JEPSEN

YugaByte DB 1.3.1

Kyle Kingsbury
2019-09-05

YugaByte DB is a distributed, multi-model transactional database based on hybrid logical clocks. In our previous analysis, we found three safety issues in 1.1.9's CQL interface, all of which were fixed by 1.2.0. In this analysis, we focus on YugaByte DB's upcoming support for serializable SQL transactions. We found two safety issues: improperly initialized `DEFAULT` columns, and G2-item (anti-dependency) cycles in transactions. The G2-item issue was fixed in 1.3.1.2-b1. We also found several issues with cluster setup and table initialization. YugaByte has written a companion blog post to this report. This work was funded by YugaByte, and conducted in accordance with the Jepsen ethics policy.

1 Updates

2019-09-05: YugaByte's blog post states YugaByte DB "passes Jepsen tests". We feel obligated to state that YugaByte DB's Jepsen test suite does not pass, though it may in the future. Race conditions in YugaByte DB's schema system can cause correctness errors. For example, inserting rows into a freshly-created table with `DEFAULT` values may result in the values for those columns initialized to NULL instead. We can also now confirm that this issue affects all default values, not just `DEFAULT NOW()`. It also appears that DDL race conditions might, under certain conditions, render tables completely unusable.

2 Background

YugaByte DB is an open-source, multi-model, distributed database. It wraps a sharded, transactional document store in multiple interfaces, including YCQL (a Cassandra-style query language), and YSQL, a Postgres-flavored SQL API. Intended for high-performance systems of record, YugaByte DB is designed for replication across datacenters worldwide.

Under the hood, YugaByte DB is comprised of a single Raft-replicated coordination service called *master*[1], and a sharded collection of *tablet servers*, which store data in Raft-replicated shards. A custom transaction protocol, loosely adapted from Spanner, provides multi-key transactions.

When we tested YugaByte DB version 1.1.9, the YSQL interface was still in beta, so we focused on YCQL. The YCQL interface does not support generalized transactions: transactions must either be comprised of multiple writes, or a single read query (which may return multiple rows from a single table). Queries with reads and writes, or multiple reads, are impossible to write in YCQL. This limited the scope of our testing; certain types of transactional workloads were simply not expressible.

Moreover, 1.1.9 supported only snapshot isolation, rather than serializability. This meant that transactions could encounter write skew or other consistency anomalies.

Serializable isolation was added in version 1.2.6, and in 1.3.1, YugaByte DB is approaching general availability for SQL, including support for generalized transactions. YugaByte asked Jepsen to help review their transactional support in preparation for the official release of SQL support.

2.1 Consistency

YugaByte DB maps SQL's `SERIALIZABLE` isolation level to serializability, and `REPEATABLE READ`, `READ COMMITTED`, and `READ UNCOMMITTED` to snapshot isolation.

[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.

Serializable and snapshot isolated transactions in YugaByte DB's key-value storage layer work similarly, acquiring write and (for serializability) read locks on records, and resolving write-write (for snapshot isolation) and read-write (for serializable) conflicts using randomly generated transaction priorities and hybrid logical clocks. Transactions proceed through a four-phase process: creating a transaction status record, writing provisional records for each update, marking the status record as committed, and (asynchronously) promoting provisional writes to standard records, before cleaning up the status record.

Hybrid logical clocks, used to resolve transaction conflicts, rely in part on well-synchronized wall clocks for correctness. YugaByte DB also employs leader leases based on `CLOCK_MONOTONIC_RAW` to provide linearizable key-value reads without the standard round trip between Raft leaders and followers, which means that the linearizability of reads depends on the rate at which each node's monotonic clock advances. Pathologically fast or slow clocks could result in stale reads.

