JEPSEN

Dgraph 1.0.2

Kyle Kingsbury

2018-08-23

Dgraph is a distributed graph database which uses Raft for per-shard replication and a custom transactional protocol, based on Omid, Reloaded, for snapshot-isolated cross-shard transactions. Dgraph claimed to offer snapshot isolation, per-client monotonicity, and linearizability. However, in Dgraph 1.0.2 through 1.0.6, we found multiple deadlocks & crashes in the cluster join and node recovery processes, duplicate upserted records, snapshot isolation violations, single-client sequential violations, records with missing fields, and in some cases, the loss of all but one inserted record. Safety issues were mostly associated with process crashes, restarts, and predicate migration, but some occured in healthy clusters during normal operation. Dgraph has made significant progress, but 4 of the 23 issues we identified remain unresolved, including the corruption of data in healthy clusters. This work was funded by Dgraph, 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—though we did find instances of long fork using other tests. We have re-checked Dgraph 1.0.13 with a corrected checker, and its long fork tests still pass.

2 Background

Dgraph is a graph database which aims to provide scalable, highly-available, and snapshot-isolated transactions over a directed labeled 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, and assigns each attribute to a group of nodes. Each group uses Raft to provide replicated, sequentially/linearizably consistent storage and queries over its triples. So long as a majority of each group's servers remain intact and connected, Dgraph can theoretically preserve safety and availability.[1]

2.1 Consistency

As of February 2018, Dgraph's FAQ said Dgraph was CP, preserving consistency with reduced availability during partitions. The design concepts documentation clarifies that Dgraph transactions provide snapshot isolation (SI), which means that every transaction observes an atomic snapshot of the database at some start time in the past, and commits atomically at some later time, only if none of the keys it has written have been altered by *other* transactions between that trans-

[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.

action's start time and commit time.

Snapshot isolation is a relatively strong consistency model, but still allows some anomalies. For instance, two concurrent transactions can read the same pair of records and update one of the two concurrently, so long as they choose *different* records to update: *write skew*. Transactions are also allowed to read from arbitrary points in the past, which implies that completed writes may not be visible to later transactions: *stale reads*. However, Dgraph's comparison page states:

> Dgraph is consistently replicated. Any read followed by a write would be visible to the client, irrespective of which replica it hit. In short, we achieve linearizable reads.

The 1.0 release blog post goes on to call Dgraph "production ready", claiming:

> Dgraph provides consistent (synchronous) replication, utilizing Raft.... Dgraph guarantees atomic consistency of writes, which means irrespective of which replica is hit for reading, any write done before is guaranteed to be available.

However, a careful reading of Dgraph's earlier transactions announcement suggests a more subtle interpretation:

> There is no need to worry about seeing a previous database state when querying a replica. From the point of view of a single client, once a transaction is committed its changes are guaranteed to be visible in all future transactions.

The first sentence suggests that stale reads are prohibited in general, which sounds like linearizability. However, the second sentence suggests that this might only hold for *individual* clients, and not *between* clients. Enforcing an order at each client separately might be something like sequential consistency, rather than linearizability. As we will see, this client-side ordering turned out to be trickier than anticipated, and Dgraph went on to introduce a stronger ordering invariant during our collaboration.

2.2 Algorithm

To provide transactional isolation across different Raft groups, Dgraph has built a custom transaction system adapted from the Omid Reloaded paper. Storage nodes (called *Alpha*) are controlled by a supervisory system (called *Zero*). Zero nodes form a single Raft cluster, which organizes Alpha nodes into shards (called *groups*). Each group runs an independent Raft cluster.

These Zero leaders provide a central coordinator for all transactions, allocating UIDs to new entities, assigning timestamps to transactions, and checking transactions for conflicts. Timestamps are assigned by reserving a block of timestamps via Raft, then issuing timestamps from that block without further coordination. Zero leaders also detect conflicting transactions by maintaining an index of the timestamp when each row was last modified. Before committing a transaction, Zero checks every row in the write set to ensure that it hasn't changed since the start of the transaction. If the transaction is valid, Zero then obtains a commit timestamp, updates every last-committed time for written rows to that value, and submits a message to Raft, marking that transaction as committed.[2]

As a transaction takes place, clients update Alpha servers with *prewrites* (tentative, uncommitted versions of rows), then contact Zero to commit or abort. Once transactions are committed or aborted, Zero streams that commit state to each Alpha server, which uses it to determine whether to promote those prewrites to stable storage, or, if the transaction wound up aborting, to delete them.

To ensure reads observe all prior committed writes, Zero also streams a high-water-mark timestamp, below which all transactions have either been committed or aborted. Clients, in turn, keep track of the most recent Raft index they've observed on each Alpha group, and include that with their requests to ensure that they always observe monotonic states on any particular group. Dgraph terms this scheme "client side sequencing".

3 Test Design

We designed a suite of Jepsen tests to verify Dgraph's safety properties, using a five node cluster with repli-

[2]Readers may question why Dgraph executes so much of its algorithm *outside* Raft, opting instead to treat Raft more as a sequentially consistent log. For concision, we omit much of Dgraph's replication and commit algorithm here, but there are additional constraints which theoretically help to ensure Dgraph's state machine is correctly coupled to Raft; among them, propagating timestamp high watermarks. Dgraph has not yet published the details of their algorithm, but the Omid Reloaded paper provides a useful starting point.

cation factor three. This means 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 began with a *bank* test, adapted from our earlier CockroachDB analysis, and designed more specific tests to explore anomalous behaviors as they arose.

3.1 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 the final read. We designed 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 creates a triple mapping that entity to value 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 any indices.

If Dgraph allows stale reads, we might read a past snapshot of the database, and miss some more recently inserted values. We can work around this problem by keeping track of acknowledged writes externally, reading each of those keys, and re-writing the values we found. Snapshot isolation should detect these write conflicts and ensure that either our read+rewrite transaction aborts, or, if it commits, that it did not overlap with any successful write transaction.

3.2 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:

> Upsert operations are intended to be run concurrently, as per the needs of the application. As such, it's possible that two concurrently running operations could try to add the same node at the same time. For example, both try to add a user with the same email address. If they do, then one of the transactions will fail with an error indicating that the transaction was aborted.

One possible problem: 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.

3.3 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.

3.4 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 to reflect some money moving between them. 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 exists, 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.

3.5 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.

