JEPSEN

Dgraph 1.1.1

Kyle Kingsbury
2020-04-30

Dgraph is a distributed graph database which uses Raft for per-shard replication and a custom transactional protocol for snapshot-isolated cross-shard transactions. Dgraph resolved all issues from our 2018 report on version 1.0.2, and requested a brief followup. We found five safety issues in version 1.1.1—some known to Dgraph already—including reads observing transient null values, logical state corruption, and the loss of large windows of acknowledged inserts. All of these issues involved tablet migration. Dgraph has addressed three of these issues in recent development builds, and we are unsure of the remaining two. This work was funded by Dgraph, and conducted in accordance with the Jepsen ethics policy.

1 Background

Dgraph is a graph database which aims to provide scalable, highly-available, and snapshot-isolated transactions over a labeled directed graph, while minimizing network communication for performance. Conceptually, Dgraph stores a set of `(entity, attribute, value)` triples. Entities (also known as subjects), are compact binary UIDs. Attributes (also known as predicates) are named edges. Values (also known as objects) are either literal values, or the UIDs of other entities. Together, these triples form an adjacency list representation of a graph. The types, cardinalities, and indices of each predicate are given by a partial schema language—when a schema is not defined, one is automatically inferred.

To read this graph, Dgraph offers a recursive query language adapted from GraphQL. Mutations are expressed by listing triples to add or remove from the graph. For convenience, Dgraph can also represent all triples associated with a given entity as a JSON object mapping attributes to values—where values are other entities, that entity's attributes and values are embedded as an object, recursively.

To store large datasets Dgraph shards the set of triples by attribute, breaks attributes into one or more *tablets*, and assigns each tablet to a *group* of nodes. Each group uses Raft to provide replicated, sequentially/linearizably consistent storage and queries over that group's triples. So long as a majority of each group's servers remain intact and connected, Dgraph should remain available.[1]

To provide transactional isolation across different Raft groups, Dgraph has built a custom transaction system. Storage nodes (called Alpha) are controlled by a supervisory system (called Zero). Zero nodes form a single Raft cluster, which supervises the Raft clusters formed by each Alpha group. Zero leaders assign transaction timestamps and detect conflicts at commit time, as well as maintaining cluster membership, and the mapping of tablets to groups.

1.1 Consistency

As a part of our collaboration, Dgraph added a section on consistency properties to their public documentation, which states that transactions in Dgraph ensure snapshot isolation (SI) plus a realtime safety property: if transaction T_1 commits before T_2 begins, than the commit timestamp of T_1 is strictly less than the start timestamp of T_2.[2]

When transactions only interact with single keys, Dgraph's real-time guarantees imply linearizability. However, Dgraph transactions are not linearizable in general, because linearizability requires that operations (i.e. transactions) appear to take place atomically,

[1] Note that Dgraph may create groups with fewer than the specified number of replicas, when the number of nodes in that group is not evenly divisible by the target replica count. Those shards have reduced fault tolerance.

[2] This property may be what Elnikety, Pedone, & Zwaenepoel refer to as "conventional snapshot isolation" in Generalized Snapshot Isolation and a Prefix-Consistent Implementation.

whereas snapshot isolation allows transactions to interleave so long as their write sets are disjoint. Linearizability over atomic transactions is strict serializability: a stronger property. However, the constraint that snapshot times are consistent with real-time order is intuitive and useful: it prevents well-known anomalies such as stale reads.

2 Test Design

We reviewed and updated the Jepsen test suite from our previous analysis, primarily updating error handling routines to adapt to new Dgraph client and server behavior since 1.0.6. We ran our tests on five-node Debian clusters, both on LXC and EC2, with replication factor three. Dgraph Alpha nodes were organized into two groups: one with three replicas, and one with two. Every node ran an instance of both Zero and Alpha.

We measured Dgraph's behavior under a variety of failure modes, including Alpha and Zero crashes, tablet moves, clock skew, and network partitions with both transitive and non-transitive topologies.

2.1 Set

Our most basic test inserts a sequence of unique numbers into Dgraph, then queries for all extant values. We then check that every successfully acknowledged insert is present in a final read. We ran two variants of this test.