3 Test Design

YugaByte independently ported the YCQL tests to YSQL, and Jepsen introduced a new, more general test of transactional isolation, designed to identify dependency cycles between transactions, up to serializability. As before, we measured YugaByte DB using several failure modes, including tablet server and master crashes; single-node, majority-minority, and non-transitive partitions; process pauses; instantaneous and stroboscopic changes to clocks, up to hundreds of seconds; and combinations of the above events.

3.1 Append Test

Jepsen is experimenting with a new type of test for transactional databases. The *append* test models the database as a collection of named lists, and performs transactions comprised of read and append operations. A read returns the value of a particular list, and an append adds a single unique element to the end of a particular list. We derive ordering dependencies between these transactions, and search for cycles in that dependency graph to identify consistency anomalies.

4 Results

We evaluated YugaByte DB on five-node Debian Stretch clusters, with replication factor 3. Three nodes ran masters, and all five ran tablet servers. We tested versions 1.3.1.0 and 1.3.1.2-b1, and deployed to both LXC and EC2 instances.

We found several issues, ranging from issues with cluster startup and table creation to resource leaks and safety violations. Each is presented below.

4.1 Create Table Constraint Violations

We encountered several problems creating tables in YugaByte DB. For example, issuing a request like

```
create table mice (
  id      int primary key,
  squeak  text
);
```

could throw errors like "duplicate key value violates unique constraint 'pg_class_oid_index'", or "duplicate key value violates unique constraint 'pg_depend_reference_index'", even when create table operations were executed by a single thread to limit concurrency.

In YugaByte DB, DDL operations like table creation and schema changes are not transactional. If a master leader election takes place during table creation, the Postgres system tables could be created, but the `create table` operation would fail, thanks to the change in leadership. When we retried table creation on the new leader, it conflicted with the originally created table, causing a duplicate key value error.

These are bugs 1962 and 1993 respectively, both of which should be fixed when DDL is properly transactional.

4.2 Create Table Type Already Exists

Similarly, creating tables using `create table ... if not exists` could, on occasion, throw errors like `ERROR: type "append3" already exists. Hint: A relation has an associated type of the same name, so you must use a name that doesn't conflict with any existing type.`

This constraint was violated because YugaByte DB allowed two requests to create the same table, each with a distinct UUID—and only detected the error once it came time to create the associated type. Why was YugaByte DB able to create two tables with the same name? Because tables are tracked in the catalog manager by their UUIDs, rather than by table names. Table names are maintained in a metadata cache, which is asynchronously updated. This design allows table creation to race, resulting in multiple tables with identical names.

4.3 And YOU Get a Table!

To work around table creation issues like these, there's a simple solution: catch the appropriate error messages, and retry. On a subsequent `create table` request, YugaByte DB's table metadata system will realize the table already exists, the `if not exists` clause will kick in, and the operation can complete as a no-op.

Typically.

On occasion, YugaByte DB would fail to notice the table had been created before, and allow every retry operation to create yet *another* table. Each table consumes some disk and memory overhead, and over a few hours the cluster could consume all available memory, before the OOM killer kicked in, and enough processes died to bring cluster operations to a halt. In one of our tests which hit this scenario, a single remaining tablet server process spun at 93-97% CPU (of 48 logical cores).

Like the `type already exists` error we discussed previously, this issue stems from the fact that table names are stored in a *cache*, rather than a linearizable data structure. Once issue 1476 is resolved, this problem should be fixed.

4.4 Crash on Table Creation

In rare cases, creating a table in YugaByte DB could cause one or more master nodes to crash, with an error message like "check failed: 'ysql_catalog_config_.get()' Must be non Null". These crashes occurred shortly after cluster creation, but since we only create tables shortly after startup, we can't say whether this issue affects long-running clusters, or is only a risk for newly created ones.

YugaByte believes this table creation issue involves a write request sent to a non-leader master node—which could happen during or after a leader election—and missing safety code to ensure that that node is actually a leader before making changes to the YSQL catalog.