3.6 Sequential

Dgraph claims to offer a sort of recency property: clients are not supposed to observe a previous database state. Dgraph's documentation sometimes claims this property is linearizability, and indeed, clients call the data structure used to enforce monotonicity a "linearizable read map", but we know from conversations with Dgraph's engineers that this order holds only on individual clients: while client A might fail to observe a completed write by client B, B should subsequently observe its prior writes. This property seems loosely analogous to sequential consistency, which implies that there exists some order of operations consistent with the order on each individual process.[3]

It's not clear how to overlay sequential consistency on a snapshot-isolated system—should we consider a sequential "operation" to be a transaction? For nonserializable histories, it might be impossible to find an order consistent with each process. However, if we restrict ourselves to scenarios for which SI implies serializability, a serial order *must* exist, and we can verify that it is compatible with the order on each process.

To do this, we establish a set of registers, each comprised 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.

4 Results

We tested Dgraph 1.0.2, and successive nightly & experimental builds, developing new tests and failure modes incrementally. We began our testing with healthy clusters, then gradually introduced network partitions, Zero crashes, Alpha crashes, and predicate

[3]We believe prefix-consistent snapshot isolation may be what Dgraph is going for, but have not had time to explore this in full.

moves, and randomized mixes thereof, at a variety of time intervals. Our findings are as follows:

4.1 Schema issues

We found a minor issue around integer handling when getting started with Dgraph: when Dgraph first encounters a new attribute on a record, it infers the schema for that attribute based on the submitted value. Notably, integers are inferred to be floats, which means that a user could write 0 without a schema, and when trying to read it back, obtain 0.0 instead. Large integer values which are not representable as floats could be silently coerced to different values: 9007199254740993 becomes 9007199254740992.0. 27670116110564327426, at the upper end of the signed 64-bit integer range, comes back as 2.7670116110564327E19: 426 fewer.

Moreover, users who use an `int` schema, which is supposedly a 64-bit signed integer, might find large values silently remapped to other numbers. Values over 2^{53} (9007199254740992) would be remapped to integers which *were* representable as 64-bit floats. Because floats can represent a wider range of integers than ints, large integers like 9223372036854775296 might be mapped to 2^{64}, then coerced *back* to signed 64-bit integers, silently overflow that type, and wind up as -9223372036854775808 (e.g. -2^{63}): a different number of the wrong sign.

The JSON spec defines numbers with arbitrary decimal precision, but does not specify how implementations will *interpret* those numbers. Dgraph now checks for floats vs. ints during JSON parsing.

4.2 Cluster Join Issues

Early in the testing process, we discovered race conditions in Dgraph's cluster join procedure.

In 1.0.3, when joining a cluster, Alpha nodes request a snapshot of their neighbors' state—normally, from the leader of their Raft group. However, if no leader has yet been obtained, the node will block indefinitely, waiting for one. Since the node is blocked, it cannot participate in leader elections, which, in turn, prevents a leader from being elected: a deadlock! Dgraph patched this by updating local leader state asynchronously.

A separate bug in 1.0.3 caused Alpha to segfault on startup: nodes would attempt to contact a leader immediately, but if no leader was known, choose a random node instead. That node might not be initialized, which could lead it to dereference a null pointer and crash. This issue was patched by *not* returning a random leader, but instead waiting until a leader became available, and retrying later.

In the same vein, a race condition allowed leaders to respond to requests before they were fully initialized, causing those leaders to segfault. Dgraph added fallback paths for methods that rely on uninitialized group state to address this issue.

While investigating these crashes, Dgraph discovered a fourth issue in 1.0.3, where Zero would allow concurrent join requests for new Alpha nodes, leading to a deadlock. Dgraph fixed this by serializing join requests.

Another lockup in 1.0.3 resulted from an interaction between Raft, which only allows one pending cluster change at a time, and concurrent join requests. If, while joining a Raft group, a new Alpha node timed out, that node would loop indefinitely, retrying the join process forever. However, subsequent join requests would fail, because the *original* join operation was still pending in Raft. That join operation was, in turn, stalled because the joining node was stuck sending join messages, rather than processing the join. To fix this, Dgraph patched the issue by removing the timeout.

Unfortunately, this patch introduced a *new* deadlock in cluster join, where nodes would refuse to serve any requests after startup, due to their `joinCluster` requests being dropped by a Raft leader which didn't have quorum. This issue was fixed in 1.0.7.

On 1.0.4, we found yet another deadlock, where nodes would get stuck in their `JoinCluster` call indefinitely. Dgraph is still investigating.

4.3 Duplicate Upserts

In building the bank test, we discovered that the test initialization process, which concurrently upserts a single initial account, resulted in dozens of copies of that account record, rather than one. We designed the upsert test specifically to stress this behavior, and found that under normal operating conditions, Dgraph would allow arbitrarily many concurrent upserts to succeed for the same key.

This occurred because the Dgraph node coordinating an upsert transaction would keep track of that transaction's *write set* (the keys that transaction wrote), but if the node responsible for actually *executing* that transaction was remote (as opposed to local), the coordinator forgot to include the write set in the commit message sent to the remote node. As a result, the remote node would assume the transaction had no conflicts (since it wrote nothing!) and commit. Correctly filling out the commit message's write set resolved this issue, in version 1.0.4.

However, 1.0.4 weakened the default safety semantics: for performance, indices are no longer checked for conflicts by default, which means that upserts are still (by default) unsafe. Instead, one must add a new index directive, `@upsert`, on any indices used for upserts. This informs Dgraph that those indices should be checked for conflicts. With the appropriate `@upsert` directives, upserts worked correctly.

4.4 Delete Anomalies

However, anomalies occurred when we introduced *deletions*. With a mix of upserts, deletes, and reads of single records identified by an indexed field `key`, we found several unusual behaviors. A query for `key = 13` could observe *multiple copies* of the same key:

```
[{:uid "0x148b", :key 13}
 {:uid "0x150c", :key 13}]
```

This implies that some upserts must have failed to observe existing records, and, by inserting, created duplicate copies. Worse yet, we can observe records with *no* associated key:

```
[{:uid "0xcf"}
 {:uid "0x110"}]
```

This is somewhat vexing, as a reasonable observer might expect that the set of records with `key = 13` would contain keys with the value 13, or, barring that, any key whatsoever. These *dangling records* suggest an inconsistency between the index and the raw triples.

