, but before its request arrives, T_1 commits, promotes its provisional records to normal ones, and, having completed, deletes the T_1 status record. T_2's request for that status record finds nothing, which T_2 takes to mean that T_1 *aborted*. T_2 then ignores T_1's committed write of k, but could observe other keys T_1 wrote, allowing read skew.

YugaByte included a fix for this issue in version 1.1.10; we have not observed read skew in any later test.[4]

[4]So long as clocks are well-behaved.

4.9 Rare Lost Inserts

Under rare conditions involving multiple network partitions, YugaByte DB 1.1.10 could discard acknowledged writes, even if they had been visible to other reads for tens of seconds. We observed this issue in the set test, which inserts unique numbers as distinct rows, and concurrently reads back every row, looking for numbers which were acknowledged before that read began, but were absent from the read itself.

In this particular test, element 427 was written successfully ~90 seconds into the test, right as a network partition was ending. 427 was present in every subsequent read, from 98.4 seconds to 141.1 seconds, when the cluster happened to be recovering from a network partition (which ended roughly 15 seconds prior). From 142 seconds through the end of the test, element 427 (and *only* 427) was missing from every read.

This issue was rare, and difficult to reproduce; we encountered only a handful of lost writes in tens of hours of testing. Each instance of data loss involved a partition resolving, or the cluster recovering from an earlier partition.

YugaByte DB changes the nodes participating in each Raft cluster dynamically, removing nodes which have been inaccessible for some time, adding new nodes to preserve the desired number of replicas, and rotating membership as a part of ongoing load balancing. YugaByte believes that when a network partition occurred concurrently with a membership change (which happens automatically in response to network partitions), writes could be replicated to some (but not all) nodes. During a subsequent leader election, a leader could come to power *without* certain writes. Normally, the Raft algorithm prohibits this. However, YugaByte DB extends Raft by introducing the concepts of voting and non-voting members—and an implementation bug allowed *all* Raft members, not just voting members, to be counted towards the majority acknowledgement required for commit. A fix is available in 1.1.13.

4.10 Read Skew Under Clock Skew

When we introduced POSIX clock fluctuations larger than `max_clock_skew_usec` during bank tests, they exhibited fluctuations in the total balance of all accounts, which should be constant in snapshot-isolated systems. These were not merely stale reads: clients observed a state of the system that should never have existed at any time. Moreover, this read skew appeared to persist after initial fluctuations, which suggests that transfer transactions observed an inconsistent view of the system and *propagated* that state back into the database, corrupting logical state.