4.5 Slow Recovery During Partitions

During network partitions, including those which isolate only a single node from the rest of the cluster, YugaByte DB generally recovers within ~10 seconds. However, roughly 1 in 20 partitions in our testing resulted in YugaByte DB going largely (but not entirely) unavailable for over a hundred seconds.

YugaByte is still investigating.

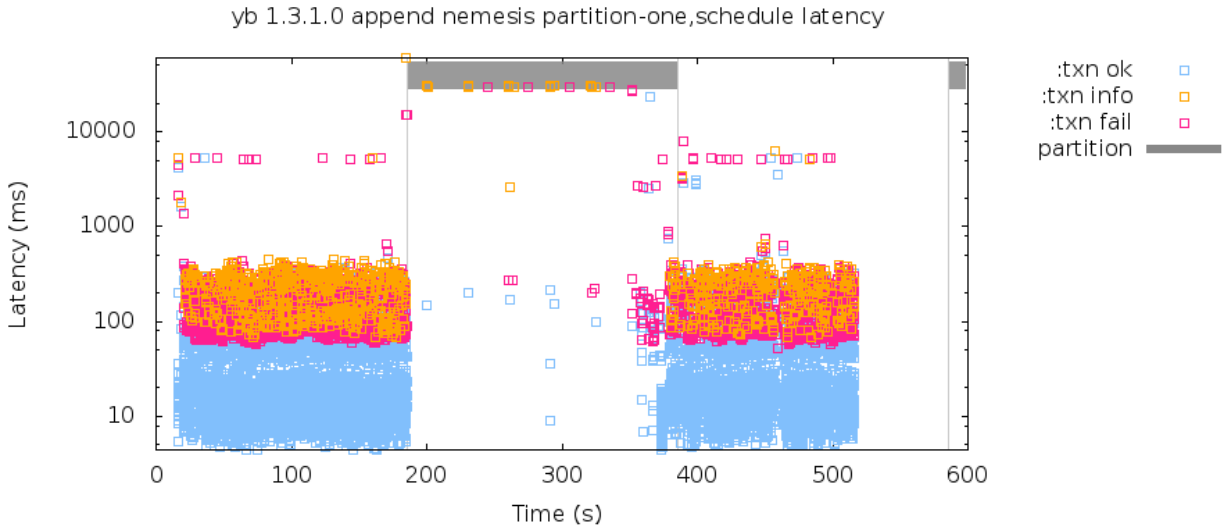

Figure 1: Request latencies over time show a cluster taking over 100 seconds to recover during a network partition.

4.6 A Plethora of Worker Processes

When we moved to longer (~1200 second) tests, a new problem arose: tests terminated by the out-of-memory killer. Nodes would leak roughly 5 MB of memory per second, before the OOM killer eventually terminated the tablet server process on that node. Worse yet, subsequent tests on the same nodes would crash *immediately* for want of memory—despite the YugaByte DB's processes being killed between tests!

There are two separate problems here. The first is that tablet servers spawn a backend postmaster process to handle transactions, which in turns spawns a worker process per connection—but when tablet servers are killed, they don't ensure the postmaster or worker processes exit as well.

The second problem is that worker processes leak over time: in some of our tests, the number of extant worker processes rose by roughly 1.5 processes/second, even when the network connections those processes were meant to handle had long since closed. Each process reported its status as "idle in tranasction (aborted)".

As far as we can tell, these processes *never* exit, and must be manually killed—perhaps because they're stuck inside a transaction, and unable to respond to the closure of their connection. YugaByte is investigating.

4.7 Undercounting Counters

With clock skew larger than the configured YB clock skew threshold, we were able to observe stale values from YSQL counters, wherein a read of a single, increment-only record could observe a value lower than the sum of all previous successful increments. This is a violation of single-key linearizability, but has not (thus far) appeared in our general-case linearizable register tests–perhaps because we limit those workloads to significantly lower throughput, to avoid running into a combinatorial explosion of the state space.