Moreover, dangling records can persist through subsequent deletions. To make matters even weirder, values can disappear due to deletion, get stuck in a dangling state, then *reappear* as full records—even in the context of a single process, which is supposed to observe transitions in a monotonic order.

These problems were caused, in part, by a race condition between the insertion of a new subject[4] and a concurrent transaction which deleted all values for that subject and some predicate. These deletes (which Dgraph calls "SP*") passed through a different codepath which was not subject to the same atomicity guarantees as regular commits. In particular, delete operations would take effect regardless of which transaction actually committed, and read transactions could observe deletions that committed after the read began—i.e., snapshots weren't actually snapshots.

In addition, Zero leader nodes failed to step down correctly: when asked to step down, they forgot to set their `leader` variable to `false`. This meant that if a node was later re-elected, it would assume it was *already* a leader, and re-use timestamps, rather than obtain a fresh block. That node would also fail to increment critical bounds on transaction timestamps used to ensure monotonicity. These issues allowed re-elected leaders to execute new transactions in the logical past. Correctly stepping down fixed this issue.

4.5 Read Skew

With a more reliable cluster join process, and working upserts, we discovered a read skew anomaly in the bank test. Clients could observe an *incomplete* transfer transaction between two accounts x and y, such that x's state was that prior to the transfer, and y's state was that after the transfer. This allowed reads to observe incorrect total balances, which fluctuated gradually through the course of the test. This behavior should be prohibited by snapshot isolation, but occurred constantly, even in healthy clusters.

Moreover, these skewed reads could be propagated *back* into the state of the database by transfer transactions, causing the total of all accounts to fluctuate further and further over time.

Skewed reads stemmed from Dgraph's use of client-side sequencing: individual clients would keep track of the last Raft index they had observed in each group, and ensure subsequent queries observed a Raft state at least that high.[5] However, there were race conditions in client-side sequencing: client A could insert a record and commit, then client B could read, observe a state *prior* to A's insert, then insert a second copy of that record and commit.

Dgraph introduced a new mode for transaction ordering: in addition to client-side sequencing, a server-side sequencing directive requires follower nodes to check with the leader and ensure that they have caught up to the leader's current state before responding to a query. This increases latency, but prevents these phantom anomalies in client-side sequencing.

[4]Recall that "subject" is another term for "entity", and "predicate" is another term for "value". Dgraph uses these interchangably.

[5]Owing to some confusion over what linearizability means, the structure storing the most recently observed Raft offset for each group is called a linearizable read map. It does not provide linearizability.

A second bug involved a race condition around SP* deletions (deletions of all objects for a given subject and predicate). When these deletion operations arrive at an Alpha leader, it applies them in order, and replicates them using Raft. Normally, once an operation is committed, Raft would ask the local node's state machine to apply that operation, obtaining a new state. By applying operations to deterministic state machines in the same order on every node, Raft obtains identical states.

However, Dgraph's state machine was not exactly deterministic: it applied some operations (notably, SP* deletions), in goroutines, rather than in a single thread. This meant deletions could be applied in different orders relative to commits, resulting in different states on different nodes. Dgraph added additional safety checks to ensure that SP* deletions would properly commute with other transactions' updates.

4.6 Lost Inserts with Network Partitions

While Dgraph was fixing those read skew issues, we uncovered a worse behavior: in pure insert workloads, Dgraph could lose acknowledged writes during network partitions. In set tests, which insert unique integer values and attempt to perform a final read, huge swaths of acknowledged values could be lost.

```
{:valid? false,
 :lost
 "#{140 149 151..155 169..174 176..178
   183 186 189 191 196 200..201 203 206
   208..210 212 214..227 229..237 242
   244 251 254 257..259 261..262 265 267
   269..271 277..278 461..466 469
   472..473 475 477..483 488..489 491
   494..495 497..498 502..505 507 510
   512..513 516 518 521..522 525 528 532
   537..538 541  544 547 550 675
   685..686 688..690 2263 2266..2269
   2272..2273 2275}",
 :recovered "#{}",
 :ok
 "#{0 6 10 12 18 23 25 29 32 34 37..39
    42 45 48 54 58 62..63 67 71 73 75
    77 81 84 86 90 106 109..115 129
    137 279 284}",
 :recovered-frac 0,
 :unexpected-frac 0,
 :unexpected "#{}",
 :lost-frac 127/2373,
 :ok-frac 41/2373},
```

Dgraph returns UIDs for each successful insert, so clearly *some* work has been done, but those records fail to appear in later queries for all objects of type "element". When this occurs, CPU use on Dgraph jumps to 100% for several minutes. To double-check that this issue was not caused by deferred indexing, we designed an alternate variant of the test which stores all elements on a single entity, removing the need for index queries. This too lost writes.

This was caused in part by Dgraph failing to advance read timestamps on leader changes; when new leaders were elected due to network partitions, Dgraph nodes could continue servicing reads below the new leader's Raft index. Dgraph fixed this issue in 1.0.5 by adding additional checks on timestamps.

4.7 Indefinite Transaction Conflicts

That single-UID set test illustrated another unusual failure mode for Dgraph: after network partitions, the cluster could get stuck in a state where every transaction writing a given key was forced to abort with a conflict, despite no apparent conflicting write transactions. This state could persist for hours after the network partitions had ended, and did not appear recoverable.

This stemmed from the same bug which caused lost inserts—transactions could obtain old read timestamps, rather than fresh ones, when a new Zero leader was elected. Transactions would read from a time hours in the past, go to commit, and discover that another transaction (long since completed) had modified their data during that time.

4.8 Unavailability with Network Partitions

Two other problems persisted in set tests. First, individual Dgraph nodes could lock up after partitions, causing all client requests to time out indefinitely. Second, partitions could push transaction timestamps far into the future, such that every client request would fail with an outdated timestamp. Clients would obtain sequential timestamps in the past, which would increment slowly. When this occurred, clients could return errors like

```
rpc error: code = Unknown desc = readTs:
 13272 less than minTs: 30014 for key:
 "\x00\x00\x04type\x02\x02element"
```

for hours, until their read timestamps caught up with the stored timestamp for that key.