The first variant uses a schema with `type` and `value` fields, and for each inserted value v, creates a new entity with type "element" and value v. To query, we search for every object with type "element", and return their corresponding values. The join from `type` to `value` attributes helps verify that Dgraph's `type` index works correctly.

The second variant omits the `type` field and instead uses a single entity; every insert of v creates a triple mapping that entity to v. This means that we can query for every value associated with that particular UID, which maps directly to the way Dgraph stores triples internally. Dgraph finds the group associated with the `value` predicate, looks up that particular entity's UID in that group, and returns all matching values, without using indices.

2.2 Upsert

An *upsert* is a common database operation in which a record is created if and only if an equivalent record does not already exist. For instance, we might wish to ensure a user record exists for a given email, but if the email is already taken, to avoid creating a second user. In SQL databases, a unique primary key can be used as the equivalence relation for upserts, but in Dgraph there are no uniqueness constraints. Instead, users perform a transaction which reads to ensure the record doesn't already exist, then inserts if necessary.

However, snapshot isolation only detects conflicts between transactions which write the same objects, but inserts, by definition, write *unique* objects and will never conflict. This allows write skew: two concurrent upserts of the same value could read an empty state, insert their respective rows, and commit, resulting in *two* records instead of one. To avoid this problem, Dgraph also treats *indices* as their own objects for the purposes of conflict detection.

> The index is stored as many key/value pairs, where each key is a combination of the predicate name and some function of the predicate value (e.g. its hash for the hash index). If two transactions modify the same key concurrently, then one will fail.

To verify that this conflict detection works correctly, we have several transactions concurrently attempt to upsert the same value, and subsequently read back all objects with that value. If upserts are safe, we should never find more than one copy for a given key.

2.3 Delete

Early experiments with Dgraph led to the suspicion that deleting records might cause anomalous behavior, especially with respect to indices, so we designed a test for repeated upserts and deletions of the same value. Axiomatically, upserts should never result in more than one record—we verify this in the upsert test. Our delete test extends this workload by concurrently attempting to delete any records for an indexed value. Since deleting can only lower the number of records, not increase it, we expect to never observe more than one record at any given time.

2.4 Bank

The bank test stresses several invariants provided by snapshot isolation. We construct a set of bank accounts, each with three attributes:

1. `type`, which is always "account". We use this to query for all accounts.
2. `key`, an integer which identifies that account.
3. `amount`, the amount of money in that account.

Our test begins with a fixed amount ($100) of money in a single account, and proceeds to randomly transfer money between accounts. Transfers proceed by reading two random accounts by key, and writing back new amounts for those accounts. Concurrently, clients read all accounts to observe the total state of the system.

Since transfers write every key that they read, snapshot isolation precludes concurrent execution of any transfers between intersecting accounts, guaranteeing transfers are serializable. Read-only transactions cannot affect the state of the system, and observe consistent snapshots, which implies they too must be serializable. From this, we can prove that the total of all account balances should be *constant*.

Because we like to live dangerously, we permute the order of reads and writes in transfer transactions at random, upsert new account records when none exist, and delete accounts which have a zero balance. This puts additional stress on Dgraph's index, which cannot assume that queries for a certain key always refer to the same entity. We also insert garbage data before aborting certain transactions, to help detect dirty reads. Different accounts use different predicates to store their keys, values, and types, which means that transfers and reads may cross multiple groups, rather than being executed on the same Raft cluster.

2.5 Long Fork

For performance reasons, some database systems implement *parallel* snapshot isolation, rather than standard snapshot isolation. Parallel snapshot isolation allows an anomaly prevented by standard SI: a *long fork*, in which non-conflicting write transactions may be visible in incompatible orders. As an example, consider four transactions over an empty initial state:

1. `(write x 1)`
2. `(write y 1)`
3. `(read x nil) (read y 1)`
4. `(read x 1) (read y nil)`

Here, we insert two records, x and y. In a serializable system, one record should have been inserted before the other. However, transaction 3 observes y inserted before x, and transaction 4 observes x inserted before y. These observations are incompatible with a total order of inserts.

To test for this behavior, we insert a sequence of unique keys, and concurrently query for small batches of those keys, hoping to observe a pair of states in which the implicit order of insertion conflicts.