This behavior, YugaByte confirms, is by design. It does not violate serializability; only linearizability & strict serializability. While YugaByte DB does claim to offer "linearizability" and "strongly consistent replication", these invariants do not hold when clock skew thresholds are exceeded.

4.8 Default Columns Initialized to NULL

When experimenting with various ways of ordering rows for tests, we discovered an odd behavior: columns which defaulted to `NOW()` could occasionally contain `NULL` instead. Take, for instance, this schema:

```
CREATE TABLE cats (
  birthday  TIMESTAMP DEFAULT NOW(),
  toes      INT
) IF NOT EXISTS;
```

… and a workload composed of a mix of inserts (`INSERT INTO cats (toes) VALUES (?)`), and reads (`SELECT birthday, toes FROM cats ORDER BY birthday`). Reads in this workload observe a mix of keys: most are timestamps, but a significant fraction are are `NULL`.

Indeed, this issue affects *every* kind of default value, not just `NOW()`. The cause? Non-transactional schema changes. Schema defaults are assigned *after* the table is created, which could allow inserts into recently created tables to believe (improperly) that no default value for the column was assigned.

4.9 Item Anti-Dependency Cycles

When master nodes crash or pause, YugaByte DB can exhibit a serializability violation called G2-item (item anti-dependency cycle) in append tests. We have observed dozens of these anomalies, and we present examples here with minor changes for readability.[2]

Consider transactions over a set of lists, each identified by a key like x or y. A transaction is a sequence of randomly mixed read and append operations. We write `[r x [5 6]]` to indicate a read of x returning the list `[5 6]`, and `[a x 7]` represents appending the number 7 to the end of x's list. The initial value of every list is `nil`; an append of `n` to a `nil` list results in `[n]`.

T_1: …, [:a x 837], …, [r y [… 874 877 883]]
T_2: …, [:a y 885], …, [r x [… 831 833 836]]

Here (and we have not, in the interest of space, provided the full values of x and y) T_1 appends 837 to x, which is not observed by T_2, and T_2 appends 885 to y, which is not observed by T_1. If this history were serializable, one of these transactions would need to observe the other's effects; since they mutually fail to observe each other, this history violates serializability.

Another example:

T_1: …, [r y [3]], … [r x [1]]
T_2: …, [r x nil], … [a y 2], …
T_3: [a x 1], …

Since T_2 appended 2 to y, and T_1 did not observe that append, T_1 must precede T_2. Since T_2 observed the initial (`nil`) state of x, and T_3 appended 1 to x, we know $T_2 < T_3$. However, that append of 1 to x *was* observed by T_1, which means $T_3 < T_1$. This cycle implies there exists no total order over transactions: this history, too, cannot be serializable.

All cycles we found related to this bug involved multiple *anti-dependencies*: a relationship between transactions T_1 and T_2, such that T_1 fails to observe some write performed by T_2, and hence must precede T_2 in any serialization order. We have not yet observed a

[2] We use smaller numbers, omit irrelevant operations from transactions, omit full lists which include hundreds of numbers, and use variable names like x instead of numeric IDs.

cycle with only a single anti-dependency, which would consistute G-single: read skew.

Moreover, these cycles occur purely with primary-key reads and updates, and do not require the use of predicates; these anomalies are therefore G2-item.

These errors are associated with the concurrent crash and restart of all master processes, or when master processes temporarily pause. YugaByte believes this has to do with a backwards-compatible codepath which, when masters are inaccessible, allows proxies to submit incorrect requests to tablet servers. YugaByte's engineers think this bug could cause "much more serious consistency issues", but we havn't observed anything worse than G2-item yet.

A patch in 1.3.1.2-b1 appears to fix this issue.