These issues were caused by improper tracking of cluster leadership. Raft elects a series of leaders, each with a unique, monotonic term, for each cluster. In Dgraph, there is (at any given point in time) a highest leader for Zero, and a highest leader for each Alpha group. Zero leaders keep track of leaders across all these clusters, and stream that information asynchronously to each Alpha node, so they can route queries to the appropriate place. However, Zero proposed those leader transitions within a goroutine. If two leader transitions occured in sequence, and the first transition's goroutine executed after the second, Alpha nodes would learn the *old*, not the *new* leader, and be unable to make progress until a new leader transition occurred. This issue was fixed in 1.0.5.

4.9 Fragile Processes

We tested Dgraph with process crashes and restarts, but these tests frequently failed, because Alpha nodes would crash unexpectedly. If, on startup, Zero was unavailable, Alpha would retry for a short time, then kill itself. This made recovering from failure complicated, as nodes might *appear* to start correctly, then crash minutes later. A service manager might also conclude, from the repeated crashes, that Dgraph was permanently broken and should not be restarted again without operator intervention. Dgraph added a retry loop, which allows Alpha to recover once Zero becomes available.

4.10 Write Loss on Node Crashes

When Alpha nodes crash and restart, our set test revealed that small windows of successfully acknowledged writes could be lost right around the time the process(es) crashed. Dgraph also constructed records with missing values, as we saw in the bank and delete tests. Nodes would also disagree as to whether records were missing—objects could be returned from queries made to some nodes, but not others.

Dgraph suggested this problem was due to client-side sequencing, and that missing values might show up after a few seconds. Unfortunately, we found that missing values persisted even after all nodes had recovered and the cluster had been stable for thousands of seconds.

Worse yet, with server-side sequencing, Dgraph could occasionally lose *all but the most recent* successfully inserted value, instead returning `nil` for tens of thousands of records.

Losing all but the most recently inserted value is a suspicious bug to say the least, and its cause turned out not to be a distributed systems problem at all! When moving a predicate from one node to another, Dgraph builds up a batch of triples belonging to that predicate, before serializing them and sending the batch to the predicate's new node. However, the temporary data structure for serialization received Go slices (i.e. pointers) to a mutable loop variable which identified the key for that triple. This meant that before serialization, *every* triple shared the most recently iterated key—effectively overwriting every previous key in that batch. Copying the key variable, instead of passing a reference, resolved this issue.

This problem was exacerbated by a bug in client-side sequencing. When an Alpha node responds to a query, it includes a logical clock (the Raft index) of its current state. Clients use that clock to ensure that future reads against that Alpha group are monotonic. However, Alpha applies Raft updates *asynchronously*. Alpha originally returned an index based on the highest contiguously applied update. However, if the current transaction *wrote* data, then those writes may have taken effect at a *higher* Raft index than the local server has actually applied. This meant that a client could write some data in one transaction, and in a subsequent transaction, fail to observe its own writes. The fix is simple: Alpha now returns the highest contiguously applied update, *or* the last index of the transaction; whichever is higher.

4.11 Unavailability after Crashes

Process crashes also induced another type of downtime: nodes could return timeouts for all requests, despite every Alpha and Zero node running, and with total network connectivity. This unavailable condition would persist indefinitely, with Alpha nodes complaining `WARNING: We don't have address of any dgraphzero server.`

This error message was slightly misleading. In this scenario, Alpha nodes *could* connect to Zero, but were unaware of which Zero node was the *leader*. Although Zero streams the current Zero leader to each Alpha node, if an Alpha node were unreachable during a leader transition, that Alpha node would never receive the message about the new leader.

Dgraph resolved this issue by pushing Zero leader information to Alpha nodes periodically, so that isolated nodes would hear about any leader transitions that occurred in their absence.

4.12 Segfault on crashes

In addition to locking up, Alpha nodes could segfault when other processes crash. As with some of the bugs we discovered in the cluster join process, nodes which received requests shortly after restarting could attempt to handle those requests before fully initializing their local state, resulting in a null pointer dereference. Adding default values to functions involved in the current cluster state resolved the issue.

4.13 Migration Read Skew & Write Loss

Even after the patches for server-side ordering and with `@upsert` schemas for keys, Dgraph continued to exhibit occasional anomalies in the bank test. After a few minutes of normal operation, without any network or node failures, the total of all account balances would fluctuate up or down. Reads might observe accounts in the middle of transfer transactions with invalid balances—for instance, a test starting with $100 could appear to contain $102 instead:

```
:value {0 45, 1 2, 2 1, 3 6, 4 10, 5 7,
        6 29, 7 2}
```

… or show accounts with missing keys:

```
:value {nil 3, 0 1, 1 15, 2 18, 3 5, 4 9,
        5 17, 6 23, 7 12},
```

… or even missing balances, despite the fact that we never construct a record without a balance. In this read, $96 of an original $100 has evaporated, leaving only $4 in account 2.

```
{0 nil, 1 nil, 2 4, 3 nil, 4 nil, 5 nil,
 6 nil}
```

Since improper account totals persisted for long durations, we suspect that these illegal reads were also promoted, by way of transfer transactions, back into the database state—permanently creating or deleting money.

To illustrate this, we can plot the total of all account balances over time, as observed by transactional reads of all accounts. Colors denote the node queried for each read. In a snapshot isolated system, every read should return 100; however, in this test run, we abruptly lose 70–80% of our account balances, and reads on all nodes fluctuate between two different totals for several minutes, before devolving into chaos.