2.6 Sequential

Earlier versions of Dgraph offered a per-client property akin to sequential consistency, which enforced that each individual client observed monotonically increasing states of the graph. To help check this property, we establish a set of registers, each composed of a key and a value. On each register separately, we perform a series of increment operations mixed with reads of that register. Since our transactions only interact with single keys, snapshot isolation implies serializability. Since the value of a register can only increase over time, we expect that for any given process, and for any given register read by that process, the value of that register should monotonically increase.

3 Previous Issues

Our analysis of Dgraph 1.0.2, completed in August 2018, left four issues unresolved: a deadlock in cluster join, an issue where transactions would time out at the end of set tests, and two snapshot isolation violations which allowed for permanent logical state corruption, associated with and without tablet moves, respectively. Dgraph has since closed all of these issues, and we'd like to review them briefly.

3.1 Deadlocks in Cluster Join

When setting up new clusters, Dgraph Alpha nodes could get stuck indefinitely at the JoinCluster phase. Dgraph believes this problem had to do with a quorum check performed by the underlying Raft library, combined with Dgraph's parallel join process. Disabling the quorum check for reads, and ensuring that nodes joined the cluster one at a time, seemed to resolve the issue by 1.0.8-rc1, and it did not appear in our review of 1.1.1 either.

3.2 Endless Timeouts

At the end of UID set tests, we found occasional cases where Dgraph could time out every read query after some point. This problem affected clusters without exogenous faults, and, once triggered, appeared to last indefinitely: we observed up to an hour without recovery. Nodes appeared to be in the middle of an automatic predicate migration which never completed.

The cause of this issue was never ascertained, but by February 2019, it was no longer reproducible.

3.3 Permanent SI Violations with Multiple Tablets

In 1.0.5-dev, bank tests—even in healthy clusters—could result in account balances drifting higher or lower over time. Effects could be limited to particular nodes. Some tests showed only transient incorrect balances, and others appeared permanently altered. This problem appeared even without predicate moves.

By version 1.0.7, Dgraph no longer exhibited snapshot isolation violations in bank tests with healthy clusters, though it still corrupted data with network partitions.

By November 2018, Dgraph had identified a cause. When a Zero leader received a commit request for a transaction T, it assigned a timestamp to that commit. If Zero was unable to communicate with its Raft peers, and a new Zero node became the leader, that new leader would begin allocating timestamps at a significantly higher number. Alphas interacting with the new Zero leader would advance their `max-applied` timestamps to match. Then assume the original Zero leader rejoined the cluster as a follower, and retried its commit proposal—this time, succeeding. Because this new proposal kept the *original* transaction timestamps, two problems could occur:

1. A read R executed after the new leader advanced the clock, but before T's commit was retried, could fail to observe T—even though T would go on to commit in the logical past of R. In essence, this allowed temporary "holes" in the timeline of transactions.

2. When an Alpha node applied a write w for key k, it would first check k's last written timestamp, and ignore w if it was lower. If w was a write from the logical *past*, w might be rejected—but other writes from the same transaction might succeed, so long as they hadn't been written recently. This allowed Dgraph to partially apply transactions.

In addition, Dgraph identified and fixed a second, related bug in the transaction commit process. When Alpha leaders received transaction commit messages from Zero, they appended those commits via Raft to their log. However, if that append process timed out, Alpha would *give up* on appending that commit message. This allowed transactions to be applied on some Alpha groups, but not on others.

Both of these issues were fixed in 1.0.11, which prevented Zero leaders-cum-followers from sending transaction commit proposals to new leaders after stepping down, and by forcing Alpha nodes to retry commits on indeterminate failures, rather than giving up on them.

3.4 Permanent SI Violations with Single Tablets

Even in healthy clusters, version 1.0.4 exhibited read skew in bank tests, leading to permanent state corruption. Some of this behavior was caused by queries returning spurious null values instead of valid data, but others were caused by read skew. Account totals could change gradually over time, fluctuate chaotically, or alternate between two different values.

While these symptoms were similar to the previous issue, they had different underlying causes: these problems were linked to tablet migration. By January 2019, Dgraph had redesigned the tablet migration code: instead of blocking writes on Alphas during a migration, it would instead block commits on the Zero leader—the node with an authoritative view of the tablet-to-group mapping. This prevented writes from sneaking onto the wrong nodes in the interval between the mapping changing on Zero, and being replicated to all Alphas. As an additional safety measure, Alpha nodes also now encode a subset of their local tablet-to-group mapping with each commit request, so that Zero can identify a potential mismatch.