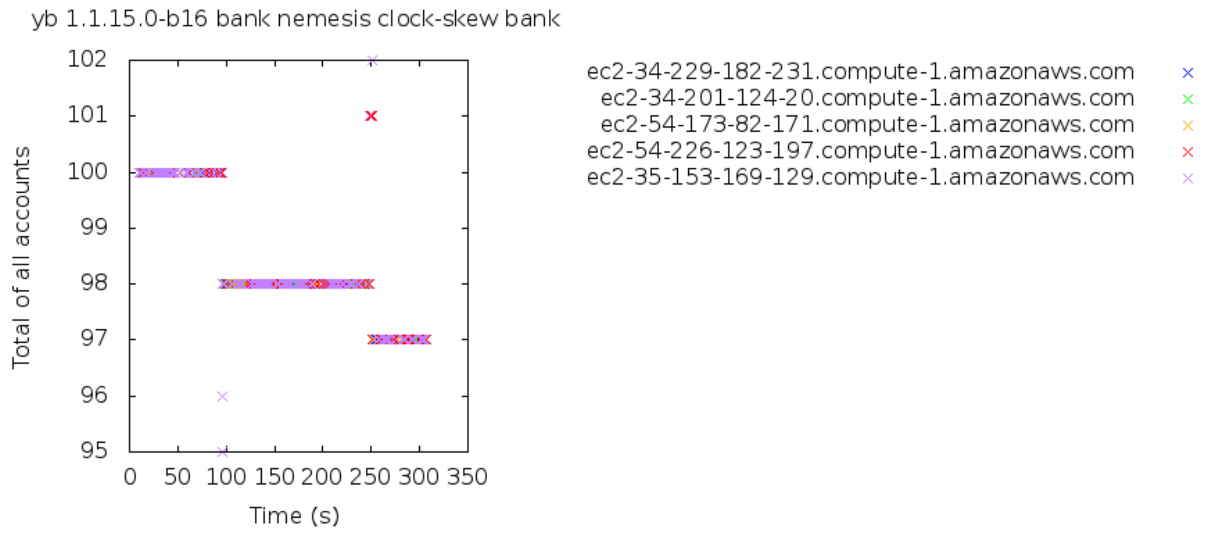


Figure 5: With clock skew, the total of all accounts can fluctuate, settling for a time on new totals.

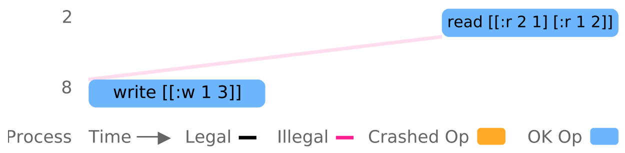
This issue turned out to be a bug in a YugaByte DB locking system. `LockBatch`, a type of lock used to ensure transactions don't execute concurrently, implemented a `MoveFrom` function which failed to transfer the lock's `status` field. If a `LockBatch` lock could not be acquired within its deadline, YugaByte DB would fail to notice that the lock hadn't actually been acquired, assume the lock was held, and proceed to execute critical sections concurrently. This issue was fixed in 1.2.0.0-b7.

4.11 Consistency Errors Under Clock Skew

In the presence of poorly synchronized clocks, YugaByte DB still exhibits some transactional anomalies. We found stale reads in both set tests and multi-key transactions.

For instance, in this set index test involving clock skew as well as tserver crashes & pauses, YugaByte DB allowed stale reads of 114 out of 20344 inserts, with the worst taking 136 milliseconds to become visible. Set tests use single-key insert transactions, but reads involve selecting multiple records across different shards, so HLC clock skew may have played a role.

We also observed nonlinearizable histories in multi-key linearizability tests. For instance, in this test, process 8 performs a write `[:w 1 3]`, setting key 1 to 3. Process 2 then performs a read `[[:r 2 1] [:r 1 2]`: it saw the value of key 2 as 1, and the value of key 1 as 2. This is impossible in a linearizable system: after the write completes, key 1 should have the value 3 until some other update occurred, but no other operations transpired during or before the read of 2. While this particular test involved multiple types of failure, we can reliably reproduce nonlinearizable histories with only clock skew, and no partitions, crashes, or pauses.



These behaviors are intrinsic to YugaByte DB's architecture, and fall outside the assumptions YugaByte DB makes about the hardware it runs on. YugaByte does not plan to change these behaviors.

We have yet to observe anomalies due to clock skew in single-key counters, or in single-key linearizable tests—perhaps because those workloads do not involve multi-shard transactions, and because our tests do not affect `CLOCK_MONOTONIC`.

№	Summary	Event Required	Fixed in
798	Concurrent create-table calls can return before table created	None	1.1.15
821	Slow/unreliable recovery from network partitions	Partition	1.1.15
822	CQL requests never time out	Partition	1.1.13
823	Repeated log messages	Partition	1.1.13
824	Rare loss of acknowledged inserts	Partition	1.1.13
862	Memory leak	None	1.1.15
864	Clients can't connect to partitioned nodes	Master unavailable	Unresolved
886	Indefinite unavailability	Master elections	1.1.15
894	Frequent read skew & data corruption	None	1.1.10
975	Read skew & data corruption	Clock skew	1.2.0

5 Discussion

We observed three safety issues in YugaByte DB. The first read skew issue is a severe violation of transactional guarantees, and occurred continuously in healthy clusters. The loss of acknowledged inserts was significantly rarer, and affected only small numbers of updates. Read skew and data corruption under clock skew was a serious problem, but required large clock offsets. These issues were resolved in 1.1.10, 1.1.13, and 1.2.0, respectively.