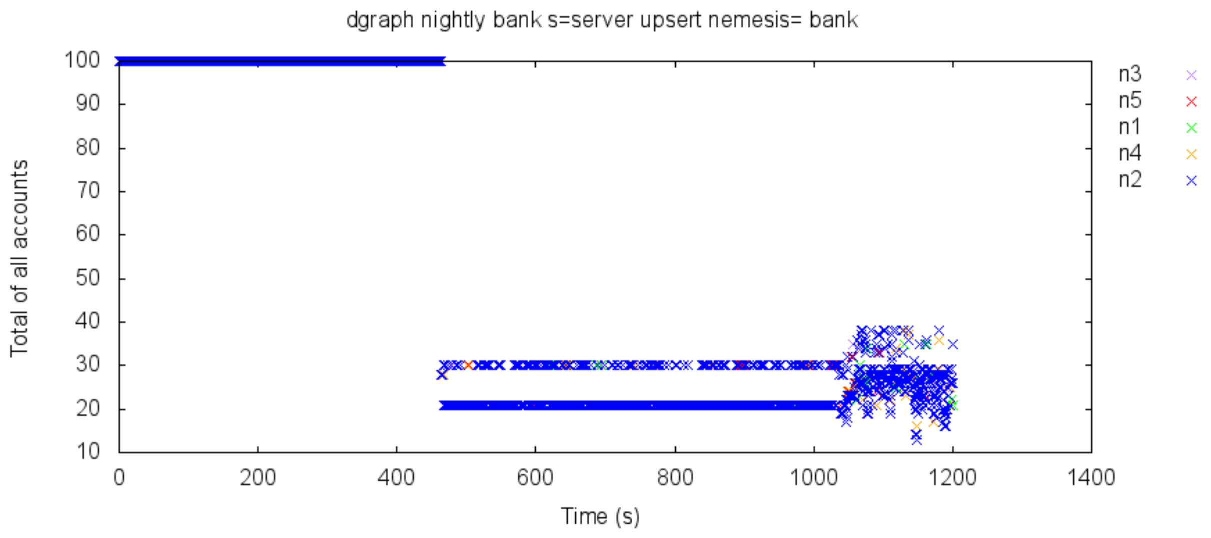


Figure 1: Plot of account totals over time, by node. After a predicate move, 70-80% of account balances appear to disappear, and reads fluctuate between two stable values for several minutes.

Reading alternating values over time could conceivably occur in a snapshot-isolated system if some (but not all!) reads from the same client observed a previous state in time. However, other test runs under similar conditions show what appear to be two independently evolving states of the system of accounts. In other databases, we might suspect split-brain, but here, both “worlds” appear visible to every node. We still don’t understand this phenomenon.

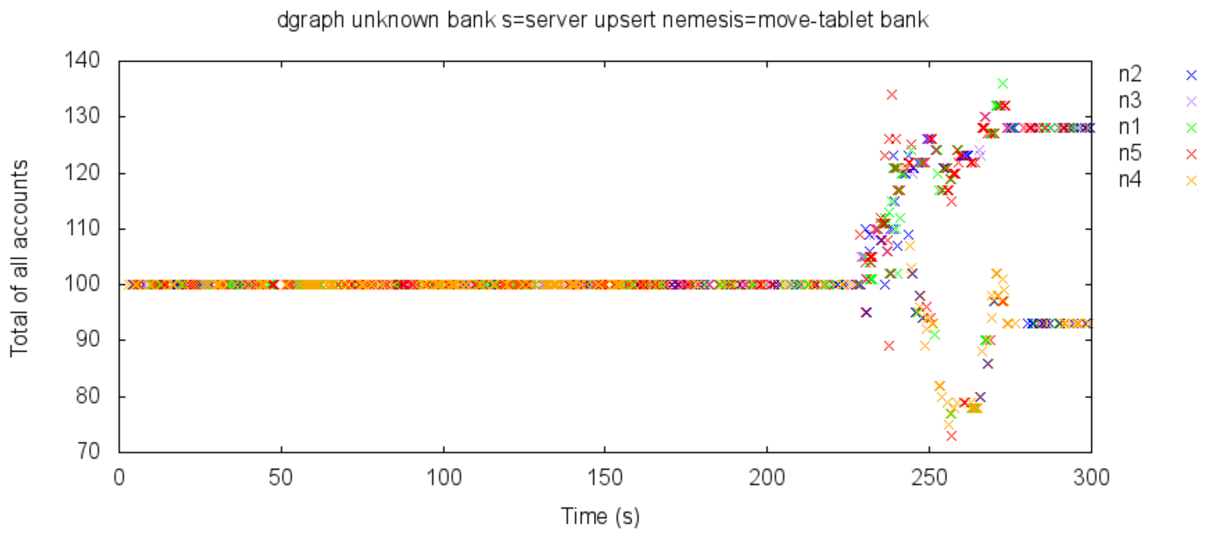


Figure 2: Plot of account totals over time. The system appears to split into two independently evolving worlds.

The start of these anomalies corresponds with Dgraph performing a routine migration of a predicate from one group of nodes to another—although anomalies persist indefinitely after the migration is complete. Moreover, the fact that we can observe missing values suggests that the problem is worse than simple read skew. We devised a nemesis[6] to stress predicate migration, and were able to reproduce this behavior much faster.

And we found something worse.

With server-side ordering, `@upsert` on schemas, and no crashes or network faults; e.g. in normal operation with the strongest possible settings, Dgraph would spontaneously lose successfully acknowledged inserts in the set test every few hours. As with the bank test, this behavior occurs when predicates are migrated. If we schedule partition migrations roughly every 15 seconds, Dgraph could reliably lose all but the most recently acknowledged insert in 60 seconds, returning hundreds of `nil` values instead.

```
{:ok-count 1,
 :valid? false,
 :lost-count 496,
 :lost
 "#{0 3 5..7 9 11..12 ... 1275 1277 1279}",
 :acknowledged-count 497,
 :recovered "#{}",
 :ok "#{1284}",
 :attempt-count 1293,
 :unexpected "#{nil}",
 :unexpected-count 1,
 :recovered-count 0},
```

This was the same same bug we observed with node crashes, but since migrations exercise the predicate serialization path heavily, they were especially likely to lose all but one record. This was fixed in 1.0.5.

Our data loss problem also involved a distributed race condition in Zero's commit path. If a transaction attempts to commit twice (say, because of a leader transition, or an internal retry), one of those commit attempts might abort, and journal that abort to Raft, while the second attempt could succeed, and journal that success to Raft as well. In this case, the second attempt would only verify that Raft had acknowledged the commit message, and wouldn't check to see if an earlier abort had already taken place. Dgraph fixed this by checking the transaction status after writing the commit to Raft, and before finalizing the commit.

In addition, a race condition between updates and predicate moves allowed Zero to commit transactions without checking to see if the affected predicates were being moved to other nodes. Zero now checks to ensure predicates aren't being moved as a part of the commit algorithm, and before moving a predicate, cancels any pending transactions as well.

4.14 Predicate Move Outages

Testing the fixes for migrations revealed an unusual failure mode: a predicate move could cause an Alpha

[6]In the Jepsen testing library, a nemesis introduces faults into a distributed system.

node to crash, and that crash would leave the rest of the cluster in a permanently unavailable state.

In this case, all requests to non-failing nodes would return `UNAVAILABLE`, as both Alpha and Zero nodes spun indefinitely, trying to establish a connection to the crashed node. Requests to the crashed node would time out. While Zero would respond to queries for the current cluster state, any attempt to make further predicate moves would time out as well.