This redesign prevented account totals from changing permanently. However, reads do *not* (for performance reasons) consult Zero, which allowed read-only transactions to observe occasional transient read skew. Upon receiving a read at time t, Alpha would block until it had applied every transaction up to t, to ensure no transaction's effects would be missing for the read at t. However, if that particular Alpha had an outdated view of the tablet to group mapping, it could read a tablet which had just been migrated to some other node, or a tablet which it was *supposed* to own, but didn't yet, resulting in stale or empty reads, respectively.

In essence, this problem stemmed from the fact that membership changes and transaction commits formed two separate, asynchronous streams of information from Zero to Alpha; a node might be processing recent transactions, but be out of date on cluster membership. To bring these streams into alignment, Dgraph added a checksum of the membership state to each batch of transaction commits, and ensured that Alpha nodes refused to service requests when their membership state hasn't caught up to the transaction stream.

This prevented reads from executing on nodes which didn't yet, or no longer, held the current copy of that tablet. Dgraph then passed bank tests even with tablet

migrations[3]; the patch was released in Dgraph 1.1.

4 New Results

In the present work, we tested Dgraph 1.1.1, as well as later development builds. We encountered five safety issues, all involving tablet migration.

4.1 Transient Missing Values

In 1.1.1 and 1.1.1-56-ge18986f1c, we observed cases where reads would return a null value for a record which should have existed. For instance, consider this bank test, in which a handful of reads observed null values for account balances…

```
{0 nil, 1 nil, 2 5, 3 13, 4 11, 5 nil,
 6 nil, 7 17}
```

… or null values for account keys:

```
{nil 15, 0 10, 2 1, 3 36, 4 4, 7 9}
```

One could interpret this behavior as read skew, since `nil` is the initial state of every record—but this problem does not resemble read skew in general. In almost every case, we observed the *absence* of data, rather than a value from the wrong timestamp.

These errors were common in version 1.1.1, occurring with essentially every tablet move. In sequential tests (issue 4540), they manifested as spurious non-monotonic reads. In bank tests (issue 4534), we saw transient reads where the total balance was lower than expected, because some accounts showed `nil` rather than their actual balance.

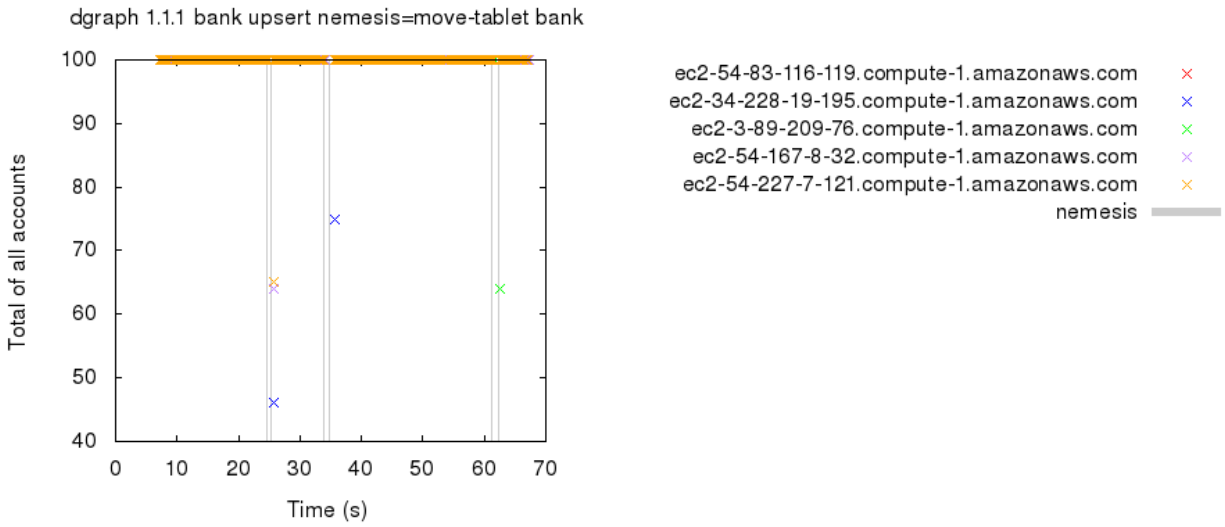

Figure 1: In this plot of total balances over time, some reads immediately following a tablet move operation (vertical grey lines) observed null values for some accounts, resulting in a low balance.