Version 1.2.0 also includes significant availability improvements, fixing a sporadic, fast-growing memory leak, master nodes getting stuck after elections, and improving recovery time from network partitions. The only issue which remains unaddressed is a relatively minor problem involving clients connecting to partitioned nodes.

YugaByte's engineers were quick to address safety and availability issues, and we're pleased to see the results of their hard work.

5.1 Recommendations

YugaByte DB 1.1.9's transactional isolation was badly broken: under normal conditions, healthy clusters exhibited frequent read skew, and those skewed reads could be written back to the database, corrupting logical state. YugaByte DB 1.1.10 could, under rare conditions involving multiple leader elections due to e.g. network partitions, lose a handful of committed writes. Version 1.1.15 also exhibited read skew and data corruption during clock skew. We are unaware of workarounds for these issues, and recommend that users upgrade to 1.2.0 or higher as quickly as possible. 1.2.0 also addresses serious availability issues.

In YugaByte DB, master nodes play a critical role. In some sharded systems, single-shard transactions can continue so long as a shard remains alive, but in YugaByte DB, a shard must be connected to a healthy group of masters in order for new clients to make progress. We recommend that users keep this in mind when making choices about node and network redundancy.

In the presence of clock skew on the order of hundreds of seconds, YugaByte DB can exhibit stale reads, and, we suspect, other uncharacterized anomalies. YugaByte already recommends the use of NTP, but, as YugaByte notes, NTP is not necessarily reliable. As a matter of general principle, we encourage users to monitor and alert on their clocks to help limit the impact of anomalies. Users should set `--max_clock_skew_usec` to comfortably enclose the clock skew they observe; YugaByte recommends a value twice as high as observed clock skew.

Using `--disable_clock_sync_error` may make it easier to detect and limit the impact of clock skew issues, but note that the default threshold for shutting down is a hundred seconds—three orders of magnitude larger than the safety envelope of 50 milliseconds. We also recommend using physical hardware where possible; virtual machines can introduce additional sources of clock error. In cloud environments, using the cloud provider's NTP servers may offer improved bounds on clock skew.

5.2 General Comments

Without clock skew, YugaByte DB 1.1.15 passed our tests for multi-key strict serializability, single-key linearizability, snapshot isolation (including bank and long fork), counters, and linearizable inserts + reads. However, users should keep in mind that YugaByte's transactional model is, at present, only intended to provide snapshot isolation plus single-key linearizability. The fact that our tests for strict serializability passed may be due to our inability to express mixed read-write queries in YCQL, or due to performance limitations in our test design: checking longer histories can be extremely slow.

YugaByte DB has updated their documentation and marketing material. The deployment checklist now offers a comprehensive discussion of clock skew and drift tolerance, and the relevant command line parameters. Clock skew options are now a part of the standard CLI documentation. The SQL page no longer claims that YSQL is verified by Jepsen; that claim should only have applied to the YCQL interface.

Jepsen is not a good measure of database performance; we evaluate pathological workloads with fixed concurrency, rather than realistic workloads with fixed request rates. In particular, we note that YugaByte DB is intended for deployment across multiple datacenters, but our tests used uniformly low-latency networks between all nodes, except for failure cases.

Finally, we should note that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we make extensive efforts to find problems, we cannot prove the correctness of any distributed system.