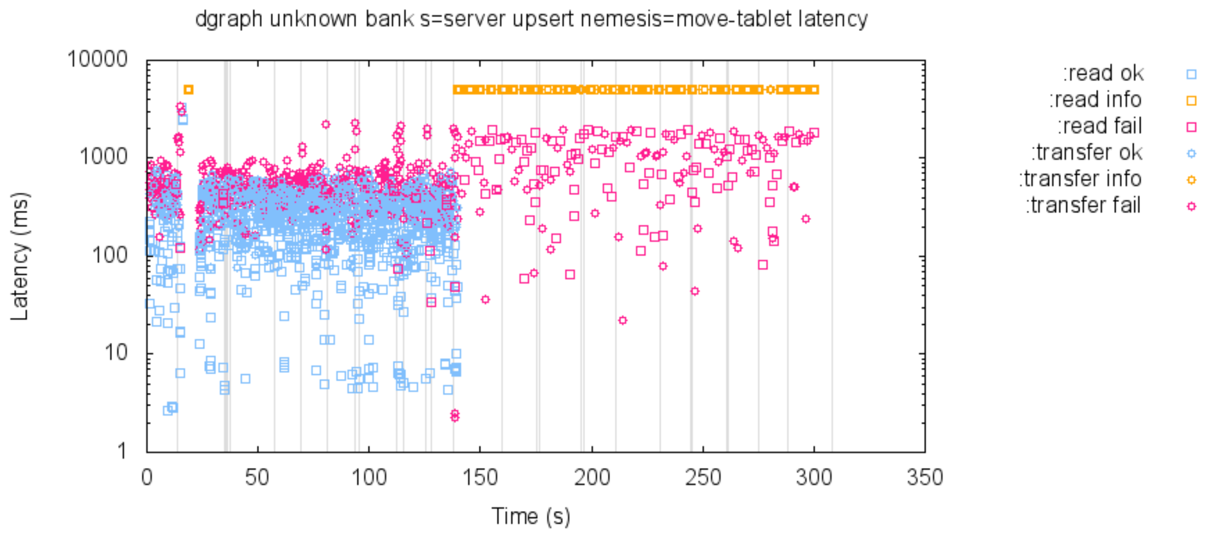


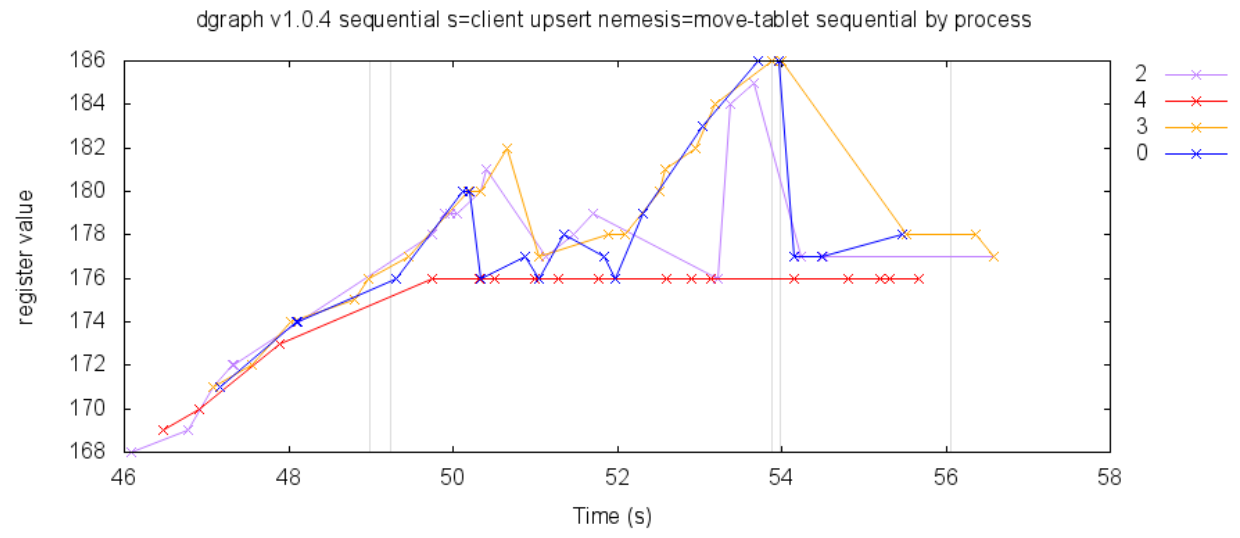
Figure 3: Plot of operation latencies over time. One node crashed at 140 seconds, taking down the cluster for the remainder of the test.

This problem arose because Dgraph would return deleted or expired values when iterating over storage, and return values which should have been overridden by a delete. Dgraph's integration tests had already identified this issue, and fixes are present in master.

4.15 Nonsequential Client-Side Sequencing

We developed the sequential test to check whether client-side sequencing ensures that clients observe monotonic states of the system. Dgraph's design concept documentation explains that clients keep a map of Alpha groups to the highest index they've observed on that group:

> In short, this map ensures that updates made by the client, or seen by the client, would never be unseen; in fact, they would be visible in a sequential order.



We expect that with client-side sequencing, different clients will read different points in time. However, each client should (independently) move forward in time, never backwards. Unfortunately, our sequential test revealed that in Dgraph 1.0.4, clients could observe newer, then older, states of the system. We measure this by incrementing (and never decrementing) a register—clients *should* observe that the register's value always rises, but instead, clients could see the value go down if predicates are allowed to move from group to group.

As discussed earlier, forgetting to make a transaction's own mutations visible to subsequent operations from that client could allow clients to observe non-monotonic states, but fixing that only reduced the *severity* of non-monotonic histories, and did not eliminate the problem. Dgraph also needed to fix a race condition between predicate moves and commits, and avoid returning deleted or expired triples.

4.16 Indefinite Periods of Query Timeouts

While confirming fixes for set tests, we discovered a new behavior: Dgraph clusters could lock up indefinitely when an automatic predicate migration occurred during a large read-write transaction. When the predicate move started, all transactions in progress would time out, and any future transactions would also time out. This condition could persist for hours, and affected the entire cluster. Dgraph is still investigating.