This issue occurred immediately following tablet migration. When a tablet was moved from one shard to another, the new shard could serve transactions whose start timestamp was *prior* to the tablet move time, and the old shard could serve transactions with a start timestamp *after* the move time—i.e. after that shard had deleted the tablet entirely. Without any data, those shards would return null values.

This bug was fixed in ec445503, which should be released as a part of 1.1.2; we have verified that this patch dramatically reduces the probability of spurious nulls.

However, we continued to see this problem infrequently with 1.1.1-56-ge18986f1c. During tablet moves, one read every few thousand seconds observed null instead of the actual value. More recent builds have not exhibited this problem, but we left issue #4575 open until a cause and fix can be confirmed.

[3]These passing results may have been somewhat premature; later testing revealed additional issues, which we describe in this report.

4.2 Read Skew Leading to Data Corruption

In versions 1.1.1 and 1.1.1-48-g157896305, bank tests occasionally exhibited permanent changes in the balance of all accounts. Under snapshot isolation, the bank workload should observe a constant balance over time. However, a tablet move could cause a single read query to observe two account balances from *different* timestamps: an anomaly called read skew. This was issue #4543.

In this history, a transfer transaction moved $2 from account 4 to account 6, which emptied account 4, and changed account 6's balance from $1 to $3. However, a later read observed account 4 *after* the transfer, and account 6 *before*, resulting in a total of $98 rather than $100. We have elided other transactions for clarity.

```
read {0 58, 1 11, 3 1, 4 2, 5 20, 6 1, 7 7}
transfer {:from 4, :to 6, :amount 2}
read {0 55, 1 11, 3 4, 5 20, 6 3, 7 7}}
read {0 55, 1 11, 3 4, 5 20, 6 1, 7 7}}
```

Transfer transactions which wrote new values based on skewed reads allowed Dgraph to propagate transient read errors into permanent changes: the total of all accounts changed from $100 to $98, and remained that way for the remainder of the test.

In this test run, process 5 begins a transfer of $2 from account 6 to account 3, while a tablet move is ongoing. While that transfer is happening, a pair of read transactions observe account 3's balance increase from $5 to $7, but no corresponding decrement is made to account 6. The transfer transaction then fails with an error message indicating that the read timestamp for that transaction was lower than the minimum timestamp available for that key. Again, we elide other operations for clarity:

```
transfer {:from 6, :to 3, :amount 2}
read {0 51, 2 1, 3 5, 4 11, 5 3, 6 8, 7 21}
read {0 51, 2 1, 3 7, 4 11, 5 3, 6 8, 7 21}
... move-tablet completes ...
read :error :old-timestamp}
```