№	Summary	Event Required	Fixed in
1962	Constraint violation error during table creation	None	Unresolved
1974	Wrong error message during `update ... on conflict`	None	1.3.3*
1991	Create table `if not exists` throws `already exists`	None	Unresolved
2001	Create table `if not exists` memory leak	None	Unresolved
1993	Create table throws `violates unique constraint`	None	Unresolved
1992	Recovery from network partitions can take 100+ seconds	Partition	Unresolved
1995	Crash on table creation	Leader election?	1.3.3*
2021	Columns with defaults may be initialized to `null`	None	Unresolved
2075	Gradual postgres process leak	None	Unresolved
2125	Anti-Dependency Cycles	Master pause/crash	1.3.1.2-b1

* Version 1.3.3 is not yet ready, but patches for these issues have been tested in development builds, and should be a part of the 1.3.3 release.

5 Discussion

In our testing, YugaByte DB 1.3.1 exhibited several issues with DDL statements, as well as resource leaks and crashes, especially on cluster startup or leader transition. We also found two safety issues: columns with defaults could be initialized to `NULL`, and G2-item anomalies when masters are inaccessible. In addition, under normal operations, YugaByte DB appears to slowly leak Postgres processes, eventually crashing due to memory pressure. YugaByte is investigating remaining issues, and plans to fix remaining high priority issues by 2.0.

Otherwise, YugaByte DB was relatively robust in our transactional tests. Although it claims to provide only serializability, and theoretically allows realtime consistency violations like stale reads and causal reverse, these anomalies were rare in our testing.

Be aware that 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.

5.1 Recommendations

Because DDL operations (e.g. creating tables) are fragile in YugaByte DB, we advise against automating them. Concurrent operations appear more likely to create issues, but even single-threaded table creation can result in anomalous behavior, including strange error messages (1962, 1991, 1993), resource leaks (2001), and crashes (1995). If automation is used, users should avoid unbounded retries, and develop a strategy for finding and purging duplicate tables. Having a human operator in the loop should reduce the impact of these issues.

While YugaByte DB generally recovers within a handful of seconds from a network partition (assuming the partition is, in fact, recoverable), it can occasionally take hundreds of seconds to resume normal operations (1992). Be aware of variable recovery times when planning for, and testing behavior during, network failures.

The `DEFAULT` issue (2075) allows columns to be improp-

erly initialized to `NULL` in freshly-created tables. Until YugaByte DB provides transactional schema changes, users should avoid writing to a table until its creation has been completed. G2 anomalies (2125) could affect many transactional workloads, though gauging their safety impact would depend on a careful analysis of the particular transaction semantics involved. We recommend upgrading to the next production release above 1.3.1.2-b1, which should resolve the issue.

YSQL is still in beta, and SQL databases have many interacting features. Our work focused on basic transactional safety: reads and writes via primary key, indexed, or unindexed columns. We are not equipped to evaluate the broad range of features in a full-scale SQL implementation. It should come as no surprise, then, that our work did not encounter other other safety issues in YugaByte DB extant in 1.3.1, such as 2111, 2061, 1946, 1546, 1386, 1250, or 990, …). Some of these issues have been addressed by YB in recent development builds, and others are pending.

As in our last analysis, YugaByte DB can violate some of its safety guarantees (e.g. linearizability) when clock skew exceeds the configured threshold, which defaults to 50 ms. We recommend measuring the clock skew in your environment over time, alerting on exceptional skew, and setting `--max_clock_skew_usec` accordingly. For comparison, Cockroach DB uses a default value of 500 ms.

Though YugaByte advertises both serializability and SQL features throughout their marketing pages, the documentation also states that YSQL is "currently in beta", and it will not be "production ready" until version 2.0. Jepsen and YugaByte do not advise relying on serializable SQL transactions for safety-critical applications at this time. However, basic DML operations appear *mostly* safe, which gives us heart that remaining issues may either be rare or limited to specific features (e.g. temp tables, compound indices, cascades, etc.) rather than fundamental problems in the transactional protocol.