4.17 Read Skew In Healthy Clusters

As Dgraph addressed read skew issues caused by predicate migration, we began to observe bank test failures without any migration, or even any failures at all. Dgraph could still return incorrect account totals, or records with missing values.

During some types of network partitions, Dgraph could exhibit what appeared to be a read-only anomaly on isolated nodes—reads on that node could jump up or down, while reads against the rest of the cluster remained at the correct total.

However, read skew is not limited to read-only transactions, nor does it require network partitions. Without predicate migrations, crashes, partitions, or any other failures, healthy Dgraph clusters can exhibit persistent read skew anomalies. In fact, these issues occur even in single-node clusters.

This suggests that there may be multiple unresolved issues in Dgraph's snapshot isolation protocol. Dgraph is still investigating.

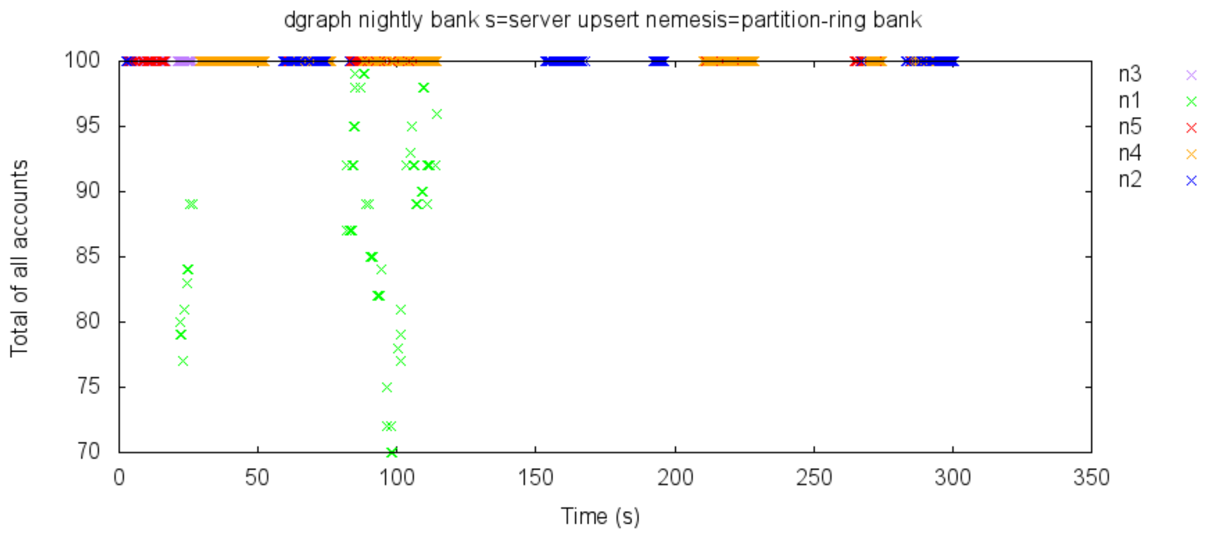


Figure 4: Total balance over time, by node, in a healthy cluster with no faults. Node n1 shows incorrect reads.

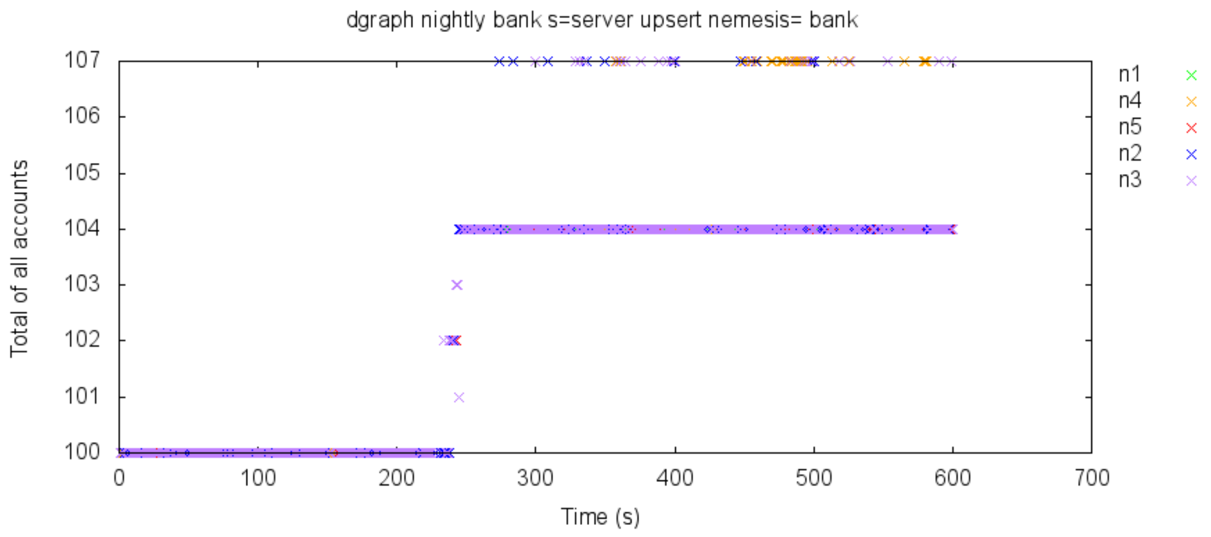


Figure 5: Total balance over time, by node, in a healthy cluster with no faults. The value abruptly jumps at 240 seconds, and fluctuates between two incorrect values.

5 Discussion

We identified multiple safety and liveness issues in Dgraph 1.0.2 through 1.0.6, including lockups and crashes in cluster join and node recovery, duplicate upserts, non-monotonic reads, snapshot isolation violations, inconsistent indices, records with missing values, and even lost inserts. Some of these issues, including write loss, could occur in healthy clusters with no faults. Note that predicate moves occur in healthy clusters, as Dgraph automatically rebalances data.

We wish to emphasize that Dgraph involved Jepsen quite early in their release process—Dgraph is barely two years old. The transaction system was initially built in two months, released in November 2017, and has only had nine months of polish since. We *expect* to find lots of bugs at this stage. It's hard to build a transaction system in that short a time frame, not only because of the engineering work involved, but also because users haven't had sufficient time to encounter, detect, and report bugs.

Moreover, we note that Dgraph has done this work entirely in public, with full issues and commit logs available for everyone to review. While there is still work to be done, we are heartened by Dgraph's commitment to improvement.