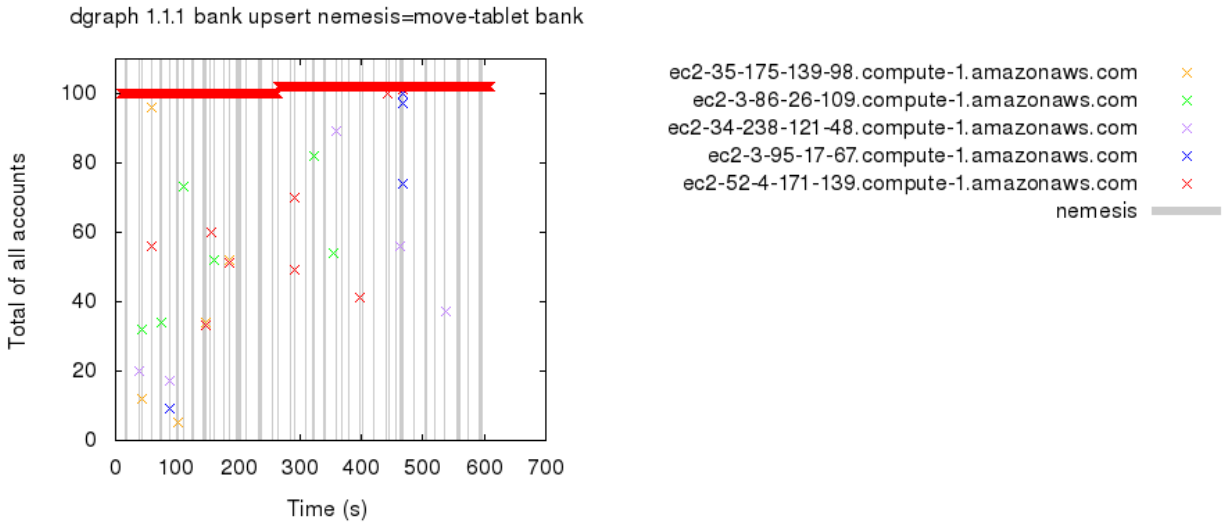

Figure 2: A plot of total account balances over time. After the read skew anomaly, the total remains $102 for the remainder of the test. Transient low values are caused by the transient null value problem discussed previously

It's not clear from this history whether process 5's transfer somehow took *partial* effect, or whether it cleanly failed and something else in Dgraph caused the value of account 3 to fluctuate. Whatever the case, the effects were permanent: for the rest of the test, every read (except those affected by transient null issues, as described previously) observed a total of $102, rather than $100.

Dgraph is still investigating this issue.

4.3 Loss of Inserted Records

In UID set tests with Dgraph 1.1.1, 1.1.1-48-g157896305, and 1.1.1-65-g2851e2d9a, we observed that windows of up to tens of thousands of acknowledged inserts could be lost. This issue (#4538) appeared to be associated with tablet moves.

For example, in this test run, Dgraph acknowledged 22,187 writes successfully. However, of those acknowledged writes, all 11,544 between 11,350 and 23,715 were lost—they failed to appear in a final read. Triples inserted both before and after that window were fine.

Dgraph believes this problem is related to posting list splits. A *posting list* is a collection of edges belonging to some attribute. When a posting list becomes large, it is split into a tree, whose root is identified by a canonical key. However, a bug in Dgraph allowed the parts of a split posting list to be accessed individually, instead of through the main key. Accessing the posting list through these secondary keys caused issues during rollups, and resulted in spurious keys being added to the database. Additional patches disabled posting split lists, and we believe this problem may be resolved in 1.1.1-59-g191232226.

4.4 Many Writes Enter, One Write Leaves

In version 1.1.1, we saw something else unusual with UID set tests: Dgraph would successfully acknowledge tens of thousands of inserts of distinct triples, and, when we asked for all of them back, return exactly one. In this test run, we successfully inserted 19,030 unique integer values, and, upon reading them back, received:

```
{:q [{:uid 0x1, :value 24333}]}
```

… rather than a list of values:

```
:q [{:uid 0x3, :value [1, 2, 4, ..., 24333]}]
```

This issue only occurred infrequently, but the impact was severe: not only was all but one write lost, but the *type* of value changed! Instead of receiving a list of integers, we got only a single number. This is particularly vexing because the schema for this attribute explicitly defines `value` to be a list: `value: [int]` .

Dgraph believes this problem (#4601) could be associated with a bug in splitting posting lists, and that recent patches have addressed the issue. Indeed, we have not encountered it in 1.1.1-65-g2851e2d9a, or subsequent builds. However, reproducing this issue has proven difficult, and without a plausible account as for *why* the posting-list bug could cause the schema to change, we are cautious about declaring it fixed.

№	Summary	Event Required	Fixed In
4534	Transient missing values	Tablet move	1.1.1-56-ge18986f1c
4575	Transient missing values (infrequent)	Tablet move	Unresolved
4543	Permanent state corruption	Tablet move	Unresolved
4538	Lost inserts	Tablet move	1.1.1-59-g191232226?
4601	Many writes enter, one write leaves	Tablet move	1.1.1-65-g2851e2d9a?

5 Discussion

Dgraph resolved all of the issues we discussed in the previous Jepsen analysis. However, we found significant new safety issues in 1.1.1 which were, in many cases, functionally identical to bugs we've seen in the past: read skew, transient missing values, and lost inserts.

This does not mean that Dgraph has failed to make progress. Indeed, the work Dgraph has undertaken in the last 18 months has dramatically improved safety. In 1.0.2, Jepsen tests routinely observed safety issues even in healthy clusters. In 1.1.1, tests with healthy clusters, clock skew, process kills, and network partitions all passed. Only tablet moves appeared susceptible to safety problems.