5.2 Clocks

One unusual question remains: YugaByte DB uses hybrid logical clocks to provide consistency across Raft clusters. However, in our testing, we observed no serializability violations in response to clock skew of hundreds of seconds, and with YugaByte DB's clock skew tolerance set to absurdly low (e.g. 1 microsecond) thresholds, and only observed linearizability violations in specific, high-throughput workloads. If these violations are possible, why are they so hard to find?

In theory, hybrid logical clocks allow databases like Spanner, CockroachDB, and YugaByteDB to provide strong consistency guarantees (e.g. seralizability or strict serializability) *without* waiting for the message delays they might otherwise require. Both YugaByte DB and Spanner require a minimum of 8 cross-datacenter message delays[3] to perform a read-write transaction. However, we also know that some databases (e.g. FaunaDB) can obtain strict serializability with a minimum of 2 cross-datacenter message delays.[4] It may be the case that the extra messages exchanged by YugaByte DB provide (via the Lamport-clock behavior of hybrid logical clocks) sufficient ordering to timestamps that the effects of even large clock skew are (typically) invisible.

It might also be that way we *test* YugaByte DB somehow masks clock errors. Perhaps our transactions are too frequent, or conflict too often, to observe divergence. Perhaps slower intra-cluster network links are required, or we need a special combination of clock error and, say, network partitions, which hasn't yet occurred in our randomized failure schedules.

Whatever the case, this is a good thing for operators: nobody wants to worry about clock safety unless they have to, and YugaByte DB appears to be mostly robust to clock skew. Keep in mind that we cannot (rigorously) test YugaByte DB's use of `CLOCK_MONOTONIC_RAW` for leader leases, but we suspect skew there is less of an issue than `CLOCK_REALTIME` synchronization.

5.3 Future Work

The focus of this work was on expanding YugaByte's YSQL tests by introducing our new, general-purpose transaction workload: the append test. We have run, but not thoroughly explored, the other YCQL, or other YSQL tests.

[3] In our previous analysis of 1.1.9, we provided a Lamport Diagram of YugaByte DB's transaction protocol, which required 2 cross-datacenter network delays for a read transaction, and 8 for a write transaction. This was based on YugaByte's transaction path documentation, which continues to suggest that read-write transactions require 10 cross-datacenter network delays. However, since March of 2019, YugaByte DB (like Spanner) creates the tablet status record concurrently with provisional writes. Read-write transactions now involve 8 network delays (assuming a single read).

[4] Transactions in FaunaDB, Spanner, Cockroach, YugaByte DB must block for various reasons, which creates additional latency to consider. A full description of their respective blocking costs is outside the scope of this discussion. We have also avoided discussion of latency costs during node failures or partitions. While these concerns are important, we feel they are best addressed via experimental measurement.

We have also not explored filesystem or disk corruption problems. Our present test design colocates masters with tablet servers, which is how YugaByte DB is typically deployed in production. However, this introduces a degree of false coupling when we create network partitions. It might be useful to explore partitions between just masters, or just tablet servers, instead of both simultaneously. Our tests are also limited in their ability to simulate `CLOCK_MONOTONIC_RAW` skew, which helps YugaByte DB ensure recency. Future analyses could find a better technique for testing `CLOCK_REALTIME` errors.

YugaByte DB's transaction protocol is complex: there are interacting systems of lock granularity, leases, timestamps, blocking for safe timestamps, Raft log ordering, membership changes, masters and tablet servers, paths for recovery, and so on. While we can test this system experimentally, Jepsen is concerned that the complexity and time dependence of the algorithm might leave important corners of the state space relatively undersampled. A formal model, combined with a model checker, might prove fruitful in exploring edge cases.

As YugaByte prepares to officially release support for SQL and serializable transactions in version 2.0, they plan to fix remaining serious bugs, and continue testing internally. YugaByte engineers are also considering a formal specification of their transactional mechanism, which could help find bugs we can't find through experimental means.

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, Karthik Ranganathan, Kannan Muthukkaruppan, Neha Deodhar, Alexander Abdugafarov, and Bogdan Matican.