№	Summary	Event Required	Fixed In
2137	Join deadlock	Join	1.0.4
2138	Join segfault	Join	1.0.4
2143	Read skew, corrupt writes	None	1.0.5
2145	Join deadlock	Join	1.0.5
2148	Partially deleted records	None	1.0.5
2149	Duplicate upserts	None	1.0.4
2152	Loss of all but one insert	Partition	1.0.5
2159	Indefinite false conflicts	Partition	1.0.5
2273	Single node deadlock	Partition	1.0.5
2286	Join deadlock	Join	1.0.7
2289	Crash on startup	Zero unavailable	1.0.5
2290	Lost inserts	Crash	1.0.5
2312	Total Deadlock	Crash	1.0.5
2321	Read skew, corrupt records	Crash or predicate move	Unresolved
2322	Segfault on startup	Crash	1.0.5
2338	Lost inserts	Predicate move	1.0.6?
2358	Per-client non-monotonicity	Predicate move	1.0.6?
2376	Join deadlock	Join	Unresolved
2377	Integers inferred as floats	No schema	1.0.6?
2378	Integers coerced to floats	Int schema	1.0.6?
2391	Read skew, corrupt records	None	Unresolved
2397	Crash, total outage on pred. move	Predicate move	1.0.6?
2405	All queries time out indefinitely	Predicate move	Unresolved

5.1 Recommendations

Dgraph has made extensive progress this year, and addressed 19 of the 23 issues we identified. However, significant problems remain: Dgraph clusters can exhibit read skew and corrupt records even in healthy clusters with no faults. We recommend that users upgrade to version 1.0.6 or higher: while it does not address all issues, it does offer offer significant safety improvements over previous releases.

For performance reasons, Dgraph does not default to the safest possible settings. Users should consider the use of `@upsert` schemas and server-side sequencing carefully.

With respect to upserts: Dgraph's data model identifies entities by an autogenerated UID, but users may want to identify entities by some other primary key, e.g. an attribute like an email address or username. By default, Dgraph will not enforce transactional isolation for indices on these keys, which could allow users to upsert multiple copies of the same record, or, if the value for that primary key ever changes, to fail to observe the correct UID for that key. To prevent these problems, we recommend the use of the @upsert schema directive on any predicates which will be used as primary keys.

Dgraph has two methods for enforcing orders over transactions: client-side (where clients track Raft offsets for each group), and server-side sequencing (where servers check with the Zero leader to ensure monotonicity). With client-side sequencing, clients may observe stale data, rather than the most recent state. Server-side sequencing prevents many anomalies, but we have not developed a rigorous formalization of its guarantees.

Dgraph plans to remove client-side sequencing altogether, instead relying on the server-side ordering mechanism. Clients will likely track logical transaction timestamps, rather than low-level Raft offsets.

5.2 Comments & Future Work

While Dgraph adapted their transactional scheme from Omid, Reloaded, they chose to avoid relying on an external consensus service: Omid uses Zookeeper, but Dgraph includes multiple built-in Raft clusters. This approach simplifies Dgraph operations, but adds internal complexity. Conversely, Dgraph, like Omid, chooses to use Raft more as a consensus *service*, rather than allowing Raft to control the entire Dgraph state machine. This allows Dgraph's state machine to execute operations in parallel, improving performance, but that concurrency introduced subtle nondeterminism and race conditions. Negotiating the balance of performance, operational complexity, and algorithmic simplicity remains a difficult challenge for distributed systems engineers.

Keeping track of leaders has led to liveness and safety issues for many systems Jepsen has analyzed, and Dgraph is no exception: we found several bugs caused by cases where nodes could not identify a leader, or believed the wrong node was the current leader. While Raft provides a well-behaved leader election system with understandable invariants, keeping track of leaders *externally* to Raft, and managing the relationships between multiple Raft clusters, proved difficult. While timeouts, stepping down, and streaming cluster state updates can mitigate these issues, we suggest that implementers ensure every node can safely handle requests intended for a leader, and that non-commutative operations performed by old leaders are correctly rejected by peers. Assuming a "leader" node is authoritative, or that there can only be one leader at a time, often proves dangerous.

Many of the problems we found in cluster join are essentially race conditions with uninitialized state: Dgraph nodes would accept requests *concurrently* with the node initialization process, then deadlock or segfault when request handlers observed invalid initial state. These problems can be addressed by forcing initialization to complete before listening, but Dgraph opted for a more robust approach, adding sensible default values for uninitialized nodes.

The availability issues we found—crashing on startup, or failing to broadcast leader state periodically—may reflect a lack of hours-in-production; these errors are easy to detect once the requisite failure conditions have occurred, and the underlying bugs are straightforward to fix. In general, this points to the importance of *reconciliation loops*: systems which experience partial failure can adapt by having a *goal state*, and a process which continuously attempts to make progress towards that goal.

We have not yet investigated filesystem-level errors, but Dgraph has tested their storage system for crash consistency using ALICE. We have not tested Dgraph with clock skew, which could impact Dgraph's timeout-based lease allocation system. We would also like to formalize Dgraph's real-time constraints on snapshot isolation, and to develop more rigorous tests for those properties.

Dgraph includes two distinct subsystems: Alpha, and Zero. Our tests colocated a Zero node with each Alpha node, and partitioned nodes from other nodes in total. This means that in any given network configuration, Alpha and Zero share the same set of visible nodes, and the same majority & minority network components. We rarely obtain cases where a majority Alpha component can only interact with, say, an outdated Zero leader in a minority Zero component. In future testing, we would like to separate Alpha and Zero nodes, and introduce different network fault topologies on each, to explore more of the state space.

We developed tests for Dgraph's linearizability, but have not fully investigated whether server-side ordering actually provides linearizability, or allows some non-linearizable anomalies, such as stale reads. Since Dgraph is still struggling to provide snapshot isolation, these tests would be premature, but we think they'll be useful down the road. We would also like to clarify the *scope* of monotonicity with respect to client- and server-side sequencing: are only transactions on the same keys ordered? What about transactions with overlapping, or disjoint keys? These remain topics for future research.

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 invaluable assistance, especially Manish Jain, Pawan Rawal, & Janardhan Reddy. Peter Alvaro gave advice on consistency levels, and extensive feedback on this report. We are also grateful to Marc Hedlund, Sarah Huffman, Tim Kordas, & Camille Fournier for their feedback on early drafts.