As of 1.1.1-59-g191232226, we have failed to observe any violations of snapshot isolation. The current Jepsen test suite passes—at least with short runs. Unfortunately, some of these bugs were difficult to reproduce, and we have not had sufficient testing time to declare these issues resolved. Since Dgraph Labs has not identified a potential cause or patch resolving 4575 and 4543, we've left these issues categorized as "unresolved"—though we cannot prove they are present in 1.1.1-59-g191232226. Dgraph Labs plans to perform additional testing as they go forward.

As always, we note that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we try hard to find problems, we cannot prove the correctness of any distributed system.

5.1 Tablet Moves

All of the issues we found had to do with tablet migrations, which raises the obvious question: why migra-

tions in particular? We suggest two potential causes.

In part, we found issues in tablet migrations because that's where we looked. Dgraph Labs knew that 1.1.1 had issues with tablet moves, but had difficulty fixing them, in part because the Jepsen test suite no longer ran reliably. Changes to Dgraph APIs prevented Jepsen from properly detecting common failure conditions, which in turn broke the retry mechanisms Jepsen uses to run tests reliably. Updating the tests to interpret Dgraph's new error types, as well as some tuning changes, allowed us to find and confirm bugs faster. With those Jepsen improvements in place, we focused on tablet moves and specific workloads in our collaboration, knowing there were extant bugs to find.

In more general terms, tablet migrations were error-prone because changing distributed state is just plain *hard*. Dgraph relies on Raft for state changes within a group, and Raft is a solid algorithm with mature implementations that handle failure well. Dgraph coordinates transactions between Raft groups by having nodes agree on which groups own which tablets. This too is relatively straightforward—as long as that mapping doesn't change. When mappings are stable, everyone's requests go to the right groups, and Raft handles it from there. When the mapping *changes*, nodes might be out of date: Dgraph doesn't, for performance reasons, use a consensus protocol for tablet mappings. Instead, nodes *asynchronously* discover mapping changes via side channels, which makes agreement trickier.

We see this pattern in many Jepsen tests: as more databases adopt proven consensus algorithms like Paxos and Raft for shard state, we've found fewer bugs at the level of individual shards. Coordinating cluster metadata and ensuring transactional correctness *across* those shards has proven more difficult. This suggests an important avenue of research for academics—and an area of caution for engineers.

5.2 Recommendations

Dgraph 1.1.1 exhibited significant violations of snapshot isolation related to tablet migration: an infrequent but normal process in Dgraph. Users could experience transient read skew, permanent state corruption, the loss of large windows of committed inserts, or the replacement of a (potentially large) set of values with just one, along with type errors: returning a single integer, rather than a list.

In version 1.1.1 and below, we recommend exercising caution during tablet moves, if possible. For instance, performing manual tablet migration during a scheduled maintenance window, rather than under normal load, could reduce the probability that transactions encounter anomalous behavior. We also advise discontinuing use of the cluster during tablet migration to prevent update transactions from observing inconsistent data, then propagating that corrupt state back into the database.

Recent development builds, like 1.1.1-59-g191232226, have not exhibited these anomalies, but we stress that Jepsen has not had sufficient time to verify these builds in detail. While we cannot make a strong claim of correctness, Version 1.1.2 incorporates many patches which should dramatically reduce the frequency of errors. We advise upgrading to 1.1.2 or higher.

5.3 Future Work

Dgraph continues to evaluate open issues and perform their own Jepsen testing.

Meanwhile, Jepsen has developed Elle: a novel checker for transactional systems, which should be well-suited to testing Dgraph. We would like to apply Elle to Dgraph, especially with respect towards per-process and realtime guarantees. We would also like to evaluate Dgraph with slow networks, process pauses, and single-node faults like filesystem corruption.

In general, our tests now go thousands of seconds without finding bugs, but we *did* occasionally find issues in development builds. Weak discriminatory power suggests it's time to redesign the tests to be more aggressive. We could increase key and tablet counts, adjust contention probabilities, and shard tests where appropriate.

This work was funded by Dgraph, and conducted in accordance with the Jepsen ethics policy. Jepsen wishes to thank the entire Dgraph team for their help—especially Manish Jain and Daniel Mai.