5.3 Future Work

We have not yet seen stale reads in single-key linearizable tests. We suspect this is because YugaByte DB relies on clocks for safety in two separate mechanisms:

1. Leader leases, which use `CLOCK_MONOTONIC` to allow Raft leaders to service reads immediately, rather than waiting for the standard round-trip.
2. Multi-shard transactions, which use `CLOCK_REALTIME` as an input to their hybrid logical clocks to obtain consistent snapshots across multiple shards.

Our tests only affect `CLOCK_REALTIME`, as `CLOCK_MONOTONIC` can't be changed on Linux systems. We tried using libfaketime, an `LD_PRELOAD` shim, to simulate drift in `CLOCK_MONOTONIC`, but were unable to complete this work before publication. It's likely that our tests missed issues relating to leader leases, and we think additional testing is warranted.

YugaByte DB's use of wall clocks for snapshot isolation suggests that when those clocks are poorly synchronized, we should observe some sort of transactional anomalies. We found three classes of failing tests under clock skew near the end of our testing process: read skew in bank tests, stale reads in set tests, and non-linearizable anomalies in multi-register tests. After our testing work together, but prior to the publication of this report, YugaByte traced the bank test issue to a bug in `LockBatch`. We'd like to thoroughly explore bank test behavior with that fix, and investigate multi-register failures in more detail: are anomalies now limited to stale reads, or can YugaByte DB violate snapshot isolation as well?

We would also like to explore the relationship between clock skew and `--max_clock_skew_usec`: is it a tight bound on allowable clock skew, or is there some margin of safety? How well does `--disable_clock_sync_error` work, and what kinds of anomalies can go unnoticed?

The tests we wrote for YugaByte DB are less complete than those we've performed against similar systems in other Jepsen analyses. We have no way to transactionally query multiple tables in the bank test, so we measure multiple keys in the same table. We cannot combine read and write transactions in multi-key linearizability tests, so we stick to strictly read-only or write-only queries. We cannot express queries which would let us observe predicate anti-dependency cycles. Since YugaByte DB's YCQL interface proscribes these kinds of queries, there is no safety risk if they don't work correctly. However, we cannot speak to YugaByte DB's transactional isolation mechanisms in general—we can only describe how they handle the limited queries we *can* express in YCQL.

Once YugaByte DB's SQL layer is ready, we would like to return and test more generalized transactions. Both YSQL and YCQL are built on the same underlying transactional mechanism, so the YCQL tests give us some degree of confidence in the behavior of YSQL as well. However, YSQL will let us write the transactions we need to check the transactional layer more rigorously. There's also the possibility of safety issues in YSQL itself, or in how it uses the underlying transactional layer. We look forward to exploring its behavior.

We have not explored filesystem or disk issues, both of which deserve attention. We also ran masters and tablet servers on the same nodes, which introduced a degree of coupling between their failure modes. Future work could explore partitions between two masters, between two tablet servers, or between masters and tablet servers, independently.

Finally, YugaByte DB has no formal model for the correctness of its 2PC-based transactional system, conflict detection, or coupling of read times to Raft. These systems interact in complex and subtle ways; proofs and model-checking could help us gain confidence in YugaByte DB's correctness.

Going forward, YugaByte plans to complete their ongoing work on serializable isolation and their SQL interface, allowing YugaByte DB to serve general OLTP workloads in a linearly-scalable fashion. YugaByte also plans to continue performance tuning, to improve fault recovery and availability during network partitions.

This work was funded by YugaByte, and conducted in accordance with the Jepsen ethics policy. We wish to thank the YugaByte team for their invaluable assistance—especially Timur Yusupov, Mikhail Bautin, Amitanand Aiyer, Sergei Politov, Ravi Murthy, Karthik Ranganathan, and Kannan Muthukkaruppan. Jessie Frazelle helped with building libfaketime, and Kit Patella made improvements to the core Jepsen library in support of this work. We are also grateful to Peter Bailis, Diego Ongaro, and Peter Alvaro for their comments on theoretical questions. Finally, thanks to Marc Hedlund and Kelly Sommers for their comments on early drafts.